2

SLAC-PUB-2418
October 1979
(I)

CONF-791037--7

EXPERIENCE USING THE 168/E MICROPROCESSOR FOR OFF-LINE DATA ANALYSIS*

Paul F. Kunz, Richard N. Fall, Michael F. Gravina,
John H. Halperin, Lorne J. Levinson, Gerard J. Oxoby,
and Quang H. Trang

Stanford Linear Accelerator Center,
Stanford University,
Stanford, California, 94305

591 0040

ABSTRACT

The 168/E is a SLAC developed microprocessor which emulates the IBM 360/370 computers with an execution speed of about one half of a IBM 370/168. These processors are used in parallel for the track finding and geometry programs of the LASS spectrometer. The system is controlled by a PDP-11 minicomputer via a three port interface which we call the Bermuda Triangle. The tape handling and downloading is controlled by one of SLAC's IBM computers via a SLAC built interface between the PDP-11 and an IBM channel. Initially, there will be a system of 6 168/E's which should be able to give six times the production capacity than can be attained by running production jobs on the SLAC Triplex system. The cost of the system, including the channel interface, is $120,000 and yet it yields the equivalent computer power of 3 IBM 370/168's. Hence, this system is an extremely cost-effective method for off-line data analysis.

INTRODUCTION

In recent years, we have seen the construction of many large spectrometers at High Energy Physics laboratories. These spectrometers are capable of taking data at such a rate that the amount of computing time required for the data analysis has become a major problem. At SLAC, for example, an experiment on the Large Aperture Solenoid Spectrometer (LASS) has collected 50 million triggers on tape, with about 40 million good events. An average of 0.5 sec of 370/168 CPU time is required for each good event to read the raw data, do the basic event reconstruction, and output the results for each successful event. The software program for these spectrometers generally takes many man-years to develop on a large computer system, and is often changed as it is better understood. It is therefore not easily removed from the large computer on which it was developed.

The goal of this project has been to add to the computer center inexpensive hardware that can execute identically the same program and get the same results as the large computer. This led to the development of the 168/E microprocessor [1,2]. It "emulates" those IBM 370 instructions that are generated by IBM's FORTRAN compiler and its speed is about one half of an IBM 370/168. We are attaching 6 of them to one of the central IBM computers. This hardware is sufficiently powerful that the elapsed time to do all the event reconstruction for an experiment can be shortened by many years.

GENERAL FEATURES OF PROCESSOR

The 168/E consists of an integer CPU, a floating point processor, memory, and an interface. They are all built on boards measuring about 12 by 16 inches, which are identical to those used by DEC in their VAX computers.

Integer CPU

The integer CPU circuit is based on the 2901, which is a LSI bit slice microprocessor chip introduced by Advanced Micro Devices in the summer of 1975. This board handles the following types of IBM 360/370 instructions: 16 bit integer, 32 bit integer, 32 bit logical, all conditional branching, and all memory addressing. It has a 150 nsec cycle time. The throughput on FORTRAN programs has been measured to be between 1.3 to 1.8 times slower than a 370/168. The only noticeably slow instruction when compared to the 370/168 is multiplication. A prototype wire-wrapped processor has been functioning since the summer of 1977. The final board will be an 8 layer printed circuit board and it costs $600 complete with components and labor for assembly.

Floating Point

The floating point processor consists of two circuit boards. It is entirely MSI logic but uses the "new" MSI circuits which have be introduced to support the LSI components. The processor handles all IBM 360/370 single precision floating point instructions with exactly the same results, bit for bit, as the 370/168. But since the single precision format of IBM contains only a 24 bit mantissa, some form of extended precision is required to do the LASS production code. For example, when calculating where two helical trajectories representing the path of charged particles in the LASS magnet intersect, we require a precision better than the resolution of the detectors. This is not possible with the IBM single precision format when the radii of curvature are large. It has been found experimentally that about 8 more bits are required in the mantissa to do the calculation with sufficient precision. On the 360/370, one declares the important variables double precision which adds 32 additional bits to the mantissa. On the 168/E, we have made a compromise between true emulation and circuit cost and complexity. The floating point processor has pseudo-double precision instructions which add 16 bits to the mantissa. Thus the processor can do either 32 bit or 48 bit floating point arithmetic. The cycle time of the processor is either 100 or 150 nsec depending on the instruction. It operates with an internal Read-Only-Memory (ROM) to control the steps in a floating point operation. The performance of the processor is about a factor of two slower than the 370/168. Again, only multiplication is a noticeably slow instruction when compared to the 370/168. The circuit boards are 8 layer printed circuit boards. The cost of the floating point processor is $1200 complete with components and labor.

Memory

The memory for the 168/E is in two parts, one for the program and the other for the data. Both are based on the Intel 2147 memory I.C. The Intel 2147 has become the industrial standard circuit although only Intel has large production experience now. Several other companies have just announced they are also producing it. This circuit contains 4,096 words by 1 bit with a 70 nsec access and cycle time. It has a unique feature in that when the memory is not being addressed, it powers down to 1/5 of its normal operating current. Thus a processor with 8 memory boards draws only as much power as one memory board plus 7/5 of one memory board. Since each memory board draws about 25 watts, considerable power is saved in the system, and considerably less heat is generated which is a major factor in I.C. component failure. The current list price is about $25 each in large quantities. The memory board is a 4 layer printed circuit board with one half containing 32 memory circuits for data and the other half containing 24 memory circuits for program. That is, one memory board contains exactly 16 K bytes of data and 4096 microinstructions which is about 8 K bytes of IBM object code or about 500 lines of FORTRAN. The cost of the board is $1600, i.e. about $50/K byte for data and $75/K byte for program. By comparison, minicomputer add-in memory is commercially available for about $30/K byte. The higher cost of the memory for the 168/E is because the memory must be fast enough to respond within a processor cycle time.

Interface

The 168/E is not capable, as currently designed, of doing any input or output instructions. It can only address its memory. The interface allows a real computer to load the memories with program and raw data and to read the processed results. The interface is very simple. Either the processor or the interface controls the memory and never both simultaneously. The interface control logic can shut off the processor so that it releases the memory busses. Then the interface can take the busses, read or write to the memory, and start the processor.

* Work supported by the Department of Energy under contract number DE-AC03-76SF00515.

Presented at the 1979 Nuclear Science Symposium, San Francisco, California October 17-19.

OBJECT CODE TRANSLATOR AND LINKER

The 168/E microprocessor does not execute IBM 360/370 instructions directly. Instead, a program called the "translator" converts IBM instructions into 168/E microinstructions. The input to the translator is either object code from the IBM FORTRAN compiler or a Link-Edited load module. The outputs are a relocatable microinstruction object code and a data set for the 168/E data memory. Most IBM instructions translate into 1 to 3 microinstructions. We call this program a translator because it does not make any fundamental change in the original code. If a certain register is used in the IBM instruction, then the same register is used in the microinstruction. In memory fetches, the displacement field and the index and base registers of the IBM instruction will be identical in the microinstruction. Even the 4 bit mask in conditional branch instructions is the same in the microinstruction and the original IBM instruction. The data set for the 168/E data memory is a copy of the constants and variables which are part of the original IBM code.

Another program we call the "linker" does two jobs. First, it does the job of the IBM Linkage-Editor by reading the relocatable microinstruction object modules and linking them together. Second, it forms an absolute memory image which can be loaded into the 168/E memories. That is, it gives an absolute address to all the COMMON blocks and also the local memory space. It resolves external references from a library of object modules where, for example, the FORTRAN Library Functions have been translated and stored. Unlike its IBM cousin, the linker has an optional input with which the user can assign the address of the COMMON blocks. This feature is used to make the data memory overlays which are described later.

CAN A MICROPROCESSOR DO THE BIG JOB?

Having built at very low cost a microprocessor that can be programmed in FORTRAN and has a speed which is no worse than twice as slow as a 370/168, is a fine achievement. But due to the design choices that have been made, it is still fair to ask the question: can it do the real number crunching job that we have with the LASS production code?

First of all, to be useful it must do a significant fraction of the time consuming part of the job. With the LASS production code, well over half of the CPU time is spent in the subroutine which finds tracks in the solenoid detectors. Thus the 168/E must be able to execute this subroutine and all the subroutines it calls to be a useful processor. This part of the program is slightly over 32 K bytes of executable code and it translates to a little over 16 K microinstructions which is 5 168/E memory boards filled on the program side. In addition, this part of the program requires about 90 K bytes of space for variables in COMMON and local to the program which is 6 168/E memory boards filled on the data side. The 168/E can thus handle this part of the program from the point of view of the amount of space it requires.

The next question is whether the subset of 360/370 instructions that the 168/E can emulate is sufficient. In this part of the code, we found the two types of FORTRAN statements which lead to IBM instructions that can not be emulated by the processor. These statements are the computed GO TO statement [for example: GO TO (10,20), N] and statements using one byte logical variables. It turns out, however, that their elimination is a good idea anyway. The computed GO TO statement is less efficient in time than a series of IF statements for a small number of possible branch addresses and use of the one byte variables is definitely less efficient in CPU time than setting flags in a 16 bit integer variable.

Thus, the 168/E processor could be used to take the most time consuming part of the production code away from the central computer. However, the event by event input to this part of the code is very large; much larger than the original raw input tape data. This is because the first part of the code unpacks the raw integer data such as wire numbers, widths, etc., into banks of floating point coordinates appropriately scaled, aligned, and corrected. The output from the time consuming routines is also much larger than the final result record, because this part of the code generates large banks of intermediate data which they pass on to subsequent routines. In order to avoid sending large amounts of data from the host computer to the processor and back again, it was also decided to run the unpacking codes on the 168/E. But with this additional code, the amount of 168/E memory required would be very large. The solution to this problem is the same as with all computers when the code is larger than the computer's memory; one must overlay the program into the processor's memory. Once overlays were necessary, it was easy to extend this technique to that code which is executed after the time consuming part, including the formatting of the result tape record. The choice was to do overlays or increase memory size. Since memory is the most expensive component of the 168/E, and overlay time would be only 10% of the execute time, we chose to do overlays. The net result was the decision to execute all of the production program in the 168/E from raw input data tape to final result data tape.

DEFINING THE OVERLAY STRUCTURE

To define an overlay structure for a program takes knowledge of the program's structure and flow. The overlays for the 168/E were defined in the following way:

1. Each overlay should be called only once per event to prevent losing real time in doing the overlay.
2. The size of the overlay is determined by the largest piece of code which satisfies the above restriction after one has tried to break the code up into the smallest pieces. In the case of the LASS production code, the solenoid track finding mentioned above is the largest overlay.
3. The number of overlays is determined by fitting the rest of the code into pieces whose size is determined by the criteria above.

Defining the overlays for the LASS production code was relatively simple, since the code proceeds from unpacking to result formatting serially in several logically separate parts. The overlays for LASS production code are as follows:

1. Unpacking raw coordinates into corrected floating point banks.
2. Counting the number of match points (or space points) in order to kill the event if there are too many, and finding beam tracks.
3. Finding tracks in the downstream spectrometer and following these tracks through the dipole to the region between the dipole and the solenoid.
4. Following these downstream tracks through the solenoid up to the target.
5. Finding tracks in the solenoid starting with points in the plane and cylindrical chambers.
6. Fitting all tracks found to a 5 parameter helix.
7. Following the tracks found in the solenoid downstream to the Cerenkov and Time-of-Flight counters.
8. Doing the vertex reconstruction on all found tracks including the beam track.
9. Formating the result record, and accumulating statistics on chamber efficiencies, etc.

With each overlay, the executable code is translated and saved as a 168/E program overlay. Unlike overlays on most real computers, subroutines which appear in more than one overlay such as SIN, COS, SQRT, etc., are simply duplicated. When an overlay is executed on the 168/E, all of the processor's program memory will be overwritten. The translation also creates a data set which contains all the constant and variable data which was internal to the subroutines. We call this the 'Local Memory' and it may be defined as all the data space a program uses which is not in a COMMON block. The local memory also needs to be loaded into the 168/E data memory when the program memory is loaded with an overlay. For the LASS production code, the local memory is typically 10% of the data memory required by an overlay.

With the overlays described above, the 168/E can handle programs much larger than can fit into its memory at one time. Still larger programs can be handled by further overlaying the remaining data memory which contains the program's COMMON blocks. In order to do this, additional knowledge of the program is needed. One would like to know exactly in which overlays a COMMON is needed, in which overlays data is stored into the COMMON and in which overlays data is fetched from the COMMON. If for example a COMMON block is used only in overlays 3, 4 and 5; then this physical data space can be used for other COMMON's which are only used in overlays 6, 7, and 8.

A method has been developed to study the whole program in this level of detail [3]. When each subroutine is compiled and object code loaded into a load module library, a data set is created which contains a summary of the COMMON block usage. We call this data set an 'Index File' and it contains one line for each variable referenced in each COMMON block. The line contains the name of the subroutine, name of the COMMON block, the variable name, its offset from the start of the COMMON block, its length, and the length of the COMMON. It also contains the Store, Fetch, and other flags generated by the FORTRAN compiler. Collecting these individual data sets into one master index file now completely details the use of every variable in every COMMON block for the entire program. The master file is easily updated, at the time a subroutine is updated into the load module library, since only the entries of the master file pertaining to that subroutine are changed. This master index file, along with a file which states which subroutines are to be used in each of the overlays, can then be used as the data base for programs which analyze the COMMON block structure of the program.

It was quickly realized that COMMON blocks could be put into one of three categories:

1. Constant. These are COMMON blocks in which all the variables never change in the course of processing an event. They may be initialized in the first phase of the production program by BLOCK DATA statements, reading disk files, and/or by calculation in subroutines called once per job.

2. Variable. These are COMMON blocks in which all the variables are generated and used on an event by event basis.

3. Mixed. These are COMMON blocks which contain both constants and variables in the sense defined above.

Since constant COMMONs never change their contents, they can be easily written into the 168/E data memory as required for a particular overlay. In a sense, they are logically similar to the local memory of the subroutines which is rewritten into data memory as required. We have chosen to do this even for constant COMMONs which are used in more than one overlay. Except for the large banks of constant COMMONs used in the unpacking overlay and the large constant COMMONs containing the magnetic field map, the total size of the constant COMMONs is less than the local memory.

The contents of the variable COMMON blocks is created by the 168/E in the course of processing an event. For practical reasons once a block has been created it remains in the 168/E data memory for as many overlays as it is needed, then it may be overwritten by other COMMON blocks, either constant or variable, in succeeding overlays.

Mixed COMMON blocks could be handled in another way, but for simplicity they were eliminated, i.e. the constants and variables were moved into other or new COMMON blocks which were either pure constant or pure variable. For the LASS production code, less than 10% of all COMMONs were 'mixed' when the code was first studied in this manner.

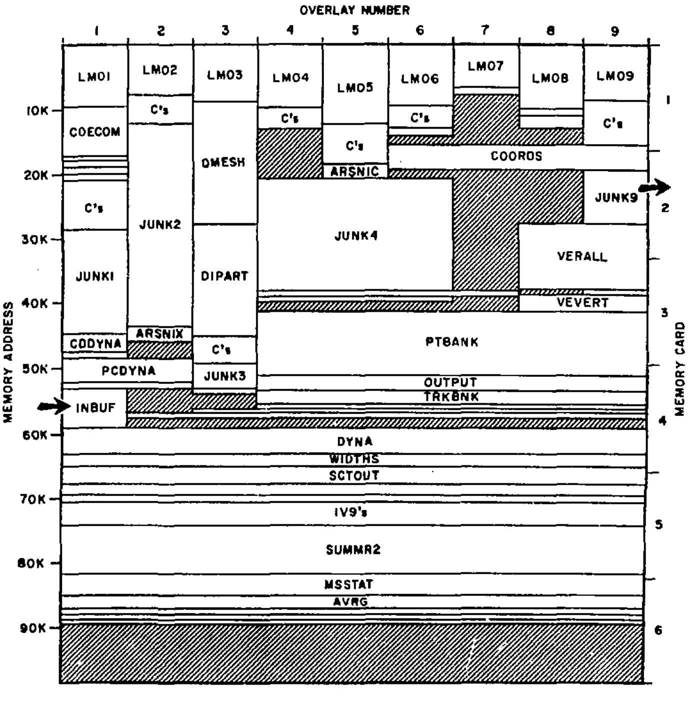

Figure 1: Data Memory Overlay Load Maps

With the master index as a data base, software tools have been developed to generate data memory load maps for all the overlays. An example is given in figure 1. The left hand vertical scale is data memory location expressed in bytes, and the nine columns are the nine overlays. Note that one first loads the local memory (LM01 through LM09) into the low addresses of the processor, then the constant COMMONs. The boxes that are large enough have their COMMON block name written in them, while groups of smaller COMMONs are designated by "C's". The unpacking overlay, number 1, has a large number of constant COMMONs and the magnet field maps have two large COMMONs labeled QMESH and DIPART in overlay number 3. Banks of coordinates generated in overlay 1 are stored in COMMONs DYNA and WIDTHS; they are used by all the following overlays. Other COMMONs such as PTBANK are generated at a later overlay, then saved until the end of processing the event.

The net effect of the data memory overlaying is a substantial saving in memory required by the 168/E processor. Since memory is the most expensive part of the processor, enough money is saved to add more processors to the system. If all the COMMONs were loaded into the memory at one time, it would require over 250 K bytes of data memory; but with the overlaying only 90 K bytes is required. On the program side, if all the code was loaded into the program memory at one time it would require over 120 K microinstruction words, while with the overlays less than 20 K micro instructions are needed. One pays the cost, however: the processor is idle during the transfer of the data and program into its memory. For the LASS production code, we have measured that the total time spent overlaying is 90 msec per event. This is less than 10% of the average event execution time on the processor which is over 1 second per event. Thus, we feel the overlaying technique is a good compromise for our production code and in the following sections we will describe the scheme for implementing the overlays.

BERMUDA TRIANGLE SYSTEM

The Bermuda Triangle system, shown in figure 2, is our method of overlaying the 168/E memory. The Bermuda Triangle is a three way interface with I/O ports to a large buffer memory, a PDP-11 UNIBUS, and a bus to the 168/E processors. Data may be transferred bidirectionally between any two ports. Two Bermuda Triangles are used, one for the program memory and one for the data memory.

The first port of the Bermuda Triangle is to the buffer memories. The program buffer memory, with 128 K words by 24 bits, is large enough to hold a single copy of all the program to be executed. The data buffer memory, with 64 K words by 32 bits (256 K bytes), is large enough to hold all the local memory and copies of the constant COMMON blocks. The data buffer memory also buffers events on input and results on output. The memory used is slower but much less expensive than the 168/E memory. The memories are implemented with general purpose memory cards purchased from Mostek Memory Systems. Their MK8000 memory card offers up to 128 K words of 24 bits. The program memory is thus a single card, while the data memory is two cards depopulated to 64 K words of 16 bits. The cycle time is 500 nsec with an access time of 375 nsec. We have used the backplane and chassis that Mostek provides for PDP-11/70 add-on memory. The signal traces on the backplane were cut across the middle so that both the program and data memories could plug into the same backplane and chassis.

The second port of the Triangle is the bus to the processors. It is a 50 line flat cable with TTL Tri-State drivers and receivers. The transfer uses a protocol which is essentially identical to the one being developed by the FASTBUS committee [4]. A 24 bit address field and 32 bit data field are used. They are time multiplexed on a set of 32 bus lines. The 4 most significant bits of the address field are decoded to select one processor with the remaining bits selecting the internal addresses of the processor's memory. Thus the bus allows direct access to any location within any processor. The rate of transfer on this bus is one word in 700 nsec. thus the transfer rate on the data side is nearly 6 M bytes per second and on the program side it is equivalent to nearly 3 M bytes per second of IBM object code.

The third port of the Triangle is a PDP-11 UNIBUS. A PDP-11/04 with 40 K bytes of memory is used as the control computer for the system. This port has 6 control registers to allow the PDP-11 to control the data flow between the three ports. Care has been taken that different software tasks in the computer have different registers that they control, thus making the software tasks easier to write.

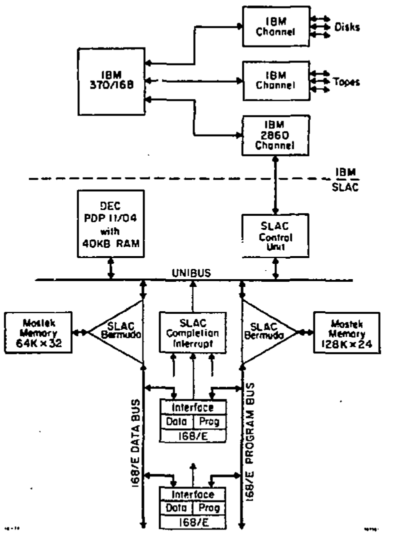

Figure 2: The Bermuda Triangle System

The buffer memories are loaded from the UNIBUS. An 8 K byte portion of the buffer memory appears as an 8 K byte portion of UNIBUS address. Both these "windows" have the same UNIBUS address, but only one is enabled at a time by a bit in their page register. Each Triangle has a 15 bit page register which is shifted left 8 bits and added to the offset from the start of the UNIBUS window to determine the buffer memory address. Thus, from the UNIBUS one can access up to 8 M bytes of memory in 8 K byte pages, where the pages can be aligned on any multiple of 256 bytes.

The processors are normally loaded from the buffer memory. From the UNIBUS port, the PDP-11 loads an address register for the buffer memory, an address register for the processor bus, and a word count register. When the word count register is loaded the Bermuda Triangle transfers the data until the word count is exhausted. It then causes an interrupt on the UNIBUS port. The results from the processor are normally loaded into the buffer memory in the same fashion. A bit in the Triangle's control status register controls the direction of transfer.

The PDP-11 gets access to the control registers of the 168/E processor by a 1 word window of the Bermuda Triangle from the UNIBUS port to the processor bus. In this case the processor bus address is taken from the same address register mentioned above. The double use of this address register is not a problem because one never attempts to gain access to the control registers of the processor while transferring data to or from it. One can also gain access to the processor control registers via either the program or data Bermuda Triangle.

CHANNEL INTERFACE

With the 168/E's and the Bermuda Triangle, the PDP-11 only needs a source for the raw data and a sink for the results. For this purpose, we designed an interface between the UNIBUS and an I/O Channel of the IBM 360/370 computer. Data is transferred between the 360/370 and the PDP-11 UNIBUS at full channel speeds (1.2 MB/sec) with the minimum software overhead on the IBM system. We have measured the CPU overhead on the 360/91 to be only 3.8 msec per event. IBM calls such a device a "Control Unit", and it looks like a tape drive or a disk to the IBM computer. This means that ordinary batch jobs can transfer data to and from the Bermuda Triangle system. The FORTRAN programmer gets access to the system by a simple FORTRAN callable subroutine.

Thus the IBM 360/370 reads the raw data from tape, sends it to the PDP-11 to be processed by the 168/E - Bermuda Triangle system, receives the results and writes the output tape. The IBM system with its 24 hour staff handles all the job scheduling, tape mounting, etc. Production jobs will be submitted to the system as is done now, and each job will first initialize the PDP-11 and buffer memories.

To synchronize the PDP-11 and 370 software, the 370 always attempts a read from the PDP-11 before a write. When the IBM computer reads results from the PDP-11, it obviously frees a buffer in the PDP-11 system, thus a write can then always be done. For normal event transfers, the control unit transfers directly to or from the data buffer memory through the 8 K byte UNIBUS window of the Bermuda Triangle, with the PDP-11 setting up the appropriate address and page registers. If the IBM computer attempts a read when no data is ready in the 168/E system, the control unit sends back a 'BUSY' response. When this signal is received, the IBM channel simply queues the read command without causing an interrupt to the CPU. When the data becomes ready for transfer, the PDP-11 loads the word count register in the control unit, and it sends a request for service to the IBM channel. This request signal wakes up the channel and the transfer is started. This is standard operating procedure for devices on a IBM 360/370 channel. The whole data transfer procedure is handled by the IBM channel. The IBM CPU is free to work on other jobs from the time it issues the Start I/O instruction until it receives an interrupt that the transfer is complete.

PDP-11 SOFTWARE

The PDP-11/04 computer has the job of controlling the 168/E overlays, the transfer of event data to and from the 168/E, and the transfer of data to and from the control unit. The job is divided into a number of software tasks, corresponding to the non-shareable hardware resources. There is a task for each processor, a task for the channel interface, and a task for each of the processor busses. As was mentioned earlier, the Bermuda Triangle was designed so that the hardware resources could easily be assigned to specific software tasks. We have chosen a small multi-tasking executive called SPEX [5] which allows all the tasks to be resident in memory and hence no disk is required on the PDP-11. It has been used as the data acquisition system in several experiments at FermiLab, Brookhaven, and CERN.

Each of these tasks is "driven" by a queue of work to do. The channel interface tasks receives raw event data and queues it to the processor work queue. When a processor becomes available, its task will take an event from the work queue, supervise its transfer to the processor, its execution through the various overlays, and the transfer of the results back into the buffer memory. The processor task will then queue the result buffer to the channel interface work queue and start working on another event from the processor work queue. Meanwhile, the channel interface task initiates the transfer of results from the buffer memory to the IBM channel. It then initiates a transfer of a new event from the IBM channel to the buffer memory and queues it to the processor work queue. The processing cycle is started by a task which supervises the initial transfer of the overlays and constant COMMON blocks into the buffer memory. It also fills the remainder of the data buffer memory with as many events as it can. The individual processor tasks will asynchronously want to send events and overlays to the processors over the two processor busses. Since the busses can only perform one operation at a time, the processor tasks queue their requests to the individual tasks which are assigned to the busses. The executive, SPEX, handles all queuing and synchronization of the tasks.

The PDP-11 itself is loaded via the channel. The IBM computer can send a hardware "Boot" command to the PDP-11 so that each job on the IBM computer completely re-initializes the whole system. The PDP software is written using a cross-assembler which is run on the IBM computers, thus there is no need for any permanent storage devices on the PDP-11 such as disks, tapes, etc. This absence of any peripherals, other than a terminal, should help make the system very reliable and reduce maintenance. The D.E.C. program, ODT-11, is loaded into the PDP-11 with the executive and associated tasks to aid in debugging. The complete software for 10 processors requires 20 K bytes of PDP-11 memory. About 700 micro-seconds of PDP-11 CPU time is needed per overlay.

SUMMARY

In October 1979 this system came into operation with one processor. It executes all of the LASS production program with essentially the same results as an IBM 370 computer. That is, identically the same points are found on all tracks in every event; only the fitted helix parameters showed some differences. Small differences were expected since the fitting of tracks to helices in the solenoid is done entirely in IBM double precision. However, the differences we see are in the least significant hexadecimal digit in the fitted track parameters except for 2% of the tracks which are very poorly defined.

We feel the importance of emulation can not be overemphasized. The LASS production code is nearly 20,000 lines of FORTRAN. It would be extremely time consuming to have re-written this code in assembly language no less microcode. Before one could have finished such a project the FORTRAN source would have undoubtedly been changed. With emulation not only doesn't the code need to be changed but also verification of the processor is easily accomplished by comparing its results on a set of events with the results from the same events run on the IBM computer. Even the smallest error in the system's hardware is detectable. For example, when we first tried to run the LASS code on the system only three errors were made. Only one of the errors led to results which were obviously wrong. The effect of the other two errors was only that a few extra reasonable looking points were added to some of the detector's coordinate banks. The cause of one of the errors was forgetting to load one of the constant COMMON blocks. The cause of the other error was a bad I.C. which only had an effect when floating point register 6 was used. Another important advantage is having only one copy of the source code which is used both for the IBM computer and the 168/E processor. In fact, the input to the translator is the link-edited load modules which are used to run the program on the IBM computer. When the production program is modified, it is relatively easy to produce a new microprogram for the processor. One just generates new memory maps for the overlays and then re-translates the IBM load modules.

We are now preparing to run many thousands of events on the system and on the IBM computer. This will check for pathological events to a level of one in ten thousand or better. We are also preparing additional 168/E processors and expect to have 6 processors on the system by the end of 1979. We will thus be able to start analyzing our 50 million events with a system nearly equivalent to 3 dedicated IBM 370/168's running 24 hours a day.

ACKNOWLEDGEMENTS

We would like to thank D.W.G.S. Leith for his support and encourgagement on this project. Special thanks must go to Hanoch Brafman of the Weizmann Institute who designed the 168/E microprocessor during his visit to SLAC. The first real test of the processor was done with the efforts of Rafi Yaari, also of the Weizmann Institute, who bought us the track reconstruction code from the TASSO spectrometer at DESY to run on the processor in February 1979. Thanks also goes to George Aiken and Al Kilert of SLAC who designed the chassis for the processor.

BIBLIOGRAPHY

(1) Paul F. Kunz, The LASS Hardware Processor, Nuc. Instr. Meth. 9 (1976) p. 435.

(2) Paul F. Kunz et. al., The LASS Hardware Processor, Proc. 11th Annual Microprogramming Workshop, SIGMICRO Newsletter 9 (1978) p. 25.

(3) Roger B. Chafee, GOODGNUS, CGTM No. 198 Stanford Linear Accelerator Center, Stanford, Calif., 94305

(4) B. Wadsworth, FASTBUS - An Emerging Laboratory Standard, a paper in this volume.

(5) SPEX: A Spectrometer Executive, General Structure, Programmers Manual, and Users Manual, J.T. Massimo, B. Nelson, L.J. Levinson, Brown University High Energy Physics Group Internal Reports 123, 124, 125 (1972)