A MICROPROCESSOR-BASED SINGLE BOARD COMPUTER FOR HIGH ENERGY PHYSICS EVENT PATTERN RECOGNITION

H. Bernstein, J.J. Gould, R. Imossi, J.K. Kopp, W.A. Love,
S. Ozaki, and E.D. Platner
Brookhaven National Laboratory*
M.A. Kramer
City College of New York+

A single board MC 68000 based computer has been assembled and bench marked against the CDC 7600 running portions of the pattern recognition code used at the MPS. This computer has a floating coprocessor to achieve throughputs equivalent to several percent that of the 7600. A major part of this work was the construction of a FORTRAN compiler including assembler, linker and library. The intention of this work is to assemble a large number of these single board computers in a parallel FASTBUS environment to act as an on-line and off-line filter for the raw data from MPS II and ISABELLE experiments.

Introduction

Anticipated computing requirements at ISABELLE are in the range of 20 to 100 CDC 7600 equivalents. Efficient development of event processing programs requires the facilities and sophisticated software of a large computing center. However, once such a program exists it should be possible to run it on a smaller, simpler computing system. The 168E and PUMA approaches achieve this goal by emulating the target machine instruction set. We believe that when the computing requirement grows as anticipated, the scale is such that it becomes feasible to develop software to translate a working higher level program on the target machine into an executable module on the simpler system. This allows the use of microprocessor-based single board computers. We set out to find how practical it was to go in the direction of minimizing the hardware cost at the expense of having to write a FORTRAN compiler.

It is well known that IC memory densities have doubled $\approx$ every year for many years. This trend is likely to continue for several more years. Thus it becomes practical to put 250-500K word RAMS on a single PC board. It is also known that similar trends have occurred in IC microcomputers, i.e., both increased complexity and increased speed. Two years ago when we started this project, three 16-bit micros were just being announced, the Z8000, 8086 and MC68000. We took a close look at these three processors and chose the MC68000 as the best choice for the sort of number crunching required by our pattern recognition software. It was obvious that to have any chance to meet our computing speed goals that a hardware floating point unit was required. At that time only the AM9511 was available, so for these benchmarks we assembled a computer centered around the MC68000 coupled to a 9511 with on-board dynamic RAMs. This paper discusses the hardware and software approaches, gives timed benchmarks and projects future developments made possible by the continuing advances in IC technology.

Preliminary to putting together a full-scale single board computer, two benchmark tests of the MC68000 were done. The first test was to do an effective mass calculation from data presented by five PWC's in a magnetic field. The MC68000 was programmed in machine language and only those instructions requiring ten or less micro-cycles were used. The pattern recognition was done by lookup tables as were all squares and square root

calculations. As a comparison, the same routine (lookup table approach) was programmed in FORTRAN on the CDC 7600. It ran only 4 times faster on the 7600 than on the 4 MHz MC68000.

As a second benchmark, a Queens chess game excercise was run using PASCAL programs on the respective machines. Only fixed point calculations were involved. The reference machine in this case was the DEC KL10. The 68000 ran half as fast as the KL10. It is apparent from these results that when the strength of the 68000 is utilized, it is a rather powerful microcomputer. In this work it was our purpose to see how well one could do with a configuration suited to full-blown pattern recognition in high energy physics.

Hardware Description

A Motorola 68000 Design Module, a 9511 Arithmetic Processing chip, memory, a Floating Point Timing device and a 90K byte per second link to a DEC PDP10 were interconnected as shown in Fig. 1. Pattern recognition programs were compiled on the PDP10, down-loaded into the 68000 memory and executed. Floating point operations were performed in the 9511 Arithmetic Processing chip.

Motorola 68000 Design Module

The design module contains a 68000 16-bit processor, 32K bytes of RAM, an 8K byte debug monitor program in PROM, a terminal connection, a host computer connection, and an interface to a Motorola standard bus. The debug monitor provides a means of controlling program execution. It also contains a variety of useful IO and conversion routines. The standard bus has arbitration capability so that external devices can reference memory on the design module. It also provides a RESET line that is used to force the 68000 into the debug monitor initialization. In addition, priority interrupt request signals are available. These are used by the PDP10 to force execution of the code that has been loaded into the modules memory.

Arithmetic Processing Unit, 9511

The 9511 provides fixed and floating point operations and a variety of trigonometric and mathematical operations. All transfers, including operands, results, status and command information, take place over an 8-bit bidirectional data bus. Operands are pushed onto an internal stack and a command is issued to perform an operation on the stack. Results are removed from the stack. The 68000 addresses the 9511 as memory address 7FFC for data transfers, and as memory address 7FFF for commands. Only 16-bit word addressing modes are allowed.

The 68000 after issuing a command to the 9511 enters a software wait loop until the 9511 returns a signal indicating that it has finished. The time needed to execute a command is operation and argument dependent. A 32-bit floating add can take from 20 μs to 115 μs. It is anticipated that the 9511 processing chip will be replaced by the ten times faster National chip NS16081 when it becomes available later in the year. In order to calculate how fast pattern recognition programs will run with the NS16081 doing the floating point, a floating point timing module was connected to the bus. This module measured how long it takes to execute a program and how much of that time was spent doing floating point operations.

Link to the PDP10

The PDP10 is connected to the Motorola standard bus via a 90K byte per second serial link. All communication between the PDP10 and the 68000 is via the 68000 memory. The PDP10 can do the following:

1. Read/write the 68000 memory.
2. Reset the 68000.
3. Generate an interrupt on channel 7.

To read a 68000 memory address, the PDP10 transmits a 36-bit word containing the memory address and a read bit. The word is assembled in the interface hardware and a bus request generated. After a μ-second or so the 68000 will release the bus. The address and control signals are then gated onto the bus and the read of the 68000 memory is completed. The interface hardware then sends a 36-bit word back to the PDP10 containing the memory address and the data at that address. An interrupt is generated at the PDP10 when the word is assembled there.

A write of 68000 memory is similar to a read except the word sent from the PDP10 contains the 16 bits of write data as well as the memory address and write bit. The interface hardware sends a write complete signal back to the PDP10 indicating it is ready for another command.

To force the 68000 into a known state before loading a program, the PDP10 sends a 36-bit word containing the reset bit. A reset forces the 68000 to start execution at the beginning of the debug monitor. The monitor writes into memory the interrupt processing vectors as part of an initialization procedure. The PDP10, as a way of controlling the 68000, changes the channel 7 interrupt vector so that it points to the starting address of a program that it wants the 68000 to start executing. It then sends a 36-bit word containing the channel 7 interrupt request bit which results in the 68000 jumping to starting location contained in the channel 7 interrupt location.

Software

Background

The Motorola MC68000 software available in the summer of 1980 consisted of a non-relocating cross-assembler, a cross-PASCAL subset, a resident relocating assembler, linker, and PASCAL. The resident software was not acceptable for our application, in that it would have required a full host MC68000 system, which would have greatly diminished the cost-effectiveness of the hardware, and only postponed the need to provide full support from a host with full access to the experiment. Further, PASCAL was not a satisfactory choice of higher level language. Most of the interested physicists are familiar with FORTRAN, not PASCAL, and the loss of time and confusion involved in learning a new language seemed likely to weigh too heavily against any possible advantages of PASCAL.

This left us with the need to obtain a cross-FORTRAN compiler with support software for the MC68000. Our reasonable expectation was that such commercial software would become available in late 1981 or early 1982, not soon enough for this project. Funds were not available to commission a special effort from a software house. We decided to make an effort at providing a minimal FORTRAN system ourselves.

Overview

The software required consisted of:

1. A relocating cross-assembler,
2. A linker,
3. A FORTRAN compiler,
4. A subroutine library,
5. A host support program.

The linker, the subroutine library, and the host support program had to be written from scratch. The relocating cross-assembler was made by modifying the Motorola cross-assembler. The FORTRAN compiler was made by adding code generation to Ryder and Hall's "PFORT Verifier."

Cross-Assembler

In M68KASM, Motorola provides a macro-assembler for the MC68000. The assembler is written in FORTRAN, allowing for simple modification. To implement relocation, addressing was extended from 24 to 32 bits, with the high order 8-bits used to designate a relocation base. The base 255 was reserved to flag external references. New assembler directives were added to direct code to the desired blocks.

```
IDENT     Name
PSECT     Name, attributes
GLOBL     Name, ---, name
XFER      Expression
```

IDENT provides a name for a module, which is also the name of the first block. PSECT allows the creation of additional blocks, or code generation in a previously defined block. A PSECT could have attributes of "CON", for concatenate, "OVR" for overlay, "LCL" for local, "GBL" for global, "REL", for relocatable, or "ABS" for absolute. Code would normally be placed in a "REL,CON,LCL" block, and common in "REL,OVR,GBL" blocks. GLOBL identifies both entries and external references. If the named symbol is defined in the current module it is an entry, else an external reference. XFER specifies the starting address of a main program.

For operational convenience, the assembler was modified to allow multiple modules to be stacked in one input file.

Linker

The linker was written in FORTRAN as a simple three-pass linker. On the first pass block definitions are extracted and absolute origins assigned. On the second pass code and data are stored in their final positions in a random file with address fields relocated as necessary. The final pass copies the random file to a sequential file in Motorola's load image format. For subroutine libraries, an optional preliminary pass copies needed routines.

FORTRAN compiler

The PFORT verifier, QCPE#374, accepts ANSI FORTRAN 66 programs, parses them for violations of the language specifications, and provides detailed cross-reference maps both within routines and between routines. The verifier generates an internal symbol

table, but no code. We used the symbol table and the parsing to "tokens" as input to a compiler. This relieved us of the need to provide error messages or listings from the compiler, and guaranteed that we would only have to compile valid programs.

In code generation we attempted to avoid redundant stores and fetches with statements by flagging current register assignments. In the absence of statement labels, those assignments are retained between statements. To maximize the use of such optimization the calling sequence was specified so that the called routine is responsible for saving and restoring registers. No attempt was made to identify common subexpressions more complex than a subscripted variable.

DO loop control variables are bound to registers by the same mechanism. Arithmetic statement functions are implemented by generating code at the point of definition and transferring parameters by value in the dummies as variables. Register binding is cleared on the ASF call, so ASF code generation is exactly the same as assignment statement code generation.

Storage allocation is partially static and partially dynamic. All variables in common blocks and all variables in statements are given static definitions. All local variables not defined in DATA statements and all temporaries are allocated on a stack. This minimizes storage requirements without violating the natural user assumptions about DATA initialized local variables. A side effect is that all subroutines are inherently recursive.

Floating point is implemented via internal function calls rather than in-line. With our current floating point chip, this represents a penalty for each operation. With a faster chip, in-line code will be essential.

Constants and FORMATS are generated in a spacial PSECT, @CHARC, with no attempt to merge redundant constants. However, most integer constants are placed directly in instructions, so no great loss of memory is expected in that case. When we change to in-line floating point, the same should be true of those constants.

Fortran Run-Time Library

On the MC68000 side, IO and FLOP respectively handle I/O requests to the PDP10 host machine, and communicate with the Floating-Point Processor. On the PDP10 side, the host program TENDER translates between the MC68000 I/O requests and DEC's versatile FORTRAN package FOROTS. TENDER also converts between PDP10 and MC68000 number formats: the MC68000 sends and receives data in its own format, so on its side the Run-Time Library is independent of the host machine.

Input/Output

By using FOROTS on the PDP10 side, the MC68000 can use all I/O services and devices available in PDP10 FORTRAN. In particular, we let FOROTS do all our formatted I/O, avoiding tricky format-scanning routines that would have been a nuisance to code on the MC68000. To read or write a record, only three FOROTS calls are needed:

1. One call to initiate the operation, which provides the device number and (for formatted transmissions) a pointer to the format description and its length.

2. For each word to be transmitted, a call to a standard entry which is the same for all types of data and for transmission in either direction.

3. Finally, a call to terminate the operation.

In the MC68000 module IO, we have entries exactly corresponding to these, and the FORTRAN program supplies the same calls and parameters as would a PDP10 program. Other than copying the parameters and data to fixed locations in lower memory, all IO needs to do is count the characters in the Format description.

The MC68000 is master in each transaction dialogue. When ready with a request, IO sets a bit in a "mailbox" memory location which TENDER constantly reads through the data link by stealing memory cycles; this reading slows up the MC68000 by about 7%. On finding the bit set, TENDER transfers the parameters or data, converts to PDP10 number format where necessary, and generates a call to the counterpart FOROTS routine. If the request is to read, TENDER will convert the datum to MC68000 number format and send it to a fixed memory location, then in all cases clear the request bit.

Meanwhile, on the MC68000 side IO is waiting in a holding loop until the request bit is cleared. Thus I/O is completely unbuffered on the MC68000, but at present that is not a serious problem. Any buffering scheme would depend very heavily on the host computer's operating system, and so it would be best not to worry about this until we are well past the present experimental stage.

FOROTS provides the usual "END" and "ERROR" return options for recognizing I/O exceptions. TENDER uses these to record any error conditions and when IO requests termination of the operation, sends a report back to the MC68000, where IO stores it in a table indexed by device number. The FORTRAN program must check for errors or end-of-file by calling functions GOF(n) or UNIT(n) which return the table entry for device n and also reset it to "OK".

Startup and Synchronization

The MC68000 program is downloaded from a PDP10 disc file. Then TENDER is loaded in the PDP10. On startup, TENDER sends an interrupt to the MC68000, initiating execution at the address in the trap pointer. This address was set in turn by the XFER pseudo-op generated on compiling the main program. The entry INITIO in IO has the job of initializing the stack pointer and then waiting to make sure the PDP10 is actually up and monitoring the "mailbox." To do this, INITIO sets the appropriate bit in the mailbox word, and waits in a holding loop until it is cleared by the PDP10. There are no problems of timing per se, since FOROTS is completely buffered and does not care what happens between calls; but we do have to be sure both machines are in the proper phase of the request-processing cycle.

Floating Point Operations

Five steps are necessary to perform a floating-point operation using the 9511:

1. One or two 32-bit operands are pushed onto the 9511 stack, in 8-bit bytes.
2. An operation code is sent.
3. The service routine FLOP waits in a holding loop until the operation is complete.
4. FLOP must check the 9511 Status Bits for error conditions - overflow, underflow, illegal operands, etc.
5. The 32-bit result is unloaded in bytes and reassembled into a 32-bit doubleword.

These steps consumed a good fraction of the total running time in our test, but only because of details that can be readily improved in our next system. Rather than discussing these details in isolation, therefore, we would like to combine a description of what we did with recommendations for the next development stage, based on our experience.

1. The 9511 data bus is only 8 bits wide, but it must be addressed as a 16-bit word. On the MC68000 this introduces irrelevant data shuffling operations. Using a chip with a 16-bit bus would eliminate most of that waste, but it would be even better to interpose a hardware register that could be addressed as a 32-bit doubleword and automatically unpack or pack bytes without further intervention by the MC68000. Such a register would also simplify the compiler and the generated coding. At present, it is necessary to call a subroutine to execute floating operations, with attendant overhead. It would be faster, and actually take less memory, to "store" the operands and op. code by sending them to special addresses (just as we use addresses 7FFC and 7FFF for 9511 registers), then fetch the result as if from memory. This would be coded in line.

2. An additional instruction would be needed in this sequence, namely a call to a holding routine to wait for completion of the operation and check for error conditions. One immediately sees that some useful computation could be done in the time between sending operands to the FP Chip and checking for completion. The 7600 FORTRAN compiler FTN does this very elegantly, but we believe even a relatively simple algorithm could save much time by, say, overlapping subscript calculations with floating operations. Another way to deal with "software floating point" is to generate interpretive code. This is not very promising because so much of the code deals not with floating operations, but with subscript calculations and loop indexing.

3. There is no need to unload a result from the FP Chip's internal stack if it will be an operand in the following operation. The compiler can easily recognize this situation.

4. The few masking and testing operations now required to check the chip's Status Register for error conditions will become a significant cause of delay. Therefore it would be efficient to use special hardware to generate an interrupt if any error bits are found to be set on completion of the operation.

Further Compiler Improvements

Next to handling floating point operations, the compiler's main problem seems to be calculating subscripts. Much time and space would be saved by including some well-known (and well understood) optimizations:

1. Detecting common subscript expressions in statements (or groups of statements) so that indices can be calculated once for all variables with the same combination of subscripts and dimensions.

2. Converting subscript calculations in DO loops to simple incrementing of an initial value that is calculated before entry to the loop.

Finally, one point is related to saving the users' time. Our present compiler is based on the Bell Telephone Laboratory's PFORT Verifier, which accepts only a rigorously standardized version of FORTRAN. In principle, this is very desirable, as programs written in this language are very easily moved to other machines. In practice, we will want to transfer many programs to the MC68000 system, and these will inevitably contain

nonstandard expressions, in particular those from CDC7600, DEC PDP10, and perhaps some IBM Fortrans. Therefore, it would probably be more desirable to have our compiler accept the widest possible range of nonstandard expressions, though without allowing new non-standard elements.

GETX Benchmark Test

To obtain a realistic comparison we developed a sample program around the central algorithm we have used in our "local" pattern section of our drift chamber analysis program. This subroutine (GETX) receives packed data from the drift chamber readouts, unpacks it and looks for associated hits in each module (a set of three drift planes). The data is normally communicated to this routine via common blocks. For the test, the full program was run on actual data and the appropriate information needed by GETX was written on a disk file for each event.

The KA-10 version was compiled using the standard DEC compiler. The bulk of the compilation of the 68K version was done by a FORTRAN cross-compiler based on PFORT, however, at the present state of that compiler, some hand corrections and coding was needed. (No attempt was made to optimize the speed however.)

A small driver program was then written to read the disk, load the common blocks, and then run GETX. To minimize the effects due to I/O timing, each event was processed 100 or 1000 times. Outputs were compared to be sure the 68000 program was working correctly. The result was that the 68000 took 9 times as long to run as the PDP KA-10.

Future Prospects

A newer version of the 68K is available that operates at 2-1/2 times the current speed (10 MHz). That processor, combined with obvious compiler improvements, notably in subroutine argument list handling, should gain a factor of 4 in the 68000 time.

A new floating point processor is also available that is 10 times as fast and can be loaded 16 bits at a time (the current processor requires eight 8-bit loads to load two floating point operands). Thus we confidently expect that the new system should operate at 5-10 times the current version, which would then be equivalent to about the processing speed of the KA-10.

Further improvements, such as special register hardware for interfacing the 68K and the floating point processor and further compiler improvements could further improve this ratio.

Conclusion

We have shown that with a 2-year-old microcomputer and an even older floating point coprocessor, it is possible to run a FORTRAN pattern recognition program at a throughput 1/9 that of the KA-10 (KA-10 $\approx$ 1/30 CDC 7600). It is expected that with an improved compiler, the fast version of the MC68000 and the new National floating point coprocessor, that these programs can run at speeds very close to that of the KA-10. Unfortunately, this is $\approx$ 30 times slower than that of our target machine, the CDC 7600. However, this factor should be gained by the micro-computer industry within the next 3 to 4 years. In the meantime we plan to concentrate our efforts on developing a bus structure such that large numbers of these single board computers can operate in parallel processing different

events. The prime candidate for this structure is the NIM "FASTBUS", on which an intelligent controller would control the event downloading into available processors and the outloading of those events that pass the pattern recognition constraints. Once this "network" is operational as a system, the precise choice of processor and coprocessor can be made on the basis of the hardware available at that time.

References

* This research was supported by the U.S. Department of Energy under Contract No. DE-AC02-76CH00016.

+ Research supported by the National Science Foundation and the City University of New York PSC-BHE Research Award Program.

* * *

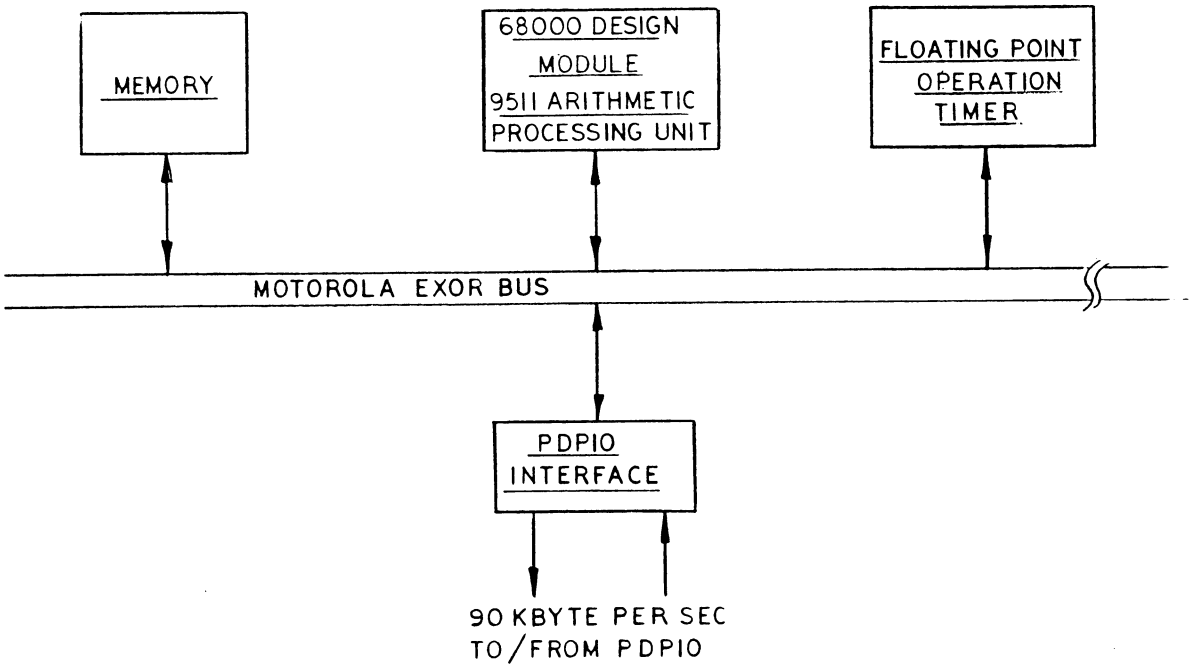

Fig. 1