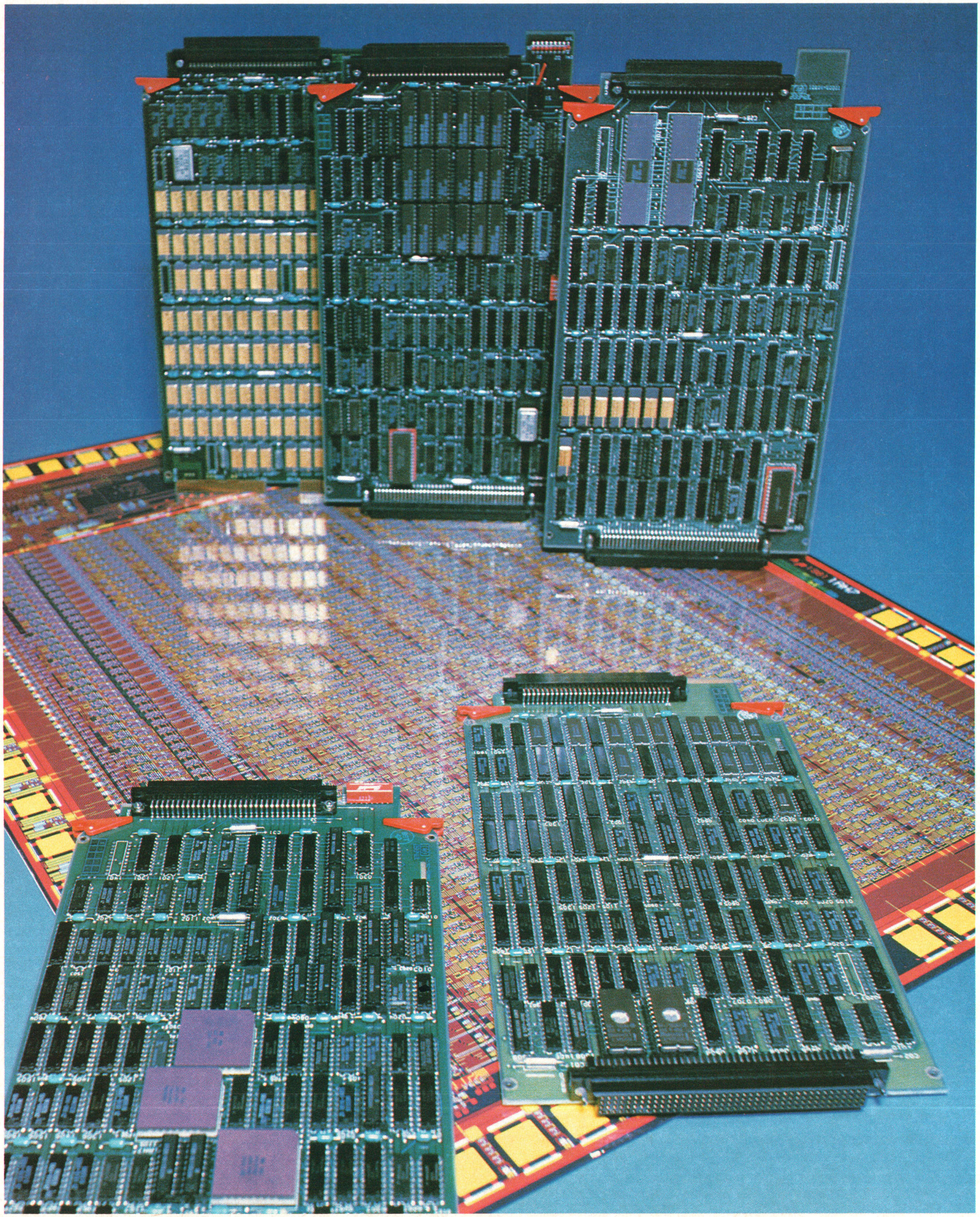
HEWLETT·PACKARD
JOURNAL

FEBRUARY 1984



HEWLETT·PACKARD JOURNAL
Technical Information from the Laboratories of Hewlett-Packard Company

FEBRUARY 1984 Volume 35 • Number 2

Contents:

In this Issue:

Real-time computers are designed to control or monitor real-world processes. These processes almost always involve other equipment, such as milling machines or semiconductor furnaces, and large numbers of sensors and activators. A real-time computer must keep up with the equipment around it. It must respond to interrupts quickly, it must transfer large amounts of data rapidly, and it must perform any necessary computations and data transfers efficiently, so that it is never overwhelmed by requests for its services. It must also be reliable, since a large amount of expensive material and equipment may depend on it.

Hewlett-Packard's premier real-time computer family is the HP 1000 product line. In this issue you'll read about the new HP 1000 A-Series Computers, the highest-performance and most reliable HP 1000s developed so far, and about RTE-A, the latest version of the HP 1000's Real-Time Executive operating system. A-Series computers range in performance from the A600's one million instructions per second to the A900's three MIPS. To keep costs lower while achieving these high performance levels, the designers of the A-Series didn't go to a fast but expensive logic family like ECL (emitter coupled logic), choosing instead to rely on advanced architectures, special hardware, and clever ways to save microcycles. In the A900, they've provided a pipelined data path, a cache memory, and three special chips that add, multiply, and divide floating-point numbers. On this month's cover are the five boards of the A900 processor. The data path board (with the large square floating-point IC packages) and the cache memory board are in the foreground. In the background are the memory controller board, the sequencer board, and the memory board with its gold-covered RAMs. Under the boards is a color print of the mask set of the floating-point divide chip.

In designing microprogrammed processors like those in the A-Series and other microcomputer-based systems, logic analyzers and logic development systems are invaluable. Last March, we published a series of articles about the HP 64000 Logic Development System. One of its subsystems, the HP 64600S Logic Timing/Hardware Analyzer, has just been upgraded with some new software that gives the designer several sophisticated new ways to process and analyze timing data collected from a system under development. The story begins on page 32.

-R. P. Dolan

Editor, Richard P. Dolan • Associate Editor, Kenneth A. Shaw • Art Director, Photographer, Arvid A. Danielson • Illustrators, Nancy S. Vanderbloom, Susan E. Wright • Administrative Services, Typography, Anne S. LoPresti, Susan E. Wright • European Production Supervisor, Henk Van Lammeren

A New Series of High-Performance Real-Time Computers

The HP 1000 A-Series consists of three compatible processors rated at up to 3 MIPS. They use a new Real-Time Executive operating system and are available in board, box, and system configurations.

by Marlu E. Allan, Nancy Schoendorf, Craig B. Chatterton, and Don M. Cross

THE NEW HP 1000 A-SERIES family of computers is designed to provide solutions to specific real-time needs in manufacturing, automation, and other performance-critical environments. Implemented with state-of-the-art technology, the new computers offer major new capabilities to meet the challenging demands of OEMs, end users, and system designers.

The family consists of three compatible processors, the A600, A700, and A900. Each processor uses the new RTE-A operating system (RTE stands for Real-Time Executive), and identical compilers and subsystem products. Each computer employs the distributed-intelligence HP 1000 L-Series I/O system, which uses an I/O processor on each I/O card.

Available in board, box, and system configurations (see Fig. 1 and Table I), these processors offer configuration flexibility for OEMs and end users across a wide spectrum of applications. Ranging in performance from 1 to 3 million instructions per second, the A-Series family of computers offers very high performance at economical prices.

The A600 Processor

The A600 processor is the lowest-price member of the A-Series product line. Based on the 2901 microprocessor, the two-board CPU supports 128K bytes of memory, expandable to 4 megabytes. This processor combines microprogrammed instruction execution with hardware assist to achieve 1-million-operations-per-second performance. The A600 comes as a rack-mount computer and in numerous system configurations. It is best suited for dedicated applications, such as numerical control, energy management, and automated testing.

The A700 Processor

The A700 processor complements the A600 processor with additional capabilities. For computation-intensive applications, the A700 can be configured with an optional hardware floating-point processor or customized by user microprogramming. Optional error correcting (ECC) memory allows memory expansion to 2 megabytes in 512K-byte increments. Alternatively, the A700 can be configured with parity memory up to 4 megabytes in 1-megabyte increments. The A700 processor is available in a four-board processor configuration in addition to rack-mount and system offerings.

The A900 Processor

The A900 processor is the highest-performance member of the A-Series product line. Using a 4K-byte cache memory, a pipelined data path, standard hardware floating-point chips, and microcoded scientific and vector instruction sets, the A900 can perform more than 3 million operations per second. Error correcting memory is standard in 768K-byte increments for a total memory capacity of 6 megabytes. A user-microprogramming package is available. The A900 is offered in rack-mount or system configurations.

RTE-A Operating System

RTE-A is the real-time operating system for all three processors in the HP 1000 A-Series family. RTE-A evolved from previous members of the RTE family and is a product of some proven real-time features from past versions and some entirely new features. The main goals of the new features in RTE-A are to provide multiuser tools and to support large programs with large amounts of data. RTE-A

Table I
HP 1000 A-Series Computers

	A900	A700	A600
System	2199C/D, 2439	2197C/D, 2437	2196C/D, 2436
Box	2139A	2137A	2136A
Board	---	2107AK	2106AK
Operating system	RTE-A	RTE-A	RTE-A
Standard memory	ECC	Parity	Parity
Optional memory	---	ECC	---
Memory: Standard	768K bytes	256K bytes	128K bytes
Maximum	6M bytes	4M bytes	4M bytes
Memory cycle time	181 ns (eff)	500 ns	454 ns
Hardware floating-point	Standard	Optional	---
Operations/second			
Base instruction set	3,000,000	1,000,000	1,000,000
Floating-point	500,000	204,000*	53,000
Direct memory access rate	3.7M bytes/s	4M bytes/s	4.27M bytes/s
User microprogramming	Yes	Yes	No
FORTRAN77	Yes	Yes	Yes
Pascal/1000	Yes	Yes	Yes
BASIC/1000C	Yes	Yes	Yes
Graphics/1000-II DGL & AGP	Yes	Yes	Yes
DSN/Distributed System	Yes	Yes	Yes

*With hardware floating-point.

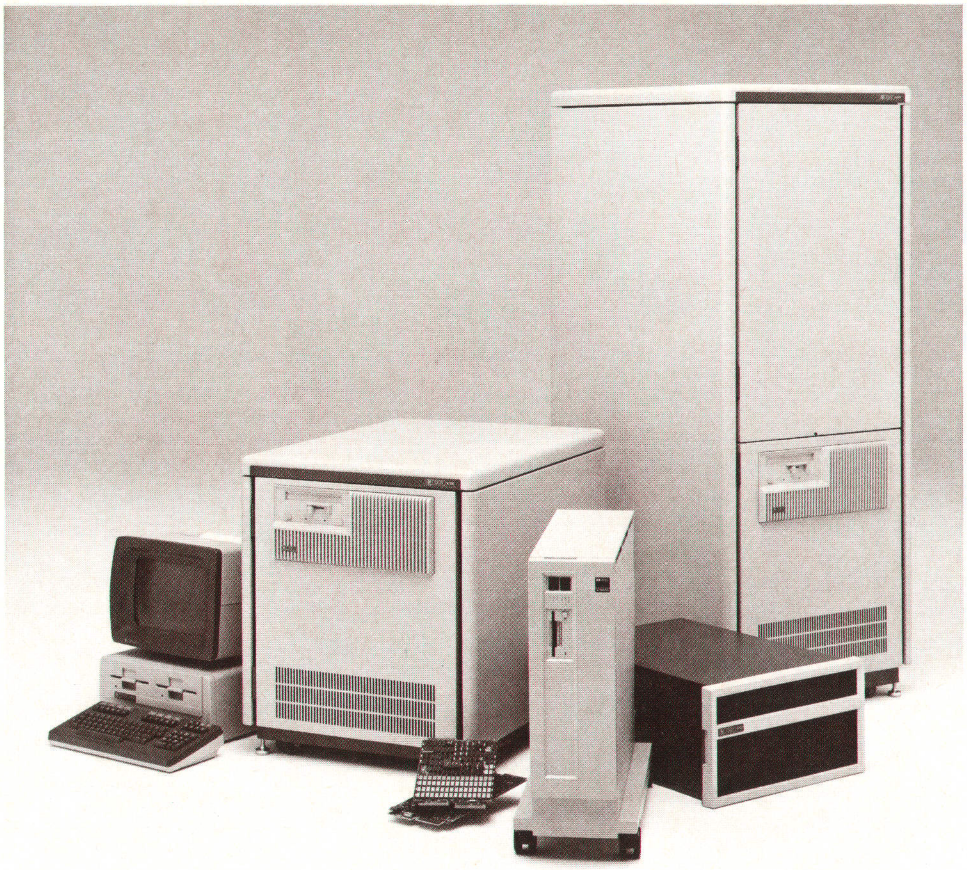


Fig. 1. *HP 1000 A-Series Computers come in various configurations and have performance ratings ranging from 1 to 3 million instructions per second.*

is implemented in a modular fashion so that one operating system can span the size and performance ranges of the entire A-Series. It is a configurable operating system and can be tailored by the user to fit any particular application.

A major enhancement in RTE-A is the multiuser environment. A modern hierarchical file system allows logical grouping of files and protection of files. It also includes time stamping of files on creation, last access, and last update. This time stamping information is used to provide an incremental backup capability for the system. Another important feature of the enhanced file system is transparent access to files on other RTE nodes in a distributed system. This enhanced file system is used as a base for a multiuser environment. Logon and logoff utilities provide identification of users and their capabilities. This identification is used in conjunction with the protection mechanisms in the file system to identify and protect files belonging to individual users. The multiuser environment is completed with a command interpreter that has on-line help facilities and an outspooling utility for programmatic and interactive outspooling of files to devices or files.

RTE-A has a number of features to support large programs and large amounts of data. Virtual memory for data is a scheme that allows users to access data in main memory and on disc as if it were all in main memory. EMA (extended memory area) is a special case of virtual memory for data. It provides faster access to data by allowing up to 2 megabytes of data in main memory. An EMA can be shared by multiple programs.

RTE-A takes advantage of new hardware features in the A-Series to provide separation of code and data for user programs. This allows transparent support of large programs (up to 7.75 megabytes of code) using a demand segment virtual memory scheme. It also allows multiple copies of the same program to share code.

A-Series Performance

The A-Series Computers were designed with excellent price/performance as an important goal. Their performance has been verified in benchmarks run against their predecessors and other currently competitive products.

Before discussing specific results, let's review how performance is typically measured. Computers are expected to perform a variety of tasks, from program development, to controlling and monitoring a milling machine, to assisting engineers in complex designs. In terms of performance, what's important to the people using these computers?

Some of the things often required are:

- Good throughput, or how much work can be done in a given amount of time. This may vary depending on what type of task is performed, e.g., floating-point computations, compiling programs, etc.
- Good response, or how fast the computer can respond to a certain input, such as an interrupt or a DMA transfer. An interrupt might be a terminal keyboard input, a sensor indicating a malfunction in a process, a satellite sending

data, etc.

- Good utilization, or how effectively the resources are being used. If only part of the machine is used a significant amount of time, then the user may be paying for something unnecessary.

A-Series Computers are often used in real-time applications, where the computer must keep up with the equipment around it. Such environments require good performance in integer and floating-point operations, good interrupt response time and I/O transfer rates, and the ability to handle large amounts of data efficiently.

Benchmarks are standard programs used to compare the performance of one computer with that of another. One should be careful when selecting benchmarks to measure performance. Sometimes a particular benchmark may be biased in terms of what it's measuring, or may exploit a particular aspect of the computer that's not used much. The best benchmark is the application intended, but this is not always practical. The benchmarks discussed here are a small sample of the ones that have been run on the A-Series. The F-Series included in the results was previously the high-end HP 1000.[1]

The Whetstone benchmarks are industry standards that were written by the National Physical Laboratory of England. The programs are written in FORTRAN, and attempt to represent average program mixes. The two most common are the single- and double-precision Whetstones. These measure performance, including floating-point, in single and double precision, respectively (32-bit and 64-bit floating-point numbers). The performance results are shown in Table II, which indicates the execution times both in minutes and normalized relative to the A900. These times were measured on a quiescent system and are elapsed times. Note that the A700, A900, and F-series times include floating-point hardware while the A600 does floating-point operations in microcode. These benchmarks are often expressed in terms of "Whetstones per second." The execution times are for 10 million Whetstone instructions, so dividing this number by the execution times yields the column labeled KWIPS (thousands of Whetstones per second). The A900 KWIPS figures are better than those of many 32-bit "super minicomputers," even though the A900 is primarily a 16-bit computer.

In applications making little use of floating-point operations, integer performance is more important. A FORTRAN benchmark was developed to measure integer performance, and the results are shown in Table III. Here, single (16-bit) and double (32-bit) integer operations were measured. Normalized times are shown, referenced to the A900. Also, a MIPS figure is included, which is the number of millions of instructions executed per second. These figures are less than the base set instruction rates, since more complex instructions are required. In this example, the A900 does very well because of its optimized data paths and good 32-bit capabilities.

Many applications require the use of discs for storage and retrieval of data. Since discs are typically slower than the CPU, their effect must be taken into account. One such application is compiling programs. The A-Series supports a variety of compilers including a macroassembler, FORTRAN, and Pascal. While the speeds of these vary, the relative performance is consistent. Compilation speeds for the Pascal compiler are shown in Fig. 2. These are in lines per minute and are measured in a quiescent system with a working set varying up to 270 pages (1024 words/page) using an HP 7925 Disc Drive. On the A900, only about 60% of the processor is used during a Pascal compilation. The remaining 40% is available for other activities.

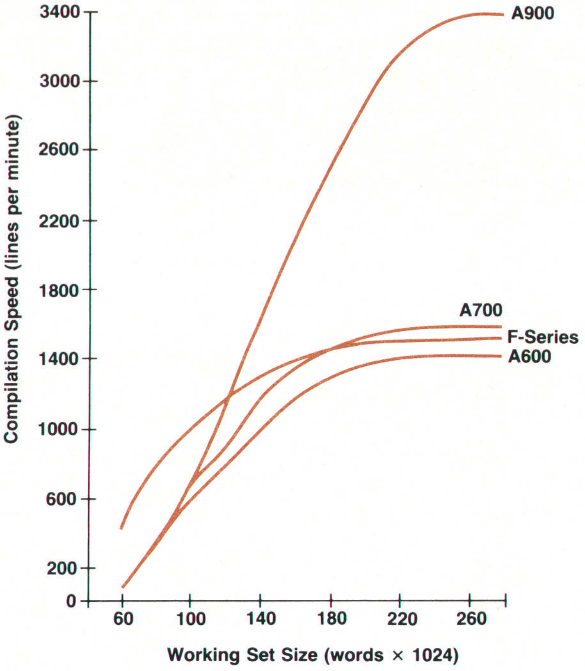


Fig. 2. *Pascal compilation speeds for the HP 1000 A, E, and F-Series Computers.*

The Pascal compiler is a VMA program, using the VMA (virtual memory area) capabilities of HP 1000 Computers. VMA allows a program to access up to 7.75M bytes of data, even though much of it may be on disc. The working set size is the amount of the data that can be in memory. The performance range is less here, since all of the A-series Computers use the same I/O system, and the disc time is now part of the execution time.

Interrupt response time is generally a good measure of how quickly a processor can respond to an external event. The data in Table IV is the elapsed time in microseconds from an interrupt until the system enters the appropriate driver. A driver is the piece of system software that communicates with a particular device or set of devices. Before entering the driver, the system must save certain state information and determine the appropriate action. Interrupt response time thus measures the operating system's performance as much as it measures the CPU's.

Even though the I/O systems are the same, much of the time is spent executing other instructions, and this results in the 3-to-1 range. An interrupt response time of 150 μs is very good for a system with the functionality of RTE-A.

In selecting the right processor for a particular application, the type of performance necessary must be evaluated. The A-Series offers a range of CPU speeds from the A600 at 1 MIPS to the A900 at 3 MIPS. While the actual instruc-

Table II
Whetstone Performance
(Times are in minutes rounded to two decimal places)

	A900 time(rel)KWIPS	A700	A600	F-series
Single Precision	.12 (1) 1344	.31 (2.6) 541	.87 (7.3) 192	.37 (3.1) 450
Double Precision	.20 (1) 821	.47 (2.4) 355	1.6 (8.0) 105	.68 (3.4) 245

Table III
Integer Performance
(Times are relative to A900)

	time (MIPS)			
Single Integer	1.0 (2.3)	3.6 (.65)	3.8 (.61)	4.6 (.50)
Double Integer	1.0 (1.9)	3.9 (.48)	4.2 (.44)	5.3 (.35)

Table IV
Interrupt Response Time
(Times are in microseconds)

Interrupt to Driver	57	142	146

tion execution rate will vary depending on the instruction mix, the range remains fairly consistent in many applications. Floating-point allows the A700 and A900 to excel, while heavier disc access compresses the range somewhat.

Reliability

HP customers have come to expect something extra in terms of reliability from HP products and the A-Series was designed with that in mind. Reliability, like quality, must be designed in. It cannot be added on later. One key to a reliable product is design margin, the attribute that enables a product to function properly over a wide range of environmental conditions and component variations.

To ensure sufficient design margin, a worst-case analysis was performed on each critical timing path in the A-Series CPUs. The A600 analysis was performed by hand while the A700 and A900 analyses were done using an HP-developed software package, which takes into account such parameters as power supply variation, output loading, temperature variation, and stripline characteristics of printed circuit boards to predict the operating margin of a digital circuit.

Before being made into a printed circuit board, each module of each of the A-Series Computers was analyzed by a group of engineers in a peer group design review. Each engineer in the review group was assigned the task of learning a portion of the module well enough to explain its detailed operation to the rest of the group. These review meetings have proved to be a very effective method of catching design errors early in a project.

After the printed circuit layout for each module was completed and digitized, the digitizer output was read by another HP-generated software package, which produced a list of all of the wires and connections on the board. The list was then checked against the schematic for the module as one final verification before boards were fabricated.

If components are operated at too high a temperature, even the most carefully designed circuit cannot deliver good long-term reliability. To ensure that each component would be operating well within its limits, thermocouples were used to look for potential hot spots that required additional cooling. The thermocouple data was used to calculate the junction temperature of each of the integrated circuits to ensure that no device was being overstressed.

As part of the development cycle, a number of typical A-Series system configurations were subjected to rigorous environmental tests designed to verify the integrity of the packaging, power supply, and processor electronics. Proper system operation was verified over a wide range of temperature, power line voltage and frequency, humidity, altitude, and vibration.

Care is also exercised during the manufacturing process to keep components from being damaged by electrostatic discharge (ESD). Often, a component will not be destroyed by ESD, but merely weakened, enabling it to pass production tests at the factory only to fail after a very few hours at the customer's site. To prevent this problem, we have implemented an extensive program to eliminate ESD damage to components during the manufacturing and testing process. The program includes antistatic mats, grounding straps for the workers, and antistatic conductive packaging for the transporting assemblies.

Mean time between failures (MTBF) calculations using RADC II methods predict the following MTBFs for the A-Series CPUs:

A600 (2156A) with 128K bytes of memory	10400 hours
A700 (2137A) with 128K bytes of memory	7400 hours
A900 (2139A) with 768K bytes of ECC memory	6100 hours

To date, field data on the 2156A and 2137A indicates that their MTBFs are actually 2 to 2.5 times better than the RADC prediction. At the time of this writing, the 2139A is too new and not enough field data is available on that product, but since it was designed using the same methodology and attention to detail that went into the 2156A and 2137A, there is every reason to believe that it too will give the high level of reliability that is expected from HP products.

Acknowledgments

The authors wish to thank all of the A-Series project members for their contributions in making the program successful. Their specific contributions are mentioned elsewhere in this issue. In addition, many people from other areas played important roles in the process of developing and releasing these products. Special thanks to Sara Dickinson and the A600 project team for their efforts, to Carl Synder for his assistance in managing the A-Series program, and to Nick Coping for his assistance during the development and system integration phases of RTE-A.

References

1. J. Cates, "F-Series Extends Computing Power of HP 1000 Computer Family," *Hewlett-Packard Journal*, Vol. 29, no. 14, October 1978.

2. *RTE-A.1 Performance Brief*, Hewlett-Packard Company, December 1982.

An Adaptable 1-MIPS Real-Time Computer

by David A. Fotland, Lee S. Moncton, and Leslie E. Neft

THE A700 COMPUTER is the midrange processor of the A-Series Computer family. Priced between the A600 and the A900, the A700 provides flexibility that allows it to adapt to a customer's needs. The A700 can be purchased with or without hardware floating-point and with or without error correcting memory, and it can be customized through user microprogramming. It is designed to operate on the earlier HP 1000 L-Series backplane and thus it can use the dozens of I/O cards that have been developed since the advent of the L-Series.

The A700 was the first member of the A-Series product line, and its inherent flexibility made it the development processor of that product line. The first objective of the A700 was to overcome the address space limitation of the L-Series, while surpassing L-Series performance by a factor of 3. Another objective was to leverage the hundreds of engineer-years of effort found in the RTE family of operating systems, languages, and subsystems. The A700 was intended to provide all of the functionality of the HP 1000 E-Series and F-Series Computers, including similar performance and microprogrammability, at lower cost, using the improved L-Series I/O system and an improved method for supporting large programs.

The A700 was the first HP 1000 to make use of bit-slice technology, the first to incorporate the high-performance SOS floating-point chip set (see page 17), the first to implement the dynamic mapping system for large address space access, the pioneer and the development processor for large-program support provided by code and data separation hardware and the VC+ enhancement to the RTE-A operating system (see page 26), and the first HP 1000 to be easily user-microprogrammable through the use of the microparaphraser microprogramming language. The A700 with hardware floating-point has better performance than the HP 1000 F-Series, formerly the top of the line, at only sixty percent of the cost.

New DMS Instructions

Since the L-Series did not have memory mapping, the A700 was free to define a new improved set of dynamic mapping system instructions. The HP 1000 uses 15-bit logical addresses, so a program can directly address 32K words. A map is a set of 32 map registers which map the 32 1K-word pages of logical address space to 32 physical pages. For backward compatibility with the HP 1000 M, E, and F-Series Computers, the A-Series has a similar format for a set of map registers. The number of map sets is increased from 4 to 32 for more flexibility and the physical page number field is extended to allow 24-bit physical addresses. This allows the operating system to allocate one DMS map to each I/O interface for increased I/O throughput. In addition, the operating system can use a separate DMS map for system available memory. The user program can be allocated two maps, one for code and one for data.

The DMS instruction set includes instructions for loading and storing maps and a new set of cross-map instructions that allow access to memory through three maps, the current execute map and two others called data1 and data2. The cross-map instructions include load, store, and move words.

CDS Instructions

The biggest architectural change was the introduction of code and data separation. Separation of code and data allows programs with large code to be handled easily and transparently without using overlays. It also provides better protection, recursion, and reentrancy. It allows code to be shared between several processes to conserve main memory. CDS was recognized early as being a desirable goal. The problem was to provide it without a major change in the existing instruction set, which would require a lot of extra hardware. The old instruction set is faithfully executed for backward compatibility. Minimal changes from the old instruction set also mean minimal changes to the existing compilers.

ECC Memory

No matter how good the design or how reliable the parts used, machines will fail from time to time. The single part in the A700 that contributes most to the failure rate is the dynamic RAM chip used in the memory. This is because these parts have a high soft failure rate compared to other logic parts and because there are many more RAM chips in the machine than any other kind of chip. An A700 with 4M bytes of memory contains 544 64K-bit dynamic RAM chips.

Error correcting memory provides higher reliability for those customers who need it by correcting single-bit errors and detecting double-bit errors. Soft errors are the most common failure of memory systems. In systems with over 512K bytes of memory the soft error rate is about one per year. Customers who need higher reliability than this can use error correcting memory.

Error correcting memory is easy to use on the A700 because it uses the same memory controller and has the same performance as parity memory. A customer can upgrade to error correction without throwing out the current controller. Error correcting and parity memory can be mixed in the same system. For example, one might want to protect the operating system and some critical applications from single-bit errors, and use less-expensive parity memory for the rest of the system.

If there are no errors, the error correcting memory runs at the same speed as parity memory. When a single-bit error is detected, the system is frozen for 200 ns while the data is corrected, and is then allowed to continue with good data.

A Hamming code is used to detect and correct memory

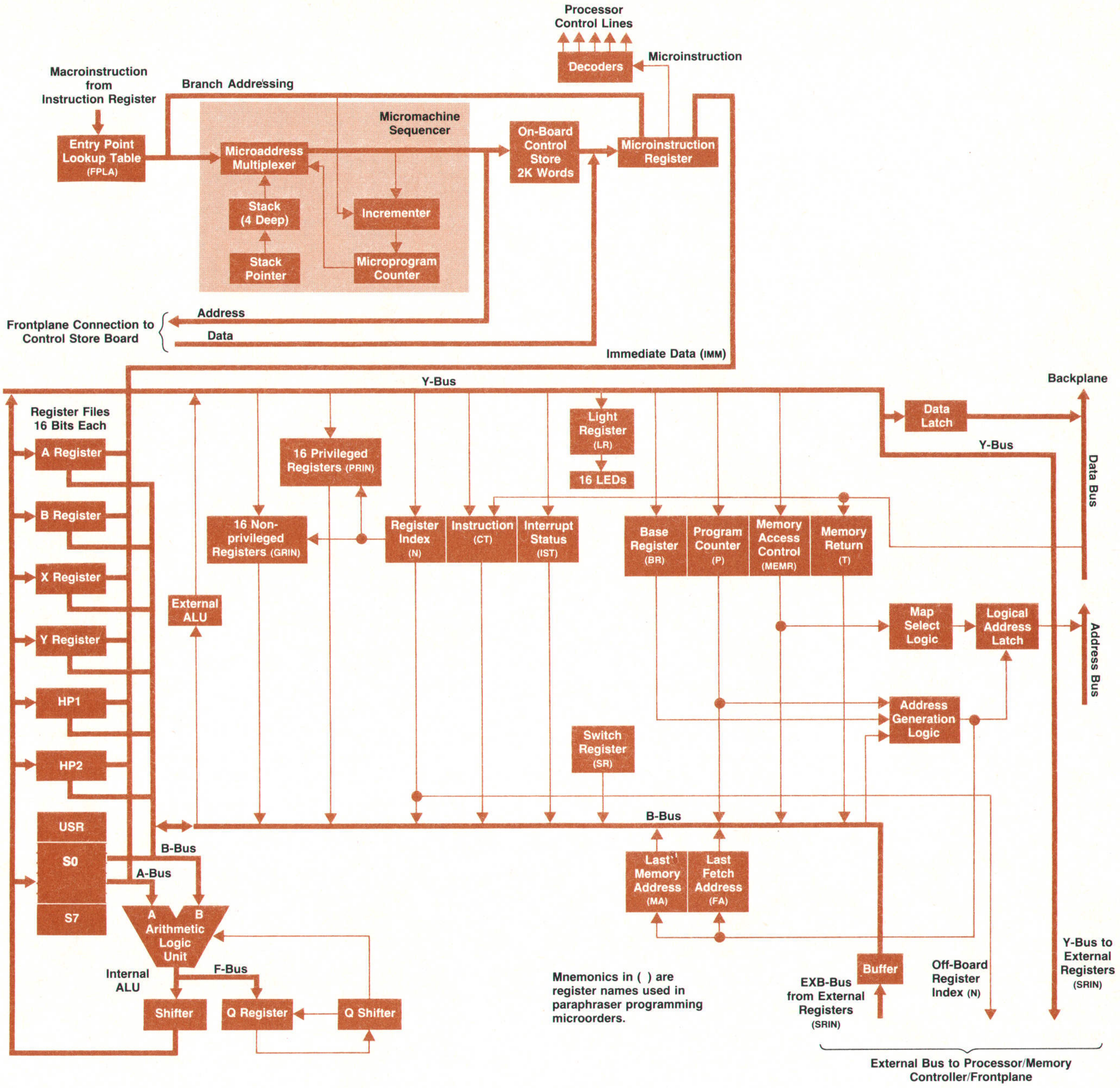


Fig. 1. *Block diagram of the A700 processor. The regularity of data flow makes microprogramming easier.*

errors in the A700. With the addition of 6 bits per 16-bit word all single-bit errors are correctable and all double-bit errors are detectable. Whenever the memory is read the parity bits are combined with the data to provide a 6-bit syndrome, which identifies the type of error and the bit number if it is a single-bit error. This syndrome is stored in an error logging RAM in the memory controller. There is one syndrome location in the RAM for each row of 64K RAM chips. By reading the error logging RAM, it is possible to determine the last chip that had an error in each row of RAMs. This information can be used to identify failed chips before they cause a problem in the system.

Performance and Ease of Microprogramming

The two major objectives in the design of the A700 micromachine were to provide fast execution of the HP 1000 instruction set, and to allow user microprogramming, which can give a substantial boost in performance to many customer applications. To accomplish these objectives, we designed an architecture that is simple and straightforward, a microinstruction set that is flexible and provides a lot of capability, and tools that aid microcode development.

We chose to base the micromachine on the 2900 bit-slice processor family, specifically the 2903 bit-slice processor and the 2911 bit-slice sequencer. These parts provide many intrinsic features, yet allow us to use our own microarchitecture. We use the 2903's numerous arithmetic and logical operations, but use our own instruction set for accessing them.

The Microarchitecture

The key features of the microarchitecture are capability and regularity. The capability allows fast performance; the regularity makes it easy to microprogram.

To optimize performance for the A700 instruction set, we used profile data which told us how often each instruction or class of instructions was executed for different types of programs. For those instructions executed most often, we provide special hardware to shorten their execution times. We avoided the common pitfall of making other instructions inordinately slow to optimize the most heavily used instructions. Even an instruction that occurs only 1% of the time can impact performance if it's extremely slow. However, it is not practical to provide special hardware for all instructions. Since much of our instruction execution time is spent in decoding the instruction (that is, determining what the instruction is), we designed the micromachine to decode all instructions quickly. By increasing the performance for instructions that had slow execution times in previous HP 1000s, we allow more flexibility in the design of applications that use assembly language (such as compilers).

Optimizing the design for user microprogramming was more difficult. To accomplish this, we looked at the types of functions that were likely to be microcoded (for example, computation and bit manipulation) and provided sufficient hardware to support these types of operations. Many of the operations were all ready provided in the 2903; we needed only to design the hooks to access them.

A block diagram of the processor is shown in Fig. 1. The regularity of the flow of data through the machine is one of the features that contributes to the ease of microprogramming. At the start of a cycle, data is enabled from the appropriate registers onto the operand buses: the A-bus and the B-bus. In one cycle, data is input to the ALU and passed to a shifter. The resulting data is available on the Y-bus at the end of the cycle. The Y-bus result can then be loaded into a register or written to main memory. In the following cycle, conditions generated by the operation just described, such as carry out of the ALU, can be tested. The user does not need to learn complex rules for the relationship between buses, registers, and conditions, since they are the same throughout the machine. Registers are updated at the end of the cycle and are available as operands for the next cycle. The conditions that are tested are the conditions that were generated during the previous cycle.

The flow of control in the micromachine is similar to high-level languages such as BASIC or FORTRAN. Jump (goto) or jump to subroutine (call) instructions are used to transfer control to nonsequential locations in micromemory.

The key to the flexibility and performance of the A700 micromachine is the microinstruction set (called microoperations) and the microinstruction format. The width of the microinstruction word was an important design decision. A longer microword means more operations can be done in one microinstruction. This does not directly translate into an increase in performance, since certain operations need to be done sequentially, such as adding two numbers and then checking for overflow. We were also developing writable control store (WCS) and PROM control store (PCS), and a shorter microword makes these boards less expensive and allows more words of micromemory on each control store board. By careful encoding and overlapping of fields, we were able to use a 32-bit microinstruction word that allows several operations to be done in one cycle (see Fig. 2). Fewer microoperations are available in a jump instruction than in an instruction that does not use a jump. However, in every microinstruction, one can perform an arithmetic or logical operation with the contents of two registers and store the result in any register in the machine. For example, one can add the contents of two registers, perform a shift on the result, and then jump to another location in micromemory, all in one cycle. To test this design decision, most of the base instruction set was microcoded before the processor hardware design was solidified.

The decoding scheme for microinstructions ensures that no combination of codable operations can damage the processor hardware (such as enabling two registers onto the same bus). Any other illegal combination of operations is detected by the microparaphraser. Thus the microprogrammer need only remember the basic relationships for data in the machine, the microinstruction formats, and a few special rules concerning interaction with main memory, I/O, and the mapping system.

A hardware timeout feature provides some protection

<table>
<tr><th>Bit</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Word Type 1</td><td colspan="5">OP1</td><td colspan="4">ABUS</td><td colspan="5">SP0</td><td colspan="4">SP2</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 2</td><td colspan="5">OP2</td><td colspan="4">ABUS</td><td colspan="5">SP0</td><td colspan="4">CNDX</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 3</td><td colspan="4">OP3</td><td colspan="6">ADRS</td><td colspan="4">SP1</td><td colspan="4">CNDX</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 4</td><td colspan="4">OP4</td><td colspan="6">ADRS</td><td colspan="4">SP1</td><td colspan="4">SP2</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 5</td><td colspan="4">OP5</td><td colspan="14">ADRL (Long Branch Address)</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 6</td><td colspan="2">OP6</td><td colspan="16">DAT (Immediate Data)</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 1S</td><td colspan="5">OP1</td><td colspan="4">ABUS</td><td colspan="5">ALUS*</td><td colspan="4">SP2</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 2S</td><td colspan="5">OP2</td><td colspan="4">ABUS</td><td colspan="5">ALUS*</td><td colspan="4">CNDX</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 3S</td><td colspan="4">OP3</td><td colspan="6">ADRS</td><td colspan="4">ALUS*</td><td colspan="4">CNDX</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 4S</td><td colspan="4">OP4</td><td colspan="6">ADRS</td><td colspan="4">ALUS*</td><td colspan="4">SP2</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>Word Type 5S</td><td colspan="4">OP5</td><td colspan="14">ADRL
(Long Jump Table Address)</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
</table>

*Special microorder in ALUS field when ALU field is coded SPEC.

Fig. 2. *A700 microinstruction formats.*

from errant user microprograms. This protects the system from a user microprogram's hanging and not allowing system interrupts. If a system interrupt goes unserviced for more than 10 ms, then the microprogram is aborted and control is returned to the operating system.

The Microparaphraser

One of the most important tools for the development of microcode on the A700 is the microparaphraser, MPARA. The microparaphraser was originally developed for the A700 and later adapted for the A900. All of the base instruction set, the floating-point instructions, and the microcode diagnostics were written using MPARA.

MPARA allows the user to write microcode in free format, Pascal-like constructs. It then translates them into microcode. The following example shows a microroutine that finds the maximum of two numbers.

```
$origin 0x3000$                    *Start routine at 3000 hex.
MAX:                               *A = arg1, B = arg2.
  s1: = a − b;                     *Compare arguments.
  if alov then goto ChangeSense;   *If arithmetic overflow then
                                   *   switch sense of compare.
  if y15 then stor,                *If result was negative, then
    a: = b;                        *   return arg2 (else arg1).
  rtn;                             *Return, max is in A.
ChangeSense:                       *
  if not y15 then stor,            *If result was positive then
    a: = b;                        *   return arg2 (else arg1).
  rtn;                             *Return, max is in A.
```

A phrase can be an arithmetic expression, such as:

s1: = a − b

or a conditional operation:

if alov then goto ChangeSense

or an individual microorder:

rtn

Phrases, which are delimited by commas, can be strung together in any order as long as they represent a legal microinstruction. A sentence is a complete microinstruction and is terminated by a semicolon. This type of format makes microcode easy to read and write and also allows plenty of room for comments (especially important for microprogramming).

Writable and PROM Control Store

Two types of user control store boards were designed to facilitate user microprogramming and to aid the design team in the development of the A700 processor. The PROM control store board (PCS) provides inexpensive storage for up to 8K of microcode. The writable control store board (WCS) allows microcode to be downloaded and altered dynamically during the debugging process.

Since the control store PROMs used in the A700 were initially expensive and not reprogrammable, it was not practical to change the base set PROMs on our breadboards whenever bugs were found during development. Therefore, we made use of a special feature of the A700 processor which allows us to overlay the base instruction set on the processor board with the microcode on the WCS board. Using this "mindswap" feature, we could download from the operating system the latest revision of microcode to the WCS, enable it, and continue to run in the operating system with the new microcode.

Hardware Floating-Point Board

The hardware floating-point board for the A700 Computer is really two boards in one. A portion of the board is designed to be a PROM control store board that can hold 4K words of microcode PROM. The first 2K is used to hold the microcode that executes the floating-point-dependent instructions in the HP 1000 instruction set, that is, single- and double-precision floating-point, the scientific instruction set (SIS), and the vector instruction set (VIS). The second 2K is available for user-microcoded routines.

The main area of the board contains the floating-point hardware, which is used during the execution of floating-point instructions. This portion of the board is designed around the computational power of three CMOS/SOS floating-point chips (see page 17). These chips are designed specifically to perform HP 1000 integer and floating-point arithmetic.

The design of the floating-point board evolved through a number of iterations. These iterations required designing a preliminary version of the hardware, then writing the microcode for the floating-point, SIS, and VIS instructions using the features in that version of the hardware. After determining the execution time of the instructions and evaluating the good and bad features offered by that particular revision of the hardware, another pass at the hardware was made. This cycle was repeated until all the hardware capabilities needed to optimize the execution of the instructions were included in the design.

One of the more significant improvements in the design of the hardware was the addition of an on-board arithmetic constant ROM (ACR), which contains single- and double-precision floating-point constants. The ACR has over 100 constants which are used during the execution of the SIS instructions and for the self-test diagnostics. The ACR is accessed indirectly through an address pointer which is automatically incremented after each use, allowing the address pointer to be set to the front of a list of constants which are to be required in a known order. The main advantage of using the ACR is in eliminating the overhead required to pass fixed-value operands (e.g., π, $\pi/4$, ln2, 1/2, etc.) from the micromachine to the floating-point board. This savings is most significant in the evaluation of the SIS instructions where many constants are required in the polynomial approximation of the functions. (The evaluation of an arctangent can require up to seven constants.)

Another important feature is the set of four on-board accumulators, each capable of holding a single-precision (32-bit) or double-precision (64-bit) floating-point number. These accumulators eliminate the necessity of repeatedly passing the same floating-point number to the board when it is required more than once during the evaluation of an expression such as $(x+y)/(x-y)$. The accumulators also increase performance by temporarily saving the intermediate results of a polynomial expression. The ability to save intermediate results on-board instead of unloading and later reloading the value significantly reduces the execution time of the SIS and VIS instructions.

Another feature included in the design of the floating-

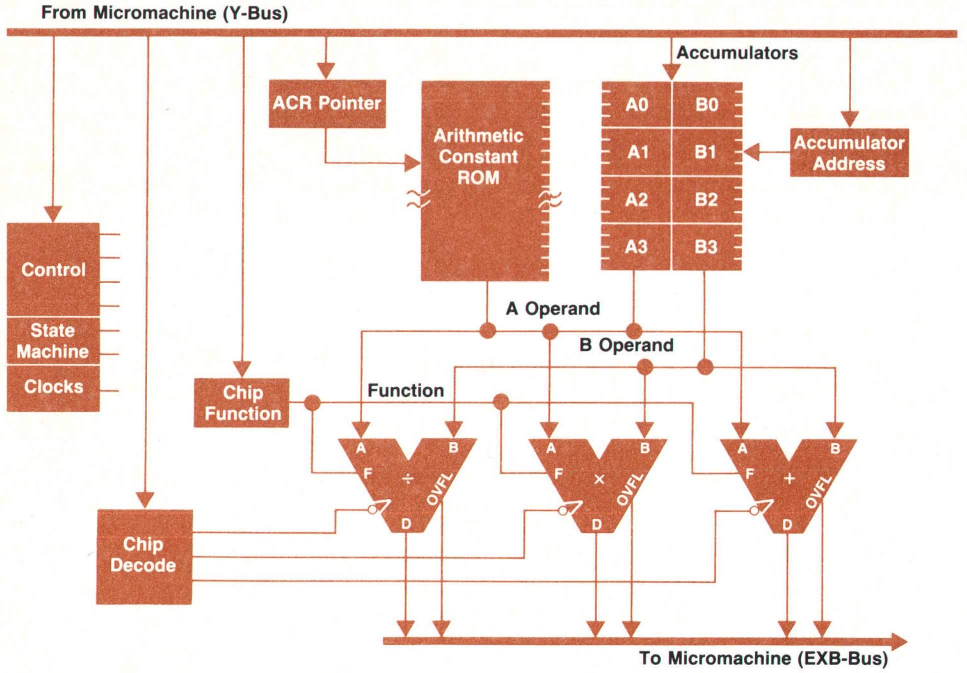


Fig. 3. *Internal data paths on the A700 hardware floating-point board.*

point board is the ability to clock operands into the floating-point chips at twice the micromachine clock frequency. Because the input clock rate of the floating-point chips is twice the speed of the processor microcycle time, operations that do not require operands from the micromachine can clock their operands into the chips twice per microcycle. All transfers between the floating-point chips and the accumulators or ACR can be performed at twice the microcycle rate. This reduces the time spent in transferring the operands of intermediate results to the chips, and results in faster execution times.

To make the addition of the optional floating-point board to an A700 system easier, the floating-point microcode will overlay the portion of the microcode on the processor card that decodes floating-point-dependent macroinstructions. The floating-point board can then intercept and redirect the execution of the floating-point instructions to the microcoded routines contained on the floating-point board. Thus no changes need be made to the A700 base set microcode when the hardware floating-point board is added to the system, making customer upgrades simpler.

Fig. 3 shows a block diagram of the internal data paths on the floating-point board. Surrounding the three math chips are state machines, control logic, four accumulators, the ACR, and clocking circuitry. The hardware floating-point board is designed to be an optional coprocessor board for the A700 CPU. To the A700 micromachine, the floating-point board looks like four 16-bit registers. These four registers are readable and writable by the micromachine and appear as four locations in an external register file that are dedicated specifically for use with the floating-point board.

To use the floating-point board, a control word is first passed to one of the four dedicated registers. This control word contains such information as which chip to use, what operation to perform with that chip, where the operands are to come from (an accumulator, the ACR, or the micromachine), and where the result is to be stored. Once a control word is passed to the floating-point board, the necessary operands are then passed to the board through the dedicated register locations.

As pairs of operands of equal significance are available on the floating-point board, the control logic clocks that pair of operands into the selected floating-point chip. When all operands have been clocked into the chip and the required propagation delay has passed, the results of the operation are available to be read by the micromachine from one of the dedicated register locations. An overflow/underflow signal is also available to the micromachine at one of the registers.

The microcode written for the floating-point board takes full advantage of the independence of the micromachine and the floating-point board. For example, during the time when an operation is being performed by the floating-point board, the micromachine can be resolving the indirect addresses for the result, fetching the next operand from memory, or doing arithmetic/logical operations of its own. This independence is used to full advantage during the execution of the VIS instructions. Each vector instruction was written to minimize the time to compute one element of the vector. This microcode generally has six operations to perform for each element of the vector:

- Execute the required floating-point operation
- Fetch the next operands from memory
- Return the results from the previous computation to memory
- Update all memory address pointers by the correct increment
- Check the loop count for the end of the instruction
- Check for system interrupts.

For each VIS instruction an inner loop was written that minimizes the time required to do this work. Because this code is optimized for execution speed and large memory bandwidth rather than ease of entry and exit, it is necessary to precede this code by a section of front-end code that synchronizes with the inner-loop code. The results of this work enable the VIS instructions to execute very close to memory speed for single-precision operations, and at memory speed for double-precision operations. This yields an improvement of 1½ to 3 times compared to the execution speeds on the HP 1000 F-Series Computer.

Acknowledgments

The authors wish to acknowledge the other members of the design team: Carolyn Guidry, Steve Kusmer, Earl Kieser, Tom Harms, and Bill Chesk, and especially our project manager Don Cross for his guidance and support. We'd like to thank our production engineer Chris Malachowsky and our product manager Scott Spielman for believing in us. Wayne Chin, Colleen Obergfell, and Dan Shewey also contributed to the success of the project.

Designing a Low-Cost 3-MIPS Computer

by Donald A. Williamson, Steven C. Steps, and Bruce A. Thompson

THE A900 COMPUTER provides approximately three times the performance of any previous HP 1000 Computer, while maintaining full software compatiblity with the other HP 1000 A-Series Computers. The cost of the A900 is noticeably lower than that of many computers of similar performance, giving it an excellent price/performance ratio. To achieve this price/performance ratio, the performance was optimized, but not by adding a lot of additional parts and complexity.

To increase the performance of a computer, the amount of work done in each machine cycle can be increased and the cycle time can be decreased. This can be accomplished by widening all of the paths to 32 or 64 bits and using a very fast technology such as emitter-coupled logic (ECL). This approach was not used in the A900 because it leads to a very high-cost computer. Instead, care was taken to add cost only where it was justified by a significant performance gain, and to minimize cost elsewhere.

Although the A900 is completely software compatible with the other members of the A-Series, it has a somewhat different hardware structure. Fig. 1 shows a basic block diagram for both the A600 and A700 Computers, while Fig. 2 shows the basic block diagram for the A900. All three machines are microprogrammed and therefore have data paths that are controlled by a microcode sequencer. The A600 and A700 have a common memory-I/O bus used by both the CPU and the I/O system to access memory. The I/O bus in the A900 is electrically and mechanically the same bus as the memory-I/O bus in the A600 and A700. However, the A900 does not fetch instructions or data across this bus. The CPU uses the bus only to communicate with the I/O system. This structure helps achieve the main goal for the A900: high performance without the normally associated high price.

Sequencer

The A900 is a microprogrammed computer. This means that each machine language instruction (macroinstruction) is emulated by a sequence of microinstructions. The format of the macroinstructions is fixed by compatibility with other HP 1000 Computers, but the format of the microinstructions is tailored to the hardware used to implement the A900.

Each microinstruction is 48 bits wide, allowing several operations to be specified in parallel. For instance, a conditional jump, an ALU operation, and a memory operation can all be coded in a single microinstruction. Thus the microprogrammer can test a condition, perform a calculation, and start reading the next operand, all simultaneously.

The microcode sequencer controls the sequence of microinstructions that are used to emulate each macroinstruction. A block diagram of the sequencer is shown in Fig. 3.

The sequencer selects a microaddress from the microprogram counter, the microsubroutine stack, or a field of the current microinstruction. The control store takes this address and generates a microinstruction which is loaded into the microinstruction register at the end of the microcycle. Usually, one of the critical timing paths in a microprogrammed machine is the decision point between a conditional branch and sequential execution. In the A900, the "condition met" signal controls a multiplexer at the output of the address selection logic. Thus the signal can arrive later in the cycle without becoming a critical timing path. This helps in reducing the cycle time of the A900 micromachine to 133 ns.

Each time a new macroinstruction is fetched, the se-

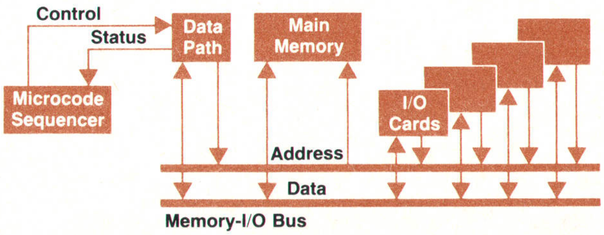


Fig. 1. *A basic block diagram for the A600 and A700 Computers, showing the common memory-I/O bus.*

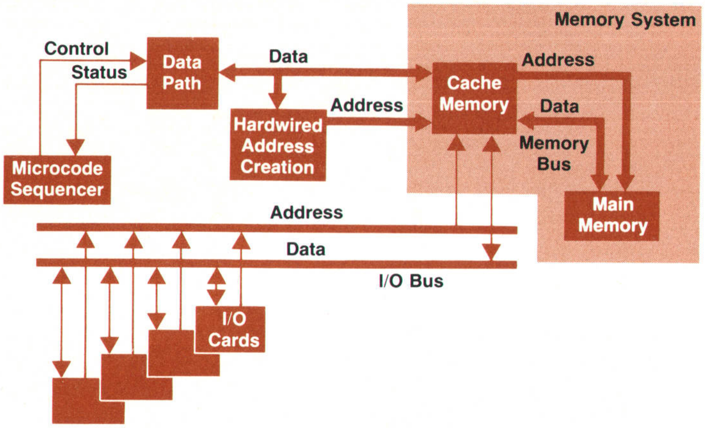


Fig. 2. *A basic block diagram for the A900 Computer, showing the separate memory and I/O buses.*

quencer must determine the sequence of microinstructions needed to emulate it. Traditionally this is done by sending the macroinstruction to a jump table (AJT) which produces the address of the first microinstruction of the emulation sequence. Since this address must then be sent to control store to produce the actual microinstruction, the decoding process takes two entire cycles. The A900 adds another type of jump table (IJT) which receives the macroinstruction and produces a microinstruction instead of a microaddress. The IJT is built with programmable logic arrays (PLAs), and can produce microinstructions for the most common macroinstructions. While the IJT is guessing the first microinstruction of the emulation sequence, the AJT is looking up the address of the second microinstruction. While the first microinstruction is executing, the control store is looking up the second microinstruction. The result is that it takes only one cycle instead of two to determine the first microinstruction of the sequence. This is a significant savings, since many important macroinstructions take only two cycles to execute.

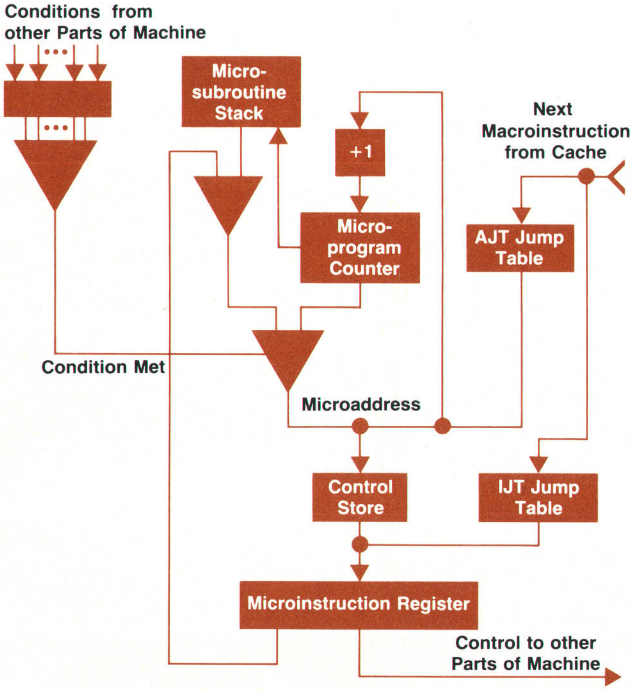

Conditions from other Parts of Machine
Micro-subroutine Stack
+1
Next Macroinstruction from Cache
Micro-program Counter
AJT Jump Table
Condition Met
Microaddress
Control Store
IJT Jump Table
Microinstruction Register
Control to other Parts of Machine

Fig. 3. *The sequencer controls the sequence of microinstructions to emulate each macroinstruction.*

Pipelined Data Path

The data path is where much of the data manipulation required by the HP 1000 instruction set is done. A block diagram of the A900 data path is shown in Fig. 4. Operands from the register file, cache memory, or other parts of the machine are operated on by the ALU, the shifter, or the floating-point unit, and the result is stored in the register file, the cache, or some other machine register. Accessing the operands, performing the operation, and storing the result take longer than the 133 ns available in an A900 microcycle. Therefore, the data path is split into two pieces by a pair of pipeline registers.

In the first cycle of a microinstruction, the operands are read and loaded into the pipeline registers. During the second cycle, the operation is performed using the values in the pipeline registers, and the result is stored. During this second cycle, the operands for the next microinstruction are being read. Even though it takes two cycles to complete a microinstruction, the parallelism allowed by the pipeline registers lets a new microinstruction start every cycle (see Fig. 5).

A side effect is that the result of a microinstruction started in cycle 1 is not stored until the end of cycle 2 and therefore cannot be used until cycle 3. The register file is paralleled by a pair of latches which can be used as accumulators. The latches become transparent if they are written at the same time that they are read. The result from a cycle 1 microinstruction can be written into one of the pipeline registers at the end of cycle 2 by storing it to an accumulator. In other words, if a result is stored to one of the accumulators it can be used immediately instead of one cycle later.

The data path is designed to maximize the amount of work that can be done by a single microinstruction. For instance, the register file is double-ported, allowing access to two operands at a time. The shifter can logically shift the 32 bits of data in the pipeline registers by 0 to 15 bits and produce a 16-bit result. Using this barrel shifter, any type of shift—arithmetic, logical, or circular—can be accomplished in a small number of cycles.

There are actually two independent paths within the

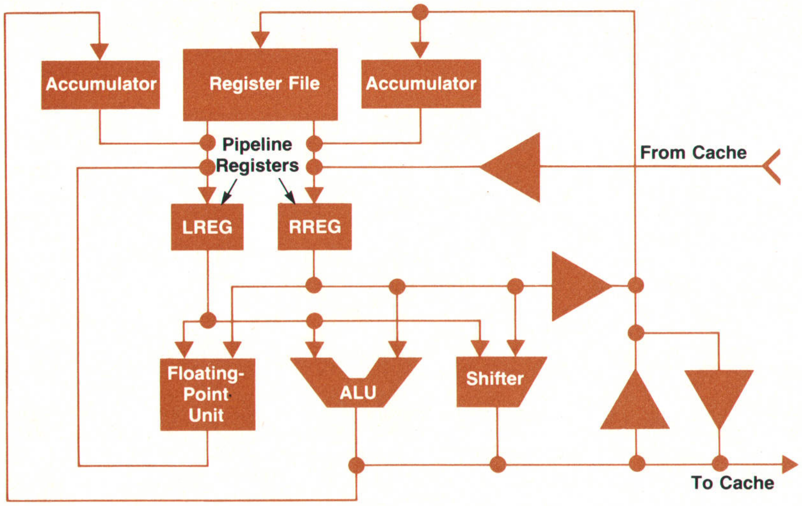


Fig. 4. *The A900 data path. The two pipeline registers make it possible to start an operation while the preceding operation is still in progress. The accumulators make it possible to use the result of a microcycle immediately instead of one microcycle later.*

data path. One of these is the path from the pipeline registers, through the ALU or shifter, and out to the cache or other register. The other is a path from the right pipeline register through a buffer to the register file. The paths are often interconnected by a bidirectional buffer pair. This allows simple operations such as storing the ALU output in the register file or passing the right-side operand to the cache. The paths can also be used independently, allowing two data transfers in a single cycle. For example, data from the cache can be passed to the register file using the right pipeline register, while data from the register file is sent to the cache via the left pipeline register and the shifter.

The floating-point unit dramatically reduces the number of cycles needed to perform floating-point operations. It uses the three LSI floating-point chips described in another article in this issue. Because the chips are tightly coupled into the data path, the 32 bits of data from the pipeline registers can be loaded into the chips each cycle. After a full set of operands is loaded, the chips produce the result, which can be unloaded via one of the pipeline registers several cycles later. The chips are also used to perform integer multiplication and division.

Memory System

The memory system includes three pieces. The first piece is the address creation logic. This logic generates the appropriate addresses for the cache, which might be used for fetching an instruction or reading the data needed for an instruction. The second piece is the cache memory. A cache memory is a very high-speed memory which keeps only a subset of main memory. The goal of a cache memory is that of any memory hierarchy: achieve performance close to that of the fastest memory (cache memory, in this case) with a cost closest to that of the slowest memory (main memory, in this case). The third piece of the memory system is the main memory itself.

One of the performance-critical parts of most computers is the path from instruction fetch to operand ready. This includes the decoding of the instruction, the extraction and merging of the appropriate fields, and the starting of the memory request. Typically, this is done by the micromachine using the main ALU. However, in the A900, almost all of this is done in hardware.

HP 1000 instructions that reference memory can be of two types. The memory reference group (MRG) instructions are the most commonly executed instructions in the HP 1000 instruction set. These instructions have a four-bit opcode, a 10-bit offset, a zero/current page bit, and a direct/indirect bit. The other type of memory reference instructions has a 16-bit opcode followed by a 15-bit address with a direct/indirect bit.

If the instruction is of the first type (MRG), the A900 hardware creates the appropriate address from the current program counter page value and the offset from the current instruction. A memory reference with this address is then started by the hardware. If the address is indirect, the address creation hardware can freeze the CPU until a direct address is read, since multiple levels of indirect addressing are allowed. If the instruction is of the non-MRG type, then the address creation hardware increments the program counter and starts a memory reference.

By the time the microcode sequencer can vector to the

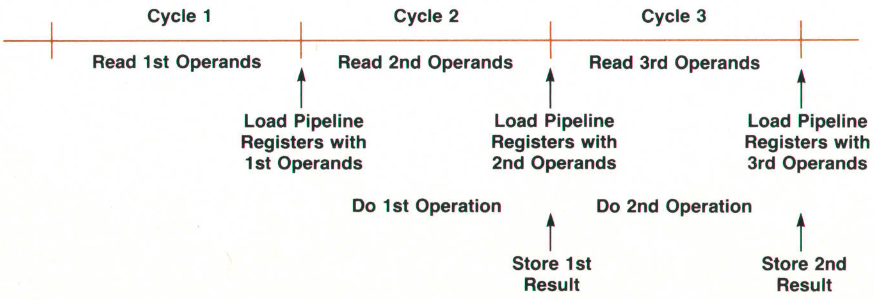


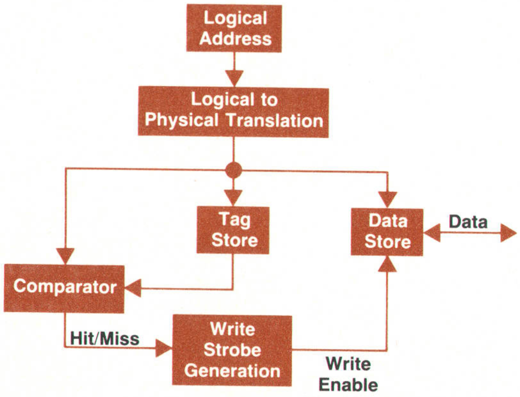
Fig. 5. *Timing in the pipelined A900 Computer. Although it takes two microcycles to complete a microinstruction, a new microinstruction is started every microcycle.*

correct microcode for the instruction, the memory operand data can be ready. This allows for a much shorter sequence of microinstructions for each machine language instruction. In fact, an instruction whose operand is another instruction (e.g., the jump instruction, JMP) can be executed in just one microcycle without the added cost and complexity of machine language instruction pipelining.

Cache Memory

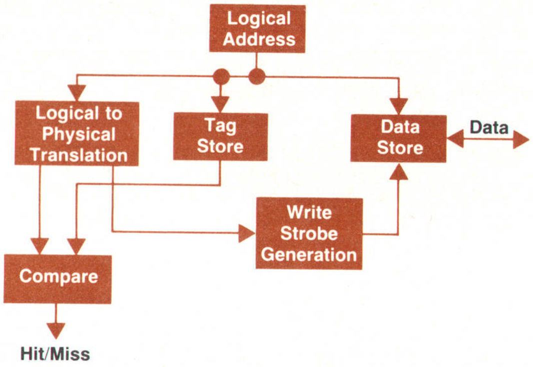
The cache memory of the A900 has a data access time of 65 ns and a cycle time of 133 ns. This allows a memory reference every microcycle, thus allowing the computer to get more done per cycle. Also, since the access time of the cache is many times faster than that of main memory, the processor cycle time can be much shorter than that of a noncache system.

A normal cache structure is shown in Fig. 6. A cache write cycle cannot begin until the appropriate address is known to be in the cache. This means the minimum cycle time for a cache write is the time for valid address, plus the time to translate logical to physical address, plus the time to access the tag RAM, plus the time to compare the tag address to the desired address (hit/miss), plus the time to generate a write strobe (or enable), plus the time for which the write strobe must exist. As shown in Fig. 6, this requires a minimum write cycle time of 144 ns, assuming proper clock edges and no skew. Fig. 7 shows the cache structure used in the A900. Note the parallel nature of the logical-to-physical translation, tag store access, and data store access. The ideal minimum cycle time for this structure is 79 ns. This fits nicely in a 133-ns cycle with plenty of room for obtaining necessary clock edges and allowing for skew. This scheme for handling writes allows the A900 cache to do a write every cycle and still have a very short cache cycle time.



9 ns	Logical Address Valid
20 ns	Logical to Physical Translation
45 ns	Tag Store Access
20 ns	Tag Compare
10 ns	Write Strobe Generation
40 ns	Write Strobe Width
144 ns	

Fig. 6. *In a normal cache memory like this one, a cache write cycle cannot begin until the appropriate address is known to be in the cache. The minimum write cycle time shown assumes proper clock edges and no skew.*



9 ns	Logical Address Valid
20 ns	Logical to Physical Translation
10 ns	Write Strobe Generation
40 ns	Write Strobe Width
79 ns	

Fig. 7. *The A900 cache memory has a parallel structure, allowing faster operation than a normal cache. The minimum write cycle time fits nicely into the 133-ns cycle with plenty of time for obtaining necessary clock edges and allowing for skew.*

In a typical cache memory system, direct memory access (DMA) from I/O cards comes directly into main memory. This means that main memory has to be double-ported, that is, it has to arbitrate between two address/data sources before starting the memory cycle. In the A900, DMA comes directly into the cache. This eases the problem of cache data consistency (i.e., keeping cache data and main memory data for the same address consistent). It also keeps the memory system simpler since main memory can now be single-ported. Another benefit is that it allows very tight coupling of the cache and main memory. On the A900, every access to the cache also causes the translated physical address to be sent directly to main memory. Since there is no arbitration, this address can be set up on the memory RAMs on the memory array cards. If a fault* is detected, almost no time is lost in starting the main memory access. This gives an unusually fast fault handling cycle. Instead of the typical 6 to 12 clocks, the A900 can handle a fault in only three additional cycles. Since the effective access time of a cache memory system is described by the equation:

$$\text{Effective access time} = \text{Hit cycle time} \times \text{hit ratio} + \text{fault cycle time} \times \text{miss ratio}$$

decreasing the fault handling cycle can be just as effective as increasing the hit ratio, which requires the expense of a larger cache.

In addition, improving the fault handling time vastly improves system performance during periods when caches are not very effective. A typical example is process switch-

*In this section, a fault means a miss, that is, the requested data is not in the cache and must be obtained from main memory. This kind of fault is not a memory error or failure, and the handling of this kind of fault should not be confused with memory error detection and correction.

ing, which occurs whenever an interrupt occurs. Thus, optimizing the A900's fault handling time improves its real-time capabilities.

To achieve its excellent fault handling time, the A900 uses a 32-bit path to and from main memory. Since the cache block size is 32 bits, a fault can be handled in just one read or write to main memory. This is an important feature in reducing the fault handling cycle time, thus giving more time to the processor.

Technology

To build a high-performance computer with a very low price, one needs not only to be clever in the design of the computer, but also to incorporate new technologies. In the design of the A900, several new technologies are used. Most of the logic in the computer uses a new fast, low-power Schottky logic family that not only provides a fast cycle time, but does so without the added heat, power supply, and cost penalities of conventional high-speed Schottky logic.

Programmable logic arrays (PLAs) are also heavily used in the A900. Small, 20-pin versions became available just in time for use on the A900. These PLAs allow most of the state machines and decoding logic to be integrated into a very small number of devices that are very easy to alter. Most of the other random logic also uses PLAs. This made the debugging of the A900 much faster than conventional designs, so the computer could be shipped much sooner.

As mentioned earlier, the floating-point operations in the A900 are done by a set of SOS (silicon on sapphire) LSI chips. By integrating the performance-sensitive part of the computer on very low-power LSI chips, the A900's power, size, and cost were minimized while achieving a very high level of performance.

Pipelining and User Microprogramming

The A900 is the first HP 1000 Computer to use pipelining in its data path to improve performance. Pipelining affects the way in which algorithms and microcode sequences are designed. A user writing an isolated line of microcode does not need to know when the different sections of the microcode line will be executed. However, in a complete microprogram, various effects of the pipelining will show up in the register transactions, conditional status checks, memory operations, etc.

All of the microprogramming examples shown here are written in the A900 microprogramming language and can be compiled to executable microcode using the A900 microparaphraser. The A900 microprogramming language looks very much like a higher-level language using free-field notation and formats. It allows the user to generate microcode without concern for the actual format of a microword. With the microparaphraser and its associated tools in the A900 microprogramming package, a programmer can quickly generate a microprogram to enhance the performance of an application with a minimum of effort. Performance enhancements of 3 to 20 times are typical.

The pipelining of the micromachine data path has the largest impact on user microprogramming, essentially causing all micromachine data transactions to take two cycles to complete. As shown in Fig. 4, two pipeline registers, LREG and RREG, are placed in the data path at the inputs of the ALU to split data path operations into two phases.

In the first phase, data flows from the dual-ported register file (or other inputs) and is clocked into LREG and RREG. In the second phase, data is taken from the pipeline registers, flows through the ALU, and is finally stored back into the register file or other write-only registers.

The effect on user microcode is that registers stored on one cycle are not updated until two microcycles later. As a result, the microinstruction sequence

```
r5:=r3;                    * Microcycle 1. Copy r3 to r5.
r3:=r5;                    * Microcycle 2. Copy old r5 to r3.
```

will swap the contents of registers r3 and r5 instead of simply copying r3 to r5 as one might expect. To have the above code sequence copy r3 to r5 you would have to add a dead cycle to allow the pipe to empty and have r5 really reflect the value of r3 before copying it back. The microinstruction sequence below will end with r5 containing the same value as r3.

```
r5:=r3;                    *Microcycle 1. copy r3 to r5.
nop;                       *Microcycle 2. This is a dead
                           *cycle to allow r5 to get the updated
                           *value of r3.
r3:=r5;                    *Microcycle 3. This does nothing
                           *important since r5 already
                           *contains the same value as r3.
```

Two special registers do exist in the A900 micromachine (the accumulators) that will reflect updated values on the very next microcycle after they are stored to. These registers are used when data must be chained through several ALU operations.

Because of data path pipelining, condition codes based on the output of the ALU will not become valid for two cycles. The microcode sequence below shows an example of testing a condition generated by the ALU.

```
nop:=0;                    *Microcycle 1. Send 0 through the
                           *ALU to test it.
nop;                       *Microcycle 2. We must wait a cycle
                           *for conditions from the ALU to
                           *become valid. Normally, an algo-
                           *rithm would be designed in such a
                           *way that the micromachine will be
                           *performing some other task here.
If TZ then go to Zero;     *Microcycle 3. This is a test for
                           *a zero output from the ALU.
                           *TZ tests for zero so this
                           *line will jump.
```

Other sections of the A900 micromachine are pipelined besides the data path. One of these areas is the memory address creation logic, which is essentially another data path. Because the memory address creation logic is pipelined, microorders that work with this logic affect either the current or the following instruction. Actions can be initiated on one microcycle and then modified on the next microcycle. An example of using these microorders is shown by the following sequence.

```
m1<=m1+1;              *Microcycle 1. Increment memory
                       *address pointer m1.
ninc;                  *Microcycle 2. Stop the increment-
                       *ing of address register m1.
                       *Register m1 will remain
                       *unchanged.
```

Another feature of the A900 micromachine that is valuable for making algorithms execute efficiently is the ability to store data path outputs to multiple destinations at once. Dual microstore fields let the programmer create an expression such as:

```
m1<<r5:=r4+1;          *Increment register
                       *r4 and store it to
                       *both r5 and memory
                       *address pointer m1.
```

Algorithms designed for a pipelined machine must be designed carefully to make use of every micromachine cycle. Algorithms are most efficient if they can be broken down into different sets of interdependent steps. When writing the code, these sets can be combined and intertwined so that the dependent steps of each process are separated by the number of steps in the pipe, in this case two cycles. In the A900 micromachine, dependent steps in a process can also be performed one after another by using special registers (accumulators) that bypass the data path pipe.

An algorithm that contains many decision points or conditional branches is more difficult to design efficiently on a pipelined machine. For these algorithms, operations that take more than one cycle to complete because of the pipe (such as condition code generation) should be ordered in such a way that they are meaningful if either path of an intervening conditional branch is taken. A feature of the A900 micromachine that lends itself well to designing this type of code efficiently is the ability to start a piped operation in one microcycle, and then modify its action in the next cycle before it completes. The short code sequence below shows an example of the use of this feature.

```
             If TZ then go to dont_inc, m1<=m1+1;
                            *Microcycle 1. This is just a
                            *conditional branch that
                            *in the same microcycle starts
                            *the piped operation of incrementing
                            *memory address pointer m1.
        inc:  nop;          *Microcycle 2. Here is one target of the
                            *conditional branch. The nop here
                            *is to show that memory address
                            *pointer m1 was incremented
                            *since no modifying microorder
                            *was used here.
                            ⋮
   dont_inc:  ninc;         *Microcycle 2. Here is the other
                            *target of the conditional
                            *branch. Here a
                            *modifying microorder is used
                            *to stop the incrementing
                            *of m1<=m1+1 before
                            *it has completed.
                            *Thus m1<=m1+1
                            *can be effectively
                            *used in both paths of the
                            *conditional branch.
```

A final key to generating efficient algorithms for a pipelined micromachine is creating efficient code for algorithms with loops. In these algorithms, the loop time is most often the determining factor in how fast the algorithm will run. Therefore, the loop itself should be designed first for efficiency, and then the entrance and exit to the loop can be added.

Acknowledgments

The authors wish to thank Simin Boschma and Jim Finnell who contributed to the hardware design, and Tom Harms, Chuck Morgan, and John Shelton who contributed to the firmware design. Thanks also to Donna Saar, Wayne Chin, and the many other people who contributed to the magical process of taking a processor from idea to production.

Floating-Point Chip Set Speeds Real-Time Computer Operation

by William H. McAllister and John R. Carlson

FLOATING-POINT ARITHMETIC performance is a prime concern in technically oriented computers. Using Hewlett-Packard's silicon-on-sapphire CMOS process we have designed a set of three monolithic floating-point processor chips for use in two HP 1000 A-Series Computers, the A900 and the A700. The chip set provides a cost-effective, high-performance solution for high-speed computation.

The set consists of three chips, one each for addition, multiplication, and division. Each chip can perform arithmetic operations on 32-bit and 64-bit floating-point numbers and on 32-bit integers.

The primary design objective was to maximize the speed of floating-point scalar (single-element) operations. This

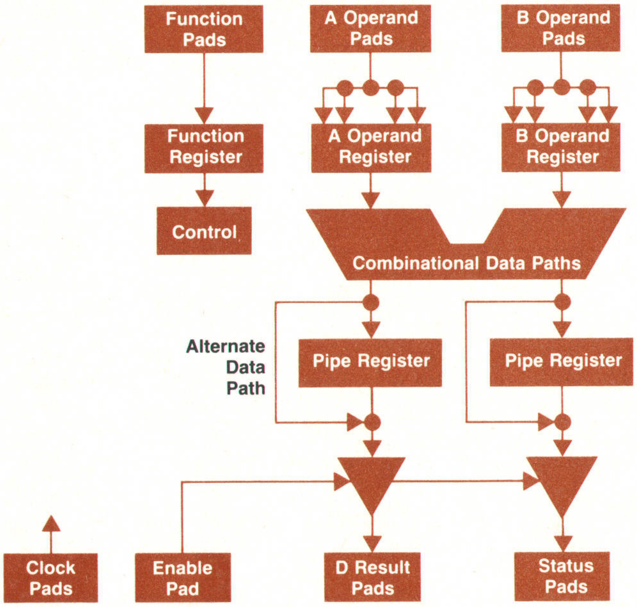


Fig. 1. *The floating-point add, multiply, and divide chips have this block diagram in common. They have similar user interfaces and operate in roughly the same manner.*

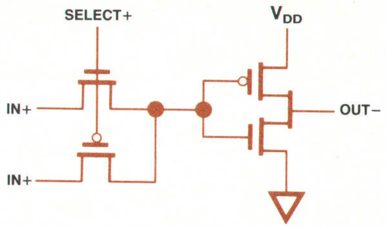


Fig. 2. *A 2:1 multiplexer implemented with two pass transistors and a level-restoring inverter. The circles indicate p-channel devices. One of the pass transistors in turned on by the SELECT+ signal. Half of the time the input transistors must pass a degraded signal, which is restored by the inverter. This circuit is slower but more compact than other multiplexer implementations.*

goal was achieved by partitioning the design into three integrated circuits. Each is individually optimized to provide the maximum speed for the most common floating-point operations. To simplify the design task and to take advantage of the inherent speed of the SOS process, we use predominantly nonclocked combinational circuits. This design technique allows each operation to proceed as quickly as the logic will allow without requiring a periodic pause to latch intermediate results.

The second design goal was to allow fast vector (multiple-element) operation. The add and multiply chips have a special pipeline register to allow the overlapping of load/unload cycles with the combinational data path delay. Using the chips in vector mode allows a string of similar operations to be performed with great speed.

Chip Operation

The block diagram in Fig. 1 is common to the three chips, which have similar user interfaces and operate in roughly the same manner. Each chip has three 16-bit buses. Two (A and B) carry the input operands and one (D) carries the output result. The 32-bit or 64-bit operands are moved to the chips in sequential 16-bit words. Function codes are loaded on a separate bus. The 32-bit or 64-bit result is unloaded in successive 16-bit words.

The operands are loaded, 16 bits at a time, into registers

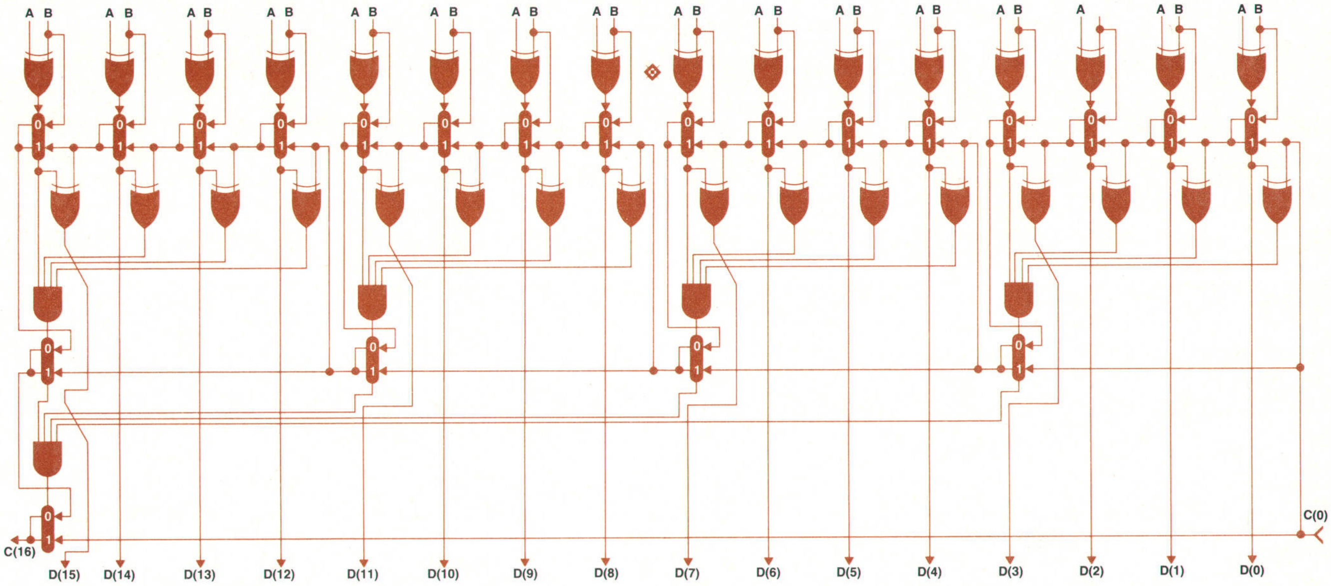


Fig. 3. *This is a 16-bit slice of a 64-bit carry propagate adder. It has three levels of carry lookahead logic. The A and B bits are the input operands to be added. They are EXCLUSIVE-ORed to produce a propagate term, which controls a 2:1 multiplexer in the carry chain and helps to form the sum D. The different parts of the carry chain allow carry signals to skip 1,4, or 16 bits at a time.*

on the selected chip in two to four clock cycles. The data propagates combinationally through the data path logic until the result is settled at the output. The result is then unloaded, 16 bits at a time, in two to four clock cycles. For the add and multiply chips, the propagation delay of the data path varies from 400 to 900 ns depending on the operation performed. The divide chip, however, is not fully combinational and requires two to four times the delay of the other two chips.

Circuit Design

The design of the floating-point chip set required the use of some novel hardware structures to limit the size of the chips and achieve maximum speed at the same time. HP's CMOS-on-sapphire process has 4-μm feature sizes and metal gate devices. The device thresholds are 1.75 volts and the chips use a +12-volt power supply. Compared to traditional CMOS which uses a silicon substrate, SOS devices exhibit less threshold variation caused by the body effect. This allows circuit design using devices as source followers. A transistor can be configured to pass the "wrong" logic level, i.e., an n-type device passing a high voltage or a p-type device passing a low voltage.

Fig. 2 shows an example of a 2:1 multiplexer implemented with two single-sided pass transistors and a level restoring inverter. The layout for this circuit is very compact owing to another feature of the SOS process. Since SOS transistors are built on individual islands of silicon, they are electrically isolated from each other by the sapphire substrate. This allows n-channel and p-channel transistors to be placed next to each other. The density advantages of the single-sided circuit are the key to the implementation of large data path arrays used on the chips. A less compact layout would have caused the chips to be too large for economical production.

As an example of the types of digital circuits used, Fig. 3 shows the schematic for a function used on all of the chips. We needed a very high-performance 64-bit adder that did not use too much area for carry propagation circuitry. An efficient carry lookahead scheme takes advantage of the 2:1 multiplexer shown in Fig. 2. It allows a 64-bit adder to be constructed with three levels of carry lookahead and a delay of only 15 full adders. In contrast, a 64-bit ripple adder requires 64 full adder delays.

Add Chip

The add chip is shown in Fig. 4. Fig. 5 is a block diagram of the data path portion of the chip. There are about 30,000 transistors on this 5.8×6.4-mm die. Most of the area is devoted to the fraction path in the center and right side of the photograph. The hardware structures in this part of the chip are the input registers, right shifter, main fraction adder, priority encoder, left shifter, incrementer, and output pipeline register. The shifters are easily recognized by their slanting pattern. The exponent path fills the upper left portion of the chip and control logic takes up the lower left portion.

After the operands are loaded, the exponent fields are compared. The larger exponent is passed on and the difference in the exponents is used to control the right shifter. The right shifter aligns the binary point of the smaller operand with that of the larger. The operands are then

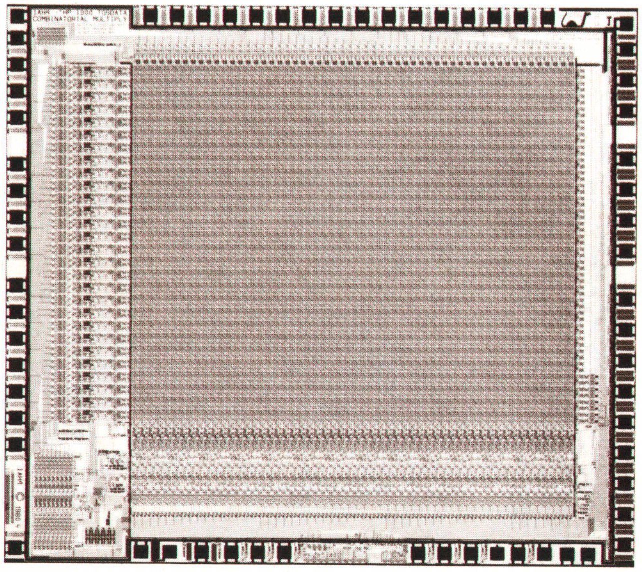


Fig. 4. *Floating-point add chip contains about 30,000 transistors and measures 5.8 by 6.4 mm.*

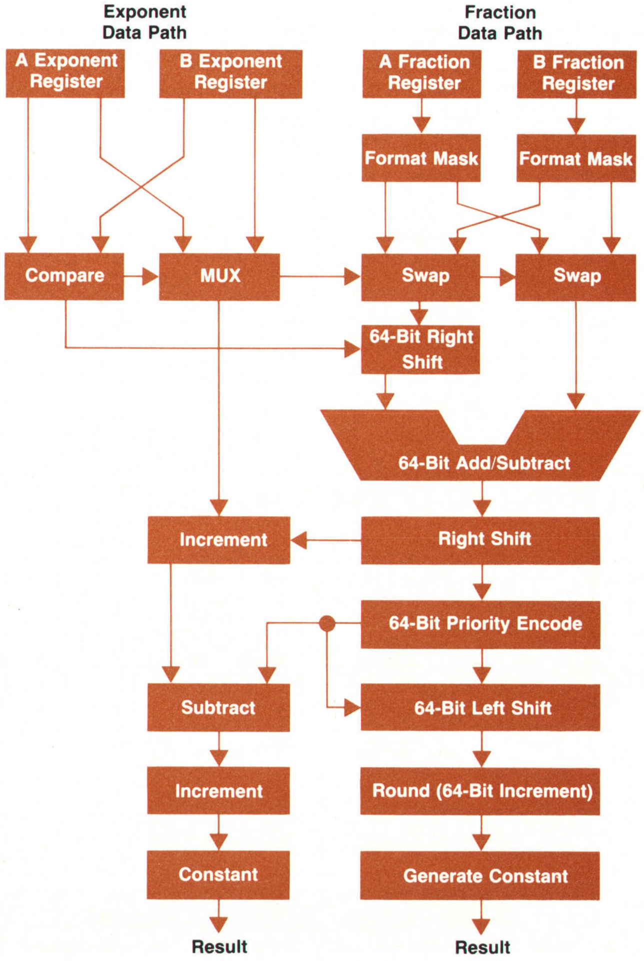


Fig. 5. *Block diagram of the add chip. After the operands are loaded, all computation is done by combinational logic.*

added or subtracted by the 64-bit main adder. If the fraction overflows, a signal is sent to the exponent path to cause an increment. The fraction is also right shifted one place to keep it in the proper scale.

If a subtraction is done there may be cancellation of leading significant bits. This result is called an unnormalized number. The required result is a normalized number. The priority encoder searches from left to right looking for the first significant bit and encodes its location. The left shifter uses this value as a shift count to renormalize the result. The shift amount is also subtracted from the exponent.

The next step is to round the result to the proper precision using an incrementer. An overflow from the fraction once again causes the exponent to increment. Finally, the exponent is checked for overflow or underflow and the proper status bits are set.

After the operands are initially loaded into the chip, all computation is done by strictly combinational logic. The worst-case delay path passes through a few hundred gates. It takes about 700 ns from the time the inputs are loaded until the result appears at the output pads.

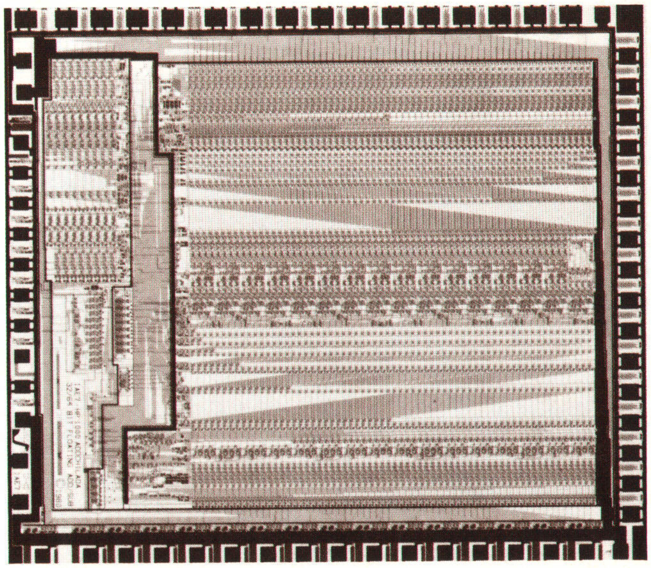
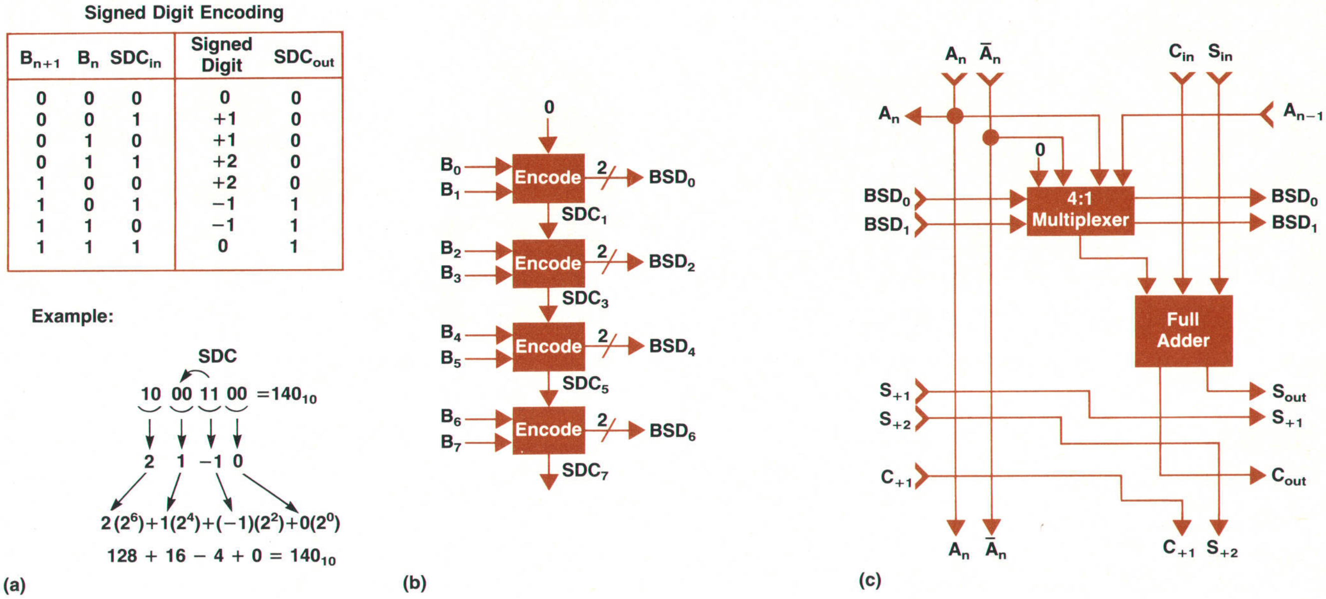


Fig. 7. *Floating-point multiply chip measures 6.1 by 7.0 mm and contains about 60,000 transistors.*

Multiplication Technique

The floating-point multiplier chip uses a technique called the modified Booth algorithm[1] to do a combinational multiplication of the operand fractions. This algorithm is used in a number of commercial monolithic multipliers and is the key to integrating this common function. The algorithm reduces the number of gate delays by nearly a factor of two with little increase in chip complexity compared to more traditional methods of multiplication. This reduction in propagation delay is accomplished by encoding one of the operands into a new form before applying it to the multiplier array.

To understand how the algorithm works, it is best to interpret the encoding in a mathematical sense. The encoding scheme can be thought of as mapping one binary operand into an equivalent set of signed digits. The particular encoding we choose turns out to reduce the number of full adder rows by a factor of two. The encoding is shown



Signed Digit Encoding

B_{n+1}	B_n	SDC_{in}	Signed Digit	SDC_{out}
0	0	0	0	0
0	0	1	+1	0
0	1	0	+1	0
0	1	1	+2	0
1	0	0	+2	0
1	0	1	−1	1
1	1	0	−1	1
1	1	1	0	1

Example:

SDC

$10\ 00\ 11\ 00 = 140_{10}$

$2 \quad 1 \quad -1 \quad 0$

$2(2^6)+1(2^4)+(-1)(2^2)+0(2^0)$

$128 + 16 - 4 + 0 = 140_{10}$

Fig. 6. *The floating-point multiplier chip encodes one of the operands (B) into an equivalent set of signed digits, thereby reducing the number of full adder rows required by a factor of two. (a) Encoding scheme. SDC is the signed digit carry. (b) A diagram showing how a binary number is converted to a signed digit representation. Each encoder acts like a full adder with three inputs—two consecutive B operand bits and a signed digit carry in. Its outputs are a signed digit and a signed digit carry out. (c) A circuit that multiplies a signed digit BSD by a binary bit A.*

in Fig. 6. The B operand bits are mapped into a set of signed digits (+2, +1, 0, −1). This representation is used in place of B to drive an array of full adders and multiplexers.

The signed-digit representation of B has the property of simplifying multiplication substantially. Each of the signed digits is simple to use as a multiplier. The 0 and +1 signed digits act just like binary digits in the traditional multiply. A +2 multiple can be obtained by a one-bit left shift. A −1 multiple takes the two's complement of the input operand (invert and add 1). A circuit that multiplies a signed digit by a binary bit is shown in Fig. 6c. It consists of a 4:1 multiplexer and a full adder. Since there is a 2-to-1 compression when encoding binary bits into signed digits, we only need half as many of these multiplier circuits.

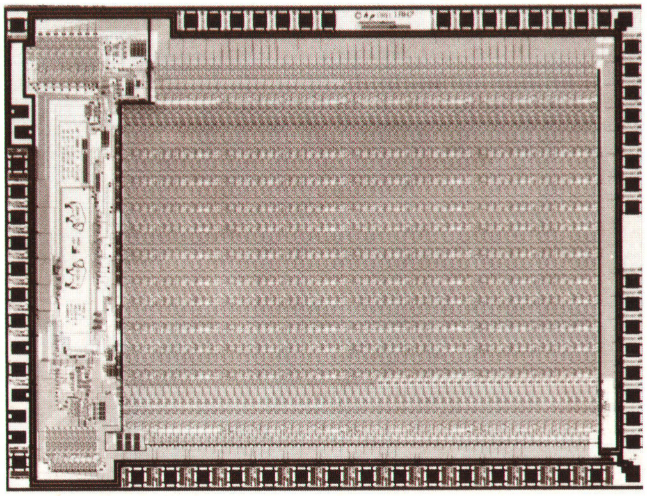


Fig. 9. *Floating-point divide chip measures 5.2 by 7.2 mm and contains 35,000 transistors.*

Multiply Chip

A photograph and block diagram of the multiply chip are shown in Figs. 7 and 8. Floating-point multiplication is relatively simple compared to addition, so the chip has a much more regular appearance than the adder. The majority of the chip area is used to perform a combinational 56-by-56-bit integer multiplication of the operand fractions. The circuitry in the lower left of Fig. 7 is the exponent data path where the operand exponents are added. The chip is 6.1 by 7.0 mm and contains about 60,000 transistors.

The A operand fraction is loaded into a register across the top of the chip. The B operand fraction is loaded into a register along the left side of the chip. The exponent fields of each operand are latched into the exponent data path at the lower left. Initially, the exponents are added together and the fractions are masked to the proper precision.

The B operand is encoded as described above. The A operand drives into the central array from top to bottom and the encoded B operand is driven across the array from left to right. The array performs an integer multiply with the most-significant bit of the product emerging at the lower

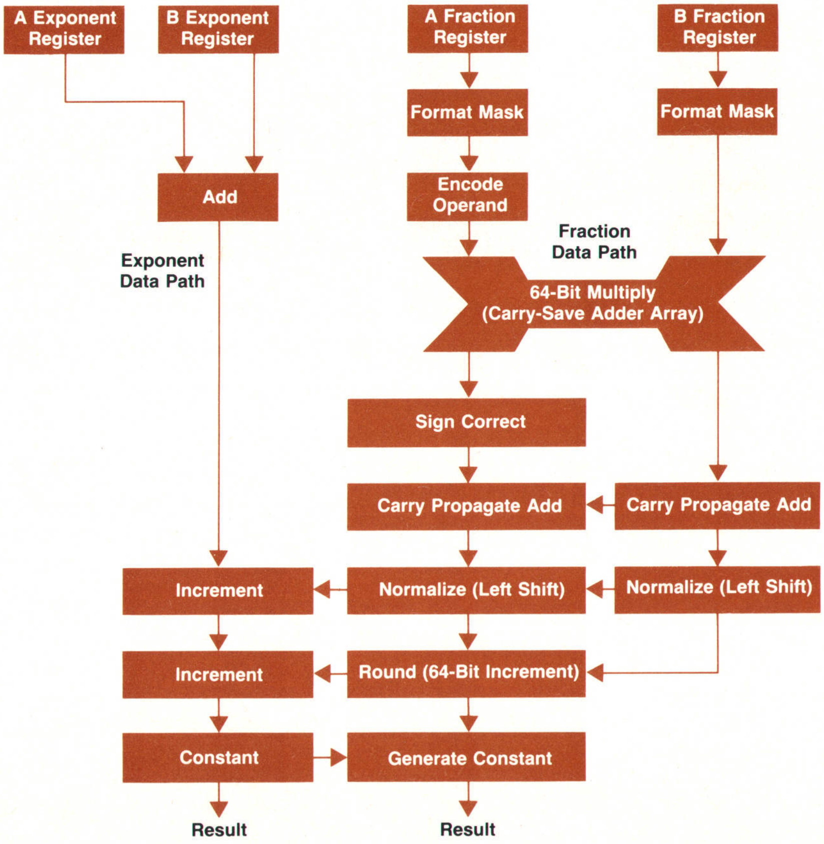


Fig. 8. *The multiply chip performs a combinational 56-by-56-bit integer multiplication of the operand fractions. The 112-bit product is normalized if necessary and rounded to the required precision.*

left corner. The product bits extend across the array to the lower right corner and continue up the right side to the very top right where the least-significant bit comes out. A 112-bit product is developed from the two 56-bit operands.

The integer product is normalized if necessary with a one-bit left shift and then rounded to the final precision. The exponent may need to be incremented if there is a fraction overflow from either normalizing or rounding. If control logic detects an overflow or underflow, the proper constants are forced into the result.

Divide Chip

The floating-point divide chip is shown in Figs. 9 and 10. The layout is similar to the multiplier with a large fraction data path in the center and a small exponent data path in the upper and lower left. The chip is 5.2 by 7.2 mm and has 35,000 transistors. The algorithm used is not fully combinational like the other two chips. It requires multiple clock cycles to produce the floating-point quotient.

The operands are first loaded into two registers across the top of the chip. The exponents are subtracted and the fraction fields are properly aligned and masked to the required precision. The signs of the dividend (A) and the divisor (B) are examined. A conditional two's complement negation is performed on each fraction to ensure that A is positive and B is negative. The B operand propagates down the chip through an array of shifters and carry propagate adders. These circuits develop the first seven multiples of the divisor (B×1 to B×7). A clock cycle is required to load

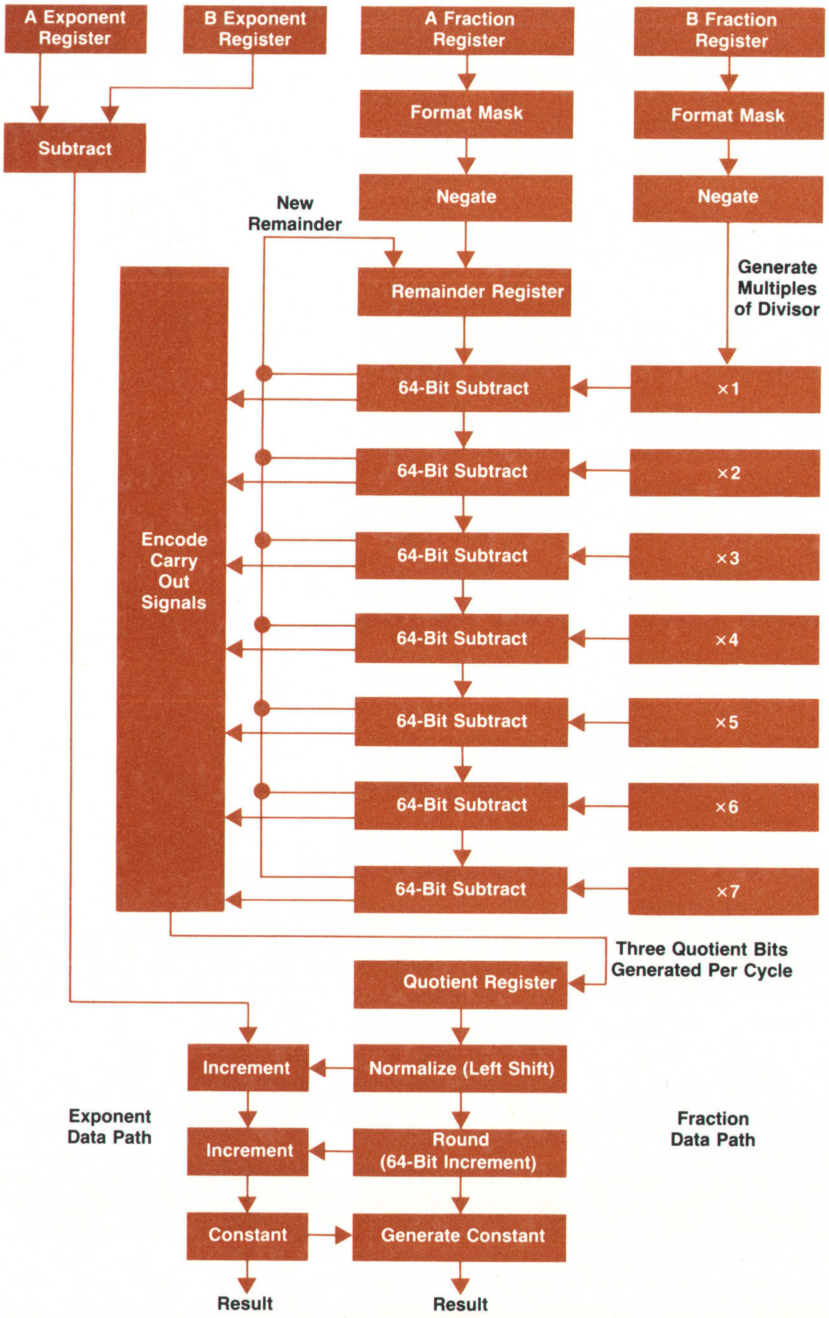


Fig. 10. *The divide chip is not fully combinational. It requires several clock cycles to generate the floating-point quotient.*

the A operand into the remainder register near the top of the chip.

After this initial setup phase, the divider is ready to generate quotient bits. The contents of the remainder register are driven down the chip through an array of seven carry propagate adders. These add circuits are actually interleaved between the multiple generator circuits mentioned above. The divisor multiples are subtracted from the remainder in parallel to create seven new potential remainders. The carryout of each adder is examined to find the smallest positive remainder. It is driven back up the chip and clocked into the remainder register. Its position in the array is encoded into three new quotient bits which shift into the quotient register. This process is repeated, generating three quotient bits each clock cycle, until the required precision is reached. The quotient is normalized and rounded. The exponent may be incremented if either of these causes an overflow of the fraction. Exponent overflow and underflow are signaled with the appropriate constants.

Conclusion

The floating-point chip set provides a powerful (up to one million operations per second) and inexpensive execution unit for HP computers. The chips may be used as flexible building blocks in a variety of system architectures. Key reasons for the success of the implementation are the density and low power of HP's CMOS-on-sapphire process and the development of algorithms and circuit designs that take advantage of the process.

Acknowledgments

The authors wish to acknowledge the contributions of Frederick Ware who pioneered the concept of this chip set and brought it to life.

References and Bibliography

1. A.D Booth, "A Signed Binary Multiplication Technique," *Quart. J. Mech. Appl. Math.*, Vol. 4, 1951, pp. 236-240.
2. S. Wasser, "High Speed Monolithic Multipliers," *Computer*, October 1978, pp. 19-29.
3. F. Ware, "64b Monolithic Floating Point Processors," *Digest of Technical Papers*, Vol. 25, 1982 IEEE International Solid-State Circuits Conference, pp. 24-25.
4. F. Ware and W. McAllister, "CMOS Floating Point Chip Set," *Electronics*, Vol. 55, no. 3, February 1982, pp. 149-153.

Comprehensive, Friendly Diagnostics Aid A-Series Troubleshooting

by Michael T. Winters and John F. Shelton

DIAGNOSTICS FOR THE A-SERIES of HP 1000 Computers are characterized by a planned systematic progression of testing, features designed into the hardware for diagnostics, and the use of an operator-oriented diagnostic design language. The progression of testing starts with a microcoded self-test and continues through to complete interface testing. The testing is in two basic sections:

- Self-test/pretest
- Kernel and interface diagnostics.

The self-test/pretest combination is executed each time the system is powered up and checks only enough of the computer to ensure that the boot loading program will execute properly. The tests are contained in ROMs and are an integral part of the CPU. These tests are an integrity check. The kernel and interface diagnostics are an in-depth check of the basic functions of the CPU, memory, and I/O functions including DMA and the interface cards. The kernel and interface diagnostics run under the diagnostic control system, which provides all utilities necessary for diagnostic loading and execution.

This sequence of testing helps the customer in two ways. If the computer has a major failure, the self-test/pretest combination will not allow the computer to run. This provides a confidence factor for the customer in that a major failure can be detected before restarting or booting of the system after power-up. This avoids a possibly dangerous condition in a process control environment. With the other diagnostics, the customer can identify and fix most failures by major chip or board replacement.

The goals for development of the diagnostics were:

- User friendliness
  - □ Conversational, including help files
  - □ Simple load and go operation with go/no-go results
  - □ Completely automatic execution
  - □ No operator intervention except where absolutely necessary
  - □ No jumper changes required during test execution
- Completeness for service engineers and manufacturing
  - □ Full error reporting for analysis
  - □ Single-test execution
  - □ Looping for single-test diagnostic and other diagnostics
  - □ Unattended loading and looping for oven testing.

Self-Test/Pretest

The self-test and the pretest check the basic integrity of the computer after power comes up, but before restarting or booting of a program. The self-test is a microcoded test performed by the CPU to check the internal registers and data paths before fetching the first pretest instruction. The pretest is a software program written in assembly language. When the pretest is executed, it checks the basic instruction set, the memory, and each I/O chip. If a failure occurs, the program will stop execution so that booting or restarting of the system will not occur. The status LEDs are then used to determine the nature of a failure. If the error is not fatal and a virtual control panel (console) is present, an error message is displayed for the operator.

Diagnostic Control System

The diagnostic control system (DCS) provides a layered structure for loading and executing diagnostics. The structure starts with the basic control module (BCM) and is added to until a complete executable diagnostic is built (see Fig. 1).

The basic control module contains the sections needed to get started. These include the auto program, format utilities, console driver, and primary load device driver. BCM also contains a basic test that is similar to the pretest and is executed before configuration. This allows simple troubleshooting in case the initial failure will not allow BCM to execute properly. BCM is self-configuring and displays the system configuration, which includes CPU type, memory size, and I/O card identification. Also, all revision levels of programs are displayed for operator verification.

Once configuration is completed, the auto program is started (if selected). It loads the remaining system modules and then starts a sequential execution of diagnostics. The modules are MSGS for extended error messages and help files, MAPS for memory management, and the DDL interpreter. The auto program is controlled by an auto file which specifies the programs and diagnostics to be loaded and executed. This is determined by the current configuration. If desired, the operator can create an auto file tailored to a system's specific needs.

BCM also contains a linking loader which allows efficient use of memory. Programs and drivers can be added without concern for absolute address requirements. The auto program calls the linking loader to load the system modules and then later the relocatable diagnostic modules.

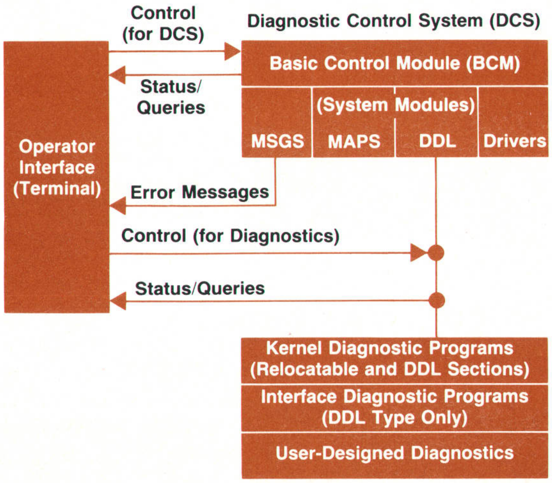

Control
(for DCS)
Diagnostic Control System (DCS)
Basic Control Module (BCM)
Status/
Queries
(System Modules)
MSGS
MAPS
DDL
Drivers
Operator
Interface
(Terminal)
Error Messages
Control (for Diagnostics)
Status/Queries
Kernel Diagnostic Programs
(Relocatable and DDL Sections)
Interface Diagnostic Programs
(DDL Type Only)
User-Designed Diagnostics

Fig. 1. *HP 1000 A-Series diagnostic control and diagnostic program structure.*

Diagnostic Design Language

One of the main modules loaded is the diagnostic design language (DDL) program, which is a BASIC-like interpreter. This makes creation or modification of diagnostics by the customer easy. The program allows direct I/O instructions, buffer/data manipulation, and simple branching. Programs can easily be saved for later use, or added to the auto file for standard execution.

A debug program is also available which allows access to the individual relocatable programs for modification. Debug was created during the development of the diagnostics for use with the relocatable programs. It became a useful tool for hardware troubleshooting and was added to the diagnostic package.

One of the advantages of the new system is that individual tests can be written in assembler relocatable format and then called by a DDL program. This has two benefits. First, writing in assembler allows instruction-by-instruction control and execution, which is very necessary for diagnostic applications. Second, DDL allows easy manipulation and control of the tests. The individual tests are not burdened with message reporting, looping, or operator interaction; these functions are handled by the DDL program.

The question may arise, "Why design a separate system for diagnostics? Why not use RTE?" There are two main reasons for not using RTE as the operating system. First, RTE assumes all hardware is functional when loaded and executed. Therefore, before any diagnostic execution, the system may use hardware that is failing and not even allow execution to start. This applies mainly to the CPU (instructions, interrupts, time base generator, etc.). Second, diagnostics require full control of the system, which cannot be allowed in RTE. For example, a system reset would cause all current I/O operations to be aborted. The diagnostic control system uses only hardware that has already been checked by the self-test, pretest, and BCM basic test.

Kernel and Interface Diagnostics

The kernel diagnostics consist of the tests required to check the basic CPU, including the base instructions, the memory controller and array, the system functions (powerfail, memory protect, time base generator, etc.), and the I/O master portion of each interface (interrupts, control/flags, and DMA). Each diagnostic contains two parts, the relocatable program and the DDL program. The relocatable program does the in-depth instruction-by-instruction verification. The DDL program controls test execution, error reporting and operator interaction. Each diagnostic contains several subsections and each can be individually selected and looped.

When executed by the auto program or a run command, the diagnostics report only a pass/fail indication. If the operator selects an individual test for execution and an error occurs, the complete error message is displayed. The message contains setup parameters, results, and a pointer to the location in the relocatable program where the test was performed.

Interface diagnostics are written only in DDL, because they are I/O related and don't need the instruction-intensive routines. Each diagnostic contains three sections: system environment, special setup, and loop to device or special access. These sections vary depending on the interface. For the system environment test, the diagnostics are loaded and executed without any hardware changes. Although this is not a complete test of the interfaces, it is as complete a test as possible. For a complete test of an interface, the diagnostic must be run individually with test hoods and/or necessary hardware changes. The loop to device test can in some cases be automatic and in others run separately. For example, the HP-IB interface diagnostic identifies devices and runs loopback tests where possible.

Virtual Control Panel

The virtual control panel (VCP) is a console-oriented replacement of a hardware front panel. It provides control and access to the CPU like a hardware front panel, that is, it enables the operator to examine or change registers and memory. It also allows loading of programs and control of their execution. The interaction with the operator is implemented by a program in ROM which also contains the pretest and loaders. The program is designed to use the system console if available and configured. The console can be local, remote using modems, or connected by a distributed system link.

Also in ROM are the loaders for the common devices available. The loaders can be invoked either by command from the operator or at completion of the pretest by selection of the start-up switches.

Having the pretest, virtual control panel, and loaders in ROM and an integral part of the CPU ensures that there is a means of loading, access, and control of the computer at a very basic level.

The BCM console and load device drivers use the same interfaces as the VCP. They also use similar routines. Thus diagnostics can be loaded and controlled from the same console, and diagnostics can be controlled from a remote console the same as from the VCP. This also applies to the loaders. For example, in a distributed system network, diagnostics can be loaded and controlled from a central computer for testing of a remote node without a terminal or local loading device.

Currently A-Series manufacturing uses the remote capability. The diagnostics used in the field are the diagnostics used in manufacturing. The oven testing and test stations are connected to a central A-Series Computer by a distributed system. The oven station is connected using the factory data link interface, which is a multiple-drop system with 12 test units connected to one central interface. The test stations are connected by a distributed systems (DS) interface which is a point-to-point connection. Using a central computer means that only one source is used for diagnostics, all units are tested the same, the test stations can use the same diagnostics when troubleshooting, and when updating is required, only the central system need be updated.

A900 Self-Test

The A900 processor contains the most extensive microcoded self-test ever written for an HP 1000. Of the control store that is reserved for the base instruction set, scientific instruction set, vector instruction set and self-test, 25% is used by the self-test. Self-test's primary use is as a confidence check every time the computer is powered up. It is designed to exercise as much of the logic as possible, and to detect when the processor is not functioning properly. Because it exercises the processor so thoroughly, it is also used in manufacturing as a debugging tool, and was used to help debug the lab prototypes.

As explained above, all A-Series Computers contain a macrocode (assembly language) self-test that checks the processor when it is first powered up. Why write a microcoded self-test to do the same job? The microcode and macrocode self-tests test the machine from very different points of view. The microcode tests the processor at the level of individual circuits, while the macrocode tests the processor at a functional level. Each type of test has certain advantages.

A microcoded test can be extremely thorough and efficient, because microcode has more control over the logic elements that make up the processor, and because it is specific to the family member (in this case, the A900). It can also be very fast. This is partly because microcode can execute more quickly than macrocode, but mainly because of the increased efficiency of the test. The A900 microcode self-test takes less than 0.1 second to test the entire processor. Another advantage that a microcode test has over a macrocode test is that a failure in the hardware could prevent macrocode from running at all, while allowing the microcode to execute well enough to report the failure.

The main limitations of a microcoded test are its size and its crude output methods. The A900 self-test has nine LEDs available with which to pass information to the outside world. It uses these LEDs to signal that there is a failure somewhere in the processor, and to indicate which board the failure is probably on. Because of the limited amount of microcode storage, the test must be as compact as possible. This limits its complexity. In the A900, fault detection was considered the primary goal. While fault isolation is theoretically easier to attain with microcode than with macrocode, space limitations prevented the A900 self-test from being as thorough in isolating faults as it is in detecting them. While it attempts to identify which board is causing the failure, it is only right about 90% of the time.

About 1.5% of the A900 processor hardware is devoted to self-test. The majority of this is made up of status and control registers. Status registers allow access to internal signals that would otherwise be difficult to observe, and control registers allow the self-test to put the processor into states that would otherwise be difficult or impossible to reach. Control registers are very useful, for example, in testing the cache memory, which is normally transparent to the rest of the processor, or in testing the error detection and correction chips on the memory controller, which have their own built-in diagnostic capabilities.

The most useful piece of self-test hardware in the A900 is a timer that generates interrupts at the micromachine level. This counter generates an interrupt eight cycles after it is turned on, and freezes certain status registers. This allows the self-test to examine the state of the machine when the timeout occurs, and allows it to escape from failures that cause microaddressing errors or cause the machine to hang indefinitely. Combined with the ability to do a vectored microjump, this timer allows the self-test to scan all of control store for microcode parity errors, thus verifying the integrity of the microcode for the entire base instruction set, scientific instruction set, and vector instruction set.

Designing and debugging a self-test presented some unique challenges. Using a processor to test itself for failures is a little bit like looking in a mirror to see if your eyes are closed. How can you trust the results of a test if you are not certain that the processor is working in the first place? Suppose, for example, that a major failure in the microaddressing logic causes the self-test to be bypassed completely on power-up, indicating that the processor is working perfectly, when in fact it is barely working at all. This is an extreme example, but there are many failure modes that can subtly mask either themselves or other faults if not properly tested.

The general rule to follow when writing a test like this is that no code that tests one part of the processor can assume that any other part is working correctly unless the other part has already been tested (this includes, of course, the self-test hardware, too). On power-up, however, nothing has been tested and nothing can be assumed to work properly. How do we begin?

On the A900, the microinterrupt timer provides an answer. The timer starts counting on power-up. If it is not turned off in eight cycles it causes an interrupt, and the code goes into an infinite loop. The self-test is arranged so that if there is a major error in the microaddressing logic, the self-test will not be able to get to the instruction that turns the counter off in time. Once we know that the major pieces of the microaddressing logic are functioning, we can use the timer in a similar way to test the rest of the microaddressing logic to assure that microinstructions can at least be fetched and executed in the correct order.

Once we can assume that the self-test code will execute in the proper order, what do we test next? Most tests rely on comparing some result with an expected value, so the next thing to test is the portion of the ALU that compares two values and recognizes a match. Self-test continues in this manner, each test using only what has already been shown to be working by previous tests.

After the A900 self-test was written, another problem arose. How do you test a self-test? We wanted to know with certainty how complete this self-test was, and to correct any bugs that would let failures slip by. It would be impossible to simulate all possible failure modes of a processor as complex as the A900. Our solution was to build a simple circuit that would allow us to force any single node (IC pin) in the processor to a logical one or zero. Then, in a time-consuming and tedious process, each node was forced to one, then to zero, and the machine was put through a power-up cycle in each case to see if an error would be detected. This task took about two engineer-months to accomplish, but when it was done, the results allowed us to refine the self-test even further, and to say with certainty how complete the self-test is. And the results? 97% of the stuck nodes were detected as error conditions and almost ninety percent of the time the correct board was indicated as the source of the error.

Acknowledgments

The development of the diagnostic package for the A-Series was a team effort. Thanks must go to Craig Chatterton (project manager), Mark Koehler (memory, system function, and I/O master), and Lotus Lin (instruction set). Thanks also to Wayne Chin for heading up the laborious process of testing the self-test, and to Christian Roy for his experienced input during the early stages of development.

New Real-Time Executive Supports Large Programs and Multiple Users

by Douglas O. Hartman, Steven R. Kusmer, Elizabeth A. Clark, Douglas V. Larson, and Billy Chu

RTE-A IS THE STANDARD SOFTWARE found on all HP 1000 A-Series Computer Systems. This includes the operating system and a large number of utility programs and libraries. It also includes VC+, an optional package that extends RTE-A's capabilities to include virtual code, spooling, and multiple users.

Various versions of RTE, the Real-Time Executive, have been the operating system for all HP 1000 Computers. RTE-A adds major new features to previous systems, including a modern program development environment and support for multiple users and large, reentrant programs.

Real-Time Systems

RTE systems are primarily used for the "real-time execute" environment. This means that these systems are running application programs that monitor or control some real-world process. Examples of these applications include computerized testing, data collection, and communications concentration. RTE is well suited to these applications, which require a different kind of system than would be found in a personal computer or in a timesharing and batch system.

Real-time systems allow programs to deal directly with external events, including I/O devices and timers. Almost all real-time systems allow multitasking, which means several different programs cooperate to accomplish a job. Real-time systems must provide these services reliably (meaning they shouldn't crash) and deterministically (meaning they should respond to events in a predictable manner).

About RTE-A

RTE-A covers the whole range of A-Series systems. Some systems are very small, doing only one thing, while others are large, with a large number of programs and peripherals. Systems can be either disc-based or memory-based. The following table shows the minimum and maximum sizes for RTE-A systems:

Resource	Minimum	Maximum
Number of programs	1	255
Number of I/O devices	1	255
Memory (bytes)	128K	24576K
Disc space (bytes)	0	20200M
Program size (bytes)	2K	7936K
Program data size (bytes)	2K	131072K

Users can have a standard configuration, known as a primary system, or can tailor a system by using a system generation program. Programs developed for smaller configurations will run unchanged on larger configurations.

RTE-A does its work using a minimum of system resources. This helps application programs make the most of A-Series hardware performance. RTE-A rarely requires more than 10% of the system's processing power. For programs doing computation on an A900, it is common to see RTE-A overhead of only 0.5%, leaving 99.5% of the CPU time available for the application. RTE-A can execute almost all system service calls in less than one millisecond of CPU time when it is running on an A900.

The A-Series and RTE-A are compatible with most of the powerful software already developed for RTE systems, including compilers, networking, data base management, and graphics. Newer products address such areas as process control, quality monitoring, and connection to programmable controllers.

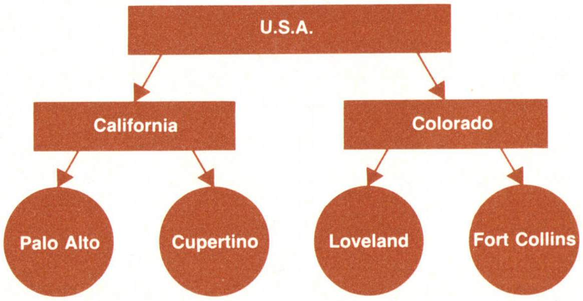


Fig. 1. *RTE-A's hierarchical file system consists of files (circles) and directories (boxes). Directories can be nested inside other directories.*

New Program Development Features

Program development includes all aspects of writing, preparing, and testing programs to accomplish the job at hand. An A-Series application typically involves a number of programs that work together. These programs pass information by means of files or common data areas. Most application programs are written in FORTRAN, Pascal, or BASIC, but sometimes assembly code or even microcode is used for sections of a program.

RTE-A has a number of features to make it easy to develop programs. The features were designed with the application programmer in mind. The goal was to reduce the amount of computer expertise needed to use the system, leaving the programmer more time to develop programs. These features include a modern file system and a helpful command interpreter and utilities.

RTE applications programmers become very familiar with files. There are files for data, files for text, and of course, a large number of files holding the application programs themselves. A typical system might have several thousand files, with a large number of interrelationships between them.

RTE-A allows the programmer to name files in ways that reflect their contents. File contents can be identified through the file name and type extension. For example, three files that are used to hold an engine test application might be EngineTest.Ftn, EngineTest.Rel, and EngineTest.Run. These are the FORTRAN source file, the compiled relocatable code, and the executable program files, respectively.

Groups of files can be further organized by keeping related files in directories. RTE-A allows a large number of directories, so it is easy to keep things organized. If files can be thought of as things in file cabinets, then directories can be thought of as drawers full of related files. Directories can be nested inside other directories, just as shoeboxes can be nested inside file drawers. Nesting depth is unlimited.

Fig. 1 shows an example of this hierarchical file structure. In Fig. 1, directories are indicated by rectangular boxes, while files are in circles. Directory U.S.A. contains all of the other directories and files. Directory California contains the files pertaining to California cities. A directory can contain any number of files, and files can appear in any directory, even a directory that has subdirectories.

These features take care of organizing files on a single computer, but what happens when you have a network of computers? The file you need may be located on some other computer. RTE-A handles this situation easily by allowing computers to pass file information back and forth across the network, without the need for special commands or utilities.

Here's an example to show how networking operates. Assume that we have written an application that updates tables in a master file with data recorded from sensors. Later we decide to split the job between several computers to put more computing power where it is needed. In the RTE-A file system, file names may include computer names. Thus the application can continue to function even though the files are now located on different systems, despite the fact that the application was not originally written

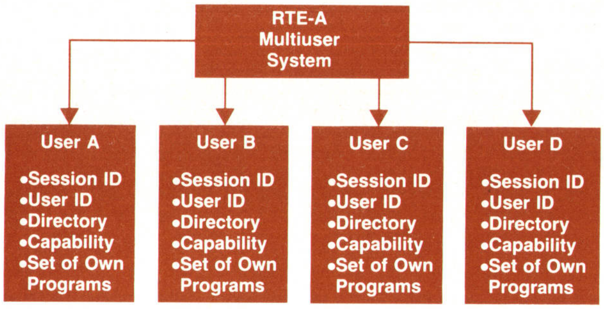


Fig. 2. *RTE-A provides a separate environment for each user.*

using networking.

In addition to hierarchical directories and networking, the RTE-A file system provides other features to help applications programmers. Some examples are:

- Files are time stamped to indicate when they were changed, and when they need to be backed up.
- Groups of files can be selected through wild-card selection, such as all names beginning with F.
- Following an errant purge operation (which removes files), it is possible to do an unpurge to recover a precious file.

Long-time RTE users will have spotted a number of new features, and may be wondering how these fit in with applications developed for older RTE systems. The answer is that RTE-A retains compatibility with previous file system applications, although the applications may need changes to take advantage of new features.

Multiuser Facility

The multiuser facility of the RTE-A operating system provides an environment in which many users can simultaneously use the system and access the system resources without interfering with each other. Each user has the experience of being the only user, and need not be concerned about the other users on the system (Fig. 2). The RTE-A multiuser system accomplishes these goals by providing an environment where users are protected from each other by user identification, by separate file directories, by the versatile Command Interpreter (CI), by having separate sets of user programs, and by logical sharing of system processes (system programs). The multiuser software includes the Command Interpreter and three other system programs called Promt, Logon, and Users.

In the RTE-A multiuser environment, each user is required to log onto the system with a logon name and optionally a password (up to 16 characters). The logon function is to identify and verify users who have authorized access to the system. For access to the system, a user must have a configuration file and an entry in the masteraccount file (Fig. 3). The masteraccount file is a protected system file containing the names of all users. This file and the users' configuration files are created by the Users system program based on information entered by the system manager.

For legal users, the logon process sets up an entry in the session table for the duration of the logon period. This session entry describes the resources and the environment in which the user is allowed to operate. At the end of the logon process, the system will run a preselected user interface, generally the Command Interpreter or the user's application program. In addition, the multiuser environment accepts remote programmatic logon requests and can also operate in a noninteractive session mode in which programs can run in the background until completion.

If error logging is used, the system is set up with an error logging file. The logon process will record the name of every attempted logon. The error log file thus provides another level of protection for monitoring of system activities.

The system allows the creation of two levels of users. Super users are allowed complete access for installing and upgrading the system. General users are allowed a more controlled access to the system.

The multiuser environment is easy and flexible to set up and administer. It can be set up with the services of a system manager in a very controlled atmosphere where security and tampering are a major concern. On the other hand, the system can be set up in a software laboratory where administration can be maintained by the engineers without the services of a system manager. Adding, maintaining, and modifying users for the multiuser system are efficiently handled by the Users system program.

User Interface

The application programmer's main contact with RTE-A is through the system Command Interpreter. This is a program that manages files, controls when programs run, and makes the facilities of RTE-A available to the user sitting at a terminal.

The Command Interpreter (which is called CI) tries to keep the user's life simple. It has a set of easy-to-use commands for operations such as copying files and running programs. In most cases the user types the right command correctly, and CI performs the requested operation, confirming it with a message such as "Copying ABC to

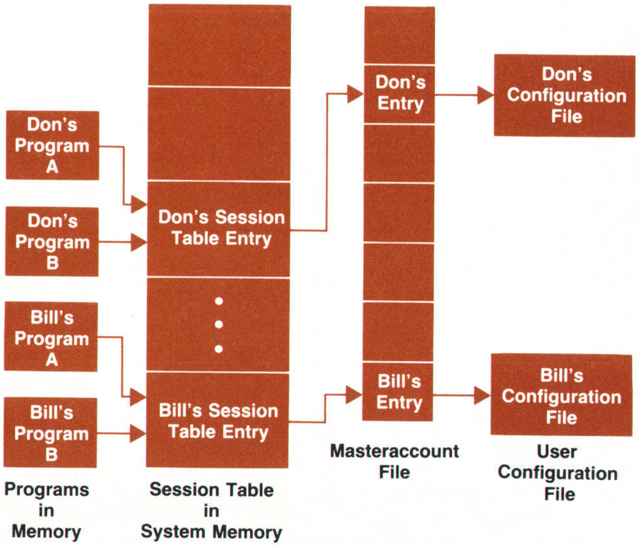


Fig. 3. *For access to the system, a user must have a configuration file and an entry in the masteraccount file. The configuration file is used to set up the initial session table entry.*

XYZ...OK." In some cases an error occurs, and CI tries to explain the situation with a message such as "No such file ABC." The user can then make use of the local editing features of HP terminals to correct and reenter the line without retyping a whole command line.

When a command seems mysterious, or when it is not obvious which command to use, RTE-A provides an "on-line manual," or help facility, that explains the functions of the various commands and gives examples of their use.

CI maintains a command stack that allows users to list, edit, and reissue previous commands. Also, CI may be directed to access a file as the source of commands. When the command file is exhausted, CI returns to interactive mode.

Sometimes a user wishes to interrupt CI while it is executing a command, such as listing a long file. Since CI is already busy processing the original command, it is not available to process the interrupting command. Under these circumstances, the Command Master program (CM) is invoked. CM is an exact copy of CI which runs at a high priority and contains the identical command set, but executes only one command and exits. When a user strikes a terminal key and a session is already active, Promt displays a CM> prompt and schedules CM. In the example of interrupting a long file listing, the break command could be used to cause CI to stop listing.

Sometimes even the CM program can be busy when a key is struck. In this case, Promt displays a System> prompt and schedules the Logon program to process the user's response. In this special mode, Logon accepts only commands that the operating system itself can process, such as off, break, and run.

In addition to the Command Interpreter, the user has approximately thirty utility programs available as a part of RTE-A, plus other programs available with HP 1000 software packages. These programs handle a wide variety of operations, including file backup and system status reporting.

Multiuser Implementation

There is a single copy of the Promt, Logon, and Command Master programs in the system. The operations of these programs are based on program-to-program class I/O functions of the RTE-A operating system. Promt is scheduled when a user interrupts the system from a terminal. Promt performs a class read on the input and determines whether to pass it to the Logon program or to the Command Master. Promt makes its decision by knowing if there is an interactive user on the terminal. If no one is on the terminal, Promt passes the input to the Logon program for logon processing. If there is a user on the terminal, Promt knows that CI is busy and passes the input to CM for execution. The Logon and CM programs are usually suspended until they receive a message from Promt. The RTE-A system wakes up the Logon program or the Command Master when Promt posts a buffer to be processed. The Promt and Logon programs contain many features that allow them to know when to perform multiuser system initialization, user logon, error detection, error correction, and special handling of the system on certain states and conditions. For example, if a needed system process is missing from the system, the Promt program automatically restores a working copy from the /PROGRAMS directory.

Features of the RTE-A multiuser environment include automatic logoff of users, noninteractive background processing, and remote programmatic logon. A user does not have to log off explicitly; the multiuser module in the system does this automatically. The system knows that the user has exited by using a program count in the user's session table. This count is incremented every time a program is created for the user. Correspondingly, the count is decremented when a program is terminated. When the count reaches zero, indicating that the user has exited from all programs including CI, the user is logged off.

To leave the terminal while continuing to execute current programs, a user can exit from the Command Interpreter into the noninteractive background session mode. This allows the remaining programs to run to completion, yet frees the terminal for another user. This background session will be logged off when the program count for the session is decremented to zero.

Another feature of the RTE-A multiuser facility is programmatic logon from remote systems. This feature allows users from other computer nodes to create a session to be

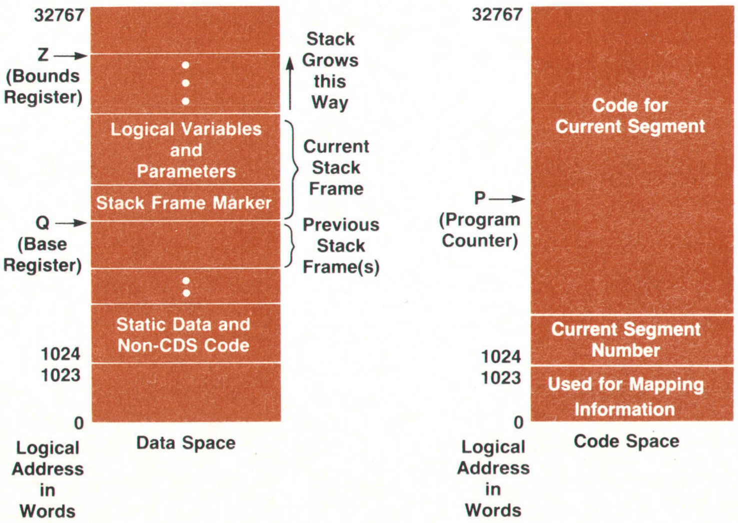


Fig. 4. *Code and data separation (CDS) provides separate memory partitions for code and data. Local variables are accessed using a stack frame.*

used by the remote system software. Programmatic logon operates within Hewlett-Packard's Distributed System Network.

Large Program Support

With RTE-A and VC+, HP 1000 A-Series Computers support new features that bring the seventeen-year-old HP 1000 architecture up to date. These features include support for large programs with up to 7.75 megabytes of code, the ability to load code into memory from disc on demand, support for reentrant code, and hardware support for procedure recursion that is up to six times faster than before.

The enhancement that makes all of this possible is called CDS, or code and data separation. A part of VC+, CDS consists of a little hardware and a little microcode. In the case of the A600, a new CPU board is required which contains new hardware and microcode. With the A700, only new microcode is required. In the A900, CDS is standard, and is implemented by a cache board and microcode. In all of the processors, a new set of virtual-control-panel PROMs is required. This new hardware supports a handful of new instructions and new interpretations of standard HP 1000 instructions.

The major effort in making CDS work, however, was in the operating system, languages, and system utilities. This enhancement to the architecture required changes in any software that has implicit knowledge of the hardware. The FORTRAN 77 and Pascal compilers are a good example, because they must generate code that makes use of CDS capabilities. The WH status-displaying program also must know about CDS, because it displays information such as the current code address of an executing program. The LINK program has to know how take the output of compilers and assemblers and create a program file.

The biggest software changes are in the operating system, which has to know how to save the state of a CDS program, how to dispatch a program that has both disc-resident and memory-resident components, how to react when a code segment fault occurs, and how to interpret the new procedure-calling convention defined by CDS.

In spite of all of these changes, the system is designed so that RTE-A can be offered without CDS capabilities, so customers without CDS-supporting hardware can still use RTE-A. This keeps the entry cost for RTE-A low. A customer only pays for CDS hardware and software when it is needed.

Stack Implementation

Traditionally, programs on the HP 1000 have stored all their code and all their data statically within the 64K-byte address space allowed. This implies that each subroutine that is in memory must have space for all of its variables in memory also. This limited address space means that only a few (usually less than 200) subroutines can reside in memory at the same time.

Part of the solution is to give the HP 1000 the ability to manipulate stack frames. This is accomplished by designing into the machine a Q-register which contains the address of the current stack frame, and instructions to take advantage of this register (see Fig. 4).

A stack frame is an area of memory used by a subroutine for the storage of local variables. This area is allocated upon entry to a procedure, and released when the procedure is exited. The advantage of this scheme is that only procedures that are currently active need to have memory space allocated to them.

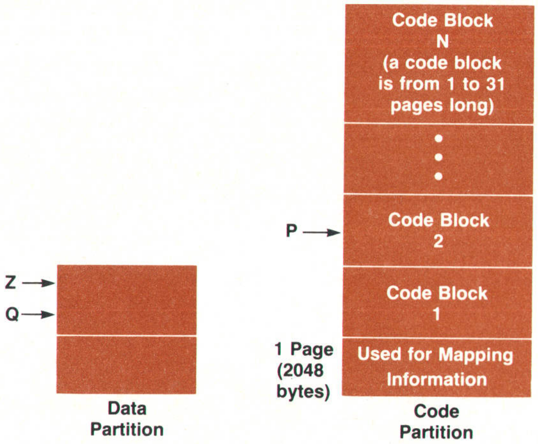


Fig. 5. *CDS allows many code segments to be mapped into the code partition. On demand, a code segment is loaded from the disc into a code block in memory. When the number of code segments exceeds the number of code blocks, replacement of the oldest block may occur.*

With this enhancement, when an address in the range 2 to 1023 is accessed, that access instead goes to the location Q+address. For example, if Q is 15000, and an attempt is made to access location 56, the actual location accessed is 15056.

Code and Data Separation

Another part of the solution to the problem of inadequate address space is solved by placing the code and the data into two separate address spaces. This was accomplished by carefully examining the instruction set and modifying those instructions that refer to either code or data. Instructions that refer to data (e.g., LDA) now access the data segment and those that refer to code (e.g., JMP) use the code segment. Each segment is an address space that can be logically mapped with one set of dynamic mapping system (DMS) maps. Thus separation is accomplished by using separate DMS maps for code and data.

Although separating code and data immediately doubles the available address space, the main benefit is that it effectively multiplies it many times. Once code and data are separated, the only subroutine that needs to be mapped in is the currently active one. This means that a program can have many more procedures than will fit into one segment by having many segments, each with a number of subroutines. Of course, a method had to be developed to allow a subroutine in one segment to call a subroutine in another segment and return. This is done by the PCAL instructions described later in this article.

As a result of these features, the effective limit on a program is that the active procedures (or more accurately, the sum of the local space used by the active procedures) must fit into the address space left for the stack in the data segment (Fig. 4). In other words, the limit is not code, but data.

Compatibility of CDS Code

An important goal of these enhancements was to allow a large degree of compatibility with previous HP 1000s. This was achieved in a variety of ways.

First, programs that would run under the earlier version of RTE-A (RTE-A.1) will also run under the latest RTE-A. This is compatibility at the relocatable level. The program must merely be linked to run on the new system. Such a program cannot, of course, use the new CDS features.

Another level of compatibility is at the source level. Programs written in FORTRAN can be recompiled using the CDS option of the FORTRAN compiler, and it will output CDS relocatable code, which can now be linked and run. One drawback is that CDS and non-CDS FORTRAN programs use different storage classes for local variables (i.e., stack for CDS and static for non-CDS). Although this shouldn't make a difference, programs that rely on nonstandard features may not work as they did before conversion. Assembly programs must be converted manually for CDS mode.

One more level of compatibility is the fact that CDS and non-CDS code can coexist in the same program. This means, for example, that a program can be converted without necessarily converting all the libraries it calls. The major restriction here is that non-CDS code is not allowed to call CDS code.

The linker for the system will automatically segment a program for the user, thus freeing the programmer from having to think about where subroutines lie in memory. Options exist to control the loading order if desired.

Code Segment Mapping

The CDS scheme supports both memory- and disc-resident code segments. This flexibility allows users to balance program memory requirements with program performance requirements. For situations where performance is critical, all code segments may be memory-resident. Where performance is less important or available memory is limited, code segments may be disc-resident with a limited number of segments occupying memory at any time.

Under RTE-A, a CDS program requires two memory partitions, one for code segments and one for data. The partitions are allocated independently from available memory. The code partition is divided into code blocks, equal subdivisions of sufficient size to contain the program's largest code segment (Fig. 5). The number of code blocks determines the number of code segments that may be concurrently memory-resident. Once a program begins execution, the number of code blocks may not be changed. Also included in the code partition is one page (2K bytes) that contains CDS data structures. This page is used by the operating system and the microcode.

When a CDS program is scheduled for execution, the memory manager allocates the necessary partitions. The code segment containing the start point of the program, as specified by the linker, is loaded from disc into the first code block. Code segments are not loaded from disc into any remaining code blocks, since it is not known which code segment(s) will be referenced.

As the program executes, it will call a procedure located in a different segment. The PCAL microcode, reading the system tables, determines if the referenced code segment is currently memory-resident. If so, the microcode alters the program's mapping registers to describe the called segment rather than the calling segment. When the called routine completes, the EXIT microcode restores the mapping registers.

If the referenced code segment is not currently memory-resident, the microcode invokes the code segment fault handler. The fault handler is part of the operating system, and is entered by an interrupt generated by the microcode. The fault handler determines if an unused code block is available to contain the referenced code segment. If it is, the segment is loaded from disc into the block. The PCAL instruction that initially caused the segment fault is reexecuted, and this time the microcode simply maps in the called segment. A similar action takes place when the EXIT instruction is executed.

In cases where the number of code segments exceeds the number of code blocks in the partition, the fault handler uses a round-robin algorithm to manage the code blocks. That is, the least-recently-loaded code segment is overlaid with the new code segment. There are several exceptions, however. Code segments may be marked as memory-locked, meaning that they are not candidates for replacement. Also, whenever possible, the calling segment is not overlaid with the called segment, since upon procedure EXIT the calling segment will be needed.

The PCAL that caused the fault cannot be reexecuted if the calling code segment has been replaced by the called segment. This situation arises if the code partition contains only one code block, or if memory-locked code segments occupy all the code blocks. Since the microcode cannot be reinvoked to accomplish the PCAL, the PCAL function is then performed by the segment fault handler after the called code segment has been loaded from disc.

Acknowledgments

The authors would like to thank project members Don Bollinger, Kevin Morgan, Naomi Munekawa, Russ Scadina, and Terry Leuthner for their contributions and suggestions. Special thanks to Nancy Schoendorf, project manager, who led and guided the project team to the successful introduction of the RTE-A operating system.

New Software Increases Capabilities of Logic Timing Analyzer

An upgraded operating software package increases the capabilities of an already powerful timing analyzer system to include statistics, marked events, postprocessing, and storage of captured trace data.

by David L. Neuder

BEING ABLE TO UPGRADE the operating software of an instrument aids in keeping a product in the forefront of a competitive market. The addition of new software features should be considered whenever they can make a significant contribution to the instrument. In examining the timing analyzer market, it was clear that there were some functions needed that traditional logic analyzers could not perform. This was the impetus for pursuing an increased feature set for the HP 64600S Logic Timing Analyzer, a subsystem used in HP's 64000 Logic Development System.[1]

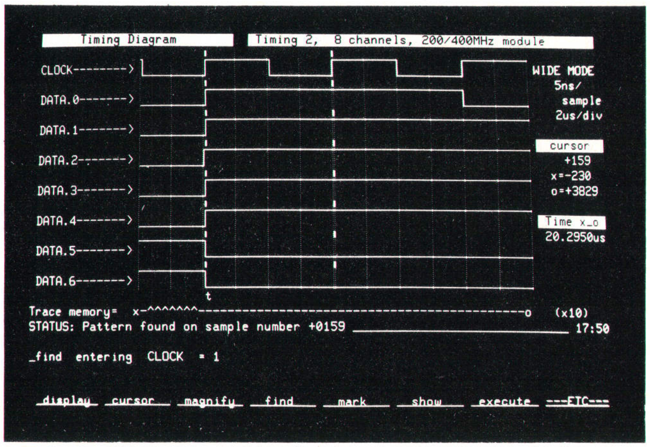
The new features are primarily associated with processing captured trace data for specific conditions, and then either calculating statistics or altering analyzer operation based on the conditions found. These features allow results to be determined more rapidly by providing more processing power to the user, and subsequently reducing the number of command operations and the amount of data manipulation required to determine a result. These features are all implemented through software changes only. There are no changes to the existing hardware and consequently, these features can be added at a minimal cost to the owner of an HP 64600S.

To review, the HP 64600S Timing Analyzer is an instrument system dedicated to the primary task of tracing signal flow on eight to sixteen channels simultaneously.[2] This is accomplished by asynchronously sampling the input channels at selected speeds between 2 Hz and 400 MHz and producing a timing diagram as output. A main feature is precise sampling of data channels with respect to time (low skew between channels), which allows a high degree of resolution of displayed waveforms. This allows signal relationships such as edges, levels, and sequences to be examined in fine detail.

What Users Wanted

Ideas for new timing analysis features came from marketing research, from analysis of competitive analyzer products, and most important, from the users of the 64600S. One enhancement suggested was the ability to find a specified event in trace memory. Users reported that scrolling the screen to locate a particular event visually was a time-consuming process; they suggested that the analyzer's microprocessor be put to work to find the specified events. Users also requested an automatic time interval function. They wanted to make a series of measurements of the duration of a pulse or the time between two edges of two signals. Discussion about simplifying the sequence of steps and commands for time interval measurement led to an idea of automatically marking the time interval by assigning events (patterns with transition or duration qualifications) to each of the 64600S's existing interval cursors or marks. These events would be found after each execution of the analyzer and the appropriate interval would be marked and measured. Then, in addition to accumulating a series of measurements for the interval, a statistical package could be added to determine the maximum, minimum, mean, and standard deviation of the interval values. Other users requested a halt for the automatic interval measurement whenever an interval exceeds or is less than a specified value so that the conditions associated with the out-of-specification interval could be studied.

Another feature users wanted was a count of the number of pulses between specified start and stop points on a timing trace. The 64600S's x and o markers could be used for the start and stop points and a new mark added to be located on each occurrence of a specified event. Thus, each occurrence of the rising edge of a pulse could be marked. Then the number of marks or pulses could be displayed and accumulated into statistics for a series of runs. The ability to halt the measurements if the number of counted



```
Timing Diagram          Timing 2,  8 channels, 200/400MHz module
Trace memory=  x-^^^^^^^-----------------------------------------o   (x10)
STATUS: Pattern found on sample number +0159 ____________________  17:50

_find  entering  CLOCK  = 1
```

Fig. 1. *Timing diagram display for finding the rising edge of a signal labeled* CLOCK.

events exceeds or is less than some value could then be used to study out-of-specification counts.

It was found that users also need to convert an asynchronous timing trace list of captured state flow into a pseudosynchronous listing of the actual state flow. A desired result would be a listing of the state flow and the relative time between states without the duplicate and transition samples associated with asynchronous sampling. One approach takes advantage of the fact that most state machines have a clock associated with the data. The idea is to probe both data and clock lines with a timing analyzer and produce a trace. The trace is then presented by storing only one sample for each transition of the designated clock. This stored sample is some specified number of samples away from the clock transition to ensure that the proper setup time of the clock is met. The result is a listing of clocked states. Of course, the sampling rate of the timing analyzer must be high enough to ensure that the input channel designated as the clock is sampled in both its high and low states. A second approach to producing state flow assumes that each state exists for some minimal amount of time. Then, by storing one sample for each state that lasts longer than this minimal time, a trace list can be produced that removes duplicate and transition samples, leaving only the state flow and the relative time between states.

Storing measurement data for later analysis would give the user the ability to store measurements at one site (test site) and do the analysis of the stored measurements at another site (home site). This idea led to the concept of visually comparing two stored measurements or comparing a stored measurement to a current measurement on the same screen at the same time.

After considering these and other user needs, an updated software package for the 64600S was developed. The new features provided by this package include:

- Finding specified events in the data acquisition memory
- Automatic marking of specified events in data acquisition memory
- Calculating statistics on marked events
- Using marked events to qualify execution rerun
- Processing asynchronous trace list data into pseudosynchronous state listings
- Storing measurement data along with the system configuration
- Visually comparing stored and current measurements.

Building on Earlier Commands

Adding the new features to the existing 64600S software while still retaining the simplicity and ease of use of this timing analyzer was a major design goal. This goal was achievable because the commands in the original version of the 64600S software allow for later expansion. For example, the keyword find was implemented in the original version and the new find features were implemented by expanding the syntax for this command. The original keyword performed a subset of the most recent version's functions, including finding the trigger, which was always

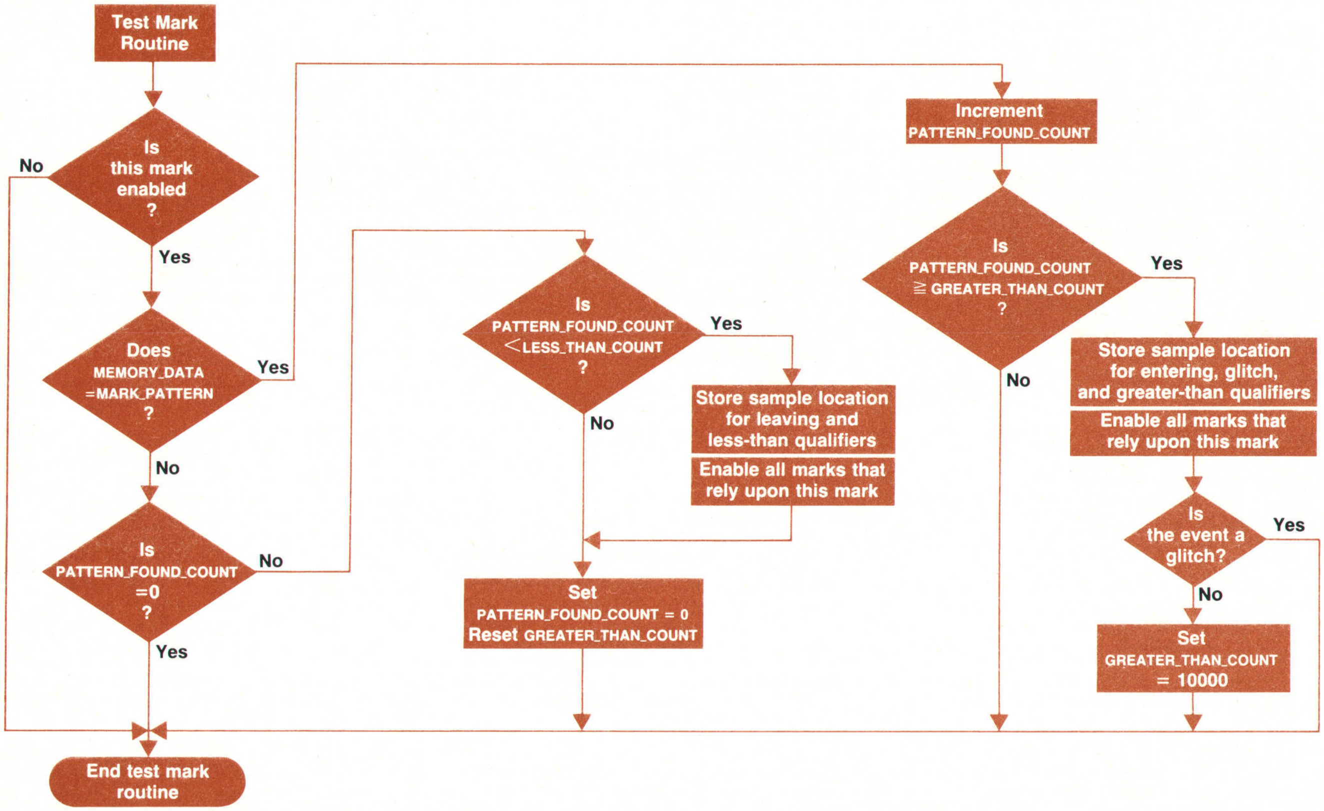

Test Mark Routine
Is this mark enabled ?
No
Yes
Does MEMORY_DATA =MARK_PATTERN ?
Yes
No
Is PATTERN_FOUND_COUNT =0 ?
No
Yes
End test mark routine
Is PATTERN_FOUND_COUNT < LESS_THAN_COUNT ?
Yes
No
Store sample location for leaving and less-than qualifiers
Enable all marks that rely upon this mark
Set PATTERN_FOUND_COUNT = 0 Reset GREATER_THAN_COUNT
Increment PATTERN_FOUND_COUNT
Is PATTERN_FOUND_COUNT ≧ GREATER_THAN_COUNT ?
Yes
No
Store sample location for entering, glitch, and greater-than qualifiers
Enable all marks that rely upon this mark
Is the event a glitch?
Yes
No
Set GREATER_THAN_COUNT = 10000

Fig. 2. *Flowchart of the software marking operation.*

associated with sample zero, or finding a mark, which was always associated with some sample number. Hence, when the new feature of finding an event in the associated trace memory was to be implemented, the find softkey was already present and clearly conveyed the meaning of this new feature.

To make the new software features easy to understand, the new command sequences closely parallel earlier commands. As an example, consider the trigger command which allows the analyzer to be triggered on transitions into patterns and on durations of patterns. When we implemented the new find commands, we used the trigger command syntax to describe an event. Thus, the find syntax is identical to the trigger syntax, and can be easily understood by users accustomed to the earlier 64600S software (see Table I).

Finding Events

This new feature allows the 64600S to search its trace memory for user-specified events. The find command is similar to the trigger command in that it allows the user to identify transitions into specified states, durations of specified states, and glitches on specified channels. The find software searches the trace memory for the specified event, and when it is found, positions the display so that the event is centered on the display. If the specified event is not found, the message Pattern not found appears and the display does not change. The find software searches forward and/or backward from the cursor as directed by the user until it locates the next occurrence of the specified event or the end of the search range is encountered. Fig. 1 shows the result of finding the rising edge of CLOCK through the use of the command find entering CLOCK = 1. Observe that the cursor is now located at the sample where the condition is found and that this sample number is noted in the message Pattern found on sample +0159.

Automatic Marking

The most common use for any timing analyzer, besides examining signal functionality, is to measure the time intervals between specified events. The standard approach to measuring time intervals is to position marks or cursors at the endpoints of the interval to be measured, and then read from the display the listed time between the marks. This is possible because the internal sampling rate of the analyzer is stable and known and the number of samples between the marks can be counted. The positioning of the marks typically involves scrolling the display screen to each of the desired endpoints and placing each mark at the specified point via a command sequence.

Table I

Trigger and Find Command Syntax

```
trigger on ⟨transition_or_level_duration⟩ ⟨LABEL⟩ = ⟨PATTERN⟩
find       ⟨transition_or_level_duration⟩ ⟨LABEL⟩ = ⟨PATTERN⟩
           ⟨transition_or_level_duration⟩ =
                  entering
                  leaving
                  any_glitch on ⟨LABEL⟩ with
                  greater_than ⟨TIME⟩ ⟨time_unit⟩_of
                  less_than ⟨TIME⟩ ⟨time_unit⟩ of
                                 nsec
                                 μsec
                                 msec
```

The new 64600S command mark on_first_occurrence_of executes finding and marking in one step. In addition, this new command will initiate an automatic search for the desired condition after every execution and mark the trace data accordingly. As an example, consider entering the commands listed below.

```
mark x on_first_occurrence_of entering CLOCK = 1 after trigger
mark o on_first_occurrence_of entering CLOCK = 1 after mark_x

execute        Display shows: Time x_o 265.0 nsec
execute        Display shows: Time x_o 270.0 nsec
```

After each execution, the 64600S's x and o marks are assigned as specified in the above definition, the time interval is automatically calculated, and the display shows the value.

Marking timing data is an operation in which specified conditions are searched for in the acquired timing trace data. The found conditions are then labeled with the appropriate labels (x, o, a, b, c, and d in the 64600S Timing Analyzer) to indicate to the user where these conditions were found.

Marking occurs after the timing analyzer has completed an execution and the execution has filled a trace memory with data. The marking operation begins with the unloading of all trace memory data into a RAM area. This unloading into RAM enables faster marking by allowing the system microprocessor faster access to the trace data. Next, the mark searching routine is initialized. Each mark has associated with it four variables that must be set to appropriate values for the searching routine to find the specified mark. First, GREATER_THAN_COUNT and LESS_THAN_COUNT must be initialized to the appropriate values as shown in Table II to match the specified mark condition. Second, PATTERN_FOUND_COUNT must be set to zero. This variable indicates the number of times the specified pattern is found. Third, the MARK_PATTERN for each mark must be determined. This is the pattern of ones, zeros, and don't cares to be found.

After mark initialization, a loop is entered in which each trace memory sample is sequentially compared with each MARK_PATTERN to determine if the corresponding mark can be found. The flowchart of the marking operation is shown in Fig. 2. This routine is called once for each mark for each sample of MEMORY_DATA for a total of 6 × 4060 times (6 × 8140 times in fast sample mode). This is where the tests are made to determine not only if MEMORY_DATA matches MARK_PATTERN, but also if the conditional qualifiers (entering, leaving, etc) of the specified mark are met. These qualifiers are specified by the two variables GREATER_THAN_COUNT and LESS_THAN_ COUNT according to their values in Table II.

After the entire trace memory is searched for marked events and the appropriate mark locations are stored, the time interval is calculated by determining the number of samples between the marks x and o and multiplying the number by the sample period. Finally, the trace data, the marked locations, and the time interval are displayed.

Statistics

The time interval measurements can now be accumulated for statistical evaluation of the interval. Following each execution (acquiring data, automatic marking, time interval calculation, and trace display), a maximum, minimum, mean, and standard deviation of the specified interval are computed and displayed. This capability is useful in circuit characterization. It allows the user to obtain a "feel" for the stability of a time interval such as setup and hold times, pulse width variations, interrupt-to-acknowledge response times, and propagation delays.

As an example, consider a case of characterizing a circuit, where it is desirable to measure the propagation delay between two signals called CLOCK and Q_OUT (CLOCK always occurs before Q_OUT). Now we can set up the appropriate marking to occur by entering the commands: mark x on_first_occurrence_ of entering CLOCK = 1 after trigger, and mark o on_first_occurrence_of entering Q_OUT = 1 after mark_x. We can turn on the statistics capability with the command indicate time_interval_x_o mean_and_standard_deviation. The trigger is set on the rising edge of Q_IN by the command trigger on entering Q_IN = 1. Then, by executing the analyzer repetitively using the command execute repetitively, the statistics can be calculated. Typical results for this type of measurement after 100 runs are shown in Table III.

One problem encountered during the design of the new 64600S software involved calculating statistics on a series of runs. The desired performance was to calculate statistics on as many runs as possible, as fast as possible, with a factor of ten increase in accuracy (i.e., mean accuracy = sample period/10) with no counter overflow. Originally, 32-bit integer arithmetic was selected as the method for doing the statistics, because the routines already existed and had the necessary speed. And, the increase in accuracy could be simply obtained by multiplying all values by a factor of ten before calculations and dividing the results by ten. But, with 32-bit arithmetic, the accumulators could possibly overflow after 77 runs. Further investigation showed that only a few 48-bit routines would be required to allow the accumulation of run statistics for up to 1000 runs. Thus, specialized 48-bit addition, subtraction, multiplication, and division routines were developed. To determine standard deviation, a 32-bit integer square root routine was developed. Thus, the statistical package for the 64600S fits the requirements of high-speed, accuracy, and the ability to accumulate statistics for up to 1000 runs of data.

Statistics can improve the measured accuracy when measuring repetitive and stable waveforms.[3] The accuracy of a timing analyzer can be defined by the equation: accuracy = ± (sample period + worst-case cross-pod channel-to-channel skew). When an interval is measured a number of times (N) and the sampling period of the analyzer is not a multiple of the period of the input interval to be measured, this equation can be rewritten to take into account the multiple runs, that is, accuracy = ±(sample period/$\sqrt{N}$ + channel-to-channel skew). This equation reflects the accuracy of the measured mean of the interval after N measurements. As an example, consider measuring a stable interval of approximately 2 μs with a sampling period of 25 ns and a skew of 3 ns. After one execution and measurement of the interval, the accuracy is ±(25 ns + 3 ns) = ± 28 ns. Now, by measuring the interval 100 times, the accuracy becomes ±(25 ns/$\sqrt{100}$ + 3 ns) = ± 5.5 ns.

Table II

Marking Condition Variable Values

Qualifier	GREATER_THAN_COUNT	LESS_THAN_COUNT
Entering	1	0
Leaving	10000	10000
Greater-than	Duration in samples + 1	0
Less-than	10000	Duration in samples
Any-glitch	1	0

Marked Events

At times it is convenient to mark each occurrence of a specified event to make it easier to locate and count the events. In addition to marking a time interval automatically, the new software for the 64600S provides the capability of marking four distinct kinds of events a multiple number of times, up to a total of 511 marks. An example of the usefulness of this capability might be marking the condition of the status lines entering a particular state. Using the following commands, we can produce the timing diagram shown in Fig. 3a with each of four microprocessor states marked (vertical bars).

```
mark a on_all_occurrences_of entering S0=1 and S1=0 and IO_M=0
        named Mem_write
mark b on_all_occurrences_of entering S0=0 and S1=1 and IO_M=0
        named Mem_read
mark c on_all_occurrences_of entering S0=1 and S1=1 and IO_M=0
        named Op_fetch
mark d on_all_occurrences_of greater_than 20 nsec_of S0=1 and
        S1=1 and IO_M=1 named Int_ack
```

By invoking another new 64600S command, process_for_data marked, only marked samples are displayed for a more readable trace list (Fig. 3b). In Fig. 3b, time count rel indicates the time between the marked samples, a feature useful for determining the time between processor states and observing the processor flow.

Table III

Typical Statistical Time Interval Results

Display	Output	Comment
Time x_o	45.0 nsec	100th time interval of x to o
Maximum	55.0 nsec	Maximum interval over 100 runs
Minimum	40.0 nsec	Minimum interval over 100 runs
Mean	42.5 nsec	Average interval after 100 runs
Stdv	4.0 nsec	Standard deviation about the mean
Runs	100	Number of runs and intervals counted

Captured Data Storage and Retrieval

In implementing the new software features of the HP 64600S Timing Analyzer, we were faced with a number of critical software concerns. One key concern was the ability to display a waveform from stored data. Because the input hardware of the analyzer could not be reloaded with the previously captured data, and since we did not have sufficient RAM available to store the measurement, it became apparent that the stored data would have to reside externally on a hard or flexible disc. Various data storage approaches were studied, but an approach that makes the data look similar to the input data was finally selected. This structure is basically a serial channel-to-channel format, which although not ideally suited for decoding into a trace list, is ideal for timing diagram display and disc space utilization. The format stores 16 channels of data, 256 words per channel, 16 data points per word, where the least-significant bit is the earliest data and the most-significant bit is the latest data. The desired format for the trace list, however, is an array of 4096 samples, 16 bits per sample, one bit per channel, where the least-significant bit is channel 0 data and the most-significant bit is channel 15 data.

One problem in reading the data from the disc memory was that each time the user scrolled the display screen, the disc would have to be accessed for more data. This was resolved by unloading enough data to allow the user to scroll through three pages of timing diagram and five pages of trace list before another disc access is required. When more data is required, another three to five pages of information is unloaded with the current display page being the center page.

Another key issue was to find/mark a specified event in a timing diagram or trace list as quickly as possible using either data acquisition memory or stored data. In finding such an event, a search occurs sample by sample in a serial fashion across all channels. Normally, this would seem to be an easy process. But, since the timing analyzer data is stored on a channel-by-channel basis as outlined above, the data of a given sample on one channel must be combined with the data from the same sample across all other channels. This building of a 16-bit word to represent the data across all 16 channels is a bit-manipulation process which is slowed by the required channel-controlled access to data acquisition memory. Initial estimates of building this word 4096 times (as would be required in a total memory search) were on the order of 1.5 s. This amount of time included no associated overhead, but it was felt that the maximum time constraint of about 3.0 s to find/mark specified events would be met.

However, another concern arose. Since the stored measurements were kept on the disc also in a channel-by-channel format, it would be even more difficult and slow to build a sample word across all 16 channels directly from the disc, because each channel record would have to be read once to extract each sample word—a total of 4096 times. This process was never attempted. Hence, it was apparent that the disc data would have to be unloaded into processor RAM before the sample words were built and the search for events could occur. But again, RAM space was at a premium and 4K of space just did not exist. The solution was to designate another overlay (see box on page 38) to the find and marking process. This overlay returns to the main display overlay the locations of the specified events or found events. In this designated overlay, there exists plenty of room to unload the data completely from the disc storage and subsequently build the sample words. In addition, by using this overlay structure and unloading data acquisition memory before building a sample word, it was found that 4096 samples could be built in about 200 ms, a significant improvement over 1.5 s!

Qualifying Data

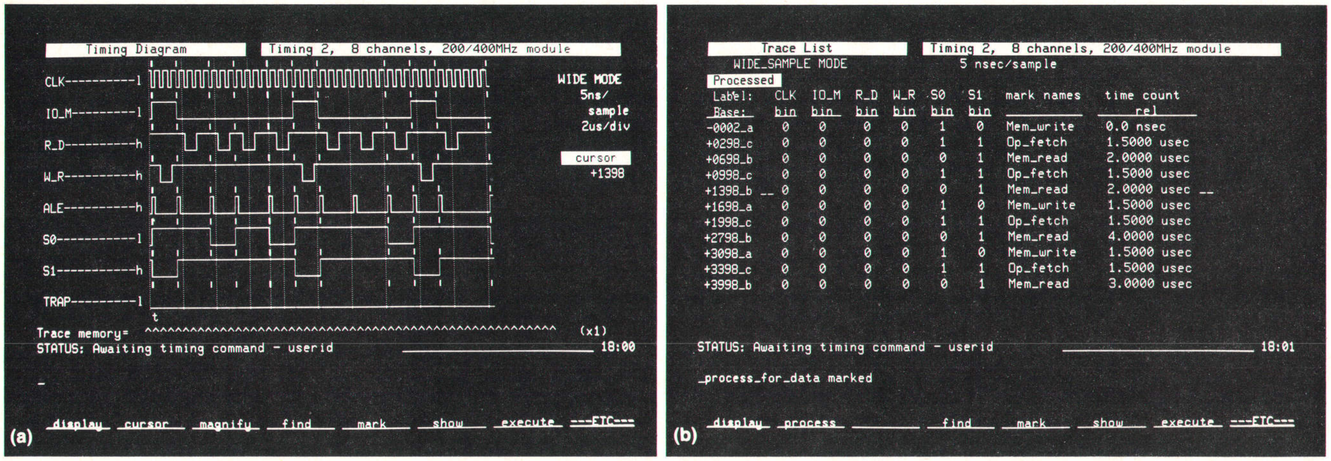
Timing analyzers incorporate a fairly extensive set of triggering capabilities for examining circuit characteristics, but they are inherently limited in certain triggering capabilities such as occurrence counting, sequential triggering, and duration triggering with resolution dependent on the sample period. One approach to expand triggering capabilities is through processing the captured data with internal software to search for trigger-like conditions. These conditions qualify the current captured data by inhibiting further collection of data. The 64600S can qualify the captured data in four ways to determine if another execution is to occur. These qualification procedures include duration qualify, count qualify, sequence qualify, and run number qualify.

Duration qualify halts repetitive execution when a time interval marked by x and o is greater than or less than a specified amount. Count qualify operates in a similar manner, except that the number of marks between x and o is counted and compared with a qualify number. When the count is greater than or less than the reference number specified, the execution of the analyzer is halted. Sequence qualify searches for a sequence of up to four marks between the x and o marks, and halts execution when the sequence is found. A typical sequence qualify command might be halt_repetitive_execution when_sequence_x_o_is mark_a then mark_b then_not mark_c. This halts the repetitive execution when the sequence is found anywhere between mark x and mark o. Remember that the time, count, and sequence qualification occurs after the trace data is captured and the analyzer is paused (not acquiring data). Therefore, if a specified event occurs while the analyzer is paused, the analyzer cannot respond to that stimulus. But when the analyzer is started again, it will again be capable of storing the stimulus and again be able to search for the halt condition.

Postprocessing Data

A timing analyzer typically presents data in the form of a timing diagram. An alternative display method is a trace list. Newer analyzers can have trace lists of more than 4K samples of data. Isolating information from a list of more than 4K samples can be a difficult task. Therefore, the new 64600S software adds a series of new commands to reduce the amount of data presented and at the same time, retain and make clearer the significant information. These new commands allow the user to observe marked samples, state flow, and clocked state flow after every execution.

The new command process_for_data greater_than ⟨TIME⟩ ⟨time_unit⟩ presents only one sample for each sequence of samples with the same data that exceeds a specified time duration. As an example, consider the process of trying to look at a high-speed 7-bit-wide data channel labeled DATA running at 40 MHz. Assume that the signals on DATA are stable for at least 20 ns (the signals take 5 ns to change states). In this example we will set up the timing analyzer to sample every 5 ns (200 MHz) by entering the command sample period_is 5 nsec. Fig. 4a is a typical listing after the analyzer executes and before postprocessing. Now, by entering the command process_for_data greater_than 10 nsec on DATA, the trace list is processed into a more readable form showing each DATA state as shown in Fig. 4b. Note that only one



```
Timing Diagram          Timing 2,  8 channels, 200/400MHz module

Trace memory=  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  (x1)
STATUS: Awaiting timing command - userid              ____________________ 18:00

_

 display   cursor   magnify   find   mark   show   execute  ---ETC---
```

```
          Trace List               Timing 2,  8 channels, 200/400MHz module
    WIDE_SAMPLE MODE                         5 nsec/sample
 Processed
 Label:     CLK  IO_M  R_D  W_R  S0   S1   mark names    time count
 Base:      bin  bin   bin  bin  bin  bin  __________    ____rel_____
 -0002_a     0    0     0    0    1    0   Mem_write     0.0 nsec
 +0298_c     0    0     0    0    1    1   Op_fetch      1.5000 usec
 +0698_b     0    0     0    0    0    1   Mem_read      2.0000 usec
 +0998_c     0    0     0    0    1    1   Op_fetch      1.5000 usec
 +1398_b __  0    0     0    0    0    1   Mem_read      2.0000 usec __
 +1698_a     0    0     0    0    1    0   Mem_write     1.5000 usec
 +1998_c     0    0     0    0    1    1   Op_fetch      1.5000 usec
 +2798_b     0    0     0    0    0    1   Mem_read      4.0000 usec
 +3098_a     0    0     0    0    1    0   Mem_write     1.5000 usec
 +3398_c     0    0     0    0    1    1   Op_fetch      1.5000 usec
 +3998_b     0    0     0    0    0    1   Mem_read      3.0000 usec


STATUS: Awaiting timing command - userid              ____________________ 18:01

_process_for_data marked

 display   process   ________   find   mark   show   execute  ---ETC---
```

Fig. 3. *Using the commands given in the text, a timing diagram marking four microprocessor states can be displayed (a). Using the new* display_for_data marked *command, a trace list of only the marked samples can be displayed (b).*

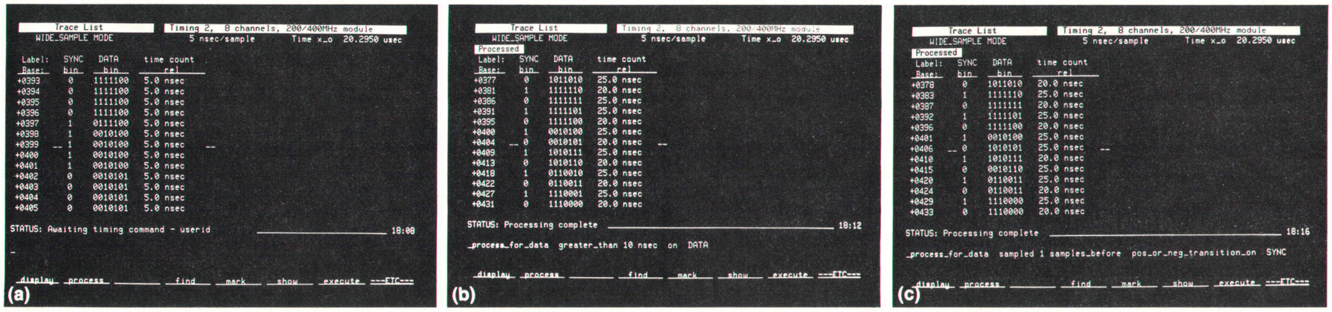
sample of each state is shown and that the relative time between these samples is also displayed.

Another new command, process_for_data sampled ⟨SAMPLES⟩ samples_⟨before_after⟩⟨pos_neg_transition⟩_on ⟨LABEL⟩, reduces the data in the trace list by only storing one sample for each transition on a specified channel. With this command, the user selects one of the asynchronously captured inputs as a clock. A clock edge and a number of samples before or after the clock are specified. This allows the user to specify a setup time that matches the characteristics of the data under test. The analyzer then processes the clock signal for the specified transition and enters the specified sample in the trace list. As an example, again consider the 7-bit-wide high-speed DATA channel. Assume that the channel has a signal SYNC that changes polarity with each valid DATA. Now, if we assume that DATA was valid for at least 6.5 ns before the positive or negative edge of SYNC and at least 1.5 ns after the same transition, we could set up a command process_for_data sampled 1 samples_before pos_or_neg_transition_on SYNC. Note that we are again assuming the same sampling period (5 ns) as the previous example. The trace list of Fig. 4a will then be processed to produce the trace list shown in Fig. 4c. The timing analyzer has stored each sample that occurs one sample before the positive or negative transition on the SYNC line.

Determining the maximum usable data rate of this feature places requirements on the signals that constitute the data and the clock. The clock must be sampled once high and once low, and therefore, must be present for at least one sample period plus the skew on a single channel (5 ns + 1.5 ns = 6.5 ns for both high and low levels). This gives a minimum clock period of 13 ns (maximum clock rate of 77 MHz) when sampling the data at 200 MHz. To sample the data accurately one sample before the clock, it must be present for a minimum of 8.0 ns. This results from a setup time equal to one sample period plus skew, and a hold time equal to skew (5 ns + 1.5 ns + 1.5 ns = 8 ns). Therefore, with a minimum clock period of 13 ns and a minimum required stable data time of 8 ns, the maximum effective sampling rate is 77 MHz with 5 ns available for the data to change. For a sixteen-channel timing analyzer with a worst-case skew of 3 ns (cross-pod channel-to-channel skew), the above analysis yields 77 MHz with 2 ns available to change data or 50 MHz with 9 ns available to change data.

Storing Data and System Configuration

Earlier analyzers could store the current setup, but rarely could they store the captured data. The 64600S Timing Analyzer now allows the user to store both the current configuration and the captured data in a file for later



```
       Trace List            Timing 2,  8 channels, 200/400MHz module
   WIDE_SAMPLE MODE                5 nsec/sample        Time x_o  20.2950 usec

 Label:   SYNC   DATA      time count
 Base:    bin    bin      ____rel____
 +0393     0    1111100   5.0 nsec
 +0394     0    1111100   5.0 nsec
 +0395     0    1111100   5.0 nsec
 +0396     0    1111100   5.0 nsec
 +0397     1    0111100   5.0 nsec
 +0398     1    0010100   5.0 nsec
 +0399  __ 1    0010100   5.0 nsec   __
 +0400     1    0010100   5.0 nsec
 +0401     1    0010100   5.0 nsec
 +0402     0    0010101   5.0 nsec
 +0403     0    0010101   5.0 nsec
 +0404     0    0010101   5.0 nsec
 +0405     0    0010101   5.0 nsec
STATUS: Awaiting timing command - userid          ____________________ 18:08

_

 display   process   ________   find   mark   show   execute  ---ETC---
```

```
       Trace List            Timing 2,  8 channels, 200/400MHz module
   WIDE_SAMPLE MODE                5 nsec/sample        Time x_o  20.2950 usec
 Processed
 Label:   SYNC   DATA      time count
 Base:    bin    bin      ____rel____
 +0377     0    1011010   25.0 nsec
 +0381     1    1111110   20.0 nsec
 +0386     0    1111111   25.0 nsec
 +0391     1    1111101   25.0 nsec
 +0395     0    1111100   20.0 nsec
 +0400     1    0010100   25.0 nsec
 +0404  __ 0    0010101   20.0 nsec   __
 +0409     1    1010111   25.0 nsec
 +0413     0    1010110   20.0 nsec
 +0418     1    0110010   25.0 nsec
 +0422     0    0110011   20.0 nsec
 +0427     1    1110001   25.0 nsec
 +0431     0    1110000   20.0 nsec
STATUS: Processing complete  ______________________________________ 18:12

_process_for_data  greater_than 10 nsec  on  DATA

 display   process   ________   find   mark   show   execute  ---ETC---
```

```
       Trace List            Timing 2,  8 channels, 200/400MHz module
   WIDE_SAMPLE MODE                5 nsec/sample        Time x_o  20.2950 usec
 Processed
 Label:   SYNC   DATA      time count
 Base:    bin    bin      ____rel____
 +0378     0    1011010   20.0 nsec
 +0383     1    1111110   25.0 nsec
 +0387     0    1111111   20.0 nsec
 +0392     1    1111101   25.0 nsec
 +0396     0    1111100   20.0 nsec
 +0401     1    0010100   25.0 nsec
 +0406  __ 0    1010101   25.0 nsec   __
 +0410     1    1010111   20.0 nsec
 +0415     0    0010110   25.0 nsec
 +0420     1    0110011   25.0 nsec
 +0424     0    0110011   20.0 nsec
 +0429     1    1110000   25.0 nsec
 +0433     0    1110000   20.0 nsec
STATUS: Processing complete  ______________________________________ 18:16

_process_for_data  sampled 1 samples_before  pos_or_neg_transition_on  SYNC

 display   process   ________   find   mark   show   execute  ---ETC---
```

Fig. 4. *(a) Trace list of high-speed data channel before postprocessing. (b) Trace list of (a) processed for data greater than 10 ns on the* DATA *lines. (c) Trace list of (a) processed for data one sample before a positive or negative transition on* SYNC*.*

analysis. Further, this data is stored in such a way that when the analyzer is reloaded, the data can be processed as if the user had never left the analyzer or the analyzer had never been turned off. In short, the full analysis capabilities of the find, mark, process, and display commands are available to be used on the stored data. The command configuration load_from ⟨FILE⟩ reloads the configuration and the associated data. Advantages of this capability include being able to process data captured at a remote site and to document data and analysis, and the convenience of being able to analyze a measurement later.

An additional and important advantage of being able to store data is that it can be retrieved and displayed concurrently with freshly captured data. This allows a stored correct waveform to be placed on the screen and compared with a currently captured suspect waveform. The required sequence of commands to set up this feature are listed below, assuming that the file FILE contains a configuration that is similar to the current analyzer configuration and that FILE also contains stored data.

```
compare_file_is ⟨FILE⟩
display ⟨display_item⟩ then ⟨display_item⟩ then........
        ⟨display_item⟩ =LABEL⟩
                       = compare_file ⟨LABEL⟩
```

Note that to display stored data concurrently with newly captured data, the trace specifications must agree in some aspects—mode, trigger position, and sample period. The user can completely specify the ordering of the data in both the timing diagram and the trace list. A typical command might be display SYNC then compare_file SYNC then DATA .0 then compare_file DATA .0 then DATA.1 then compare_file DATA.1 then DATA.2 then compare file_DATA.2. This command produces a timing diagram as shown in Fig. 5. Note that the character x follows each of the labels that come from the specified compare file. Also note how the traces can be visually compared to find differences. All processing commands work in reference to the currently captured data and will only process the compare file to the extent that the same sample number that is processed in the current data will be processed in the compare file. This can be useful in comparing state flow of one trace with state flow of another, if both were captured with the same trigger.

Acknowledgments

Joel Zellmer contributed invaluable assistance in design, design review, and debugging all of the new 64600S software. Ted Hanna, John Scharrer, and Marvin Wilson also contributed to the feature design process. Bryce Goodwin contributed valuable software support.

References

1. *Hewlett-Packard Journal*, March 1983.

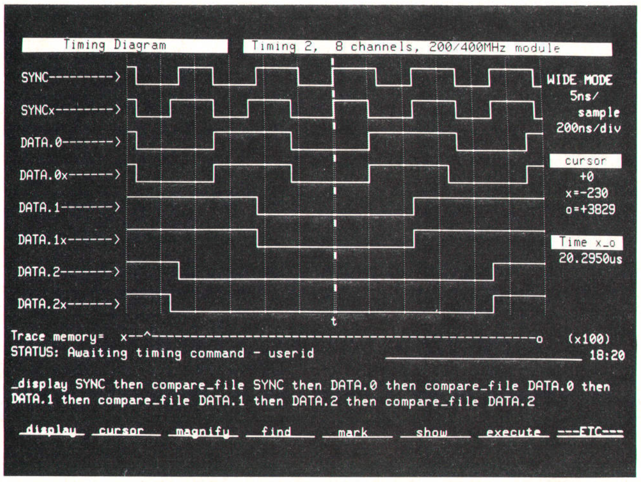
2. J.A. Zellmer, J.E. Hanna, and D.L. Neuder, "A Modular Timing Analyzer for the 64000 System," *ibid.*

3. *Time Interval Averaging*, Application Note 162-1, Hewlett-Packard Company.

Overlay Memory Structure

The HP 64600S Timing Analyzer has a processor RAM space of approximately 22K words, but the code to run the analyzer is approximately 95K words. Therefore, an overlay structure is used with a resident portion of code of 6K words and an overlay area of 16K words. The resident code contains the data structures that define the analyzer configuration, the utility routines used by all overlays, and the routines that control the overlay loading and execution. Typically, each overlay was partitioned to correspond with each display mode of the timing analyzer, where all the commands associated with a particular display mode are immediate in response (no additional overlays must be called to execute the command). With the addition of the new marking function, this is no longer true. Because not enough processor RAM space was available to unload the data acquisition memory or stored data, a separate overlay is needed just for the marking function. Thus, whenever a mark command is executed or the newly acquired data is to be marked, the current display overlay is swapped out and the mark overlay is placed in processor RAM. When marking is finished, the swapped-out display overlay is reinstalled and if required, the display is then updated.

One problem which had to be solved during the swapping of overlays occurred while a user was executing a measurement in repetitive mode where marking was requested. In this mode, a user would still like to be able to enter commands even though the code that responds to the commands is swapped out 50% of the time. Problems arose in that the command line would be initialized each time the display overlay was swapped back in. This made it difficult to execute even simple commands. The problem was solved by modifying the structure of the overlays so that 1) all variables of the display overlay are in high portions of memory so that marking overlay does not interfere, and 2) modifying the nature of the overlay swapping such that the call to the marking overlay is like a call to a procedure and correspondingly, the return to the display overlay is directly after the call. This leaves all the current variables of the display overlay untouched by the marking overlay. Therefore, any command that a user enters while the display overlay is swapped out is left intact. This gives the user the impression that the command response is a little slower because of the code swapping, but other than that, everything appears normal.



```
Trace memory=  x--^---------------------------------------------o   (x100)
STATUS: Awaiting timing command - userid                         18:20

_display SYNC then compare_file SYNC then DATA.0 then compare_file DATA.0 then
DATA.1 then compare_file DATA.1 then DATA.2 then compare_file DATA.2
```

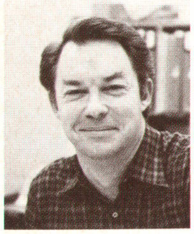
Fig. 5. *Timing diagram display illustrating new compare function.*

Authors

February 1984

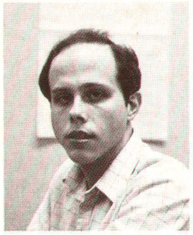
3

Don M. Cross



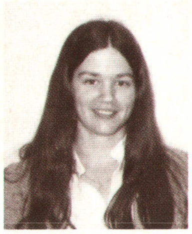
Don Cross joined HP 24 years ago as an assembly technician, then worked his way up through a variety of titles, including test technician, test equipment designer, design engineer, and project manager. He was project manager for the A700 Computer. A member of the IEEE Computer Society, He's a native of Phoenix, Arizona, and now lives in Los Altos, California. He's married, has two children in college, and his hobbies are woodworking and stained glass.

Craig B. Chatterton



With HP since 1976, Craig Chatterton has been a development engineer for HP 1000 F-Series Computers and served as project manager for the A900 Computer's microcode and diagnostics. He received his BSEE and MSEE degrees from the University of Illinois in 1975 and 1978, and is a member of the IEEE and the ACM. Now a resident of San Jose, California, Craig was born in Hinsdale, Illinois. His interests include playing the piano, singing with the HP chorus, and sailing his Laser.

Marlu E. Allan



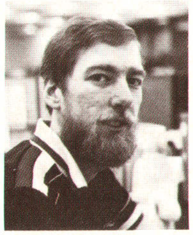
Now a project manager for manufacturing application software, Marlu Allan previously served as project manager for development of the A900 CPU. A graduate of the University of Michigan, she received her BS degree in computer engineering in 1977 and joined HP soon afterwards. She's married, lives in San Jose, California, and enjoys sports, especially golf, tennis, and basketball.

Nancy Schoendorf

Nancy Schoendorf received her BS degree in computer science and mathematics from Iowa State University in 1975. She began her HP career in 1976 as a software production engineer, then contributed to the development of the RTE-4B and RTE-XL operating systems and served as project manager for RTE-A.1 and RTE-A. In 1980 she received her MBA degree from the University of Santa Clara. Now a section manager with HP's Data Systems Division, she lives in Los Altos, California, has a one-year old daughter, and is married to another HP employee. She enjoys cooking, water skiing, traveling, and spending time with her family.

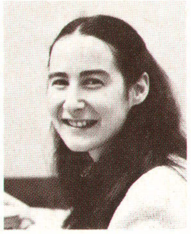
7

David A. Fotland



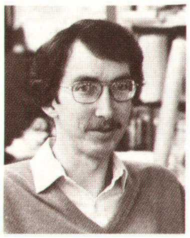
Dave Fotland is a native of Cleveland, Ohio, and a graduate of Case Western Reserve University. He received both a BS degree in electrical engineering and an MS in computer engineering in 1979, and joined HP the same year as an R&D engineer in the Data Systems Division lab. He designed one of the A700 processor boards and worked on RTE-A. Dave is married, lives in San Jose, California, and plays volleyball and go.

Leslie E. Neft



Leslie Neft graduated from Carnegie-Mellon University in 1978 with a BS degree in electrical and biomedical engineering. For the next five years or so, she designed hardware—including one of the A700 processor boards—and wrote drivers as an R&D engineer with HP's Data Systems Division. She's now in technical marketing. Leslie was born in Pittsburgh, Pennsylvania, and now lives in Cupertino, California. Her interests include volleyball, bicycling, and ballet.

Lee S. Moncton



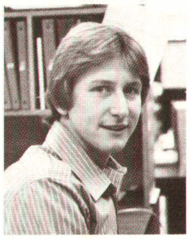
Lee Moncton is a project manager in HP's Data Systems Division R&D lab. With HP since 1979, he contributed to the design of the HP 1000 XL memory and the A700 floating-point board and microcode. Before joining HP, he designed military electronics for more than two years. He's a member of the IEEE. Born in Williamsville, New York, Lee attended Rensselaer Polytechnic Institute, graduating with a BSEE degree in 1976. He received his MSEE from Pennsylvania State University in 1979. He lives in Santa Clara, California, and enjoys volleyball, sailing his Hobie Cat, and skiing.

12

Donald A. Williamson

Don Williamson received his BSEE degree from Case Western Reserve University in 1973 and his MSEE from the University of Illinois in 1975. He's a member of the IEEE and has been with HP since 1975, contributing to the design of the 7902 Disc Drive controller and various processors, including two boards of the A900 Computer. He's now a full-time HP fellow at Stanford University, working on an advanced engineering degree. A resident of Cupertino, California, he is married, has a daughter, and works on old cars for relaxation.

Bruce A. Thompson



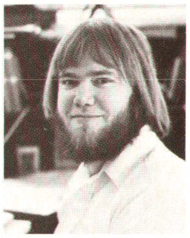
Bruce Thompson has been designing firmware and operating systems for HP since 1980. He contributed to the firmware design of the A900 Computer and is named as an inventor on a patent application for the A900 micromachine architecture. Born in Waukegan, Illinois, he received his BS degree in electrical engineering from the University of Illinois in 1979. He's married, lives in San Jose, California, and enjoys skiing, sailing, scuba diving, hiking, running, and bicycling.

Steven C. Steps

Steve Steps received BS degrees in electrical engineering and computer science from Kansas State University in 1975 and an MS degree in electrical engineering from the University of Southern California in 1976. He joined HP in 1976, contributed to the cache design of the HP 3000 Series 64 Computer and the cache and I/O design of the A900 Computer, and served as project manager for the A600+ Computer. His A900 work resulted in five patent applications. Steve is active in his church and in the American Rose Society and the San Jose Astronomical Association. A native of Topeka, Kansas, he now lives in San Jose, California, and is married to another HP engineer. Besides gardening and astronomy, he's also interested in photography.

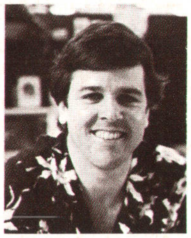
17

John R. Carlson



John Carlson received his BS degree in mathematics from California State University at Hayward in 1973 and his MS in computer science from the University of California at Davis in 1975. Between degrees he worked as a system programmer. He joined HP in 1975, contributed to the design of the HP 300 Computer's microcode and microsequencer, designed a cache memory controller chip, wrote a microprogramming language compiler, and designed the floating-point divide chip used in A-Series Computers. He's coauthor of a paper on the floating-point chip set and is a member of the ACM. John is married, lives in Fremont, California, and enjoys photography and backpacking. He's originally from Concordia, Kansas.

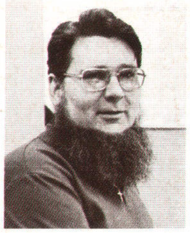
William H. McAllister



Willy McAllister joined HP in 1980. He was one of the designers of the SOS floating-point chip set and has coauthored two professional papers on the chips. He's now a project manager for integrated circuit development. Willy grew up in southern California. He received his BSEE degree from the University of California at Santa Barbara in 1974 and his MSEE from Stanford University in 1976. Before coming to HP, he designed standard-cell ICs for special-purpose processors for four years. He's married, has a son, lives in Cupertino, California, and enjoys volleyball, water and snow skiing, and good jazz.

23

Michael T. Winters



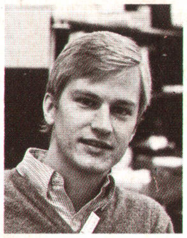
With HP since 1972, Mike Winters has developed diagnostics for HP 2100, HP 21MX, and HP 1000 L-Series and A-Series Computers. He also developed the virtual control panels for the L-Series and the A-Series. Before becoming a designer, he was a service engineer for field support. Mike is married, has a son and a daughter, and has worked with Cub and Girl Scout groups. Born in Burbank, California, he now lives in San Jose, California, and enjoys square dancing and camping.

John F. Shelton

John Shelton joined HP in 1981, developed the microcoded self-test for the A900 Computer, and is now a hardware designer. He's named as an inventor on a patent application for the fast decoding scheme used in the A900. He was born in Washington, D.C., and attended Massachusetts Institute of Technology, graduating with an SB degree in 1976. He also holds an MSEE from the University of California at Berkeley, which he received in 1981. He is married, has a son, and lives in Aptos, California.

26

Steven R. Kusmer

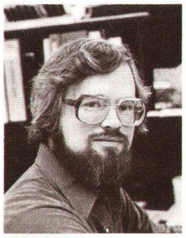


Steve Kusmer worked on the definition of code and data separation on the A-Series Computers and on the A700 microcode. A Cornell University graduate, he received his BS degree in electrical engineering in 1979 and came to HP the same year. He's a member of the IEEE, a runner, holder of a black belt in Korean karate, and a native of Cleveland, Ohio. He and his wife, a pianist, now live in San Francisco, California.

Douglas O. Hartman

Doug Hartman received his BS degree in electrical and computer engineering from the University of Michigan in 1979 and joined HP soon afterwards. He has developed HP 1000 system software and served as a technical lead for RTE-A, and is currently marketing UNIX-based systems. A member of the ACM, he received his MS degree in computer science from Stanford University in 1982. Doug was born in Ann Arbor, Michigan, and now lives in Santa Clara, California, "where it is much warmer than Michigan." He enjoys skiing, bicycling, photography, and music.

Douglas V. Larson

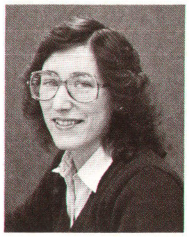


After joining HP in 1979, Doug Larson worked in software QA for two years and then implemented the LINK programs for RTE-A.1 and RTE-A. A native of Minneapolis, Minnesota, he graduated from the University of Minnesota with a BS degree in computer science in 1978. He's also a veteran of six years as a U.S. Navy electronics technician. His interests include playing go and bridge, and reading science fiction "while trying to keep up with the mortgage payments." He lives in Santa Clara, California.

Billy Chu

Bill Chu received his BA degree in mathematics and computer science from the University of Texas in 1968 and came to HP in 1972 with experience as a minicomputer software engineer. He received his MSEE degree in computer engineering from Stanford University in 1974. Since joining HP, he has participated in the design of the RTE-L, RTE-XL, and RTE-A operating systems. Bill is married, has a daughter, and lives in Sunnyvale, California. He is a deacon in his church and enjoys tennis, gardening, family outings, and working on his house.

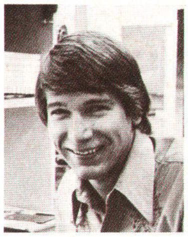
Elizabeth A. Clark



Now a technical lead for RTE-A enhancement, Beth Clark has been with HP since 1980 and has contributed to the RTE-L, RTE-XL, RTE-A.1, and RTE-A projects. She developed the memory manager and dispatcher for RTE-A. A native of York, Pennsylvania, she received her BS degree in computer science from Virginia Polytechnic Institute and State University in 1980. She's married, lives in Santa Clara, California, and enjoys choral singing, traveling, and indoor plants.

32

David L. Neuder



Dave Neuder joined HP in 1979 as an R&D engineer and worked on the hardware and software for the 64600S Timing Analyzer. His work has resulted in one patent related to the 64600S software. He studied electrical engineering at Michigan State University, earning a BSEE degree in 1977 and an MSEE degree in 1979. Born in Wyandotte, Michigan, Dave is a member of the IEEE and lives in Colorado springs, Colorado. Outside of work he serves as president of his church's Sunday school class and enjoys photography, skiing, dancing, mountain climbing, and backbacking.

Hewlett-Packard Company, 3000 Hanover
Street, Palo Alto, California 94304

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD JOURNAL

FEBRUARY 1984 Volume 35 • Number 2

Technical Information from the Laboratories of
Hewlett-Packard Company
Hewlett-Packard Company, 3000 Hanover Street
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
Van Heuven Goedhartlaan 121
1181 KK Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To change your address or delete your name from our mailing list please send us your old address label. Send changes to Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, California 94304 U.S.A. Allow 60 days.