ATTACHING AN ARRAY PROCESSOR IN THE UNIX ENVIRONMENT

by

Clement T. Cole

Memorandum No. UCB/ERL M83/23

12 April 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ATTACHING AN ARRAY PROCESSOR IN THE UNIX ENVIRONMENT

Clement T. Cole

Electronic Research Laboratory

Department of Electrical Engineering and Computer Sciences

University of California

Berkeley, California 94720

ABSTRACT

In this report the work necessary to attach a commercial array processor, the Floating Point Systems FPS-164, to a VAX 11/780 running 4.X BSD UNIX is described. The FPS-164, its surrounding program development system, and the interaction with potential application programs are also presented.

April 8, 1983

1. Introduction

This is a report concerning issues related to the use of an array processor attached to a UNIX machine for the purposes of running CAD/CAM application software. After a brief examination of the hardware involved in the project, the software system built upon that hardware is described. First, a justification for the use of such a system is presented. The ultimate goal is improved overall system performance measured on the basis of cost effectiveness.

Throughout the report the term *attached array processor* refers to a single processor attached to a general host computer for the purpose of speeding up certain classes of computational tasks. It takes the name *array* because most of the tasks maintain their storage in one-dimensional (vector) form. The machine is not a *standalone* computing engine, in that it is *attached* as a peripheral to some other host computer. The host is typically an *minicomputer* or *midicomputer* that includes hardware to retain large amounts of data, communicates with users or other computers and is usually able to run the given application on itself is a somewhat limited manner.

The complete case for attaching an array processor is not offered here. It has been presented elsewhere.[Col82, Kar82] Instead, this report covers the requirements for attaching such a processor and how an application such a SPICE2[Nag73, Coh76] uses such a processor. This work is a step in an on going research effort aimed at improving the performance of electrical simulation[Coh81a, Vla83, Qua83] and device analysis[Old79, Old80, Nan81] for integrated circuits (ICs). Early results have indicated a *minimum* of a 2.0 to 3.5 time speed increase for SPICE2 compared to the the VAX 11/780 UNIX program using the VAX floating point accelerator. Preliminary results indicate that a speed up of up to 4 times (a 5 to 10 times total speed up) would be possible by use of compiler optimizations and clever

programming.

1.1. Overview

The format of this report is as follows: An introduction and overview; some results of the effort; examples of concepts associated with parallel processing in general and array processing in specific; an examination of some fundamental array processor hardware implementation issues, along with a presentation of the hardware architecture of the FPS-164; a examination of software issues for this computing system; a description of the conversion process itself; a few suggestions for future work.

The intent of this report is to be a detailed description of basics of attaching an array processor, to act as a guide to the software, and to give a starting point for further work.

To provide a basis of comparison and establish some terminology, the report includes some examples of contemporary computers and the classifications normally applied to them according to their use. Typically, they are segregated on the basis of instruction rate, memory addressing capability, floating point operations performed per second, etc.. These terms are found in the Appendix. Also included in the Appendix are the software terms used throughout the report. A reader unfamiliar with this terminology should review the Appendix. Of particular interest are the names of the different programs which make up the entire FPS-164 software system.

1.2. Technology Issues and Formal Computer Classes

According to Enslow,[Ens77] four system characteristics which serve as motivation for continued development and use of such parallel processing schemes are:

1.) Throughput

2.) Flexibility

3.) Availability

4.) Reliability

This report focuses on *achieving greater throughput* for the system. (More work per unit time without significant compromise of any of the other characteristics). The way to increase the throughput is to decrease the time it takes to perform a given computation. One way to do this is simply to use faster electronic parts. As the switching times of modern devices approaches the physical limits imposed by the speed of light and as propagation delays along the *computational path* become more significant, alternative approaches to speedup are necessary. In theory, given a computational task which takes T time units to execute, if it can be partitioned into N independent tasks then applying N processors to the problem could result in consuming only T/N time units to execute - a speedup of N times. While this limit is rarely achieved for practical problems, a well-partitioned algorithm can gain a significant improvement in the throughput from the N processors.

Flynn[Fly66] classifies computer systems into four groups, depending on properties of the instruction and data streams.

Typical uniprocessors (serial computers) are classified as:

1.) SISD: single instruction - single data stream.

Parallel computer systems are classed as:

2.) SIMD: single instruction stream - multiple data stream

3.) MISD: multiple instruction - single data stream

4.) MIMD: multiple instruction - multiple data stream.

Parallelism is achieved by replicating the instruction and/or data streams.

Typically, the faster machines employ techniques which classify them in one or more of the 3 parallel categories. From *minicomputers to supercomputers*, the move is from essentially no parallelism to a large amount of parallelism and, as a consequence, the speed and cost of the computation increase proportionally. Siewiorek, Bell, and Newell[Sie82] represent this graphically in Figure 1.

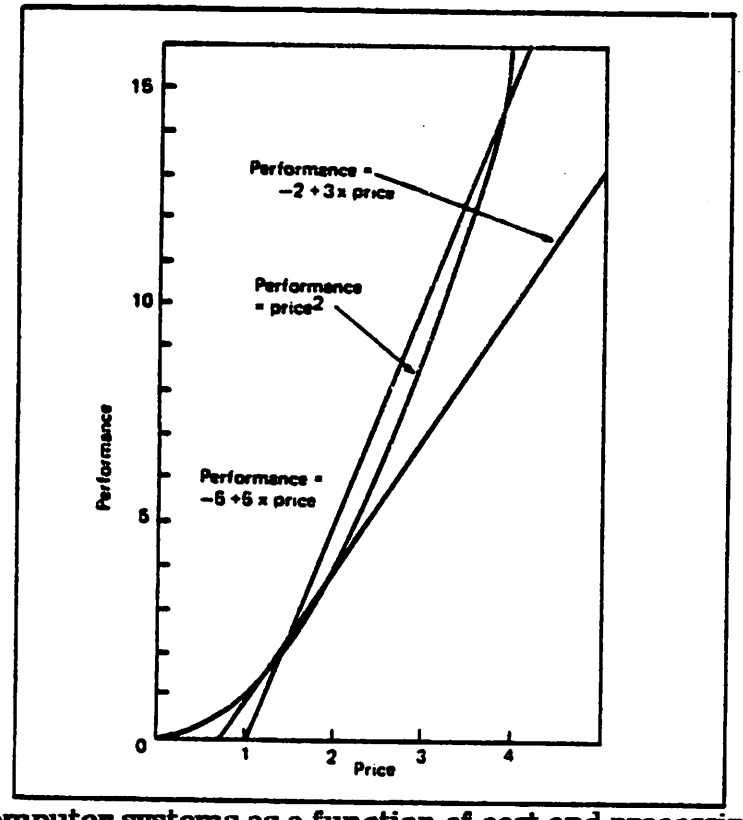

Figure 1. Computer systems as a function of cost and processing rate.

1.3. Cost Effectiveness Issues

Consider a simple interactive task, such as *writing and editing* a program, the speed and computational power of the supercomputer are quite underutilized, and the money spent to procure such a system is hardly justified for this type of computational task. A supercomputer must be kept busy a significant portion of the time to be cost-effective.[1] However, consider a task which consumes hours of cpu time, such as circuit simulation of a large integrated circuit, or logic simulation of a sophisticated central processor. The cost associated with the supercomputer is weighed against the fact that the simulation may not even be *able to be run* on a smaller machine. The ability to run a program on a machine is a concept which considers both the space the program and associated data occupy as well as the runtime of the program. For example, if a simulation takes 10 hours of CRAY-1 cpu time to run to completion, it might take on the order of a week of VAX cpu time to run to completion. In most cases, such a problem would be considered *unable to be run* on the VAX. When the ability to run a program is considered, the user must make a the assumption that the computing engine will not fail during the whole of his program run time. The longer the program runs, the more chance of a failure.[2]

Even though the owner of a midicomputer or minicomputer may have the legitimate need for the sophistication and speed of a supercomputer, the enormous initial costs and continuing maintenance costs associated with such a machine (roughly an order of magnitude greater than the typical minicomputer/midicomputer system) could easily prove to be an insurmountable obstacle. The cost of machine is not just the simple outlay of dollars for the

[1] Cost effectiveness might be defined as: (cost per run) = (cpu time) * (cost per second), where (cost per second) = (purchase price)/(60,000 hours) * 3600 seconds/hour. [Mar81] A finer analysis would consider programming and software development time, maintenance, etc. which is examined later in this section.

[2] A failure could be most anything, from human error, to power loss; from hardware malfunction to operating system malfunction.

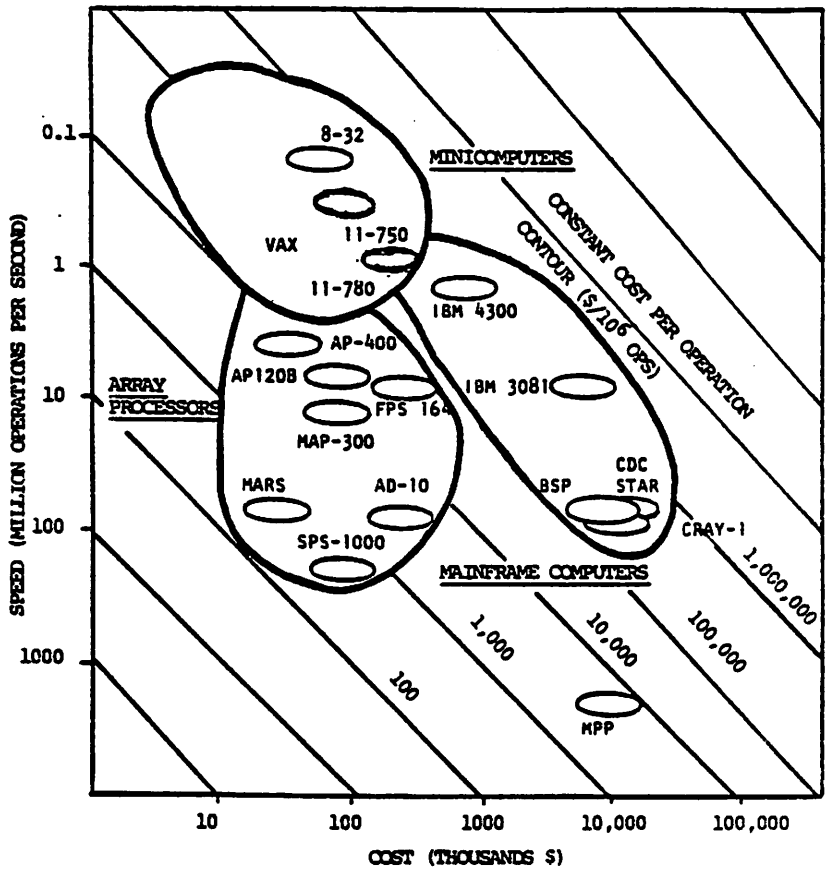

Figure 2. Speed vs Cost. Kar81

the cpu. The size of the maintenance staff (both software and hardware) is related to the size of the machine - the larger the machine, the larger the staff to maintain it. An alternative to this situation is found in the *attached array processor*. This processor, being attached to a minicomputer or midicomputer, does not significantly increase the maintenance costs because it to not a complete host. Instead it increases the complexity of the total computing system, HOST + AP, slightly, but not as much as a two completely separate hosts.[3] Thus it can be operated and maintained my the current existing staff, while still holding the capital outlay to a minimum.

The issue is illustrated graphically in Figure 2. The domain of problems represented by this curve includes numerically intensive problems, such as

[3] The "it's easier to administrate" argument was one of the arguments presented when the "dual processor" host such as the Purdue University Dual VAX 11/780 Gob81 systems were created.

those solved with the FORTRAN programming language but does not include interactive tasks, such as editing and compiling. Figure 2 shows constant cost-per-operation-contours. The machines of interest here are the VAX 11/750 the VAX 11/780, the IBM 4341 and the IBM 3081. If you assume a cost of between $100K and $150K for a VAX 11/750 and a cost of $250K to $500K for the VAX 11/780, the observer will notice that both of these machines sit on the same cost contour as the machines which cost in the ranges of $450K to $1M for the IBM 4341 and $5M to $10M for the IBM 3081. The array processors are situated on a contour below that of the CRAY-1 style supercomputers, meaning that they give the user more processing power for the dollar spent. The observer should note that these type of machines give the same amount of raw processing power as that of the maxi-computer like the IBM 3081, but at the same price as the VAX 11/780. In the curves presented, Karplus has included only the cost of the array processor and has assumed that the user of the AP, already owns the minicomputer or midicomputer with with it is attached. This is not unreasonable because, when the array processor is added to the host, the host is still available to be used as a processor in its own right. Even if the host is near saturation in use, the array processor need not be a significant load as shown later. Most likely, the array processor will help reduce the load on the host.

2. The Attached Peripheral Array Processor

Historically, the term array processor has been associated with all of the previously defined classes of computer systems (SIMD, MISD, MIMD). As examples, there are systems composed of arrays of processing elements (SIMD) such as the ILLIAC, systems designed to process arrays of characters or strings (MISD) such as the CDC 6000, systems designed to facilitate the processing of vector arrays of data (SISD, MISD, or MIMD depending on the particular system) such as the CRAY-1.

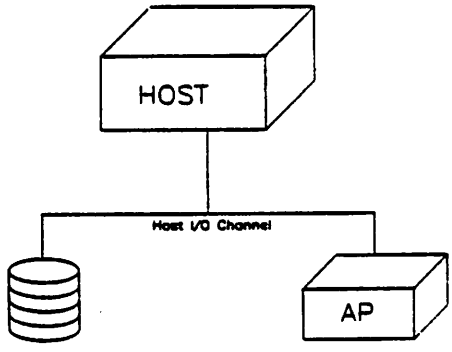

Figure 3. An HOST with an attached AP

To distinguish these types of systems from the attached array processor, a more general definition is necessary and some distinguishing features of attached peripheral array processors must be examined. The term *array processor* as used in this report, refers to a single peripheral processor which is attached to a general purpose host computer system so that the tandem combination provides a much greater computational (number crunching) capability

than achieved by the host computer system alone. [The81] Figure 3., is an example of such a connection.

In terms of Flynn's classes, the array/host configuration can thus be viewed as an MIMD architecture,[4] with the number of processors $Np \geq 2$.
Karplus and Cohen[Kar81] distinguish an attached array processor as a special digital computing device with all of the following features:

1.) It is designed as a peripheral for a conventional host computer and is limited to enhance the performance of the host in specific numerical computing tasks.

2.) It achieves high performance through parallelism and/or pipelining.

3.) It includes an arithmetic section containing at least one adder and one multiplier capable of operating in parallel with each other.

4.) It can be programmed by the user to accommodate a variety of arithmetic problems.

5.) Typically operates on data in multi-dimensional (array) form.

A simple explanation of two of the main points are described in the appendix. The serious reader should refer to these sections to obtain a detailed understanding of the interaction between the speed of a computation and the implementation of an array processor.

2.1. The Floating Point System's FPS-164

The specific attached array processor used in the project is the Floating Point System's FPS-164.

The hardware architecture of this computing engine is complex. The machine contains a 64 bit word, 8 functional units, 7 high speed data paths and

[4] It is likely that parallel I/O capability will exist. This is just another processing element.

one low performance interconnect to a host a VAX 11/780 Unibus Adapter, an APOLLO Multibus, or an IBM Channel.DEC79a, Apo81, IBM70

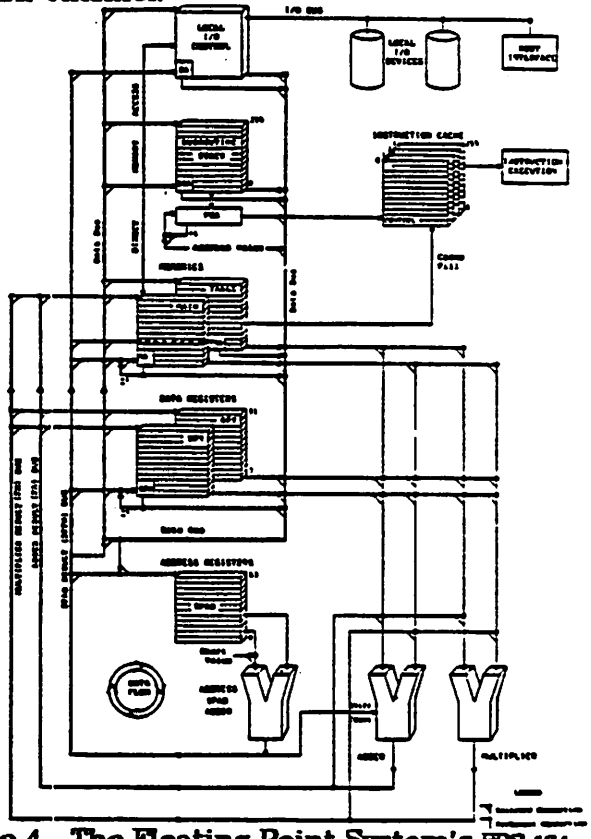

Figure 4., The Floating Point System's FPS-164.

The FPS-164 is contains three primary architectural features: (see Figure 4.)

- multiple functional units
- multiple interconnects
- multi-operation instructions

In order to exploit the full power of the computing engine, all 8 functional units must be kept operating at the same time. The FPS-164 is a dyadic computer, which means that each functional unit is capable of operating on two items at once. Thus, during each instruction cycle the AP can perform either two data computations, two memory accesses, an address calculation, read two internal registers, write two internal registers, or initiate a conditional branch. Each

functional unit is pipelined so that on each 167 nanosecond cycle, the AP can move data from one pipe stage (segment) to another. Figure 5, shows the four independent types functional units of FPS-164.

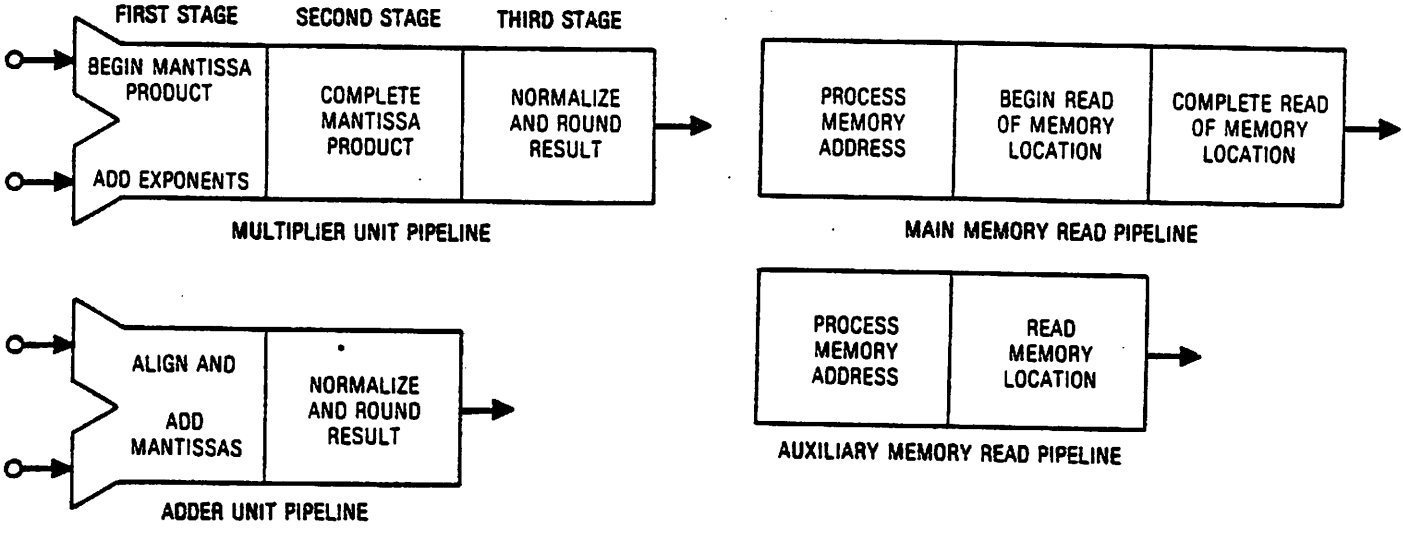

Figure 5. FPS functional unit stages.[Cha81]

These functional units consist of either two or three segment pipelines and may run in parallel. This means that a complete arithmetic operation is not completed until two or three cycles after it is initiated. The short pipeline size means that a small penalty will be paid when the pipeline must be flushed.

Unfortunately, this relatively short pipeline means that that machine will not perform as well as a longer pipelined machine such as the CRAY-1 for long vector operations without software aids.[5]

2.1.1. Functional Units

The important features of the FPS-164 are:

Arithmetic Functional Unit. The *adder unit* performs floating point additions and subtractions and data format conversion. It also performs integer arithmetic, shifts, and divide/square root approximations.

Multipler Functional Unit. The *multipler unit* performs integer and floating point multiplication.

Memory Functional Unit. The primary store for the data words is main memory. It is also the secondary store for instruction words. It has a size maximum of 12 megabytes due to the current memory technology (16K-DRAMS). The memory unit also contains a set of simple base and limit registers to aid in relocation of data and instruction text.

Auxiliary Memory. The *auxiliary memory* is smaller and optional. It is used to contain read only constants such as cosine tables, specifically used for FFT algorithms. This allows frequently used constants to be loaded into the AP once, instead of every time a program is run. The size of this memory may range from 64K bytes to 256K bytes.

Data Registers. The *X and Y data registers* each contain 32, 64 bit registers. One register per register file can be read and one per file can be written per cycle.

---

[5] Such as reworking the application program's internal data structures to tune them towards the pipeline structure of the *specific* processor that will execute the program. In the case of the CRAY-1, general experience has shown that data structures that are tuned in "8's" (or multiples there of), tends to perform better than data structures that are say of size "5". The CRAY-1 is built around 8 registers, 8 pipeline stages etc.

Address Calculation Unit. The *address calculation unit* adds, subtracts, and performs logical operations on memory addresses, loop counters and integer data contained in the 32 bit wide address registers.

Address Registers. The *address registers* are similar to index registers of a normal computer. The FPS-164 contains 64 of these.

Instruction Memory Unit. The *instruction memory unit* provides for the primary storage of instructions. This separate memory allows instruction fetchs to take place simultaneously with data fetches without mutual interference. Unfortunately, the instruction memory is only 32k bytes in size. The FPS-164 can fetch code from primary memory but at the cost of performance.

Branch Control The *branch control* logic is used to decide which instruction to execute next in parallel with the completion of the current instruction. The FPS-164 tries to execute both parts of the branch if it can, (i.e., if there are enough functional units available). In this way it is not penalized significantly by a branch.

2.1.2. Interconnection

The FPS-164's multiple functional unit architecture means that data paths must exist between the functional units in order to keep all of the units operating. Figure 6. shows the interconnection topology of the FPS-164.

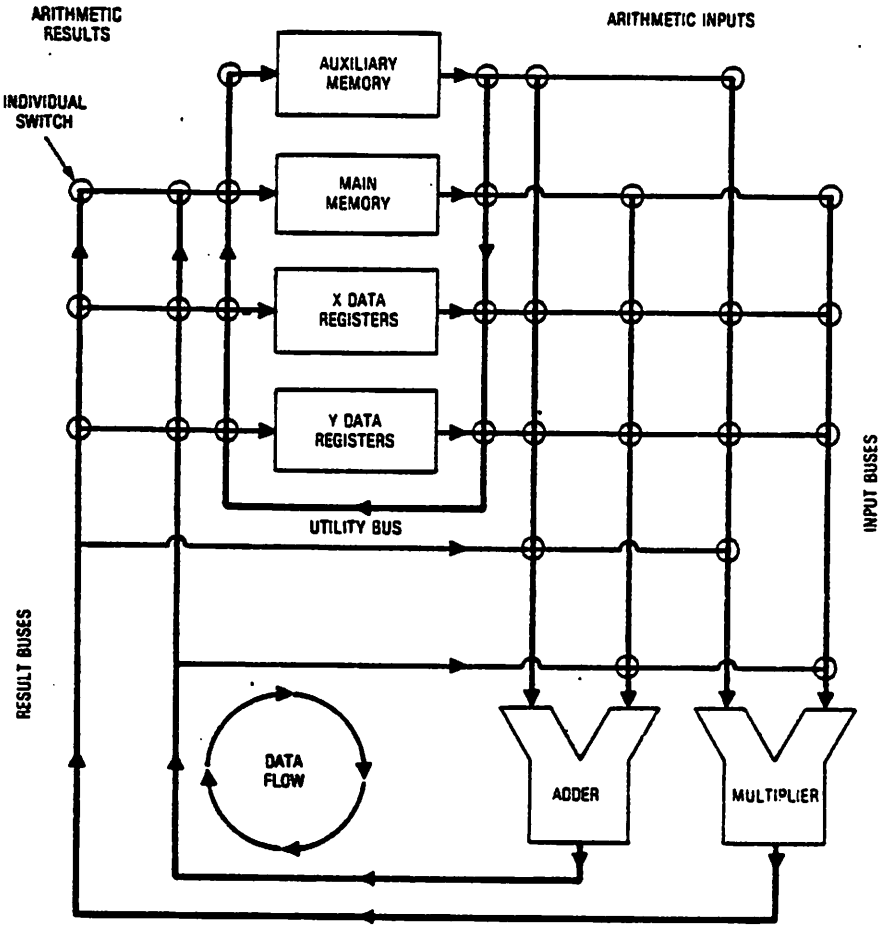

Figure 6., The FPS-164 interconnection network.

The FPS-164 can produce six individual results at one time, from:

- the multipler
- the adder
- the X register
- the Y register
- primary memory
- the auxiliary memory

There are eight possible inputs for those results, from:

- the two memories,
- the X register
- the Y register
- any of the four arithmetic operators

Due to the physically large number of interconnections, the FPS-164 only uses 30 of the possible 48 interconnections. For a 64 bit wide word, this means the FPS-164 contains an interconnection network of 1920 connections. The crossbar is implemented with seven dedicated busses each 64 bits wide. Four of the busses are used to supply operands to the arithmetic units, and two are used to carry the results away. The seventh bus is a utility bus which is not dedicated to the processor and is used to link the register file to the memory file for fast register flushing and loading.

2.1.3. Multi-operation Instructions

To run at full speed, the 64 bit wide word of the FPS-164 is divided into 10 subinstructions, called "parcels". (see Figure 7.) Each instruction in the FPS-164 can be thought of as direct access to the micro-instruction of a conventional processor. Parcel's are analogous to single instructions on a conventional computer. The programmer (or compiler writer) is responsible for keeping all of the parcels filled and non-conflicting. If all parcels are filled, then 10 simultaneous operations will take place. By sharp programming, a complete vector loop is possible in one complete FPS-164 instruction. With the 167 nanosecond cycle time, and if all parcels are kept filled, the machine will operate at 60 million total operations per second which becomes 12 million floating point operations per second (12 MFLOPS).

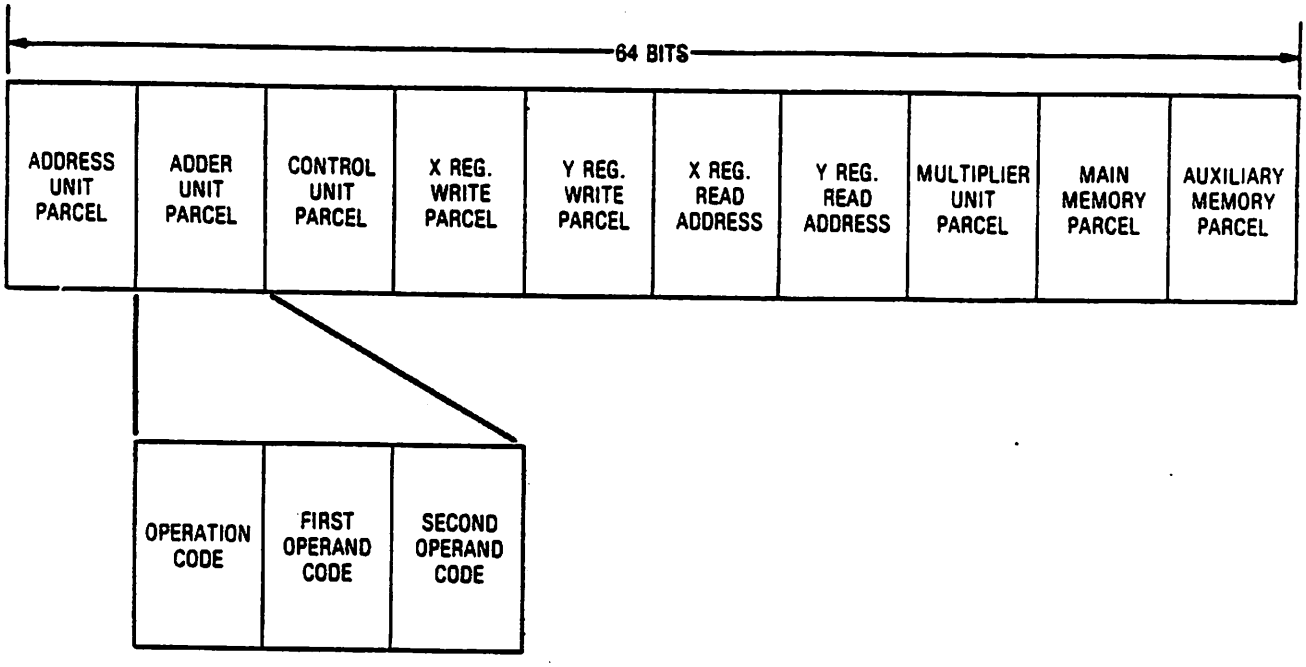

Figure 7., The FPS-164 Multi-Operation Instruction Format.

3. The Software System

As with an conventional processor, an array processor must be supported by some set of utility programs that allow the program to create and execute programs to run on that computing engine. The FPS-164 is no exception. The name for the this set of programs is the PDS, for Program Development Software. Included in these utilities are an assembler, a linker, a librarian, and a compiler.

3.1. The Compilation Process

In a conventional computer, a program writer takes the source to his program and submits it to a translator, such as a FORTRAN compiler or an assembler for the HOST machine. The output of this program is an object module, that contains the set of machine instructions that represent that program entity (subroutine or complete program). This object can be archived together in another

file, with a *librarian* such as the UNIX archiver, ar(1).[CSR81] The libraries, along with other object modules, may be *bound* together to produce a executable program with a *linker*, such as the UNIX linker, ld(1).[CSR81] Figure 8, is an representation of this process.

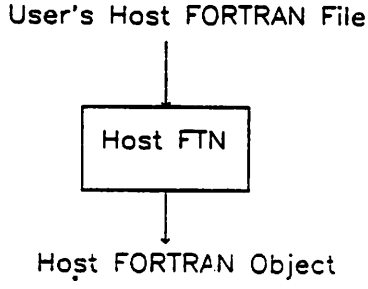

Figure 8., Standard Compilation Process.

The standard compilation process breaks down with an array processor because it has been built the assumption that *every instruction generated by the translator will be executed in what appears to be a local manner*. On an attached array processor, instructions are being executed on the behalf of the user in a *peripheral*. The standard HOST translators have no knowledge of this peripheral so they can not produce a set of instructions to be executed on the peripheral processor. For a first-order view, we simply replace the standard HOST translators with ones for the AP. The problem with this view is that when a user runs a program on the HOST, the HOST needs an object module that contains

HOST instructions to execute and the AP's translators emit instructions for the AP.

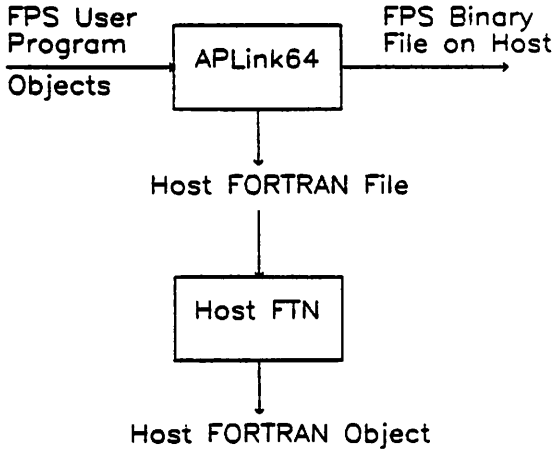

Figure 9., AP Compilation Process.

As a result, the programmer must pick the set of routines that he wishes to be executed remotely. These routines are submitted to either the assembler or the translator for the AP. The programmer then binds together all of the routines that will run on the AP with the AP's binding program (APLINK64 in this case). The array processor's binder will emit a module that contains the instructions that run on the AP. The problem is that the "main" part of the program runs on the HOST.

Notice that there is a gap between the HOST and the AP: the AP translator creates instructions for the AP and the HOST's translators create instructions for the HOST. This rift is spanned by a feature of the binder. Along with the execution module for the AP, the AP translator also emits a set of subroutine stubs in *HOST FORTRAN*. The module of stubs is called the HASI. (HOST-AP Software Inter-

face). The HASI is input to the HOST FORTRAN compiler to produce a HOST object module. The object module can then be bound with the main object module and other objects to produce a HOST execution image. The resultant image is a "complete" HOST program that can be executed locally. Contained within HASI stubs are a set of calls to the AP's runtime executive (APEX). Figure 9, shows the AP compilation process.

This is the "classical" way the AP is used, but if the AP has been equipped with a local disk, the AP can be used more efficiently.

3.2. APEX/SUM

The AP Executive is actually a collection of routines that are written for the most part HOST FORTRAN.[6] This collection of routines has the major responsibility for communication and synchronization between the AP and the HOST. The operations performed by the HASI through APEX calls are[FPS82a, FPS82b]

[6] HOST FORTRAN was used for all of the HOST independent routines, and most of the dependent routines. For the actual I/O calls and the I/O waits etc, there exists two C modules that are FORTRAN callable. This made the UNIX interface a bit simplier.

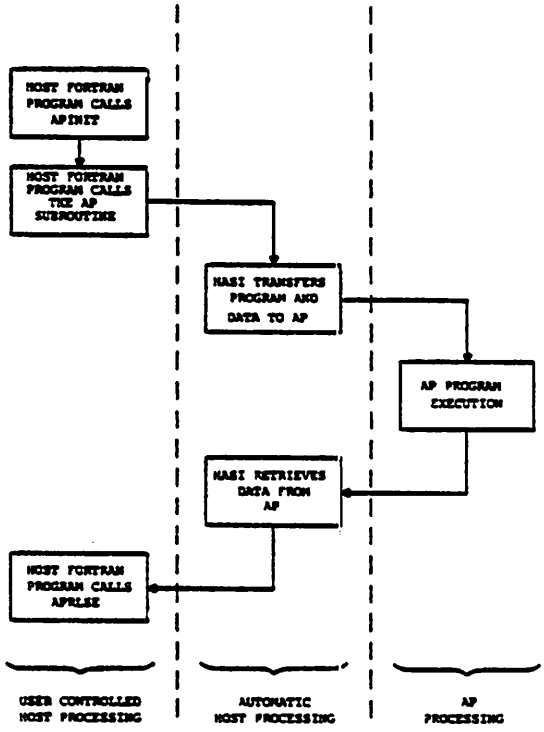

Figure 10., HOST - APEX - SUM - AP relationship.

1. Assign a AP to the running HOST process
2. Initialize the APEX data base for later communication
3. Initialize the AP if need be.
4. Initialize the HISP if need be.
5. If need be, load and initialize from a HOST disk file, the SUM image binary.
6. Load AP local routines for the HOST process from either HOST process memory or for a HOST disk file.
7. Transfer data from the HOST process image to the AP process image.
8. Transfer data from the AP process image back to the HOST image.
9. Start up an AP process.

10. Handle AP exception conditions.

11. Provide AP debugger support if need be.

12. Terminate an AP process if need be.

13. Release the AP when the HOST process terminates.

The APEX calls cooperate with a set of routines in the SUM. It is the SUM that actually does much of the work on the AP. Below APEX and the SUM is the device driver and the HISP (Host Interface Support Processor).[FPS82c] See Figure 10, for a view of the HOST/APEX/SUM/AP distribution. The driver specifics are covered later.

When an application program is written with APEX linkage to AP routines, it is written to be executed on the *HOST*. The program will call routines that will execute on the AP. Because the program has been written for execution on the *HOST* all I/O (to terminals, files, devices etc.) is directed to the HOST. Thus is does not make sense to attempt I/O in the AP. In order to perform I/O from the HOST, a protocol would have to be maintained by the two processors to insure that the data move correctly between them. Such a protocol would be overly complex and does not exist.

As a result, the programmer of an application must make sure that all routines running on the AP are free of any I/O statements.[7] This results in having only the most computationally intensive portions running on the AP. There is an obvious inefficiency with this method of execution. The data for each subroutine must be moved between the HOST and the AP each time the program transfers control from the HOST to the AP or vise versa.

[7] For a large application program, such as SPICE, the task of removing that I/O is large.

With large data arrays, considerable overhead is associated with keeping valid the data in memory on both processors.[8]

3.3. SJE/SFM

The single job executive (SJE) is a way to use the AP without requiring the programmer to decide *a priori* which modules will best utilize the AP. With the SJE, the entire program runs on the AP. Here in, lies another problem: something must run on the HOST to actually get the AP started. As described previously, APEX is responsible for starting programs and the transfer of images from the HOST to the AP. To this end, the SJE is a standard HOST FORTRAN program that makes calls to APEX. (see Figure 11.)

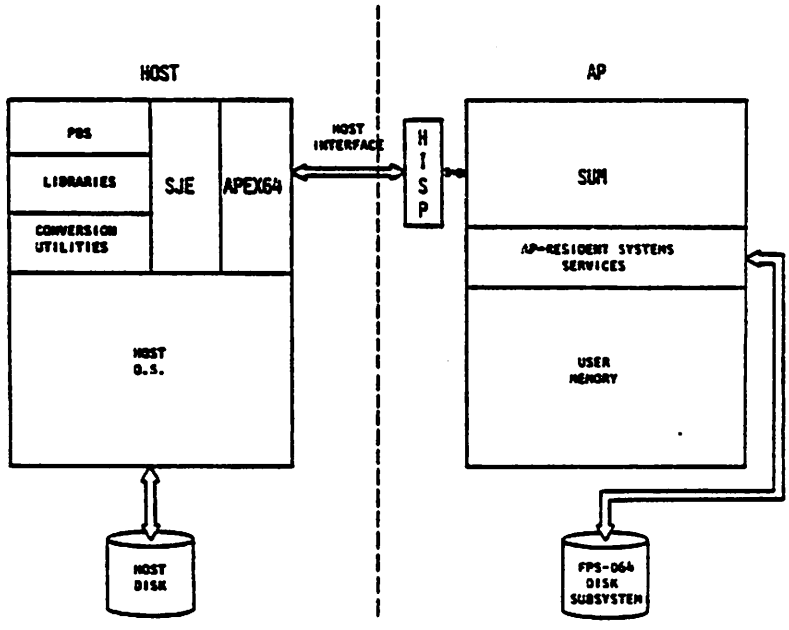

Figure 11., SJE - APEX - SUM - SFM relationship.

It is a conversational program which communicates with the application's *user*.

[8] The data validity problem is a classic problem in distributed systems.

Notice that when writting an application program to run under SJE, the programmer uses the AP's translators just like he used standard HOST translators. It os not necessary to have the binder produce the HASL. SJE will move the image from the HOST to the AP. When using an SJE bound application program, it is the *user* of that application who must be conscious of where the program is running.

The difference between standard APEX mode and SJE mode for a program should be examined. SJE need not move data from the HOST for the active AP application while it is running.[9] Instead, the user of the application moves his input data files from the HOST over to the AP's "local" disk. Once the application has finished, he must move the results back from the AP local disk to the HOST's local file system. The portion of the AP's operating system that maintains the file system is referred to as the Scratch File Manager (SFM). It takes this name because files are not permanent. Each time the AP is reloaded, all files are lost. Hopefully, future versions of SJE and the SUM will allow a more permanent way of retaining data.

A major limitation to the SFM is the size of a file. When a file is created, all of the space for the file is allocated. Thus when a program runs, if it does not use all of the space allocated to it, it "wastes" space on the disk. If the program needs more space than requested, it is terminated prematurely. This is problem if a user of SJE is interactively running a single application many times. If the application program must create the output ahead of time, then it will have to make it "large enough" for that data run. If the user then runs the program, but is running many small runs, he will see the disk fill up even though he is only using a small amount of space.

Of interest is the SFM/SUM relation to a broken program. It has been observed that when a program terminates prematurely (from a divide by zero

[9] Other than the normal terminal I/O traffic, which will be obviously low bandwidth data

fault, etc.), the SFM/SUM does not flush the data in the output buffer to the disk. The SUM is using a technique called "write behind" which buffers data in the main memory until it "has to" flush the data out to the disk. This technique uses the disk transfer speed efficiently (data is optimally written out in full buffer increments). Unfortunately, this efficiency is a hindrance to the application program writer because it makes it difficult to tell were an error is occurring when debugging.[10] Thus liberal use of *write statements* will not always help debug a program because the data written will stay in the SUM's buffers and not be flushed out to the disk.

As described before, the user of an SJE application program uses the AP in a slightly different manner than the APEX application program. Under APEX, all I/O (such as input files) is handled by the HOST; but active data must be moved between the two machines as the program runs. Under SJE, the user must move his input files from the HOST to the AP before he starts and his output files back when it finishes. What is more, the SJE user must be aware of the fact that the AP has different data formats from the HOST. Thus before moving data from the HOST, the user may have to convert his data files to a form that is suitable for the AP. It follows logically that the inverse must be performed at program end. The data conversion routines are contained within a library associated with the AP on the HOST computer. The associated conversion routines reside in the utility library.

[10] Though to be fair, the debugger built into the SJE helps a great deal to this end. By liberal use of watch points, break points and stack traces, the program writer has a fairly good chance of catching his bugs.

3.4. The Driver

The Driver is the piece of operating-system-resident code that supports the AP. It translates an I/O request into an I/O request to the AP or an AP start/stop request. The driver routines are called from APEX.[11] The driver in turn communicates with the HISP via a shared memory called the *message ram* (MRAM). The driver is lowest level code that runs on the HOST. The primary responsibilities of the driver are:

1.) Support the notion of an UNIX I/O device.[Rit78] This means that the driver must contain the normal entry points: open, close, read, write and ioctl along with the special entry points such as the probe routine for automatic device recognition at UNIX boot time.

2.) Start up an I/O transfer to or from the AP. This includes preforming all VAX specific operations like page lock down/free up, obtaining and releasing Unibus resources, and address translation into the Unibus address space.

3.) Handle any interrupts from the AP. The AP can interrupt the HOST asynchronously, so the driver must retain all interrupt state.

3.4.1. The Message Ram

As described before, the driver communicates with the HISP via a special shared memory called the MRAM. This is 32 byte section of memory contained on the VAX interface board that may be read or written by both the VAX and the HISP. It is addressed by the VAX as "memory" locations in the Unibus memory space. Figure 12. is a picture of the MRAM. The last four bytes of the MRAM are not actually memory, they are the Control and Status bytes for the interface. The VAX

[11] The Unibus Diagnostic, VUIT84, called the driver though a subroutine called: DGPIO - Diagnostic Programmed I/O.

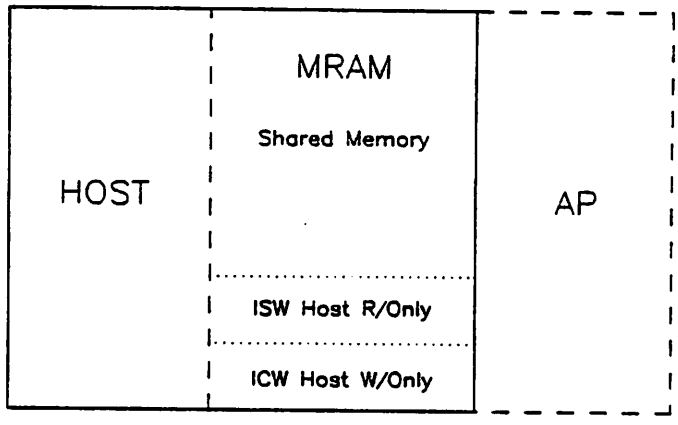

Figure 12., VAX/HISP MRAM.

writes the ICW to inform the HISP of a change in status, the HISP will write into the ISW to send status information back to the VAX.

3.4.2. VAX/HISP MRAM Protocol

Whenever you have two processes that share a resource a protocol must be obeyed by both processes. For the MRAM, the default case is that the HISP controls it. Independent of a particular operating system, if the VAX wishes to read or write this resource, the VAX must take control from the HISP by writing a 1 into the SETLOK bit in the ICW. The VAX then must wait for the LOKACK bit to be asserted in the ISW. When this second bit is asserted (by the HISP) the VAX UNIX driver "knows" that it has complete control of the MRAM. The HISP will not attempt to modify the MRAM until the VAX relinquishes control by asserting the CLRLOK bit in the ICW.

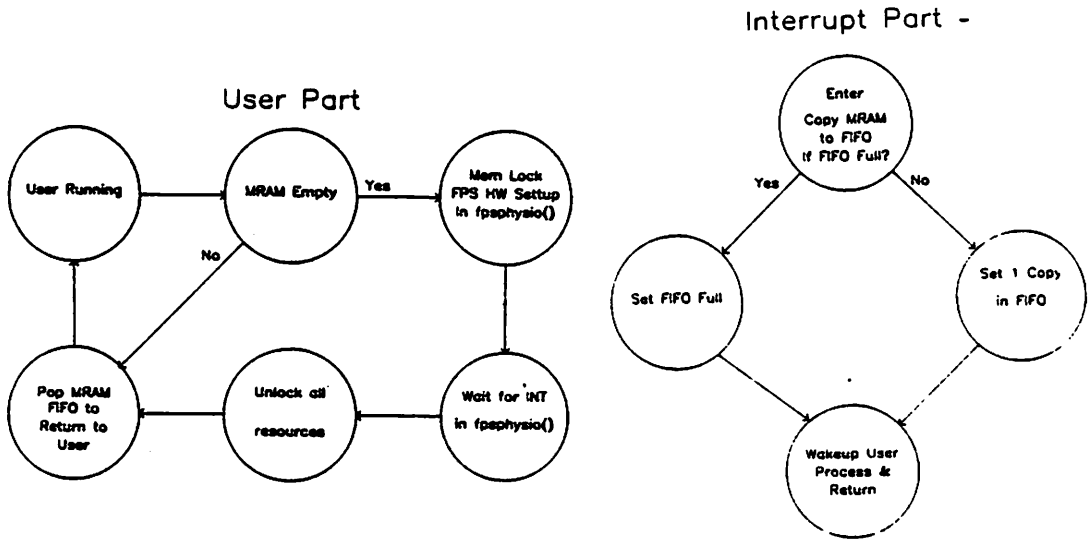

Figure 13., VAX Interrupt State Diagram.

Whenever the VAX wishes to gain the attention of the HISP, it uses the HISP attention bit in the ICW. Equally, whenever the HISP wishes to get the VAX's attention, it will depcsit a one in the Interrupt Indication (INTIND) bit in the ISW. If the VAX has interrupts enabled (turning on the INTENB bit in the ICW) and the processor is at a low enough priority, the VAX will take an interrupt from the FPS. The interrupt causes the driver to do the following:

1.) lock the MRAM.

2.) make a local copy of the MRAM.

3.) release the MRAM.

4.) return to normal processing.

When the MRAM is released, the driver must clear the INTIND flag by posting a CLRINT

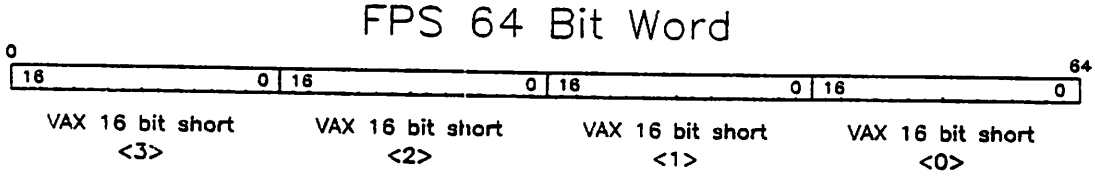

Figure 14., FPS-164 word to 4 VAX shorts mapping.

in the ICW. Figure 13. is a state diagram of the interrupt process.

3.4.3. The HCSR

Within the MRAM are 8 bytes of status information contains the state of the HISP and the AP. These bytes are known as the HCSR and make up one FPS word. FPS documents the bits within the HCSR using the FPS bit numbering which is opposite of the VAX.[12] In the VAX HOST Manual,[FPS82c] the HCSR is referred to an HCSR(Upper) and HCSR(lower) for the two 32 bit upper and lower portions in VAX longs. The issue is further clouded by looking at the HCSR as four VAX 16 bit unsigned short integers (shorts). This is more natural because the HCSR[0..3][13]

[12] One of the more confusion parts of the driver is the mix between FPS 64 bit words, VAX 16 bit shorts (FPS sometimes calls these words), VAX 32 bit longs (FPS sometimes calls these words also). Danny Cohen describes this type of problem in more detail for a variety of computing systems.[Coh80]

[13] Notice C array reference naming conventions of a[0] for the first position within the array, not FORTRAN convention of a[1] for the same place within the array.

are the four shorts as the driver will see them though the MRAM. Figure 14 is the mapping between the 4 shorts and the 1 FPS long for the HCSR. HCSR[2] contains the interrupt masks. Whenever an interrupt is taken by the VAX from the FPS, after the driver copy's the MRAM, it then looks at the type of interrupt the VAX received and the posts UNIX I/O status information based on the type of interrupt received. In all cases the MRAM data is copied and saved for use by APEX.[14]

3.4.4. Interrupt Status Information

Of particular interest is the problem associated with the AP and asynchronous interrupts to the HOST. Most peripherals conform to the *disk drive model* of the world, that is, *the peripheral will never interrupt the HOST without the HOST having previously made a request to the disk.* When a disk drive makes an I/O request, the disk controller will interrupt it with one of a number of interrupts (I/O done, I/O error, bad block error, etc.) In all cases, the interrupt is only generated after the HOST has requested some service from the peripheral. The interrupt is a message from the peripheral, informing the HOST of the result of the service request. In each case, the HOST has some outstanding I/O request waiting for the transfer complete.[15] UNIX synchronizes its processes by causing the process that makes the I/O request, to block (halt executing) in the driver until "disk" finishes servicing the request. With a peripheral that produces asynchronous interrupts such as the AP, the HOST can be interrupted asynchronously of a I/O request. Which is to say, *there is not a user buffer available in the HOST operating system for the status information from the peripheral (the*

[14] This is sort of a lie. Current UNIX does not support power fail. APEX has no direct support for such, so this interrupt is tossed on the floor. If you get this interrupt, UNIX will have tossed cookies itself so this is not a problem. Further, there is a special type of "on-line" interrupt that should be ignored. See the driver code for the details.[Bro82]

[15] Some operating systems, such as DEC's VMS, do not cause the user process requesting the I/O to block while waiting. But in all cases, the process has some sort buffer allocated for the status information from the peripheral. Thus when the I/O is complete, the peripheral driver places that status information in the buffer and starts up the driver.

AP) and a process waiting for that information.

Consider the following problem: The user of the AP, sends to the AP an image to be processed. The AP's device driver will start up a standard I/O request to the AP to transfer the image it. When the image has been moved in the AP's memory (saved on the disk as it were) the AP will interrupt the HOST with a HISP_DONE interrupt to inform it that the HISP has completed a I/O transfer. The HOST can then go back and continue doing other work (possibly starting up another transfer to the AP). The AP will then start executing the image that was just transferred. At some later time, the AP will send the HOST a TASK_DONE interrupt to inform the HOST that the AP is finished and ready for more work. The problem with this scenario is that there is not a pending I/O request from the user to receive that interrupt. To make sure that APEX and the SUM stay in synchronization, the driver must buffer up the information in the MRAM from the asynchronous interrupt and save it for later interpretation by APEX.

Due to the APEX/HISP protocol, it is possible for two interrupts to be outstanding from the HISP. The driver therefore implements a 2 position FIFO for the MRAM in the HOST. When the first interrupt is taken, the driver places the status in the first position. A state flag is set in the driver (FPS_MESG_RAM_FIRST_INT). If another interrupt is taken, the driver, copies the MRAM into the second position and asserts the MESSAGE_RAM_FULL flag. If more interrupts come in before the the driver's FIFO is emptied, the status is lost after an error message is printed on console. If this message ever prints, something confused in the interface and you could (probably do) have hardware problems on either the VAX or the AP.

Associated with APEX is a subroutine to read status information from the AP. This is implemented in the driver as an IOCTL called: WAIT_FOR_VAX_INTERRUPT. When this IOCTL is called, if there is status information in the FIFO, it is returned immediately and the FIFO is popped one position. If the FIFO is empty, the pro-

cess is put to sleep until information comes back from the AP. Earlier their was a reference to the interrupt handler posting UNIX I/O status information. This status information is waking up of a process that may be waiting (blocked in the driver) for that status information.

Of further interest is a convention maintained by APEX. When APEX calls the driver, if the driver contains interrupt status information in its FIFO, the driver does not execute the I/O currently requested. It returns normally, *as though the I/O had taken place.* APEX will then call the driver to obtain the status information from the last I/O, but instead of returning that information, it will receive the MRAM information in the driver's FIFO. APEX will recognize that this was not the information from the last I/O request (because of course the last I/O never really occurred) and eventually resend the missed I/O over again.

3.4.5. Interrupt Lock Out

There is an interresting subtlety here. If the user requests an I/O and does a context switch into the driver, the driver will obviously check to see if there is status information in the driver. If there is none, the driver will continue setting up the I/O. What happens if the AP has the HISP post an interrupt during the time that the driver is trying to set up the next I/O. When the driver is setting up an I/O, it is in a critical section,[Sha74] which means that interrupts are locked out. With interrupts locked out, the VAX will not see the interrupt until after it has started the I/O. According to protocol, the I/O should never get started. The data in the HISP needs to be read. The driver has interrupts locked out, so the VAX can not take the data from the HISP.

The protocol is modified to work as follows, when the driver is setting up an I/O, after is usurps control of the Message RAM, it looks at the INTIND bit in the ISW. If this bit is set, it means that that the HISP is trying to interrupt the VAX, but can not because interrupts are locked out. If this is the case, the driver will release

the MRAM *without starting up the I/O.* It then lowers the interrupt priority level, and takes the interrupt. Once the interrupt has been processed, the FIFO will have data in it. The driver then returns to the user in a normal way, but *without starting up the I/O.* Again, APEX will read out the MRAM from that last interrupt.

3.4.6. Command Parcels

Also contained within the MRAM is the location of the FPS equivalent to a device *address register* and a *word/byte count register.*

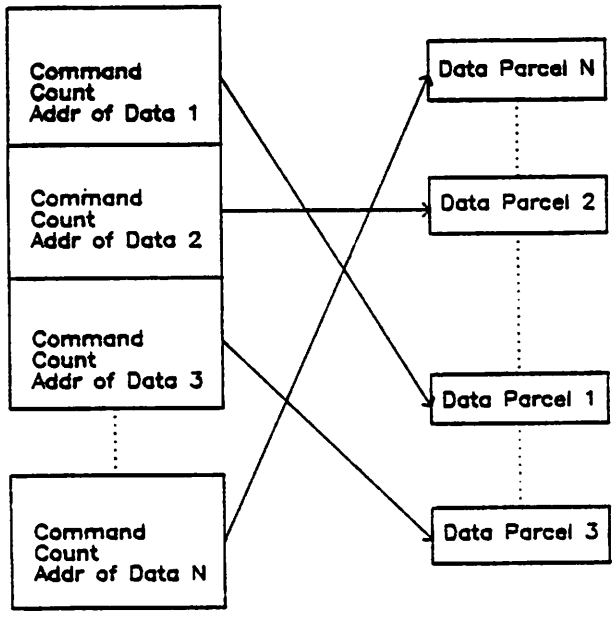

Figure 15., FPS-164 Command Parcel Chain.

With a normal DEC I/O device, the I/O hardware contains a register that is loaded by the driver with the address of the memory location to start an I/O into or from. Associated with that register is a limit or count register that contains the number of bytes from the base that the I/O will include. The FPS-164 does not contain these two registers directly. Instead the MRAM contains two registers that work in a similar manner but instead of pointing to the actual data for the

transfer, the base address points to the base address of a "chain of command parcels," and the "count" is the number of parcels in that chain. Each parcel is a 12 bytes (3 VAX longs) of I/O information (see Figure 15). These parcels comprise of a command word, a count word and an address pointer. A parcel is three VAX "longs" in size. This unusual I/O scheme is used because FPS observed that under most operating systems, such as DEC's VMS, the time it takes to perform a single I/O call was high.[Kas80] UNIX is no exception to the high system call overhead problem.[Joy80] A great deal of the time is spend doing a context switch between a user program and the operating system to perform the actual system call. As a result of the hardware, each APEX call to the driver is potentially able to contain a number I/O operations. Thus the HOST task only pays one context switch for a number of I/O operations. Unfortunately this "non-standard" hardware interface raises a number of ugly problems for the driver.

3.4.7. Physical I/O Subsystem

Most operating systems have a singular view of all peripheral hardware. In this way the device drivers can share much of the operating system code, minimizing the amount of "device specific" code necessary. In UNIX all I/O to major peripherals is done though the *physical I/O* system.[Rit78, Lio77] This routine is responsible for allowing physical I/O into a user's own address space (thus avoiding an internal data copy).[16] The physical I/O system will lock down any page that may be needed for the operation and call the device specific "strategy routine." Once the strategy routine completes, the physical I/O system returns to the user the status of the I/O operation. The strategy routine is responsible for obtaining any physical resources for that device, such as Unibus Map registers,[DEC79b] deciding which I/O to perform next (seek scheduling for disk is

[16] Physical I/O to the user's data buffer is known as *raw I/O*. When the kernel supplies the I/O buffers (for read-ahead, write behind etc), it will call physical I/O on it own.

done here), and calling a device specific "start" routine. The start routine will actually load the base address of the I/O (after suitable address translation) into the interface's address register, loads the count register and starts up the I/O and then return to the strategy routine. The strategy routine will then put the user to sleep and wait for the I/O to complete. When the completion interrupt is received, the device specific interrupt handler will send a wakeup to the process sleeping in the strategy routine. The process, now awake, looks at the completion codes from the interrupt, frees up the resources that it needs and returns to physical I/O.

Unfortunately, with the FPS-164 life is not as simple. The above separation only works if all routines obey the same basic hardware strategy, and thus physical I/O can perform page lock-down and address translation in the start routine. The major problems with the FPS-164 hardware encountered are the two just mentioned: page lock down and address translation.

3.4.8. Page Lock Down

When I/O is started up, the VAX lacks the ability to let an I/O device fault on a page reference. Thus before any I/O takes place, the page that will receive the I/O must be marked valid and locked into main memory. VMS and UNIX handle this problem with two different methods.[DEC81] Under VMS, every page table entry (PTE) has an associated reference count. Whenever an I/O requests comes in that requires a page be locked down, the count is incremented. When that I/O is completed, the lock count is decremented. Once the reference count goes to 0, the page is marked to be reclaimed. UNIX does not support asynchronous I/O or shared memory, thus it never has needed of a reference count on PTE's. As a result, a page may only be locked down once. If the kernel ever detects a request to lock down a page a second time the driver panics.[17] The FPS

[17] Panic is a UNIX term used when the kernel voluntarily shut itself down, after deciding

hardware's parcel chain can (and often will) have two different pointers to the same page in memory.

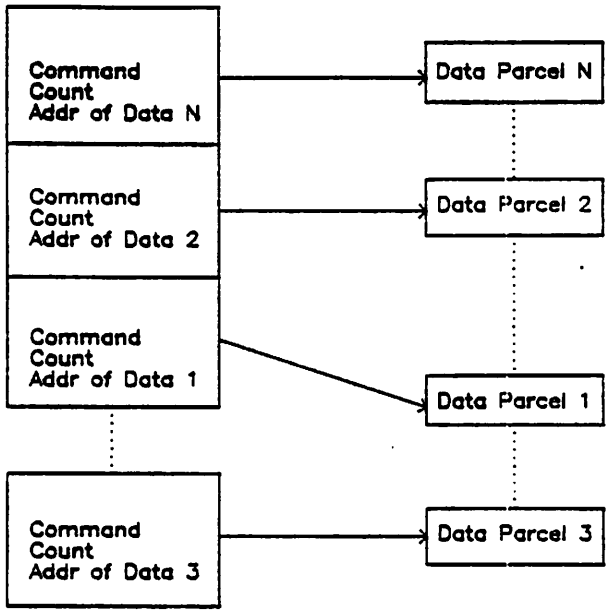

Figure 16., Parcels After Sorting.

Under VMS, the driver needs to just take the address from each parcel and lock down each page. Under UNIX this method will result in attempting to lock the same page down twice. To protect from this problem, the driver walks the parcel chain and makes a copy of the chain in a local copy in kernel memory. It then sorts this copy with a kernel version of "quicker sort."[CSR81] At that point the driver knows that it can make a linear scan of all pages needed. References that are to the same (or adjacent) pages will appear next to each other.

Figure 16 represents the three possible cases that can occur after the sort. In the first case; complete overlap; the driver just ignores the second entry. In

---

that something serious (that should never happen), has in fact occurred (hell just froze over). The kernel will shutdown before more damage is done. In IBM terms this is called a "major supervisor error" or a "1411."

the second case, it will have either adjacent pages or an overlap of the first entry into the second. In this case, the driver extends the limit of the first entry to include the second entry and then ignores the second entry. In the third case, the driver has two distinct sets of page entrys. The driver can therefore lock down the first set of pages because the sorting of the parcel list guarantees that all entries following will not be the in (or below) the range of all entries previously seen.

3.4.9. Address Translation

On the VAX there are multiple address spaces available: user virtual, kernel virtual, physical and unibus physical.[DEC79b] When a process runs for a user, the addresses that the user generates are "user virtual" addresses. Each address is passed through a virtual to physical hardware translation to produce that actual page in physical main memory. If that page is not available (not actually contained within the physical memory), the user is said to "page fault." The VAX receives a "page fault" interrupt and the operating system's paging software must fetch that page from secondary storage and add an entry to the user's active pages (a PTE is created). The program can then be restarted and the translation hardware can proceed without faulting.

Unfortunately in the case of I/O, the translation is not automatic. The I/O hardware works with physical address. Thus when either the user or the kernel wishes to perform an I/O, the address placed in the I/O registers must have been translated in software to be the physical address. As described earlier, that physical page must be "locked" into memory because the I/O hardware can not fault.[18] To cloud the issue further, the VAX includes a way to retrofit I/O inter-

[18] Page locking is used in the process of deciding if a page is "allowed" to be migrated to secondary storage when more main memory needs to created for the current process. A locked page is a page that can not be removed.

faces from the PDP-11 to this newer hardware.

3.4.10. The Unibus Adapter

The PDP-11's I/O architecture was built around a 18 bit bus called the *Unibus*. The VAX contains a separate "Unibus Adapter"[DEC79b] that provides the glue between the 32 bit address of the VAX and the 18 bit addresses of the PDP-11.

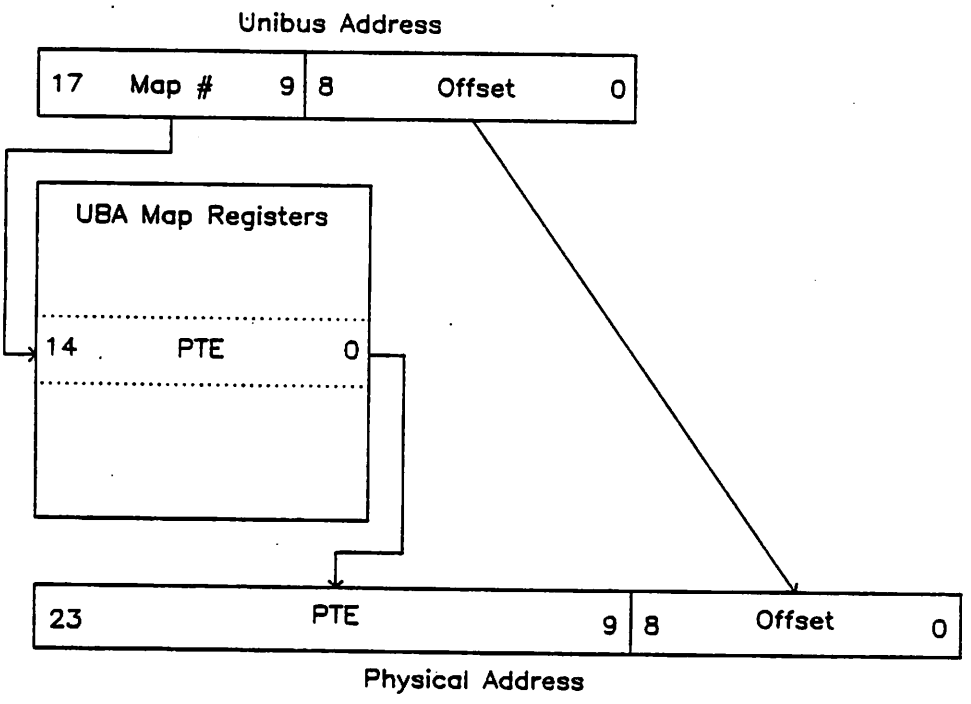

Figure 17., Unibus to Physical Address Mapping.

The adapter contains a set of registers that map between the 18 bit addresses generated by the Unibus devices and physical addresses in the VAX main memory. These mapping registers are shared between all users and all devices wishing to perform I/O between the two memory spaces. UNIX, by convention, uses the strategy routine to obtain enough map registers for the I/O. Each routine calls the ubasetup() routine to perform that operation. When the kernel calls ubasetup(), it sends the virtual address of where the I/O is to be performed,

a count for the amount of data to be moved, some uba specific information,[19] and which virtual address map to base the I/O from. The setup routine will then translate the virtual address into page table entries, grab enough registers for the transfer and load the uba's map registers with the correct PTE's. By convention, the ubasetup() returns a 32 bit magic number that is then broken down, into which uba, how many registers, and the starting register. The driver can then use this result to load the device specific register with a 18 bit unibus address that will point to the map registers that the driver just received from the allocation routine. Figure 17 is a pictorial view of the translation process.

3.4.11. The FPS-164 Address Translation

The FPS-164 uses the Unibus as its interconnect with the VAX. Therefore the driver must set up the map registers for the I/O. Unfortunately the operation described above is not quite what must happen for the FPS-164. As described previously, the user passes the address of the command parcel chain to the driver. Obviously, the interface hardware needs the address of the parcel list in order for it to perform it's I/O. Unfortunately, the addresses contained within the parcel list, are also user virtual address and thus they must also be translated. Because the Unibus can only operate on 18 bit addresses, the parcel list must only contain 18 bit addresses. To alleviate this problem, the driver makes yet another copy of the parcel list in kernel memory. This version will not be sorted, but rather it is modified to contain 18 bit addresses. Thus when the hardware is given the base address of the command parcel list, the address that it is pointed at will always be the address of this kernel copy, not the actual address user's version.

During the process of locking down memory described earlier, the kernel

[19] Uba specific information might include, which uba to use, which data path in the uba to use, etc.

takes the virtual address that it just locked down and calls ubasetup() with that address. It then modifies all parcels in the kernel copy of the parcel list with value returned from the ubasetup() routine.

3.4.12. FPS Physical I/O

The two problems with physical I/O are taken care of in a driver routine specifically written for the FPS-164 called: fpsphysio(). It is this routine, not the normal UNIX physical I/O routine (physio()), that must guarantee that pages are locked and address translated. Once the I/O actually takes place, fpsphysio() is only half way to completing its job. After the I/O has completed (the HISP returns a HISP_DONE interrupt), the fps strategy routine will return to fpsphysio(). Fpsphysio() must then walk through the sorted copy of the parcel list and release all unibus registers that it used and then it must unlock all memory pages that it had previously locked.

3.4.13. The Timer

As described previously, UNIX does not allow asynchronous I/O. This design feature causes another problem here. When an I/O transfer is in progress the process is put to sleep, and continues be blocked until an interrupt is received from the FPS-164 that causes UNIX to finish the I/O and restart the process. If the FPS-164 hangs, (which happened frequently in the development stages of the project), the process would stay blocked forever. UNIX has a mechanism for sending an interrupt to a process from the keyboard (^C or <DEL>). Unfortunately this mechanism will not allow the user to unhang the process doing I/O with the FPS-164. This is an artifact of the implementation of ^C and the interaction with the kernel "sleep" mechanism.

When a process is put to sleep in the kernel, the kernel will pick a priority at which the process is put to sleep. This priority determines how ^C is handled.

If the priority is "low enough" and an user interrupt (^C) is entered while the process is asleep, the process will "immediately" abort the I/O in progress and return to the user. If the priority is "high enough," the interrupt is posted but the process is not restarted. When the process is restarted by whatever the normal awaking procedure for this driver, the process will then recognize the ^C interrupt and abort immediately after that point. The problem described is two-fold, if the process sleeps at a low priority, then the the user will abort the I/O *without finishing*. If he aborts, then the final cleanup phases of physio() will not be executed. If the clean up phase in physio() is not executed, then resources such as the Unibus map registers will not be reclaimed and locked down pages free up. If the process is sleeping at a high priority then the ^C interrupt is not recognized. As a compromise, there is a timer associated with each I/O transfer. It is currently set for 20 hours.[20] If the hardware does not respond, when the timer triggers it will awaken the process from its sleep state. The process will then tidy up its environment and return to the user with a "timeout" error.

The user still can send a ^C, but can also run a program from another terminal called "nukefps." This special program will use an alternate entry point in the driver, and reset the timer to 1 second. Nukefps will make sure that the timer goes off before it returns. If the timer goes off, APEX will receive a fatal error and stop all processing.

3.4.14. Init for New User

The last piece of the driver that must be examined is a special I/O control routine (an IOCTL) for APEX. When APEX starts up for a new user of the AP, it will

[20] The APEX will occasionally send a *TR-WAIT*[FPS32b] channel program when it wishes to wait for the FPS-164 to preform some computation. Some computations, such a SPICE2 run, may take over 20 hours to compute.

enter the driver and request that the driver restart the AP. The driver will reset the Interface and wait for an interrupt, it will set an interrupt to the HISP and make sure the HISP is operating. If both of those operations fail, the driver will preform a hard restart of the AP. APEX will reload the SUM (if need be) and the SUM/APEX protocol will begin.[Bro82, Mil82]

3.5. The Utility Library

When writing production FORTRAN programs, a common problem is writing programs to run efficiently while still allowing the programs to be portable between a number of different manufacturers. A popular solution is to use a preprocessor, such as RATFOR, that will transform the user's source code into a host specific FORTRAN program. This solves the portability problem, but leaves the efficiency problem. RATFOR solves the efficiency problem by looking at a common efficiency problem: I/O statements.

Standard FORTRAN I/O, while being portable, is awkward for most common programming tasks such as character manipulation. As a result, most manufactures "enhance" their compilers with special, efficient, but non-portable I/O primitives. The RATFOR solution is to define a set of *primative routines* to perform the I/O for the user program. A program writer can use these calls rather than the standard FORTRAN calls and still maintain portability. The efficiency issue is therefore handled by a system programmer who understands the subtleties of the particular machine that RATFOR has been "ported" too.

For the FPS PDS these primatives are called the *utility library*.[FPS82d] By definition they are written with efficiency as the most important design goal. The UNIX version is written in three languages: FORTRAN, C, and UNIX/VAX assembler. In the case of the data conversion routines, most of these were originally written in VMS dependent FORTRAN. In many cases these routines were sanitized and converted to standard FORTRAN-77. Some routines that are difficult to write in

FORTRAN, such as the time and date conversion routines, were rewritten into C.[Ker78] C was also used for all of the I/O primitives. These routines are difficult to write, and C provided a efficient, clean, and portable way of implementing them. The FPS PDS uses 64 bit integers, but since C does not support such a data type an arbitrary precision arithmetic package, developed at BTL, was used. Although these routines are functionally correct, they are not efficient and thus extremely slow. Because efficiency is the most important issue for this library, the 64 bit package was eventually written in assembler.

The UNIX manual pages for this library can be found in the Appendix. One routine from it must be discussed, the CLI. The CLI is the command line interpreter. When a user runs an application program, APAL for example, he will type some command line for the operating system. The HOST operating system will pass that line to the application program for interpretation. In the C/UNIX environment, these are the "argc/argv" parameters passed to every C program. The CLI accepts these user typed parameters and in turn must interpret them for the application program which called it. Unfortunately, in the UNIX environment there is a problem. When FPS defined the CLI they had never encountered a computing system that was as flexible as UNIX. Under other systems, such as DEC's VMS or Prime's PRIMOS, there is a specific order to parameters typed on a command line. Under UNIX we have free form, "mixed case" command line input. The CLI can not decode all switches if any switches have optional sub-parameters, as is the case with some of the switches for the librarian. The manual page for each program should be referred to for an explanation of the switches. Furthermore, FPS assumed that all terminal I/O, such as the CLI's switches would be "case folded."

To handle case folding, a set of compromises is used. All programs in the FPS PDS use upper case for their switch definitions. VMS folds all file names to

upper case, so the VMS version of the CLI can blindly case fold its input. The UNIX version can not fold the input because due a small flaw in the semantics of the CLI definition. In general, the CLI can not distinguish between a switch as input and a filename as input. Under UNIX filenames are mixed case. Case folding therefore can not be done. This is unfortunate for the user, because it more natural to type lower case than upper case. In order to allow lower case for each program in the PDS, each program would have had to be modified in the supposedly HOST independent portions. This annoyance was weighed against the fact that a UNIX user rarely runs a program such as the compiler "directly." Instead, that user usually creates a *Makefile* and has the make(1) program[Fel78a] call the compiler (or assembler or linker or ...) for the user. The make(1) program can handle upper case as easily has lower case.

3.6. Some Results

As a demonstration that the conversion was successful, the SPICE2[21] program was compiled and run. No optimization was used by the compiler (i.e. APFTN64 was run with OPT=0). No modifications were done to the program other than the usual supplying of the system dependent routines such as the time call and the routine to move characters from one memory location to another.[22] The results of these runs are included in the Table 1. All numbers are in cpu seconds.

Circuit	VAX	FPS-164	Speed Up Factor	Corrected by 1.5
Benchmark	372.90	75.13	4.96	3.31
UA741 Op Amp	95.33	31.38	3.04	2.03
Proprietary Digital Circuit	4589.18	1271.97	3.61	2.41

Table 1., FPS-164/VAX Speed Differential for SPICE2.

The three circuits are the standard SPICE Benchmark suite, distributed with SPICE, the UA741 Operational Amplifier, and a industrial digital circuit from a VLSI computer chip. The encouraging result is that without *any* optimization the FPS-164 ran 3.0 to 4.5 times faster than the VAX UNIX FORTRAN version. One industrial firm has seen that with just using optimization from the APFTN64 compiler, the speed up is in the order of 3 to 5 times for SPICE 2G5 between the APFTN64 and VMS FORTRAN. The VMS FORTRAN compiler is reported to produce code that runs approximately 1.2 to 1.5 faster than the code produced by the UNIX FORTRAN compiler.

[21] A prelease version 2G6 SPICE was used for the bench mark. At the time of writing, 2G6 is in beta test, and will be available for public distribution shortly.

[22] The move routine was non-trivial on this machine due to FORTRAN-77's lack of dealing with characters in a rational manner. The FPS-164 being a word addressed machine, stores 8 characters in word. A character pointer is thus the tuple: Word Address, Index within Word. Thus an arbitrary move routine can not use a simple, indirection through an address (as in this case with a byte addressed machine). Instead a routine was written that called a routine to pack and unpack characters from a FPS-164 word.

Using 1.5 for the correction, we get a range between 2.0 and 3.5 times for the speed up. This result leads to the conclusion that by using compiler optimization and using smarter data structures, such as used in CLASSIE[Vla83] a speed up in the order of 5 to 10 times would not seem unreasonable.

4. The Conversion Process

At the time of this writing, Floating Point System does not support UNIX as an operating system for the FPS-164. The CAD group at UC Berkeley uses both UNIX and VMS. As a result, the need arose to "port" the PDS to the UNIX environment. Figure 18. shows the time line for the conversion process.

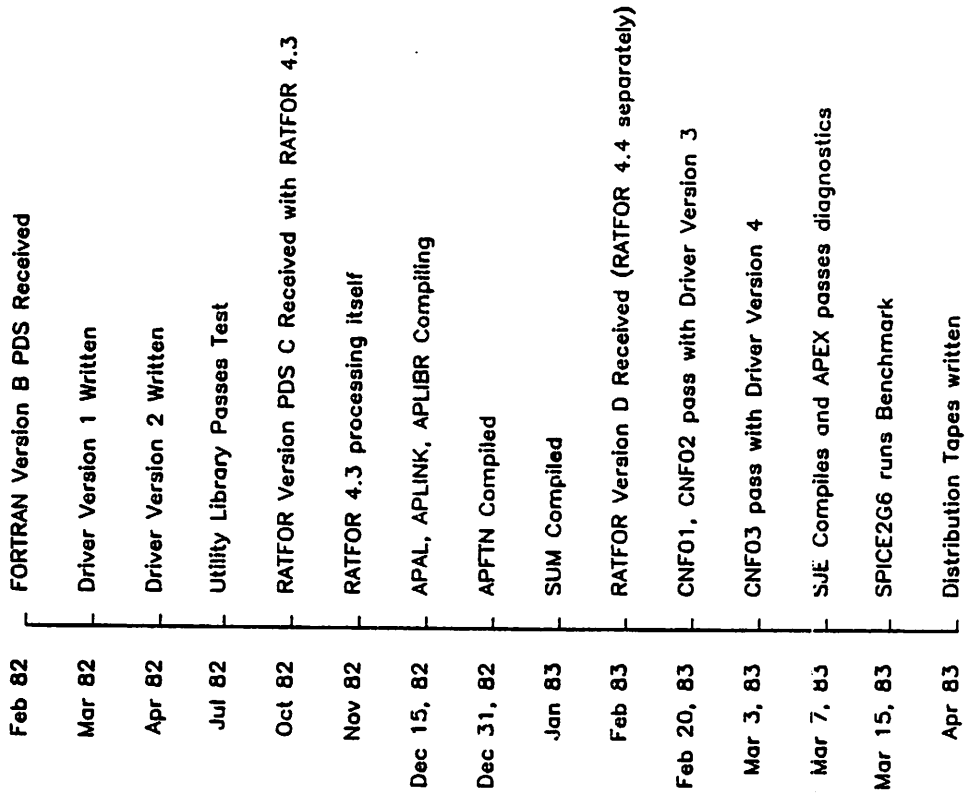

Figure 18., Unix Conversion Time Line.

Most of the PDS, as distributed from FPS, is written is a host-dependent FORTRAN. It is written to make calls to the Utility library and each program may make calls to a a few host local modules. These modules are written in C for portabil-

[23] An example of the type is routines in C are the trap handlers for SJE or the routine that returns the size of a file in a some strange FPS size format. C was picked because it is Portable and interfaces well with operating system routines without the readability problems of using FORTRAN for the same type of routine.

ity and ease of understanding.[23]

4.1. History of Conversion

In February of 1982, UC Berkeley received a copy of the VAX/ VMS PDS and Driver - Release B. At this time, the conversion process began. At the same time, the Whippany Electrical Power Systems laboratory at Bell Telephone Laboratories began the same process. The two groups joined forces to speed the process up. The work was originally split between the two groups in this manner:

- UC Berkeley would provide all Unix dependent (OS portions)
- BTL would provide all Unix F77 dependent portions.

This split required UC Berkeley to deal with the Driver and all of the I/O portions of the code. BTL worked on the conversion of the PDS for the Unix FORTRAN-77 compiler, F77. This included many of the routines in the Utility library written in FORTRAN-77.[Fel78b] As time went on, most of the I/O portions of the library moved to UC Berkeley and BTL spent most of its time working on the data conversion routines.[24] In the end all of the PDS conversion was moved to UC Berkeley.

In March of 1982, I wrote the first version of driver while at Whippany. At the end of the week, the FPS VAX Unibus Interface Diagnostic (VUIT64) ran through every test except the last (interrupt test). About a month later, the members of both team when to the FPS factory to work with the FPS Engineering staff to make the interrupt test work. By the end of the second trip, the FPS-164 was capable of running VUIT64. The driver was left alone as the next phase of the conversion pro-

[24] An example of data conversion routines are the routines that convert between an FPS floating point formats and a VAX Unix printable ASCII string. These routines were originally written in VAX / VMS FORTRAN, and some of these were relying on VMS local FORTRAN compiler options for correct interpretation. They are numerically difficult routines. The conversion to the Unix F77 was no small effort. All VMSism's have been removed and now all FORTRAN code should work on any FORTRAN 77 compiler without strange options.

cess went into effect.

During this time, more of the Utility library was written (rewritten) as both teams tried to bring the PDS up under Unix. This effort resulted in limited success. After discussions with the FPS engineering staff, all groups determined that the Release B of the PDS was too old to be a useful starting point. In late August, FPS sent Version C of the code to UC Berkeley. It was during this time that both teams learned from FPS engineering that the PDS was actually written in RATFOR.[Ker76] The *distributed* PDS was the *host dependent, post processed* output from the RATFOR preprocessor.

Discussions were initiated that produced an agreement between FPS and UC Berkeley such that UC Berkeley was released the RATFOR source under non-disclosure agreement. On September 22, 1982, the major conversion work began again. By late December 1982 most of the Release C was processed into Unix F77 format. At this time, the debugging process revealed many bugs in the Release C version of the PDS. At this time, FPS was shipping to the field the Release D version. The team at UC Berkeley was reluctant to take yet another distribution, feeling that it would be more of a set back than a help. After a month of negotiation, UC Berkeley agreed and the new tapes were mailed. On March 6, 1983 SPICE2 ran for the first time as an application program under SJE, UNIX and the FPS-164.

4.2. RATFOR Headers

The PDS, as distributed by FPS, is the output of a RATFOR preprocessing step. These FORTRAN programs are compiled by the host FORTRAN compiler to produce executable programs for the host. For future maintenance of the FPS PDS software, it is essential to be aware of how the PDS is put together in the RATFOR source form. The installer of the PDS on a host need not worry about the RATFOR process other than be aware of it's existence and recognize that it is a potential

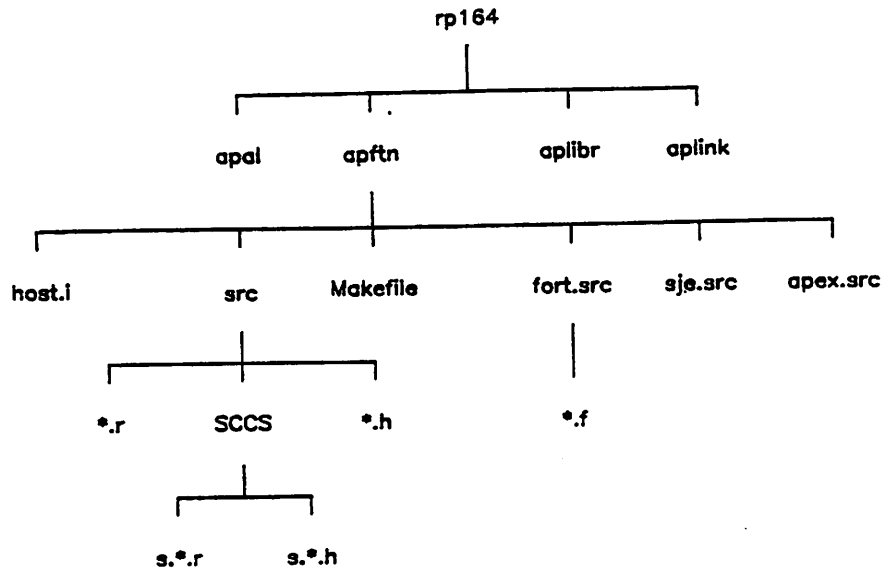

Figure 19., A RATFOR/PDS source directory.

source of error,

A piece of the source tree is shown in Figure 19. The top most level in the tree is the "rp164" directory. Each piece of the PDS is a subdirectory within that directory. Above, "rp164" is the top of the master data base tree, "master.src." For more detail here, refer to the appendix.

For each of the PDS programs in RATFOR form[25] there exists a file in each master subdirectory for that program called "host.i."[Nak82] This file contains the host specific defines, such as conversion constants for the local machine's numerical representation and the FPS-164 representation or the mapping between a function to perform an *Exclusive-OR* of two 32 bit integers and return a 32 integer result. If the local operating system, in this case VAX UNIX, has a

[25] APAL64, APDEBUG64, APEX64, APFTN64, APLIBR64, APLINK64, APSIM64, SJE64 are all members of the PDS that must be processed with RATFOR. Each of these exist in the subdirectory:

primative in the compiler for this function, then the local intrinsic is defined in this file. RATFOR will replace the macro with the intrinsic.

Also found within the "host.i" file is the *absolute pathname* for any of the *include* files used in the source code. If a source program trys to include one of these files, the define within "host.i" will inform RATFOR where to find the file to be included - causing RATFOR to replace the *include line* in the source file with a lines from the *included file*.

The RATFOR source for each program is found an a subdirectory called: "src." Under the "src" directory another directory is found called: "SCCS." All sources are in SCCS[Dol78] format. The sources found in the "src" directory have been checked out of the SCCS database. All modifications of the PDS software since the original FPS Version D release may be followed. Simple comments explaining each modification are included. The prt(sccs) command is used to read these comments.

In the top level directory (.../master.src/rp164/<whatever>) there are at least two other directories. The directory "de01" contains the raw VAX VMS specific files from FPS. Some of these files are needed by the UNIX version and thus have been checked into SCCS format in the "src" directory if they were used. The other files have no bearing on the UNIX version and are ignored. The unused files are left as added documentation on the *functionality* of certain pieces of the FPS code.

The other important directory is the directory "fort.src." This contains the output of the RATFOR preprocessor. The distribution uses *symbolic links*,[CSR82] of the new 4.2 BSD file system, to point from the distribution area into these directories. If the make(1) program[Fel78a] is run in the top directory, all RATFOR sources that have changed will be preprocessed, run through the host fortran compiler, and a test version will be generated in the current directory. Running

make(1) with the argument "clean" will cause the objects to be removed and general directory clean up will be preformed.

The FPS FORTRAN Compiler (APFTN64) contains three other subdirectories. The compiler must be able to support both the SJE product and the APEX product. As a result, the 3 files that are different between the two versions of the compiler are contained within the subdirectorys: "sje.src" and "apex.src." Finally, the subdirectory "unix.files" contains a set of UNIX specific files that are included when RATFOR is run on the input sources.

The Debugger and SJE both need a small number of host specific subroutines that are specific to these programs, such as the interrupt trap handler or the routine that returns the size of a host file in FPS disk blocks. In each case these routines are written in C and stored in the "src" and the SCCS directories.

5. Suggestions for future work

The FPS software can be used in its current form, but this is just the starting point for more work. Programs can now be run on the AP, but at this time, none are running optimally on this hardware configuration. SPICE2 is currently running with the compiler optimization set to the lowest level. A possible future project would be to exploit more of the parallelism of the machine with the use of compiler optimization. A research project, similar to the the research begun by Andrei Vladimirescu[Vla83] can now begin using this new processor. Other programs that are memory intensive, such as WOMBAT[Spi83] (assuming a C Compiler) can begin to explore the memory pipeline of the FPS-164.

The FPS-164 would make a useful addition to a local-area-network if it could be directly attached to a local Ethernet. However, in its current form, the AP must be connected via the VAX. By attaching the FPS-164 to an Multibus Unix processor such as the SUN 68000 Computer System[Bec80] instead of the VAX, the SUN board could handle all of the APEX calls, while allowing many users on the Ethernet to have access to the AP, instead of limiting its use to users that have access to the VAX. Such a environment is similar to that proposed by many researchers doing CAD/CAM. In order to move to this type of environment, the SJE and SUM will need to be reworked somewhat. SJE must become a process manager, and the SUM will need extensions to handle permanent files.[26]

The ability to move data between the host and the AP more invisibly might be explored. An example, would be to allow the AP's disk to become part of the local operating system's disk. Thus a user could:

```
cd <fps_disk_directory>      ! move the current directory to the AP
ls                           ! obtain a catalog of files in that directory
```

[26] The file naming conventions will need a little work also. They currently work fine for the SFM, but when multiple users need to attach to the machine, directories and subdirectories will need to be implemented.

Further the ability to automatically run a program on the AP with out having to enter SJE would seem a logical extension. The UNIX exec(2) system call, which is used to create a new process, could quite easily be extended to perform this operation. The current facility to execute a command procedure (execute a shell fill in UNIX parlance), is exactly the same operation as loading a co-processor. At the present time, when UNIX notices that you are trying to "execute" a text file, not an object file, UNIX loads and executes the UNIX command interpreter (the shell). The shell in turn is passed the user's text file as a parameter for it to interpret. If follows logically that when UNIX notices that the user is trying to execute an object linked with the FPS binder, not the UNIX binder, UNIX simply loads and executes a UNIX program (something similar to an extended version of SJE) that will load the AP with an object file. This program in turn is offered as a parameter the name of the FPS-164 object file to load and start running on the AP.

Much new CAD/CAM development has moved from FORTRAN, to more modern programming languages. Programs such as WOMBAT,[Spi83] KIC,[Bilar] and SPLICE2,[Kle83] are written in the C programming language. Until a C compiler is available for the FPS-164, these programs will not be able to use the machine, which is unfortunate.

6. Acknowledgements

Many people helped me in many ways as this work was in progress. I can not name them all. There are a few that should be named for there moral and intellectual support throughout: Dr. Richard Newton and Dr. Donald O. Pederson of UC Berkeley EECS Department, for advise on how to make a research project proceed; Tektronix, and Bell Telephone Laboratories for financial support, Floating Point Systems for providing the FPS-164; the Engineering Systems Group of the Digital Equipment Corporation for providing the VAX 11/780 computer to which the AP was attached; Dr. Richard LeFaivre and Dr. Ian Getreu of Tektronix, Mike Carey of UC Berkeley/Carnegie-Mellon University and Dr. Eugene Bartel of Carnegie-Mellon University for helping me pursue a graduate degree and supporting me throughout the entire endeavor; Rick Spickelmier of UC Berkeley, for being the best "grammer" program I've used; Tom Quarles, Peter Moore, Ken Keller and Jim Kleckner of the UC Berkeley for help in solving different programming problems; Ken Brown and Steve Nakamoto of Floating Point Systems for answering my many "non-normal" questions about how the FPS-164 was supposed to work; Robert Rodriguez, Richard Drake and Edward Whelan of BTL Whippany Electrical Power System's group for help on the BTL end of this project; the entire UC Berkeley CAD group for moral support; Paul Hansen of UC Berkeley, whom with I co-wrote a paper on array processing that made me put together my thoughts on how the system was interrelated; Mark Hofmann, Scott Baden and Giles Billingsley, men of the world, for constantly keeping me laughing, and most of all to my father for teaching me the three most important tools I used in the project: to never give up, never lose your sense of humor, and never let "your schoolin get the way of your education."

I got by with a lot of help from my friends.

attached A small computing element with large main storage, on same order as a midicomputer or maxicomputer, with a computing element similar to a maxicomputer or supercomputer in floating point speed, i.e. 1-15 MFLOPS. This machine is *attached* to another computer as a peripheral. The attached processor (AP) is number cruncher, and as such does little if any I/O. The FPS-164 and MAP-200 are examples of attached array processors.

integral A small processing element integrating into the main CPU as a hardware option. The address range of these processors will be the same as the CPU they are connected. Such a processor is added to speed the normal processing rate of the main cpu. An example of these might be the floating point accelerator for the PDP-11 or VAX or the array processors that are available as options for certain IBM machines.

7.2. Software Terms

In order to understand the software system described later, some of the major software terms are listed here.

AP Attached Array Processor - In this case the Floating Pointing System's FPS-164.

APEX AP Executive - The host resident routines that interact with the AP and the SUM to actually transfer data and control information between the host programs and the AP resident programs.

Binder A program that takes multiple object files, and possiblely library files, and "binds" them together into one runnable object image. It finds all occurrence of an external reference within a module and "binds" it to its actual location else where in the complete object image.

7. Appendix

7.1. Computer Types

mini A minicomputer is a relatively small computer, such as an Hewlett-Packard HP1000 or a Digital Equipment Corporation PDP 11. Typically, such a computer is characterized by having 16 to 18 bits of address range, 8 or 16 bit word size, and an instruction rate of less than .5 million instructions per second (MIPS).

midi A midicomputer is a medium scale computer, examples being the DEC VAX 11 series, or Data General ECLIPSE. Machines of this range typically have at least 20 to 24 bits of directly addressable main memory, a minimum 16 bit word size (and possibly 24 or 32 bits) and an instruction rate of at between .5 and 1 MIPS. Furthermore, these machine might be capable of executing floating point operations at a rate of 500,000 per second, or 0.5 MFLOPS.

maxi A maxicomputer or full-sized computer is a large scale machine such as the IBM 370 or Burroughs B6700. These machines have 24 bits or greater of addressable main memory, a 24 to 60 bit word size, and at least 2 MIPS of processing power. Also included is floating point hardware capable of at least a 1 MFLOP rate.

super A supercomputer is a processor like the CRAY-1 or CDC Cyber 205. Such machines typically have at least 16 megabytes of main memory, (although the CRAY-1 is somewhat constrained in main memory capability) at least 10 MIPS, with 100 MIPS not being unusual. Further, the floating point processor is typically capable of from 10 to 100 MFLOPS, with some short bursts exceeding 250 MFLOPS. [Sie82]

Driver The host kernel resident routine that communicates with the HOST/AP interface hardware.

HASI Host AP Software Interface - A set of FORTRAN stubs that call APEX routines to allow communication between the HOST and the AP.

HISP Host Interface Support Processor - A bit slice computing engine that resides in the AP used to move data and status information from the AP back to the Host.

MRAM The message ram is a shared memory between the AP and the VAX. It is used as the communications channel between the two processors for processor specific information.

PTE Page Table Entry - an entry in the system tables for virtual to physical address translation.

PDS Program Development Software - the collection of utilities used to create an object program run on a computer. This is the term that includes, the assembler (APAL64), the linker (APLINK64) and librarian (APLIBR64), the FORTRAN compiler (APFTN64), as well as the debugger (APDEBUG64) and simulator (APSIM64).

SFM Scratch File Manager - The set of SUM calls that maintain the file system on the AP's disk.

SJE Single Job Executive - a utility program, similar in function to the UNIX *shell*[Bou78] that is used to create files, execute programs on the AP and to the move files between the host computer and the AP.

SUM Single User Monitor - the AP resident "operating system."

Translator A program used to "translate" a textual format file into a binary format.. An example is a FORTRAN compiler that translates from an ASCII or EBCIDIC representation of a FORTRAN program (or subprogram) and pro-

duces a binary object of the machine instructions for those routines.

7.3. Computation Issues for Array Processors

The array processor, like the super computer, use two major hardware techniques to obtain speed: Parallelism and Pipelining. A brief explanation follows describing the basics of these techniques and how they are used by the FPS-164. To exploit the parallelism, data must be moved from one functional unit to another. This problem, interconnection, is inherent to an array processor. and as such is also examined.

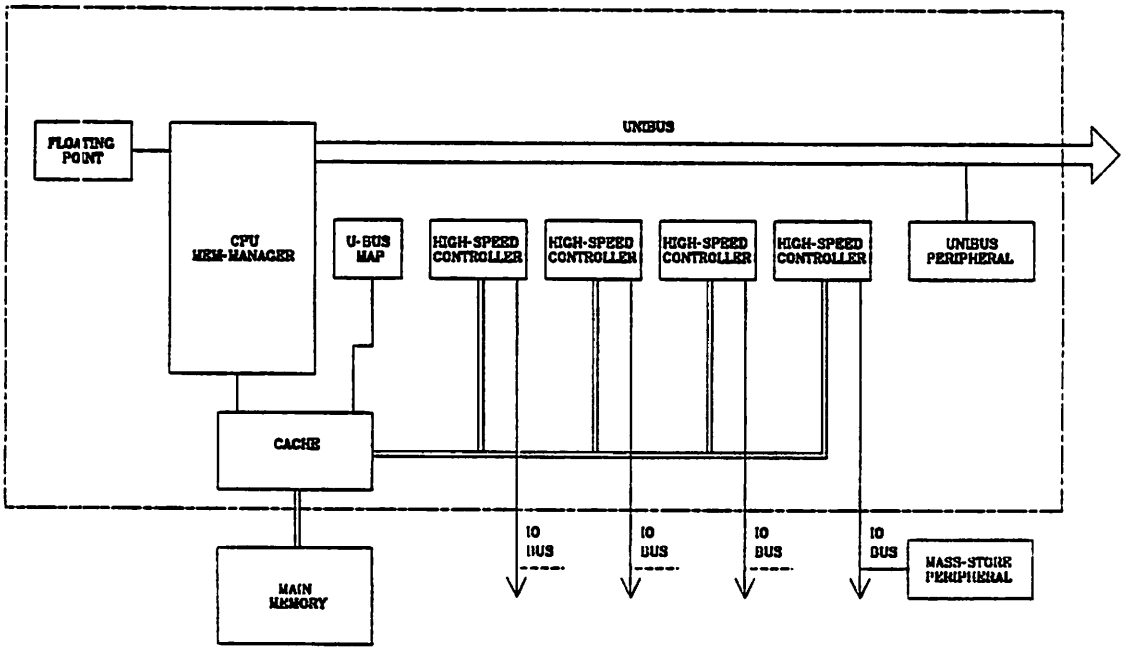

Figure 20. DEC PDP 11/70 block diagram with parallel I/O processors.

7.3.1. Parallelism and Pipelining

Both parallel (overlap) and pipeline processing rely on the simple notion of *division of labor*: a total job is partitioned into individual subjobs, to be parceled out to different working units. [Sto80] The term *parallelism* describes the simultaneous execution of completely independent tasks. Synchro-parallelism

denotes the condition of identical units working in unison. Many early and recent parallel machines (Burrough's Illiac IV, Goodyear's STARAN, SDC's PEPE, Burrough's BSP, CDC's Cyber 205, etc.) consist of a number of arithmetic/logic units operating *in parallel* under the supervision of a single control processor. Parallelism in I/O and instruction preparation/execution are two common example of parallelism. This parallel structure is commonly used on minicomputer and midicomputers such as the DEC PDP 11/70. [DEC76] Figure 20 illustrates the first of these.

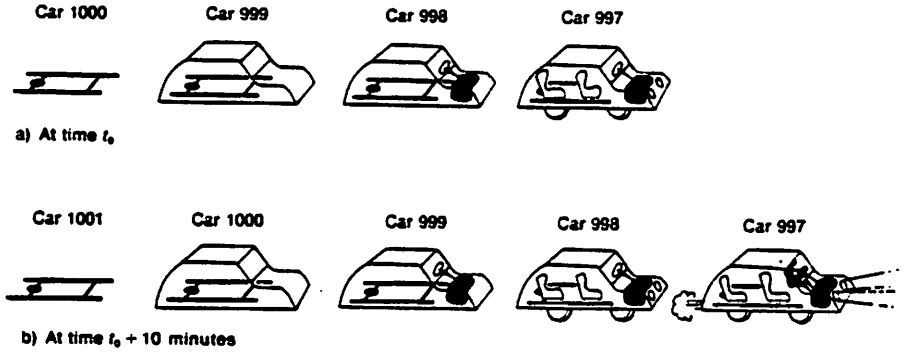

Figure 21. An automobile assembly line.

The term *pipelining* describes the execution of precedence constrained subjobs in accomplishing an overall task. Concurrency is possible since each subjob is being completed simultaneously by independent *segments*, which in turn pass their complete subjobs to the next station(segment) in sequence (the pipe) for further processing. A simple analogy is an automobile assembly line, where each station is specialized to perform one part of the overall task. Figure

21 illustrates this simple example. [Sto80]

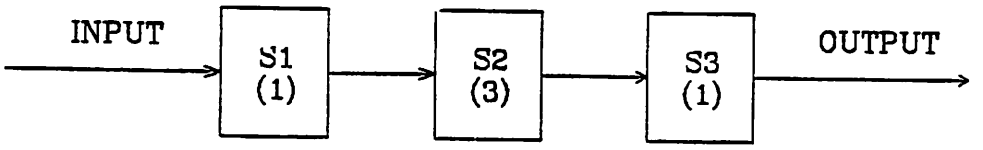

Figure 22. Pipelined processing task with 3 stages.

Table 2. Initiation sequence for a 3-stage pipeline.

	Time States Through the Pipe																	
Stages	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16	t17	t18
S1	A	B	C	D	E	F	.	.	.									
S2		A	A	A	B	B	B	C	C	C	D	D	D	E	E	E	F	F
S3					A			B			C			D			E	.

Examples of pipelined architectures include the TI ASC, SDC's PEPE, IBM 360/91, CRAY-1, CDC Star-100, etc. [Ram77] Such machines are sometimes called *vector processors* since the chain of operands streaming into/out of such pipelined arithmetic units are largely organized as vector data structures. Vectoring is preformed to amortize the cost associated with *filling the pipe* over a large number of computations. For example, floating point multiplication may consist of concurrently performing exponent addition and mantissa multiplication. Such operation requires at least 2 function-units capable of simultaneous operations, with the advantage of a possible factor of 2 speedup. As a series of operands is provided, a series of results is produced, one every clock-tick.

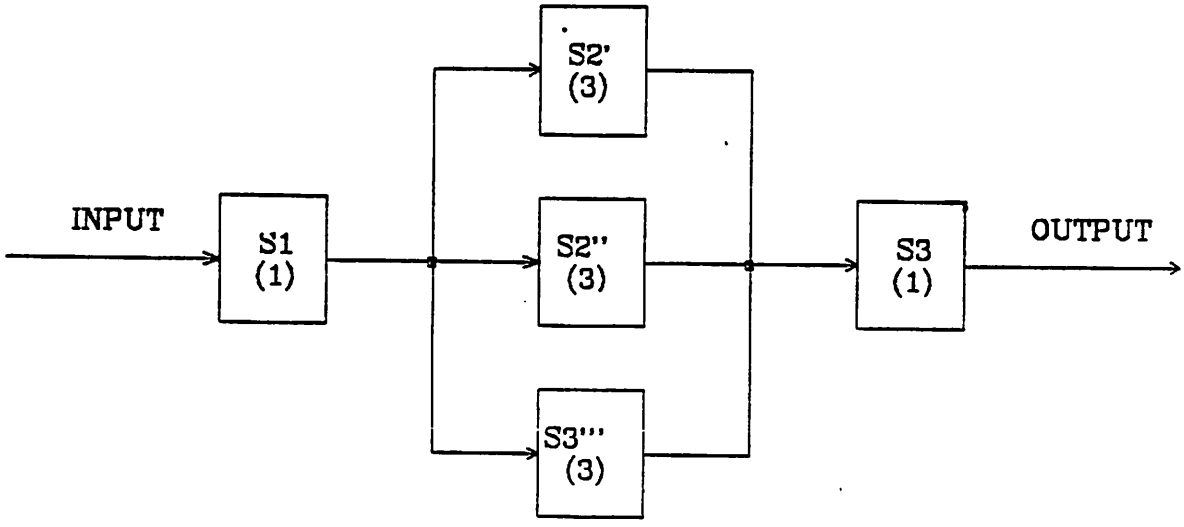

Figure 23. Pipelined processing task with 3 parallel stage 2 segments.

Table 3. Initiation sequence for the pipeline shown in Figure 23.

	Time States Through the Pipe																	
Stages	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16	t17	t18
S1	A	B	C	D	E	F	.	.	.									
S2'		A	A	A	D	D	D	G	G	G	J	J	J	M	M	M	N	N
S2''			B	B	B	E	E	E	H	H	H	K	K	K	N	N	N	Q
S2'''				C	C	C	F	F	F	I	I	I	L	L	L	O	O	O
S3					A	B	C	D	E	F	G	H	I	J	K	L	M	N

An array processor is designed to employ the advantages of both types of concurrency: parallelism and pipelining. For example, consider a processing task which takes 5 time units for completion. A serial processor would take 5*N time units to complete N instructions. Consider further that the 5 time-unit

task can be partitioned and pipelined as shown in Figure 22, where the *stage time* is indicated in parentheses. The initiation sequence of such a processor is shown in Table 2.

Such a system can be shown to require {5+3*(N-1)} time units to process N uninterrupted instructions. For large values of N, this represents a speedup factor of approximately 1.7 over serial processing (speedup Sp = T1/Tp).

Now, consider *paralleling* 3 stages which are identical to stage 2 in function. (You will note that stage 2 is the *bottleneck* of the computation pipeline.) The resulting pipelined processor would appear as shown in Figure 23. The initiation sequence for our modified pipeline processor is shown in Table 3.

Such a system can be shown to require {5+(N-1)} time units to process N uninterrupted instructions. For large values of N, this represents a speedup factor of approximately 5 over serial processing. This is illustrated graphically in Figure 24.

As methods to speed computation and enhance overall system efficiency, parallel and pipelined techniques provide distinct advantages. Next we consider some of the basic issues of such multiple processor systems.

7.3.2. Interconnection Issues

Enslow's paper on multiprocessor organization[Ens77] and Feng's paper on interconnection networks[Fen77] serve to guide this brief examination of issues relating to interconnection schemes for multiprocessor systems. Enslow characterizes the hardware organization by the nature of the system used in organizing the primary functional units (processor, memory and input/output channels). The 3 basic schemes are categorized as:

1.) Time shared busses.

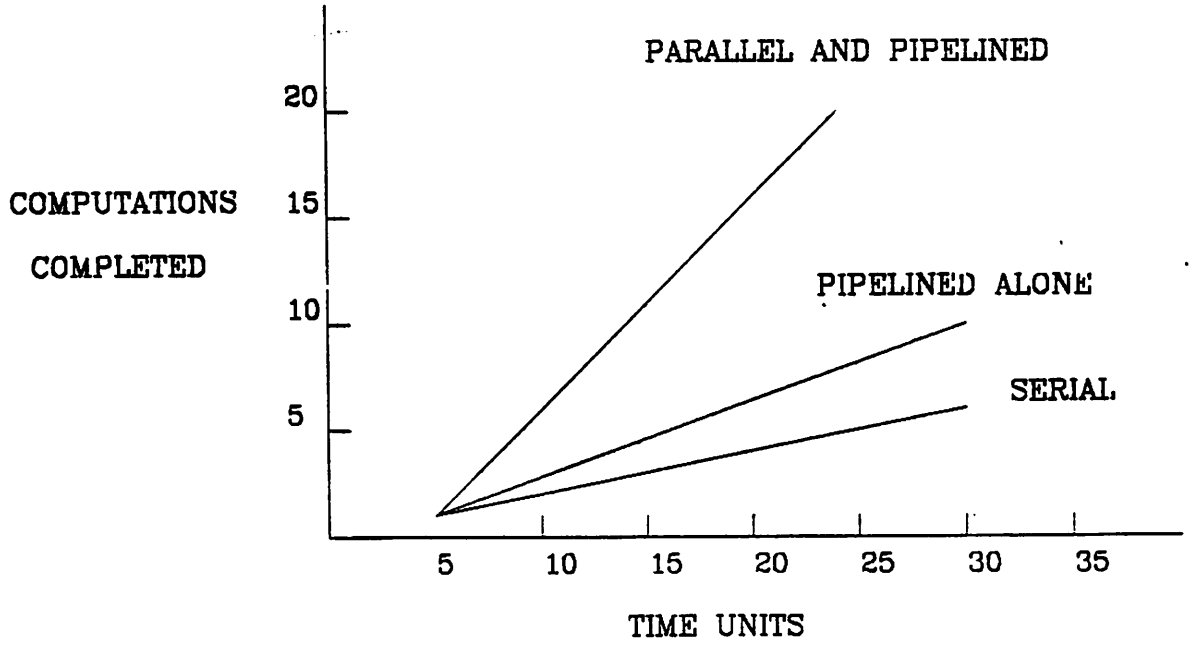

Figure 24. Comparison of serial, pipelined, and parallel-pipelined computation times.

2.) Cross-bar switch matrix.

3.) Multibus, multiport memories.

This serves to partition the higher level aspects of the system. A more recent analysis by Feng indicates that the system designer must consider 4 topics when selecting an interconnection network for his processing element environment. These are:

1.) Operation mode - either synchronous, asynchronous, or combined depending on whether the connection requests are issued statically, dynamically, or a combination of both.

2.) Control strategy - either distributed or centralized, depending on who manages the switching element.

3.) Switching methodology - either circuit switching (a physically established path) or packet switching, which relies on *chunks* of information flowing through a network in no predetermined path. (Note: this topic relates to *computer networks* at a higher level and won't be considered in this paper.)

4.) Network topology - either static or dynamic depending on the ease and timeliness of interconnection reconfiguration capabilities.

The practical aspects of these four topics usually consider the the multiprocessor system at a higher level than the internal structure of an array processor. The question is: given a number of processing elements (computers) what is the best strategy (in terms of operation mode, control, switching method, and topology) to *lash them all together* in a productive system? However, these are also important considerations to apply to the structure of an attached array processor itself and will be dealt with in the following sections. For each of the 3 topics, the general notions and how they apply to array processors is presented. We will then present specific examples of each idea by describing the implementation of the Floating Point Systems FPS-164[Cha81] and Computer Signal Processing Incorporation MAP-200[Coh81b] array processors.

7.3.2.1. Operation Mode

The choice between synchronous and asynchronous operation can be influenced by several factors. Fully synchronous operation will typically lend itself to simple pipelined implementations. Since the sequencing of stages is *metered* and known, control relies on the ability to test and generate hardware pipeline interlocks. Such interlocks are necessary since stage-times may not be equal or may vary with the type of processing being done. [Ram77]

Asynchronous operation will typically lend itself to parallel implementations

containing multiple (identical) physical or virtual execution units. In this case, if no data dependencies exist, it is possible to allow computations to finish *out of sequence* relative to when they were initiated. Thus, such sequence reordering can come as a result of certain hazard conditions or simply because one event takes less computation time than another.

Various problems result from both modes of operation. Obviously, where strict data dependencies exist, or control flow changes occur (branch, jump, etc.), the initiation sequence will possibly degenerate into purely sequential execution, with performance comparable to a serial processor. Likewise, for asynchronous operation of parallel execution units, similar hazards may occur when two active instructions are simultaneously in process, where one execution unit may need information provided by another unit. Ramamoorthy and Li[Ram77] present a fine essay on such control problems associated with asynchronous systems. They consider:

1.) read after write

2.) write after write

3.) write after read.

Special consideration is given to the first of these three problems, and they suggest solutions which apply to the other two.

Briefly, the FPS-164 is a fully synchronous pipelined architecture, while the Computer Signal Processing Inc. MAP-200 uses an asynchronous, nonpipelined architecture to achieve parallelism. (Figures 25 and 26 show the structure of each of these attached array processors.)

7.3.2.2. Control Strategy

Controlling the execution unit(s) in an attached array processor can be realized either centrally or on a distributed basis. Synchronous operation lends

itself to centralized control, and is commonly found in pipelined systems. A current text by Peter Kogge[Kog81] presents a thorough examination of the control problems of synchronous pipeline architectures and includes many useful design techniques and examples which are also mentioned by Ramamoorthy and Li. (We have considered this topic extensively in our previously completed homework assignments. Please refer to them for our conclusions and observations.)

Distributed control is likely to be found in asynchronous systems, where the flow of information may proceed on the basis of *availability of resources*, not necessarily a fixed point in time. An analogy might be a line of cars on a freeway during a moderate traffic jam. As the car ahead of you is able to advance, you will *fill up the gap* he leaves behind. The driver of each car determines when to advance based on the *space* available to him, which is an example of an asynchronous pipeline under distributed control.

The FPS-164, being a fully synchronous pipelined architecture relies on centralized control, while the MAP-200 relies on FIFO's and queues which allows data-driven processing, suggesting a fully distributed control structure. Figure 25 illustrates the MAP-200 architecture.

7.3.2.3. Topology.

A key issue which determines many of the choices in the design of an attached array processor is the topology - how the processing elements are physically linked. The paths between various elements can either be passive and thus dedicated (static), or they may be reconfigurable through active switching elements (dynamic). Combinations of both static and dynamic topologies are also possible. The spectrum of choices for static topologies range from a single common data bus approach (as used in the IBM 360/91 arithmetic unit) to fully interconnected schemes. Even if we consider bidirectional data paths, the fully connected scheme is Order[N*N] in complexity and quickly becomes

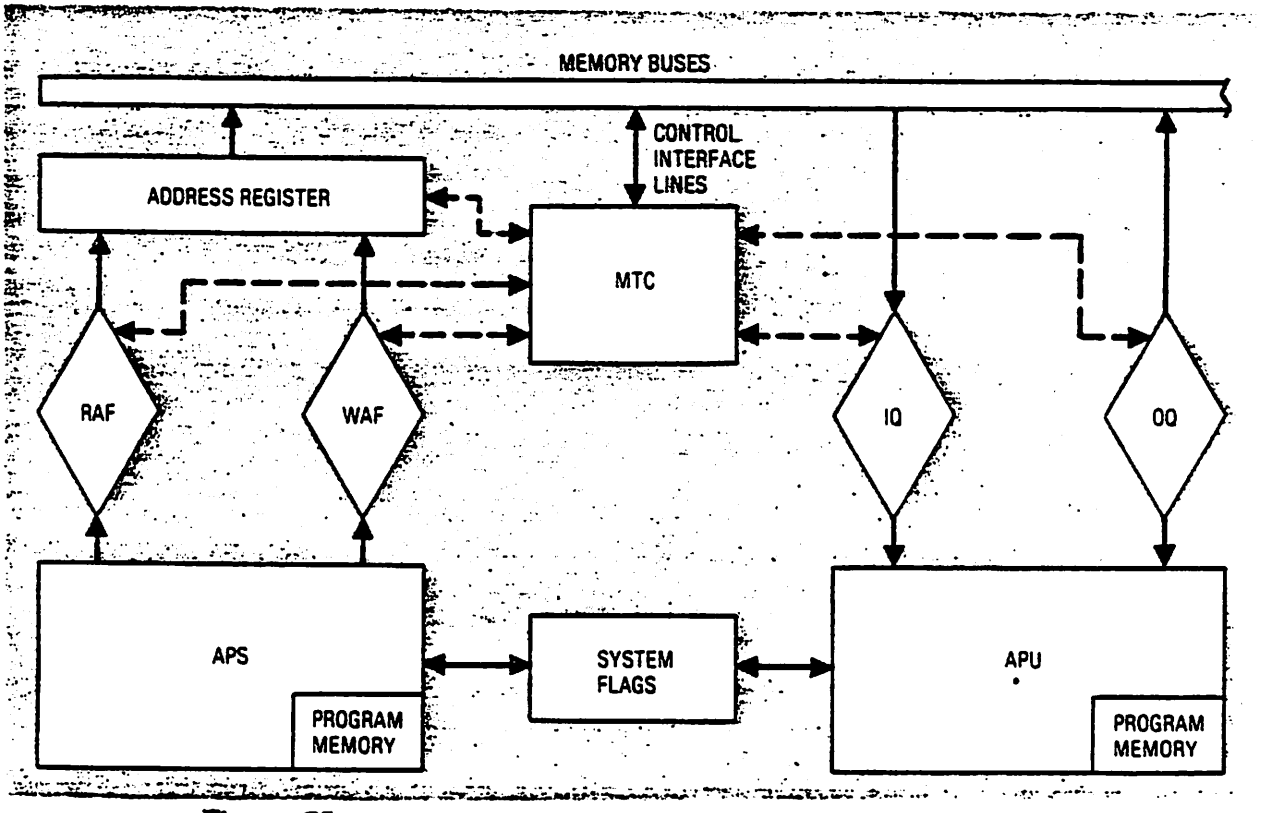

Figure 25. CSPI MAP-200 asynchronous architecture.

prohibitively expensive. A possible solution is *timesharing* in a sense, in which the data path can be dynamically reconfigured to make the proper connection between processing elements. Again, a spectrum of choices between a single stage and multiple stages is possible, with a full-crossbar switch providing total flexibility (any output can be mapped to any input without blocking).

The choice between static and dynamic may well be driven by how much time overhead is associated with setting up the path in the dynamic case, versus the expense of having the fully static connection always available. In terms of utilization of resources, the dynamic solution is more economical, even though it may be considerably slower. Feng provides many examples of the various forms and presents the design decisions involved in choosing a topology.

The FPS-164, as can be seen in Figure 26 employs a cross-bar structure for its topological interconnection scheme. The MAP-200, as seen previously in Figure

25 is a bus-interconnected architecture, with some being shared and others strictly static. This correlates with the overall asynchronous system architecture.

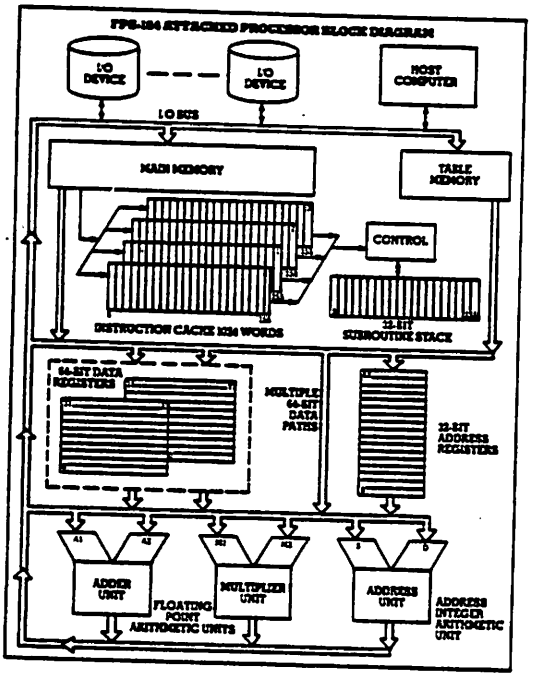

Figure 26. FPS-164 internal structure.

7.4. Installer's Guide

To install the PDS software on the FPS-164, three steps must executed.

1.) Read the magnetic tape

2.) Install the system dependent portions (driver, system directories).

3.) Compile all source code into executable binary code.

4.) Install all executable code.

7.4.1. The Distribution Tape

The FPS UNIX Distribution is contained on one 2400 foot magnetic tape. It has been written with the standard UNIX tar(1) program in ASCII at 1600 bits-per-inch

(1600 BPI) with the normal UNIX blocking factor of 20 blocks-per-record. This will yield a tape made up of 10240 bytes ASCII records. The last record has not been filled to the full 10240 bytes. All standard unix file names and directories are contained within the tape. Tar(1) will build all directories as it reads the tape.

The tape can be read on a 4.1 BSD UNIX system with:

```
% cd <where_ever_you_have_50_megs_of_space>
% tar x
```

When the tape has been read you should have a directory entry in the current directory called: "src" A directory listing of that directory should yield:

Makefile	aplibr/	apslib/	man/	util/
apal/	aplink/	driver.4.1/	sje/	vuit/
apdebug/	apmath/	driver.4.1C/	sum/	
apex/	aprlib/	fpskey/	report/	
apftn/	apsim/	lev3/	tools/	

Certain FPS customers are not licensed for SJE. and will not receive the directories "sje" and "aprlib." The directories: "apal," "apdebug," "apex," "apftn," "aplibr," "aplink," "apsim," "sje," "sum" contain the obvious piece of FPS PDS code. The "apmath" library is the FPS math library. The libraries, "aprlib" and "apslib", are runtime libraries for the compiler. The utility library is contained within the directory, "util." The driver diagnostic is found in "vuit". The confidence tests is found in "lev3". The directory, "fpskey," contains header files for the build process. The "nukefps" and other tools are in "tools." The "man" directory has standard UNIX manual pages for the whole system. A copy of this report is found in the "report" directory. Last of all, are two versions of the driver, the two directorys left. The 4.1C version is the latest, but can not run on a 4.1 system.

7.4.2. Dependent System Installation

To install the FPS PDS system, you will need to follow these directions carefully. First, two bugs in the UNIX F77 compiler must be corrected and installed. The first fix is for APEX's huge data initialization and the second one for the lexical and semantic analyzer in APFTN.

To add these fixes on a normal 4.1 system.

```
cd /usr/src/cmd/f77
```

Near the beginning of the file "data.c," is a format statement in the form:

```
static char datafmt[] = "%s\t%05ld\t%05ld\t%d" ;
```

This must be changed to:

```
static char datafmt[] = "%s\t%09ld\t%09ld\t%d" ;
```

This fixes a bug with large data statement initializers.

In the file "defs," the define of MAXLAB was changed from 125 to 256. This fixes a bug with computed goto's. This will allow a switch/case statement of all of the alphabetics (e.g. alpha = [0..255]).

After making those fixes, type:

```
make
```

Now, make sure no user is running the UNIX FORTRAN compiler by running the UNIX ps(1) program. Once the coast is clear, type:

```
make install
```

You now have a new FORTRAN compiler. The new 4.2 BSD UNIX system has the second change, the first will not be included because that part of the compiler was changed.

At this point you should create a directory for the PDS libraries. The distributed "Makefile" assumes this to be: "/usr/lib/fps" and is writable by the "userid" of the person building the PDS (root most likely). The Makefile is parameterized for this directory to be changed, though it is not suggested.

Next create the directory for the binaries. It it currently named "bin" and in the same directory as the "src" directory. This is also parameterized, so read the Makefiles if you wish to change this directory.

Now copy the driver files ("fps.c" and "fps.h") into the operating system source directories: "/usr/sys/dev" and "/usr/sys/h" The file named: "/usr/src/conf/files," must be updated with the line:

```
dev/fps.c          optional fps device-driver
```

The configuration file for your machine must be updated to include a line for the FPS-164. The line from UCBCAD looks like:

```
device          fps0 at uba? csr 0162000          vector fpsint
```

Now the system can be configured with:

```
% cd /usr/sys/conf
% config <Your_Host_Name>
% cd ../<Your_Host_name>
% make depend
% make
```

At this time, the current directory contains a copy of a runnable UNIX system with the FPS-164 device driver linked into it. The current version of the operating system should be saved and the new version installed in its place with:

```
% mv /vmunix /saveunix
% mv vmunix /vmunix
% sync
```

The system then can be rebooted with:

```
% /etc/shutdown -r +10 "Reboot With FPS-164"
```

The 10 can be changed to how many minutes you would like to wait before kicking the users off the machine. While you are waiting for the shutdown, you should turn on the FPS-164 and make sure its ON-LINE light comes on. If it does not, check to make sure the switches on the diagnostic microprocessor's front panel are correct.

UNIX will reboot normally with the addition of restarting the AP. A word of

caution, the FPS-164 can take a *long* time for the restart. The time it takes is a function of the amount of memory in the AP. UNIX must wait for the AP to restart before it continues to autoconfigure. The UNIX wait time is tunable in the driver header file (fps.h). Unfortunately, A message prints out on the console when UNIX is waiting. This message is normal, and is included so that an operator will not become nervous when the machine is "idle" for 15 seconds or so on reboot.

After rebooting, the unibus diagnostic should be run to verify everything is functioning properly. This program is called: "vuit64." To run it:

```
% cd <where_you_read_the_tape>/src/vuit
% make
% vuit64
> apnum=1
> r w e
> exit
```

The details of this diagnostic are in the standard FPS manuals. Assuming no errors, you should proceed.

7.4.3. The Make Process

The simplest part of the process is the build process. This is done with:

```
% cd <where_ever_you_read_the_tape>
% make |& cat -u > errs&
```

This will compile everything and log the results in the file "errs." If you are curious of the state of the build, you might try:

```
% tail -f errs
```

This will print the last lines of the errs file on the terminal. This process takes about 4-6 hours on the UCBCAD VAX 11/780 with a nominal load.

7.4.4. The Install Process

This is just like a make, except you need to inform the make system that you wish to install the binaries.

```
% cd <where_ever_you_read_the_tape>
% make install |& cat -u >> errs&
```

Notice that you did a >> not a >. You can again, tail(1) the errs file. The install will run in about 10 minutes.

7.4.5. A Quick Test

After everything is installed and properly built, the three confidence test should be run. This is accomplished by:

```
% cd <where_ever_you_read_the_tape>/src/lev3
% make
% cnf01
% cnf02
% cnf03
```

References

Apo81.Apollo, Technical Staff, *Apollo Domain Architecture*, Apollo Computer Corporation, N. Billerica, MA, February 1981.

Bec80.Bechtolsheim, Andreas and Baskett, Forest, *The SUN Workstation*, Computer Science Department, Stanford University, November 12, 1980.

Bilar.Billingsley, Giles and Keller, Ken, "Program Reference for KIC2," *Electronics Research Laboratory Memorandum*, vol. ERL-???, Berkeley, CA, To Appear.

Bou78.Bourne, Steven R., "An Introduction to the UNIX Shell," in *Unix Programmers Manual*, vol. 2A, Bell Labortories, November 12, 1978.

Bro82.Brown, Ken W., *Private Communication*, Floating Point Systems, April 1982.

Cha81.Charlesworth, Alan E., "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, vol. 14, no. 9, pp. 18-27, September 1981.

Coh80.Cohen, Danny, "On Holy Wars and a Plea for Peace," *IEN*, no. 137, USC/ISI, April 1, 1980.

Coh76.Cohen, Ellis, "Program Reference for SPICE2," *Electronics Research Laboratory Memorandum*, vol. ERL-M592, Berkeley, CA, June 14, 1976.

Coh81a.Cohen, Ellis, "Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System," *PhD Thesis, EECS - UCB*, Berkeley, CA, May 22, 1981.

Coh81b.Cohler, Edmund U. and Storer, James E., "Functionally Parallel Architecture for Array Processors," *Computer*, vol. 14, no. 9, pp. 28-36, September 1981.

Col82.Cole, Clement T. and Hansen, Paul M., *On Attached Array Processors*, EECS-UCB, May 1982.

CSR81.CSRG, Technical Staff, *Seventh Edition, Virtual VAX-11 Version*, Computer Systems Research Group, June, 1981. Computer Science Division, Dept of EECS, U. C. Berkeley

CSR82.CSRG, Technical Staff, *4.2 BSD Users Guide*, Computer Systems Research Group, EECS-UCB, June 1982.

DEC76.DEC, *PDP 11/70 Processor Handbook*, Digital Equipment Corporation, Maynard, MA, 1976.

DEC79a.DEC, Technical Staff, *VAX 11/780: Architecture Handbook*, Digital Equipment Corp., Maynard, MA, 1979.

DEC79b.DEC, Technical Staff, *VAX 11/780: Hardware Handbook*, Digital Equipment Corp., Maynard, MA, 1979.

DEC81.DEC, Technical Staff, *VAX/VMS Internals and Data Structures*, AA-K785A-TE, Digital Equipment Corp., Maynard, MA, April 1981.

Dol78.Dolotta, T. A., Haight, R. C., and Mashey, J. R., "The Programmer's Workbench," *The Bell System Technical Journal*, vol. 57, no. 6 Part 2, July-August 1978.

Ens77.Enslow, Philip Jr., "Multiprocessor Organization - A Survey," *ACM Computing Surveys*, vol. 9, no. 1, pp. 103-129, March 1977.

Fel78a.Feldman, Stuart I., "Make - A program for Maintaining Computer Programs," in *Unix Programmers Manual*, vol. 2A, Bell Labortories, August 15, 1978.

Fel78b.Feldman, Stuart I. and Weinberger, Peter J., "A Portable FORTRAN 77 Compiler," in *Unix Programmers Manual*, vol. 2A, Bell Labortories, November 12, 1978.

Fen77.Feng, Tse-Yun and Li, H.F., "An Overview of Parallel Processors and Processing," *ACM Computing Surveys*, vol. 9, no. 1, pp. 1-2, March 1977.

Fly66.Flynn, Michael J., "Very High-speed Computing Systems," *Proceedings of the IEEE*, no. 54, pp. 1901-1909, 1966.

FPS82a.FPS, Technical Staff, *FPS-164: Operating System Manual*, 860-7491-00[0,2,3]A, Floating Point Systems, Inc, February 1982.

FPS82b.FPS, Technical Staff, *FPS-164: DAPEX Designer's Guide*, 860-7491-001A, Floating Point Systems, Inc, January 1982.

FPS82c.FPS, Technical Staff, *FPS-164: VAX Host Manual*, 860-7493-000A, Floating Point Systems, Inc, February 1982.

FPS82d.FPS, Technical Staff, *FPS-164: Utilities Designer's Guide*, 860-7490-000, Floating Point Systems, Inc, January 1982.

Gob81.Goble, George H. and Marsh, Michael H., *A Dual Processor VAX 11/780*, TR-EE 81-31, Purdue University, West Lafayette, Indiana, September 1981.

IBM70.IBM, Technical Staff, *IBM 360 Principles of Operation*, GN22-0354 (Ninth Edition), November 1970.

Joy80.Joy, William N., *Comments on the Performance of UNIX on the VAX*, Computer Systems Research Group, EECS-UCB, March 1980.

Kar81.Karplus, Walter J. and Cohen, Danny, "Architectural and Software Issues in the Design and Application of Peripheral Array Processors," *Computer*, vol. 14, no. 9, pp. 11-17, September 1981.

Kar82.Karplus, Walter J., *Peripheral Array Processors*, Society for Computer Simulation, October 1982.

Kas80.Kashtan, David, *UNIX on the VAX - Some Performance Comparisons*, SRI International, January 1980.

Ker76.Kernighan, Brian W. and Plauger, P. J., *Software Tools*, Addison-Wesley, Reading Massachusetts, 1976.

Ker78.Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

Kle83.Kleckner, James E., "Advanced Mixed-Mode Simulation Techniques," *PhD Thesis, EECS - UCB*, Berkeley, CA, 1983.

Kog81.Kogge, Peter M., *The Architecture of Pipelined Computers*, McGraw-Hill and Hemisphere Publishing Company, New York, NY, 1981.

Lio77.Lions, John, *Commentary on the Unix Operating System*, University of New South Wales and Western Electric, New South Wales and Murry Hill, NJ, 1977.

Mar81.Maron, Neil and Brengle, Thomas A., "Integrating an Array Processor into a Scientific Computing System," *Computer*, vol. 14, no. 9, pp. 41-44, September 1981.

Mil82.Miller, Ed, *Private Communication*, Floating Point Systems, April 1982.

Nag73.Nagel, L. and Pederson, D., "Simulation Program with Integrated Circuit Emphasis (SPICE)," *16th Midwest Symposium on Circuit Theory*, Waterloo, Ontario, April 12, 1973.

Nak82.Nakamoto, Steven, *Private Communication*, Floating Point Systems, April 1982.

Nan81.Nandgraonkar, S. N., Oldham, W. G., and Neureuther, A. R., "Integrated Circuit Process Modeling with SAMPLE," *Proceedings of the 4th Biennial University/Government/Industry Microelectronic Symposium*, pp. 32-40, Presented at Mississippi State University, May 26, 1981.

Old79.Oldham, W. G., Nandgraonkar, S. N., Neureuther, A. R., and O'Toole, M. M., "A General Simulator for VLSI Lithography and Etching Process: Part I - Application to Projection Lithography," *IEEE Transactions on Electronic*

Devices, vol. ED-26, no. 4, pp. 717-722, April 1979.

Old80.Oldham, W. G., Neureuther, A. R., Sung, C., and Nandgraonkar, S. N., "A General Simulator for VLSI Lithography and Etching Process: Part II - Application to Deposition and Etching," *IEEE Transactions on Electronic Devices*, vol. ED-27, no. 8, pp. 1455-1459, August 1980.

Qua83.Quarles, Thomas, "SPICE3: User's Manual," *Master Report, EECS - UCB*, Berkeley, CA, 1983.

Ram77.Ramamoorthy, C.V. and Li, H.F., "Pipeline Architecture," *ACM Computing Surveys*, vol. 9, no. 1, pp. 61-102, March 1977.

Rit78.Ritchie, Dennis M., "The Unix I/O System," in *Unix Programmers Manual*, vol. 2B, Bell Labortories, November 12, 1978.

Sha74.Shaw, Alan C., *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

Sie82.Siewiorek, Daniel P., Bell, C. Gordon, and Newell, Allen, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, NY, 1982.

Spi83.Spickelmier, Rick L., "Verification of Circuit Interconnectivity," *MS Report, EECS - UCB*, Berkeley, CA, May 1983.

Sto80.Stone, Harold S., *Introduction to Computer Architecture*, SRA, Palo Alto, CA, 1980.

The81.Theis, Douglas J., "Array Processor Architecture," *Computer*, vol. 14, no. 9, pp. 8-9, September 1981.

Vla83.Vladimirescu, Andrei, "Simulating VLSI Circuits," *PhD Thesis, EECS - UCB*, Berkeley, CA, Jan 1983.