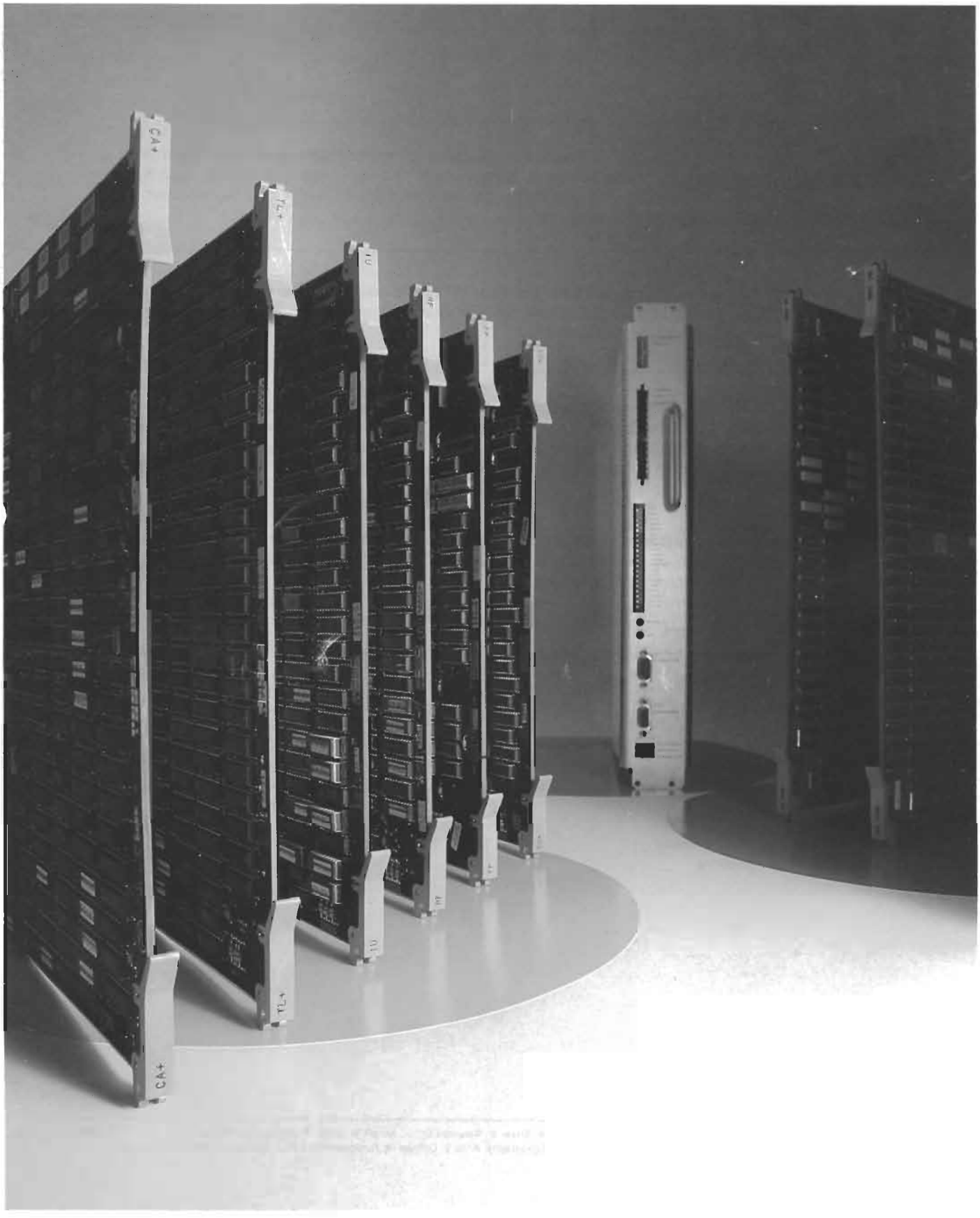
HEWLETT-PACKARD
JOURNAL

MARCH 1987



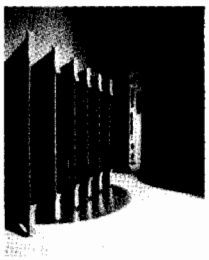
HEWLETT-PACKARD
JOURNAL

March 1987 Volume 38 • Number 3

Articles

Departments

In this Issue



This issue continues our series on HP Precision Architecture topics. On page 4 you'll find the hardware design story of the first two members of HP's new generation of computers based on HP Precision Architecture. The HP 9000 Model 840 Computer runs the HP-UX operating system and is designed for technical and real-time applications. HP-UX, HP's version of AT&T's UNIX® System V operating system, was featured in our December 1986 issue. The HP 3000 Series 930 runs the MPE XL operating system and is designed for business data processing. We've received a paper on MPE XL and will be publishing it later this year. Both the Model 840 and the Series 930 have the same processor, which is noteworthy because it uses a relatively old-fashioned integrated circuit technology, TTL, and yet achieves about four times the speed of the fastest of HP's previous-generation computers. It was, of course, this potential speed of RISC-like architectures (RISC means reduced instruction set computer) that triggered the development of HP Precision Architecture. Future computers implementing the architecture in state-of-the-art VLSI technology are expected to be even faster. The Model 840/Series 930 processor is the six smallest boards on this month's cover. The two larger boards are an 8M-byte memory module and the board with the handle is the system monitor.

HP Precision Architecture is more than just a RISC architecture. For any architecture, compilers must be developed to provide a high-level language interface to the machine. Whether the speed potential of a RISC architecture is realized, particularly for commercial languages requiring many complex operations, is largely a function of compiler design. How the HP Precision Architecture compiler designers approached some of the more challenging problems is detailed in the paper on page 29, which also gives performance data showing how well they succeeded in solving these problems.

The HP 3000 Series 930 can have a large number of terminals connected to it through intelligent hardware modules called HP 2345A Distributed Terminal Controllers, each of which accommodates 48 terminals. The theory, design, and operation of the DTC are described in the article on page 21. Queueing theory was used to predict its performance. On page 18 is a description of the production test system for the Model 840 and Series 930 computers.

I can't remember ever having anything controversial in these pages, so the Viewpoints article on page 38 may be the first time. It's a paper presented to the Mathematics Panel of the American Association for the Advancement of Science by Zvonko Fazarinc of HP Laboratories. We hope you'll find his ideas on the teaching of infinitesimal calculus thought-provoking.

-R. P. Dolan

What's Ahead

Next month's issue will feature the design of the HP 8175A Data Generator and its arbitrary waveform generator option. There will be two research reports, one on software reliability and one on surface mount solder joint failure modes. Another paper will discuss the design and application of HP's PL-10 software package, a master planning tool for the semiconductor manufacturing industry.

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers. Letters must be brief and are subject to editing. Letters should be addressed to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

Hardware Design of the First HP Precision Architecture Computers

The HP 3000 Series 930 and the HP 9000 Model 840 are implemented with commercial TTL logic.

by David A. Fotland, John F. Shelton, William R. Bryg, Ross V. La Fetra, Simin I. Boschma, Allan S. Yeh, and Edward M. Jacobs

THE HP 9000 MODEL 840 and the HP 3000 Series 930 are the first technical and commercial computer products, respectively, to use the new Hewlett-Packard Precision Architecture.[1] HP Precision Architecture combines a simplified, RISC-like instruction set with a powerful coprocessor architecture, a 64-bit virtual memory addressing system, a new high-performance I/O architecture,[2] and provision for multiprocessors.

The HP 9000 Model 840 and the HP 3000 Series 930 are both based on the same processor, memory system, and I/O system. The processor consists of five printed circuit boards, each 8.4 by 11.3 inches, containing off-the-shelf TTL logic. It uses FAST™ TTL, 25-ns and 35-ns static RAMs, and 25-ns and 35-ns PALs™. These five boards include the processor pipeline, which fetches and executes an instruction every 125 ns, a 4096-entry translation lookaside buffer (TLB) for high-speed address translation, and 128K bytes of cache memory. An additional (sixth) board contains the hardware floating-point coprocessor. Each board contains about 150 ICs.

A 20-Mbyte/s bus called the MidBus connects the CPU, main memory, high-speed I/O cards, and I/O channels. There are six memory slots, and memory comes in two-board 8M-byte sets. This gives a maximum of 24M bytes of memory. Memory uses 256K-bit nibble-mode dynamic RAMs with single-bit error correction and double-bit error detection. There are seven general-purpose I/O slots, which can be used for high-speed I/O cards or I/O channels. The I/O channel is a two-board set. Most I/O is handled by cards on an HP CIO bus connected to the MidBus through an I/O channel. The HP CIO bus is a 5-Mbyte/s I/O bus.

A system monitor card monitors power supply levels and temperature and provides front-panel functions and system overtemperature shutdown. An access port card allows remote field support access for diagnosis.

The differences between the HP 9000 Model 840 and the

FAST is a trademark of Fairchild Camera and Instruments Corporation.
PAL is a registered trademark of Monolithic Memories.

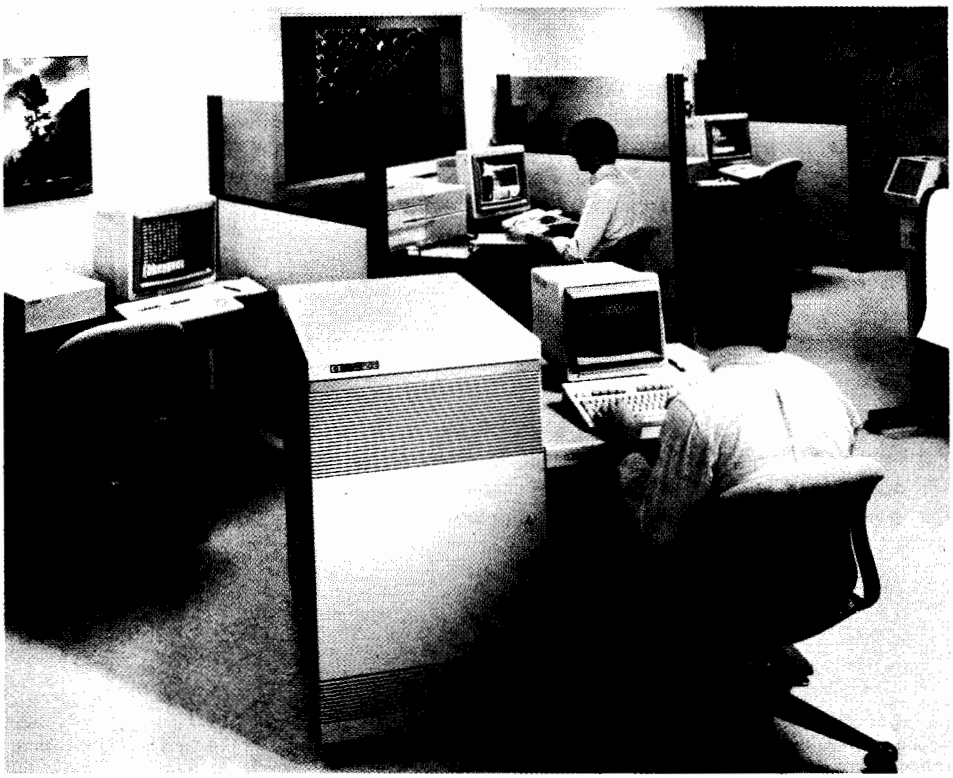


Fig. 1. *The HP 9000 Model 840 Computer is the first HP Precision Architecture computer for technical and real-time applications. Its operating system is HP-UX, HP's version of AT&T's UNIX System V operating system.*

HP 3000 Series 930 are in configurability and software. The HP 9000 Model 840 (Fig. 1) is a technical machine. It runs HP-UX, HP's version of AT&T's UNIX® System V operating system with real-time extensions.[3] The Model 840 is a single-bay, one-meter-high system. Its input/output system has one HP CIO channel, and up to 12 CIO cards. Terminal I/O is done using a six-port multiplexer card, limiting the number of terminals to sixty. 8M bytes of memory is standard.

The HP 3000 Series 930 (Fig. 2) is a business machine. It runs MPE XL,[4] a new version of HP's proprietary MPE operating system, and provides compatibility for existing HP 3000 customers. The Series 930 has two one-meter-high bays to provide more I/O capacity. It has three CIO channels, and 16M bytes of memory is standard. The second bay contains two CIO buses and up to two HP 2345A Distributed Terminal Controllers.[5] Terminal I/O for MPE XL is done using an IEEE 802.2 local area network and the HP 2345As, each of which can handle up to 48 terminals and can be located near a work group using a LAN cable.

History of the Project

Development of HP Precision Architecture began at HP Laboratories in early 1982. The processor instruction set and virtual memory system were well-defined by the end of 1982. The TTL implementation project began in April 1983.

The project's goals were low factory cost, good performance, and very fast design time, since this was to be the software development machine for the HP Precision program. We used internal HP design tools for schematic capture and timing analysis, and FTL, a simulator developed at Amdahl Corporation, for gate level simulation of the entire system. We did not build wirewrap breadboards, but went straight to printed circuit boards.

Simulation of the processor started in the fall of 1983, and we had working processors by early 1984. A complete processor with cache, TLB, and main memory was delivered to the software developers in July 1984. This version of the machine did not have an I/O channel or a hardware floating-point coprocessor since the architectures for these units were not complete. I/O was done with a parallel interface to an HP 9000 Series 200 Computer.

This machine was sufficient for software development, and we built 36 of them over the next five months. This version used bench power supplies and had a very small cabinet. It ran at 30 MHz, rather than the 32 MHz of the final machine. Over the next six months the final cabinet, power system, and system monitor were designed and the I/O channel was completed.

In January 1985 we put together our first lab prototype system. This system looked very much like the final product. It had working I/O channels, up to 24M bytes of memory, and a full-speed processor. Between January and September we built almost 200 for use as software development machines. This machine had only 32K bytes of cache memory and a 1024-entry TLB. It could execute about 3.5 million instructions per second (MIPS).

Enhancements were added for higher performance and better manufacturability during 1985. Newer, denser static RAMs were available, so we quadrupled the size of the caches and TLBs. Some other minor changes were made to eliminate bottlenecks in the processor and the final performance rating is 4.5 MIPS. We also completed the design

UNIX is a registered trademark of AT&T.

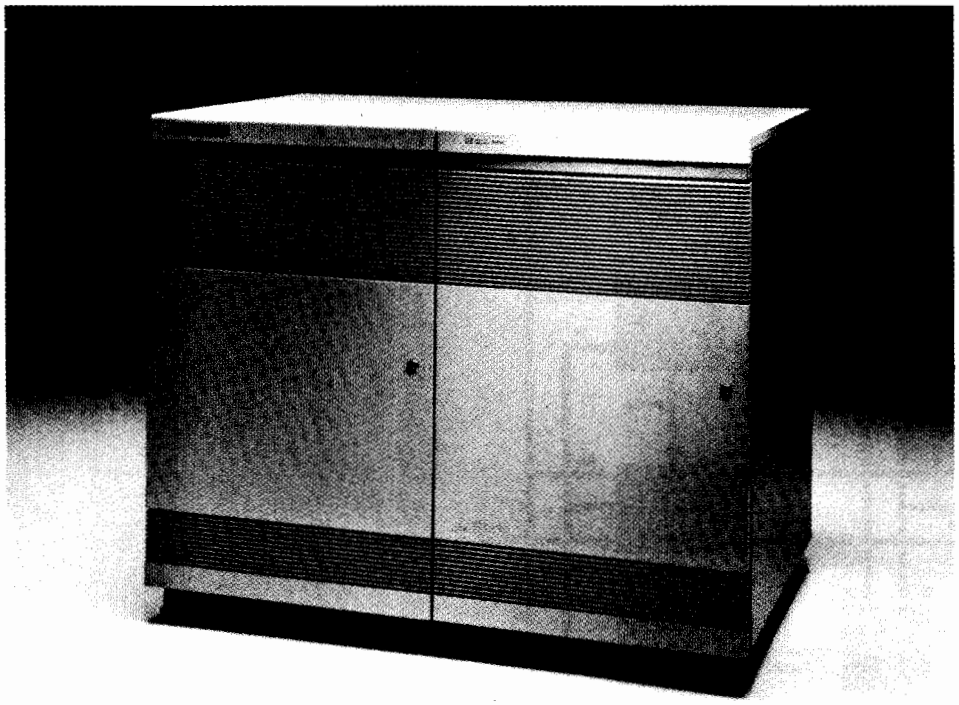


Fig. 2. *The HP 3000 Series 930 Computer is the first HP Precision Architecture computer for commercial applications. Its operating system is HP's proprietary MPE XL.*

of the floating-point coprocessor, and started building full-functionality production prototypes in May 1986. The first production Series 840 was shipped in November 1986.

CPU Design

The simplicity of HP Precision Architecture allowed the entire CPU and floating-point coprocessor to be implemented on six medium-size boards, even though it was designed using mostly MSI TTL, a technology with a fairly low level of integration. These six boards and the six major buses internal to the CPU are organized as shown in Fig. 3.

Instruction Unit

The I-unit (instruction unit) controls the flow of instructions. It executes branch instructions and handles traps and interrupts. The I-unit also creates and distributes the system clocks that keep all of the elements of the processor synchronized. Instruction execution begins when the I-unit creates the address of the instruction to be executed and sends this address to the I-cache, which contains the instructions to be executed. The I-cache sends the instruction back on the NI (next instruction) bus, which is distributed to all of the processor boards.

Instruction decoding is decentralized, with each board decoding only as much of the instruction as is necessary for that board to do its job.

Register File Board

The register file board supplies the operands (the values to be operated on) for the instruction. It maintains thirty-two general registers. Each register is thirty-two bits wide. In addition, the register file maintains copies of the twenty-five control registers specified by HP Precision Architecture.

In many computer architectures, operands can be stored in memory. In HP Precision Architecture, all operands are stored in the general registers or encoded in the instructions. The only instructions that access data memory are load instructions and store instructions. The addresses for the load and store instructions are created from values in the general registers and from values encoded in the instructions.

The register file board drives the register values out to the rest of the CPU on the X (index) and B (base) buses. Sometimes in a pipelined implementation like the Model 840/Series 930 processor, a result being created by one instruction or data being loaded from memory is needed immediately by the following instruction before there is time to store the result or the data into a general register. The register file board recognizes these cases and routes the data around the general registers to the instruction that needs it.

Execution Unit

The E-unit (execution unit) performs arithmetic calculations on the operands. It executes the arithmetic instructions and creates the addresses for load and store instructions. It contains a 32-bit ALU (arithmetic logic unit) for arithmetic and logical calculations, a barrel shifter for shift instructions, and complex mask/merge circuitry for extracting and depositing bit strings. It also contains a preshifter on one input to the ALU. This is used in address calculations and for special instructions used in software multiply routines (the Model 840/Series 930 does not execute multiply instructions directly in hardware.)

The E-unit sends its result back to the register file over the R (result) bus. If the instruction is a load or store instruction, then the address is sent to the cache controller and TLB boards on the CADR (cache address) bus. The E-unit also creates a condition code based on its result. That condition code is sent to the I-unit to be used for conditional

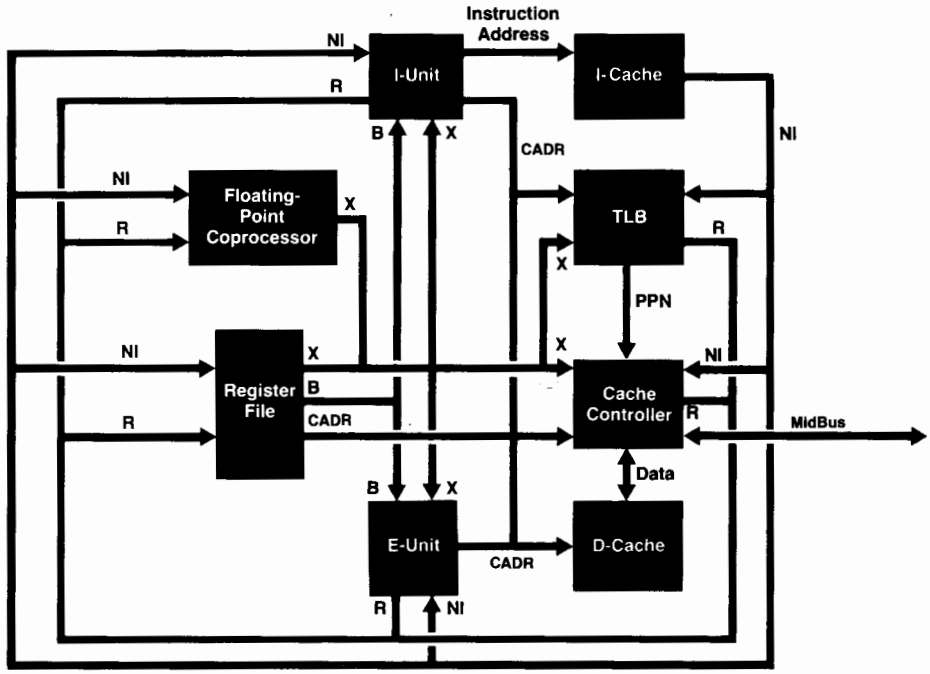


Fig. 3. *Block diagram of the CPU for the HP 9000 Model 840 and HP 3000 Series 930 Computers.*

branches and conditional skips.

TLB, Cache, and Coprocessor

The TLB (translation lookaside buffer) controls the access to virtual memory. HP Precision Architecture provides a very large global address space. Addresses in the Model 840/Series 930 processor are 48 bits long. The architecture will support up to 64-bit addresses. This huge address space increases performance by making memory management easier in both software and hardware. However, it would be impossible to support that much physical memory, so it's necessary to translate the large virtual addresses into smaller physical addresses.

The TLB performs this translation from the virtual memory address (the address that the processor sees) to the physical memory address (the address that the memory system sees). It also keeps track of protection information (i.e., which user is allowed to access which portions of virtual memory).

The large global virtual address space specified by the architecture allows effective use of a TLB with many entries. The Model 840/Series 930 TLB has entries for 4096 pages of virtual memory. Each page contains 2K bytes of code or data. This is an enormous number of entries compared to other computers, and makes the performance penalty for the virtual-to-physical translation very small (the penalty comes from the processing necessary when the TLB does not contain an entry for the virtual memory page that the processor is trying to access.)

The cache controller manages two high-speed cache memories, one for code and one for data. In other architectures, caches, if they are used, are forced to be transparent, that is, invisible to the software. HP Precision Architecture allows explicit software management of the caches, thereby making it possible to separate the code and data caches, doubling the available bandwidth between the caches and the rest of the processor.

The floating-point coprocessor works in parallel with the main processor to do floating-point calculations. This allows the processor to continue processing during the several cycles that it may take for the floating-point coprocessor to complete the execution of a floating-point instruction.

Architectural Impact

Because the architecture is simple and regular, it allows for a nonmicrocoded implementation such as this one. This means that the implementation is a fairly straightforward interpretation of the architecture, and architectural features have a large impact on the processor organization.

For example, all instructions are exactly 32 bits in length. Most instructions use one or two general registers as operands, and the register addresses of these operands are always encoded in the same place in the 32-bit instruction. This means that prefetching of the instruction and its operands can be done without regard to the decoding of the instruction or to the execution of the previous instruction.

As the instruction comes out of the cache and is distributed to the processor boards, the register file board receives the instruction from the NI (next instruction) bus. The register file board immediately prefetches from the register file the two operands necessary for the instruction, so that at the beginning of the next cycle, when the rest of the processor is prepared to execute the instruction, the operands have already been obtained and are ready to be driven out to the rest of the processor on the X and B buses.

Because of the simplicity of the instruction encoding, it was very efficient to distribute the instruction decoding among all of the processor boards, with each board decoding only the piece of the instruction that applies to that board. Each board, then, receives the instruction from the NI bus as it comes out of the cache and prepares to execute it during the following cycle.

Because the architecture relies so heavily on register values for operands, the register file is central to the instruction flow. Each instruction begins with the prefetching of the instruction from the cache and the prefetching of the operands from the register file. From here, the operands fan out to the rest of the processor, which consists of several short, parallel data paths.

This, again, is a result of the architecture, which heavily emphasizes these short, parallel data paths as a means of increasing performance. The E-unit, for example, has a barrel shifter combined with sophisticated mask/merge circuitry for bit manipulation. It also contains a 32-bit ALU. The results from these two pieces of circuitry are never needed in the same instruction, allowing them to be placed in parallel so that neither one will impact the speed of the instructions that use the other.

The I-unit can calculate branch target addresses in parallel with E-unit calculations, which allows for instructions that calculate an arithmetic result and conditionally branch based on that result, all in one cycle. This makes possible very efficient loops and range checking in the code.

Another example of the parallelism encouraged by the architecture can be found in the cache and TLB. Because the architecture does not permit one physical address to be referenced by more than one virtual address, the cache and TLB accesses can be done in parallel. At the same time that the TLB is performing the translation from virtual to physical address, the cache is obtaining the data (or code), and reading a tag that indicates which physical address that data belongs to. When the TLB has completed the translation, the physical address is sent to the cache over the PPN (physical page number) bus, and compared to the physical address that the cache has read from its tags to determine whether this data is really the data that is needed. In most architectures, these two processes must be done serially, resulting in a much longer cache access time.

Processor Pipeline

The processor is pipelined. This means that several instructions are in various stages of execution at any one time. Whereas this implementation technique must be made transparent in most architectures, it is supported, and in fact encouraged, by HP Precision Architecture.

The pipeline has three stages, as shown in Fig. 4. Each stage takes 125 ns, and is subdivided into two minor stages. The first stage is referred to as the fetch stage. During the first half of this cycle the address of the instruction is sent to the instruction cache from the I-unit. During the second

half, the instruction is returned and distributed, instruction decoding is begun, and the register operands are read out of the register file.

The second stage is referred to as the execute stage. It is during the first half of this stage that the arithmetic result is calculated if this is an arithmetic instruction, or the data address is calculated if this instruction is a load or store instruction. If this is a branch instruction, the branch target address is also calculated during the first half of this stage. For arithmetic instructions and for conditional branch instruction, the condition is calculated during the second half of this cycle. For loads and stores, the second half of the cycle is used to drive the data address to the cache controller and the TLB.

Notice that if the instruction is a branch instruction, because of the pipeline the following instruction is being fetched while the branch target is being calculated. In most architectures, this would result in an instruction being fetched that was not going to be executed, and would result in a wasted cycle with every branch. In HP Precision Architecture, branches are delayed by one instruction. In other words, one additional instruction is executed after the branch instruction before the branch target is reached. This is an example of how the architecture supports pipelined implementations, resulting in a performance improvement over traditional architectures.

The third pipeline stage is referred to as the load/store stage. During this stage, load data is returned from the cache, or stored data is written to the cache. It is during the first half of this stage that the E-unit result is actually written into the register file, and during the first half of the following cycle that the load data is written into the register file. Notice that the register file is only written during the first half of any cycle. This is because during the second half cycle it is necessary to read from the register file the operands being used by the instruction that is currently in its fetch stage.

Cache and TLB Design

The cache and TLB (translation lookaside buffer) speed up memory accesses by keeping recently accessed data and virtual address translations in local high-speed memory. They are designed to give the best performance without increasing the CPU's basic cycle time. They take advantage of the architecture, which allows a simple design, and they are pipelined to get increased bandwidth without increased hardware.

The cache is a high-speed memory that shortens typical main memory access times by keeping copies of the most recently accessed data. The cache is divided into a 64K-byte instruction cache and a 64K-byte data cache, each of which is divided into 4096 16-byte blocks. Each block has an address tag that specifies the block of memory from which it came. When the processor accesses data or instructions, the block is copied from main memory into the instruction or data cache, as appropriate. All further references use the copy in the cache, until the cache block is needed for a different block from memory. At that point, the block that is removed is written back to memory only if it has been modified by the processor.

Similarly, the TLB speeds up virtual address translations by acting as a cache for recent translations. Both virtual memory and physical memory are divided into pages of 2K bytes, and each TLB entry maps a virtual page number to a physical page number. Each virtual address is made up of a virtual part (i.e., the virtual page number) and a physical part (i.e., the offset within the page). The TLB translates the virtual page number to get a physical page number. This is concatenated with the page offset to generate the physical address.

To allow the implementation of large, high-speed caches, HP Precision Architecture disallows address aliasing, that is, the capability of having two different virtual addresses pointing to the same physical location. This allows the Model 840/Series 930 processor to access the cache and TLB in parallel, without the problems and constraints this has had in other architectures. Thus, the access time is the worst of the TLB and cache access times, rather than the sum of them (see Fig. 5).

TLB Operation

The virtual memory space is divided into virtual pages of 2048 bytes each. A data structure in main memory called the page table keeps information about each virtual page that is currently in use (i.e., a copy of that page exists in the physical memory). This table has one entry for each virtual page, and contains information on the protection

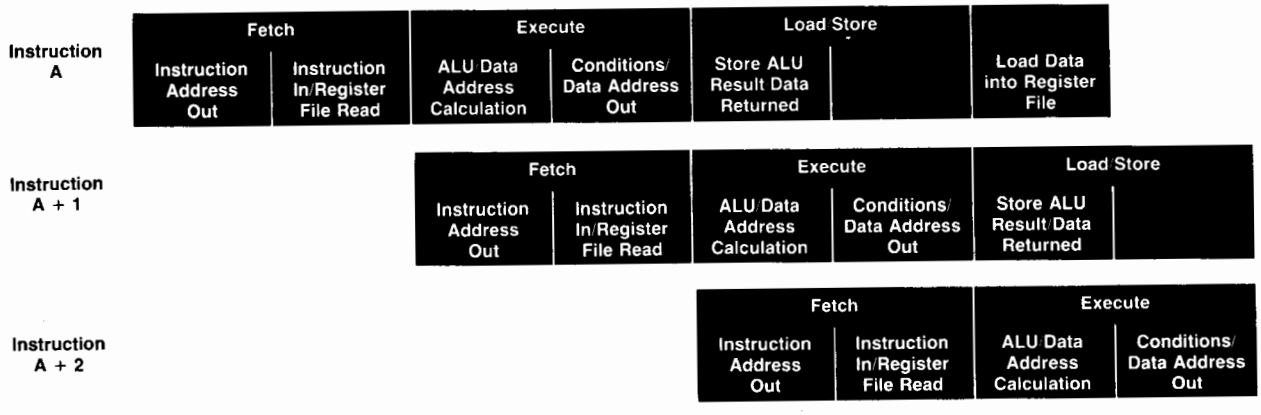


Fig. 4. *CPU pipeline.*

of that page (which users are allowed what kind of access to the page) and the physical page number of that page (where the page exists in physical memory).

The TLB acts as a cache for this information. Just as the instruction and data caches keep copies of recently accessed memory locations, the TLB keeps copies of the information for recently accessed pages. The Model 840/Series 930 TLB has entries for 4096 pages, 2048 for code and 2048 for data.

With each memory access (both instruction fetches and data loads and stores), the TLB checks the protection information for that page and the physical address of that page. It sends the physical address to the cache, so that the cache can check to see if the word being accessed is contained in the cache. If the protection information indicates that the user is not allowed the kind of access being attempted, the TLB signals the register file board to back out of the instruction and signals the I-unit to raise a trap.

TLB miss handling is done in software on the Model 840/Series 930. This means that if none of the entries in the TLB corresponds to the virtual address being accessed, a trap is raised. Software must then get the information about that page from the page table, place it in a TLB entry, and reexecute the instruction.

The TLB is pipelined so that it can perform both an instruction address translation and a data address translation each cycle. The first half of every cycle, it reads the tag and translation out of the TLB for the instruction being fetched (see Fig. 6). During the second half of the cycle, it checks the tag for a match (also known as a hit) and performs a protection check.

Also during the second half of each cycle, the tag and translation are read out of the TLB for any memory access instruction currently in the execute phase. This is checked for a match and protection during the following half cycle, that is, the first half cycle of that instruction's load/store phase. Thus, each half cycle the TLB starts a new translation, which will be completed one cycle later.

The TLB is a direct mapped TLB. It has 2048 entries for instructions and 2048 entries for data. Direct mapped means that each virtual page translation can exist in only one entry of the TLB, and if a program accesses another page that maps to the same entry, the first one is replaced. Although direct mapped TLBs (and caches) have greater miss rates than set-associative TLBs of the same size, the direct mapped TLB minimizes cycle time, which has a greater impact on performance.

The TLB is addressed by the 9 LSBs of the virtual page number hashed with the 11 LSBs of the space ID to create an 11-bit index. The hash takes the two LSBs of the space ID, then flips the next 9 bits of the space ID around and exclusive-ORs them with the 9 LSBs of the virtual page number. In addition, one address bit selects instruction or data, since the TLB is split for instruction and data accesses, although these use the same hardware. The address hash reduces the likelihood of a program's heavily using a single TLB entry for two different pages. In addition, the large size of the TLB greatly reduces overall miss rates.

In addition to the physical page number of the page translated and a virtual tag to identify the corresponding virtual page, each TLB entry has extra information to implement the HP Precision protection scheme. This includes an access rights field, an access ID, and several status bits. The access rights field specifies how the page can be used (e.g., read only, read/execute, etc.) and the necessary privilege level to use it. The access ID is a key that must match one of the four protection IDs (in control registers) that processes can have, in addition to the access rights check. The status bits include an entry valid bit, a dirty bit, two different break-on-access bits, and an I/O bit which marks whether the page corresponds to an I/O module.

Since TLBs do not contain the translations for all pages in memory simultaneously, they occasionally do not have the desired translation and a miss occurs. The instruction cannot complete without the translation, so the TLB causes a trap. This causes the current processor state to be saved in control registers, and execution continues in the software TLB miss handler. If the page is actually in memory, the handler will insert the needed TLB entry and retry the offending instruction. If the desired page is not in memory

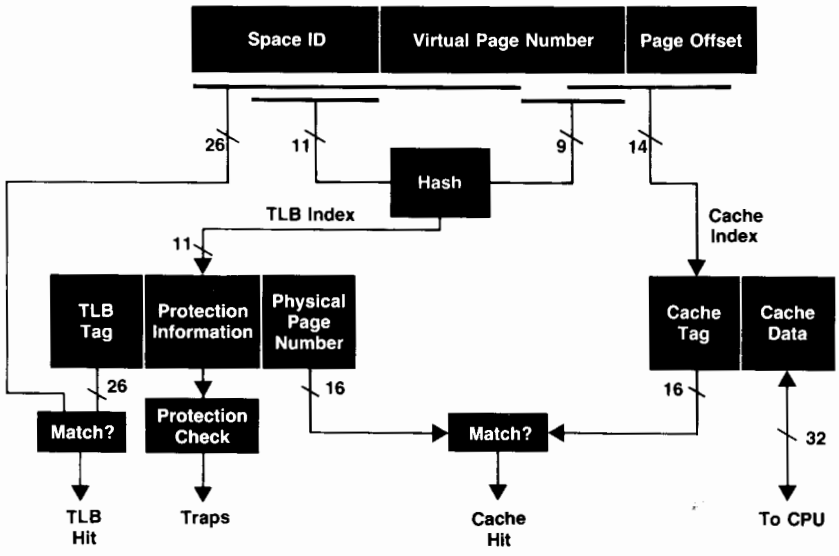


Fig. 5. *Cache/TLB block diagram.*

(i.e., on disc), the handler will invoke the page fault handler. Because the instruction is restarted after a TLB miss, a TLB must be able to contain two arbitrary translations to be able to complete any instruction, one to fetch the instruction and one to access data requested by the instruction (if load or store). If it cannot, there are cases where the program will get stuck first replacing the entry for fetch, then for data, and back again. Hence the TLB is split, half for instructions and half for data, to guarantee forward progress.

One side effect of the very large HP Precision address space, 256 terabytes (2^{48} bytes) in this case, is that all processes execute in the same unified address space. This allows different processes (or programs) to share code or data more simply, since the addresses are the same. Also, since there is only one address space (as opposed to separate address spaces for each user), the operating system does not have to flush all entries out of the TLB whenever it switches processes. This cuts down on the number of TLB misses.

Cache Operation

The cache tags and control, like the TLB, are pipelined. At the same time the TLB entry is read, the cache tags are read, and the cache tag comparison is performed when the TLB tag comparison and protection check are performed, both for instructions and for data. However, the data RAMs (for both the instruction and the data caches) are not pipelined. Instead, the data RAMs for the instruction cache and the data cache are implemented using separate RAMs so that the access can span an entire cycle. This allows the use of larger, slower RAMs for the data, without affecting the cycle time. Otherwise, the cycle time would have to be long enough to allow reading the slower data RAM in a half cycle.

Like the TLB, the instruction cache and data cache are both direct mapped to minimize cycle time. They are as large as possible with current high-speed RAMs, thus keeping down miss rates. In addition, there are several features that allow the processor to keep running even though the cache is servicing a miss.

In the case of a data cache miss, the cache allows the processor to continue running until either the processor needs the data (from a load) or the processor executes another cache access. The first case is called a load/use interlock, and occurs when the cache receives a load instruction for a particular register, and before the cache can supply the data, it receives another instruction that uses that target register. This is detected by comparing the load target for any load instruction in progress with the register fields of instructions being fetched, and causing the processor to freeze if there is a match. As soon as the data is returned to the processor, the interlock goes away and the processor can continue.

When there is a cache miss, the cache receives the data from main memory in a 4-word block, one word per cycle. To speed things up, as soon as the cache receives the requested data, it passes it through to the processor while it is also putting it into the cache. This allows execution to continue, even though the cache might still be servicing the miss.

When there is an instruction cache miss, the cache freezes the processor immediately, since it needs the instruction to continue. However, as soon as it receives the requested instruction from memory, it passes the instruction and the following instructions through to the processor to allow it to continue. The processor continues receiving the instructions straight from memory until either the end of the block is reached or the processor executes a taken branch. At the end of the block, the processor goes back to getting its instructions normally, from the cache. If the processor branches, the cache will freeze the processor until it finishes servicing the miss, then allow execution to continue. These optimizations improve performance by reducing the average cache miss penalty. Cache performance is measured by measuring the total miss penalty, which is the product of the miss rate and the penalty for each miss. The miss rate is minimized by making the cache as large as possible. The miss penalty is minimized by allowing the processor to execute whenever possible, even during cache miss servicing.

HP Precision Architecture allows a somewhat simpler cache than would otherwise have been possible, by putting the burden on software to keep the instruction and data caches consistent with each other and with any I/O being performed. The architecture provides cache flush and purge instructions, which software can use to guarantee that the copy in memory is up to date. Thus, the hardware does not have to check for a program modifying instruc-

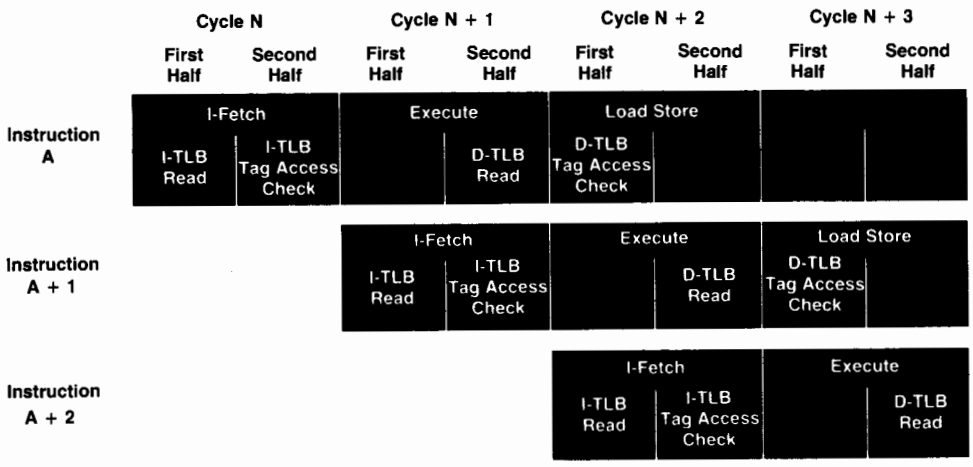


Fig. 6. *TLB pipeline.*

tions, or whether DMA is accessing data that is currently in the cache.

Performance

Since this machine was the first HP Precision Architecture machine, we thought that it should be instrumented for hardware performance measurements. Thus the analysis interface card was born. This card is a coprocessor, and it has two functions. First, it depipes the instructions and presents the data to a frontplane interface to a logic analyzer. It can show three 32-bit buses and some control signals to the analyzer. The three buses can be chosen to show the instruction address, instruction, either source register, the ALU result, the load/store address, or the load/store data. Control signals indicate whether the instruction was executed or nullified, or if a taken branch or a trap occurred.

The analysis interface card and a disassembler for the HP 64000 Logic Development System were used extensively for both hardware and software debugging. The interface card was also used to take instruction traces for running performance simulations for other machine organizations.

The other function of the analyzer card is to collect performance statistics. It contains five 32-bit counters. Three of the counters can each count one of 32 predefined events. The other two counters form a pair, only one of which is readable by the processor. This pair of counters can count one of the 32 events in four ways: simple counting (like the other three), maximum duration of the event, number of times the event duration exceeds a threshold, or event occurrences masked by a one-zero-don't care comparator on one of the CPU buses. For example, the data cache miss rate can be measured by having one counter count data cache accesses and another count data cache misses. These statistics helped confirm the results of the cache and TLB simulations that were used to make trade-offs when the machine was being designed.

Events that can be counted are:

- Cycles
- Fetched instructions
- Executed instructions
- Loads
- Stores
- Branches taken
- Branches not taken
- Branches nullifying next instruction
- Arithmetic operations nullifying next instruction
- All architectural nullified instructions
- All nullified instructions
- External interrupts
- Traps
- Instruction cache misses
- Data cache misses
- Dirty data cache misses
- Instruction cache accesses
- Data cache accesses
- Instruction TLB misses
- Data TLB misses
- Instruction TLB accesses
- Data TLB accesses
- I/O accesses
- Load/use interlocks
- Cache wait cycles
- Coprocessor wait cycles
- Interlock and wait cycles
- Time spent at privilege level 0, 1, 2, or 3
- Time spent with interrupts off
- Time spent in virtual code space
- One write port interlocks.

MIPS Calculations

A frequently used measure of raw CPU power is millions of instructions per second, or MIPS. The MIPS rating of a computer is calculated as one over the cycle time times the cycles per instruction:

$$\text{MIPS} = \frac{1}{\text{cycle time}\,(\mu s) \times \text{CPI}}$$

MIPS ratings are a good measure of performance when comparing machines with the same computer architecture, but can be misleading when comparing different architectures, since one architecture may take fewer instructions to complete the job than another. The best way to compare machines is to run the same application on both machines. Even on machines with the same architecture, the MIPS rating is calculated using a standard instruction mix or measured when running a standard jobstream. If the application differs from the standard then the MIPS rating might not be a good predictor of performance when running the application. The instantaneous MIPS rate of a computer is quite variable, as can be seen from Fig. 7, so any MIPS number quoted is only a long-term average. MIPS rates vary with the job executed and with time. MIPS ratings also have the drawback of not taking into account operating system efficiency, compiler efficiency, or I/O system efficiency. Therefore, MIPS is not a good metric for predicting applications performance on systems running different operating systems.

Cycles per instruction, or CPI, is the key measurement of how well a CPU uses its available power. Ideally, one instruction per cycle will be executed, so the CPI will be one. Calculating the CPI is straightforward: sum up the product of the penalties and their frequency, and add one:

$$\text{CPI} = 1 + \Sigma\, P_i F_i$$

There are two ways of minimizing the CPI: reduce the penalty (P_i) or reduce the frequency (F_i).

Model 840 MIPS

For the HP 9000 Model 840 and HP 3000 Series 930 Computers, the cycle time is 125 ns. If there were no penalties, all instructions would take one cycle (CPI = 1), so the maximum possible performance is 8 MIPS.

The measured MIPS rate for the Model 840 varies from about 3.5 to 8 MIPS with an average of 4.5 to 5. (Fig. 7 shows measured performance during typical operation.)

The MIPS rating for the Model 840 can be calculated using the statistics gathered with the analysis interface card described above. First, there are nullified instructions. This happens when the compiler can't schedule a branch, and

when an arithmetic instruction skips on a condition. About 7% of all instructions are branches specifying nullify. About 2% of instructions are conditional skips that are taken. The load/use interlock happens when a load instruction is followed by an instruction that uses the data returned. About 14% of the instructions have this interlock. As the optimizing compiler improves, these numbers should go down. The Model 840 has an interlock that most HP Precision Architecture machines won't have: the one write port interlock. Since the load instruction result comes back one cycle after the load is executed, the architecture expects the register files to be able to write two results at the same time, one from the load instruction, and one from the ALU. The Model 840/Series 930 register file is built from static RAMs and can only store one result at a time. If a load instruction is followed by an instruction that stores a result, there will be a one-cycle interlock. This happens on about 8% of instructions.

Every instruction has the possibility of missing the I-cache. Nullified instructions are fetched, so they can miss the I-cache also. The I-cache miss rate is about 3% and the miss penalty is 5 to 8 cycles with an average of about 7. The contribution to the CPI can be calculated by multiplying the miss rate (0.03) times the penalty (7) times the frequency of access (1.09). The frequency of access is 1.09 because for every one access per executed instruction, there is 0.09 access (9%) because of nullified and skipped instructions. This makes the CPI contribution 0.23 (see "Cache Performance," below). Every instruction can also miss the I-TLB. TLB misses cause a trap and are handled in software. This software is not doing any useful work so it is not counted as instructions executed for calculating MIPS. The miss handler is about 40 instructions and takes an average of about 70 cycles. The I-TLB miss rate is about 0.05%, so the CPI contribution is about 0.04 (0.0005 times 70.) Load and store instructions can miss the D-cache and the D-TLB. Loads are about 25% of the instruction mix and stores are about 15%. A D-cache miss can cost anywhere from 1 to 15 cycles with an average of about 8. The D-cache miss rate is about 3%. The D-TLB miss rate is about 0.1%. The D-cache contribution to the CPI is about 0.1, and the D-TLB contribution is about 0.03. There is also a CPI contribution from flushing the cache for I/O. This is another function that most machines do in hardware, so it should not be counted in the instructions executed. The contribution to the CPI is about 0.04. The CPI is therefore:

Basic instruction	1.00
Nullify	0.09
Load/use	0.14
One write port	0.08
I-cache miss	0.23
I-TLB miss	0.04
D-cache miss	0.10
D-TLB miss	0.03
I/O cache flush	0.04
Total CPI	1.75

This gives 4.57 MIPS as the calculated performance.

Cache Performance

Looking at the numbers in the preceding section, we can easily see that the cache is the major contributor to the CPI on this machine.

The function of a cache is to make the memory look faster by remembering recent memory accesses in the hope that they will be used again. Most programs have some sort of locality, that is, memory references in the recent past are likely to be used again in the near future. If this were not true, the cache would not work.

Caches normally take some amount of time, usually one cycle, to fetch data. This is how high performance is achieved; memory normally takes more than one cycle. Of concern are the less frequent cases when the cache takes

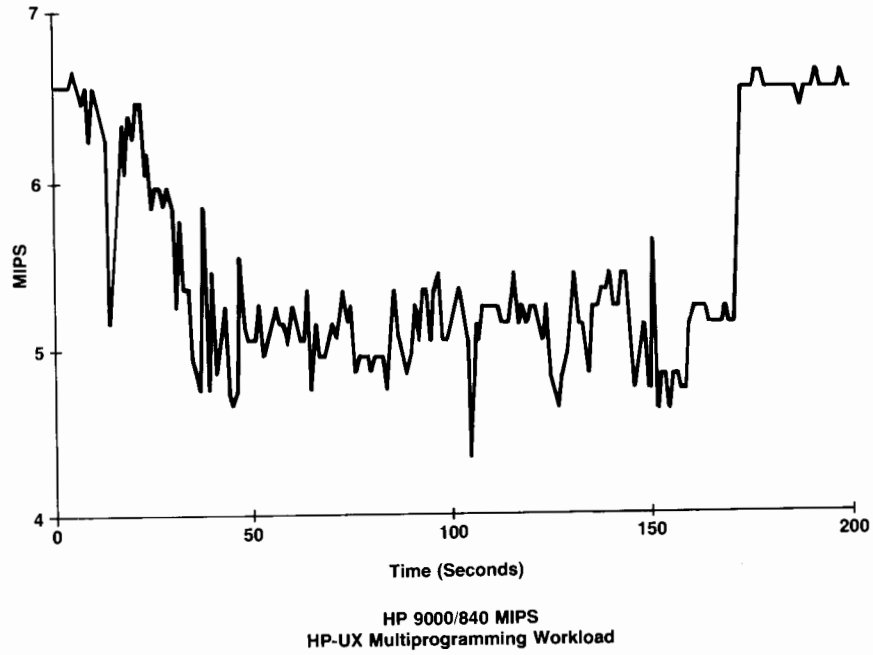


Fig. 7. *HP 9000 Model 840 measured performance in millions of instructions per second (MIPS) during a typical operating period of 200 seconds.*

more than one cycle to return the needed data. An example of this is when a piece of data is used for the first time. It cannot be in the cache, and must be fetched from memory. It is these less frequent cases that determine cache (and processor) performance.

The largest, and frequently only, penalty associated with caches is the miss penalty. It is the time the cache requires to get information (data or instructions) from memory. The cache needs to start a memory transaction, get the data back, and save it. All this time is counted as the miss penalty.

Effects of Separate Caches

The processor is capable of fetching an instruction every machine cycle, and some of these cycles, about 40%, also require a data cache reference. To prevent a large CPI increase (about 0.4), it is necessary to use the instruction and data caches during the same cycle. If the caches are combined (instruction and data cache are the same), the cache must be accessed twice per cycle. This becomes difficult to do with large caches, because the RAMs are just not fast enough. Splitting the cache into an instruction cache and a data cache allows both caches to be used in the same cycle without making them faster.

To prevent thrashing (excessive cache missing), a combined cache should also be at least two-way (two-set) associative. However, multiway associativity also slows down the cache, making it more difficult to build. Splitting the cache also eliminates this problem.

The decision to have separate instruction and data caches allowed us to use a large cache; a 128K-byte combined cache would have been difficult and expensive to build. The split cache does have a slightly lower hit rate than the combined cache, but this presented less of a problem than alternative cache organizations.

The cache is direct mapped, which means that each virtual address has exactly one place it can go in the cache. This is also known as a one-way associative cache. In two-way or four-way associative caches, a single virtual address can go in two or four places in the cache. Any number of ways can be built, but the expense is great, so normally, two-way or four-way associativity is used. Since each virtual address can go in more than one spot, there is less likely to be a conflict between two addresses. Thus the miss rate of the cache is lower. Simulations of the Model 840/Series 930 cache show that the miss rate would be about 40% lower if a four-way associative cache had been used. But this would have cost nearly four times as much in hardware, and would have required an increase in the cycle time of the machine, since the cache access takes longer when the cache must determine which way (set) contains the data. The benefits did not seem commensurate with the cost.

Write-Through versus Write-To

When the processor writes to the cache, the cache can do the write to memory at once (a write-through cache), or it can save the data and do the write to memory later (a write-to cache). The write-through cache has the benefit of always having a correct copy of the data in memory, something nice for I/O. A write-to cache reduces the bus traffic considerably, since only a small number of writes generate a bus transaction.

Keeping the memory up to date is a problem that HP Precision Architecture leaves to software. That removes most of the addtional complexity normally associated with the hardware on a write-to cache, and made the choice of a write-to cache clear.

Caches are something that software has traditionally not been able to control. But HP Precision Architecture provides explicit instructions for the software to maintain the caches. These are the purge and flush instructions, and all their variations. A purge removes the information from the cache without saving anything. A flush does the same, except that it writes the contents back into memory (for a write-to cache) before destroying it. The instructions come in two flavors, one for the instruction cache and one for the data cache. There is no purge instruction cache instruction, since a program can never write to (change) the instruction cache.

Why does this make a difference for performance? The explicit purge and flush instructions take time to execute. Traditionally this time has not been required. But the hardware complexity is significantly less, and therefore we can build faster caches. Although it is difficult to measure, we believe that the cost of the explicit instructions is less than the performance gain from the simplicity of the design.

Memory transactions in the Model 840/Series 930 processor occur on the MidBus, and are 16-byte (4-word) reads and writes, named READ16 and WRITE16. After a cache miss, the bus states are:

READ16	WRITE16
(cache miss)	(cache miss)
Address	Address
Dead Cycle	Dead Cycle
Tristate	Data 0
Data 0	Data 1
Data 1	Data 2
Data 2	Data 3
Data 3	Tristate
Tristate	

The basic bus cycle is the same as the processor's, 125 ns. Two things are key: the latency and the data rate. The latency is how long it takes to get the first word of data back (time from address to first data). The data rate is how fast the data comes back once it starts going. If the latency is high, the miss penalty will be high. It may be best to load more data in that case. If the data rate is the same as the instruction rate, bypassing the instruction cache can make a lot of sense. In general, the latency is largely determined by the bus and the memory (dynamic RAMs) used. It is hard to reduce. The data rate is also determined by the bus, but it is easily controlled by adjusting the bus width and cycle time.

Effects of Bypassing

Cache line bypassing is the concept of using the instructions or data as they are loaded into the cache, rather than waiting for the cache miss to finish. This reduces the penalty of the cache miss, but it has a few problems, too. In the Model 840/Series 930 processor, about three-quarters

of the instruction cache misses occur on the first word of a line, and the rest are evenly distributed among the other three words. The instruction cache penalty on the Model 840/Series 930 processor is 9 cycles with no bypassing, but is from 5 to 8 cycles if bypassing is used. The effective miss penalty is calculated by summing 75% of 5, 8% of 6, 8% of 7, and 8% of 8. This comes out to 5.12, much better than 9.

When actually measured, the effective miss penalty is 5.35, close to the calculated value. Comparing these numbers (still incomplete) at this point gives a CPI contribution of 0.16 with bypassing, and 0.27 without bypassing. These numbers were calculated by multiplying the I-cache miss rate (assumed 3%) by the respective miss penalties.

Some other things must be considered. If the processor is not executing the instructions in the same order and at the same speed as they are coming into the cache, the analysis breaks down. The Model 840/Series 930 processor is capable of doing this, but there are two important exceptions: a processor freeze and a taken branch. If the processor freezes for any reason other than the instruction cache miss, the instructions coming from memory are now too early for the processor. In the case of a branch that is taken, the instructions coming from memory simply are not the proper instructions. In both cases, the processor is refrozen and the instructions from memory are ignored. However, this is what happens anyway; there is no additional penalty for bypassing. On the Model 840/Series 930 processor, this refreeze adds 1.90 cycles to the effective I-cache miss penalty. This brings the total miss penalty to 7.25. Recalculating the CPI, a bypassed I-cache adds 0.22, while no bypassing adds 0.27. The processor performance gain by using bypassing is 0.05 CPI multiplied by 8 raw MIPS, or 0.40 MIPS, about a 10% gain.

Bypassing is also done on the data cache. Here it is not nearly so important. Since the cache can only handle one cache operation at a time and the cache operation is not finished until the miss has been handled, bypassing only works for one access per line. The CPI contributions, calculated in a similar way as above (but more complicated) are 0.10 for bypassing versus 0.13 for no bypassing. Bypassing is done on the data cache simply because it was easy to implement. The logic existed for the instruction cache (where it makes a larger difference), and was easy to multiplex between the two caches.

The following table summarizes the bypass and no-bypass performance calculations.

	no bypass (CPI)	bypass (CPI)	difference (CPI)	(MIPS)
I-cache	0.27	0.22	0.05	0.40
D-cache	0.13	0.10	0.03	0.24

Bypassing has one other problem. If the bus supports retries (a third party requests that the current bus transaction be ignored and tried again), the retry must be known before the first data word is used. Normally this means that the retry signal must be present with or before the first data word. Because of control complexity, the retry signal must be present on the MidBus one cycle before the first data word in the Model 840/Series 930 processor.

Critical Word First, Line Size, and Cache Size

Critical word first is an idea that only makes sense with cache line bypassing. It is the idea of rearranging the data on the transaction so that the needed word comes first, and the rest come later. For example, if the second word (word 1) is needed first, the memory would return the data in this order: word 1, word 2, word 3, then word 0. Taking a look back at the miss penalty for the instruction cache (5.35), we can see that this doesn't make a lot of sense. A miss penalty of 5 is the best we could do. Critical word first also potentially introduces a penalty on the 32-byte memory transactions, a discussion of which is beyond the scope of this paper. The Model 840/Series 930 cache and memory do not support critical word first.

Caches load one line at a time from memory. How big should this line be? Typically, the larger the line size, the more efficiently it can be loaded from memory. But with large lines, it is more likely that words will be loaded that will never be used. Since the instruction cache is normally used in a regular way, it benefits more from a large line size than the data cache. The Model 840/Series 930 processor uses a line size of 16 bytes (4 words) for both the instruction and the data caches. A line size of eight words would have been better for the instruction cache, but would have been difficult to implement since the Model 840/Series 930 has combined, pipelined cache tags.

Determining how big to build a cache is sometimes difficult. The larger the cache, the lower the miss rate will be, as long as you stay in the same global address space. Some processors do not have a single address space large enough to handle multiprocessing. Operating systems can get around this problem by flushing the TLB on process switches. Sometimes it is necessary to flush the cache, too. If the processor must flush the cache on a process switch, there comes a point where building larger caches may not help, and may even hurt system performance. This would put a practical limit on the size of TLBs and caches in such systems. The larger they are, the longer they take to flush. Also, the larger they are, the less likely they are to be fully used before a process is switched out again. HP Precision Architecture solves this problem by having a single large address space. All processes share this common address space, and no flushing needs to be done on either the TLB or the cache during process switches. With HP Precision Architecture, larger caches are always higher-performance, as long as the cycle time is not affected.

Floating-Point Coprocessor

In HP Precision Architecture, floating-point operations are handled by a coprocessor. This coprocessor runs concurrently with the main CPU and has the sole job of supporting floating-point arithmetic. Floating-point arithmetic is well-suited for a coprocessor because it involves calculations that require multiple cycles to perform. This means that although the floating-point instruction occupies one position in the instruction stream, the main CPU can receive and execute subsequent non-floating-point instructions concurrently with subsequent cycles of the floating-

point instruction.

As a true coprocessor, the floating-point board decodes its own instructions. It also has its own set of sixteen 64-bit registers. Data is sent to the floating-point board by loads from the cache and results are retrieved via stores to the cache. The floating-point operations are performed according to the IEEE standard for binary floating-point arithmetic.

The floating-point board is not equipped to handle all floating-point operations required, nor all floating-point number values. When an unsupported operation or data value is detected, the floating-point board indicates this fact to the main CPU, which then handles the operation in software. The occurrence of these operations requiring software is rare and does not significantly affect performance, but does ensure that the IEEE standard is completely satisfied.

Overlapped Processing

During the initial investigation for the floating-point board, a simple design was considered, which did not allow overlap of any type of floating-point instuctions. However, several types of floating-point applications (e.g., matrix multiply, vector sum, 3×3 graphics transformation, etc.) were examined and estimates of performance were made. It quickly became apparent that the initial design did not meet the performance goals. Furthermore, in most of the applications examined, allowing nonconflicting floating-point loads and stores to be processed while floating-point operations (flops) were being executed increased the performance enough to meet the goals. Therefore, the floating-point coprocessor implements this capability.

Once the capability of the board was decided, the design was implemented in a simple and straightforward manner. It was determined that microcode would be used to execute flops, while floating-point loads and stores would be implemented completely in hardware. A proprietary HP floating-point add, multiply, and divide chip set used in the HP 9000 Model 550 was selected to do the floating-point calculations. Also, a special assembler was written to convert the source microcode into a listing file and a burn file, with the latter being used to program the microcode PROMs.

The design choices led to the partitioning of the floating-point board into two distinct parts, the micromachine and the state machine (see Fig. 8). These two control pieces share a common data bus and both have access to the register file. They share these data paths on a time basis, with each in control for half of the 125-ns cycle time. The state machine does all of the interfacing with the main CPU and provides control for floating-point loads and stores. It also dispatches the micromachine to execute flops and signals the main CPU when traps or freezes are necessary.

The organization of the floating-point board into these two distinct control blocks is well-suited for allowing floating-point loads and stores to be processed while flops are in progress. When the state machine receives a floating-point load or store it simultaneously determines whether the micromachine is busy, and if so, determines whether the new load or store conflicts with the flop in progress by accessing a floating-point register used in the flop. If there is no conflict, the state machine processes the floating-point load or store immediately; otherwise, it waits until the flop is completed.

Self-Test

A full self-test strategy is built into the floating-point board design. A special, machine dependent instruction was created, which when executed causes the micromachine to perform a detailed diagnostic test of the floating-point board. This test signals pass or error conditions by setting bits in one of the floating-point registers. It also puts the one's complement of these bits into another register so that the validity of the information in the first register can be verified. Also included in the self-test strategy is the ability of the main CPU to distinguish between a malfunctioning floating-point board and the lack of a floating-point board in the system.

At power-up, a self-test is run on the floating-point board. The test includes the microcoded self-test as well as one instruction of each type of floating-point operation to ensure that the board is performing correctly. If this is determined to be the case, then the system is configured with the floating-point board enabled. If this is not the case, an error is signaled during initialization. The floating-point card can then be removed and the system rebooted without the floating-point board enabled. Then all floating-point operations are performed in software.

Development Methods

Once the floating-point hardware was solid enough to run in a system environment, a software test package was written to aid in catching numerical errors. This package allows us to test any flop with any operands. It also offers a pattern mode, during which operands are generated in a user-controlled pattern and continually used in flops. It functions by checking results received from the floating-point hardware with results derived from software floating-point routines. During development, if discrepancies were found between the two results, the operands, operation, and results were logged to an error file. In the pattern mode of operation, this software package caught a few numerical flaws both in the floating-point hardware and in the software floating-point routines. The mistakes in the software floating-point routines were equally important to correct

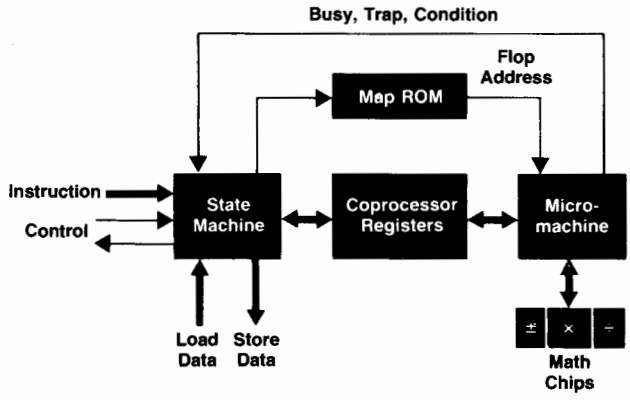


Fig. 8. *Floating-point coprocessor block diagram.*

because in the absence of hardware, they would be used to do the calculations.

To get the maximum performance from the floating-point board, the compiler groups and especially those connected with the optimizer were brought into the design process in the early stages. They were given information on how long each flop took to execute and what constitutes a conflicting load or store versus a nonconflicting load or store. Using this information along with their knowledge of the operating characteristics of the main CPU, they altered their compilers and optimizers so that the instruction ordering and floating-point register allocation they generated would take the best advantage of the floating-point hardware that existed.

The inclusion of the floating-point coprocessor with the main CPU allows the hardware to be used in a technical environment. Its position in the system as a coprocessor enhances the overall system performance by allowing the CPU to do non-floating-point activities such as address generation and integer arithmetic while multicycle flops are in progress. The floating-point board's ability to do floating-point loads and stores concurrently with flops means that while one result is being calculated, previous results can be stored and operands for future flops can be loaded. In summary, the design of the floating-point board, its placement in the system, and its influence on the compilers and optimizers all serve to get the maximum technical performance from the technology used to implement the design.

Memory System

The Model 840/Series 930 memory system is designed to maximize performance by keeping the latency from address to first word as small as possible. All signals were potential critical path signals and had to be analyzed carefully to ensure that the timing goals were met. The memory strobe lines RAS and CAS were closely analyzed so that the skew was minimized.

The DRAMs are accessed using nibble mode so that a read operation can return a word of data every 125 ns after a latency period of 300 ns. The memory controller is implemented using TTL technology.

The memory system communicates with the processor and the I/O system through the MidBus, which is a synchronous high-speed bus. There is parity checking on the MidBus for the address, data, and control lines. The memory generates and checks parity on data reads and data writes to improve the reliability of the memory system.

The memory can be accessed in either 16-byte or 32-byte transactions. The 16-byte transaction takes seven cycles and the 32-byte transaction requires 13 cycles. The maximum memory bandwidth for 16-byte transactions is 18.285 Mbytes/s and for 32-byte transactions is 19.7 Mbytes/s.

At power-up it is necessary to initialize the memory controller. The architected control and status registers are visible to the software in the I/O address space. The boot code initializes the memory controller, setting up the physical memory's address range via MidBus I/O transactions to the I/O registers resident on the controller.

Each 8M-byte main memory module is physically located on two boards. The memory controller board contains three banks of DRAMs and the memory array contains five banks of DRAMs. One memory controller communicates with one memory array card. While this increases the manufacturing price compared to a product that extends the reach of the controller to many memory cards, it has the advantage of reducing latency, since fewer DRAMs are addressed and the address and data buses are shorter.

27 bits of the 32-bit physical address are used. This limits addressability to a 128M-byte physical space.

Error Correction

Error-correcting memory is standard. A 32-bit error-detection and correction (EDC) chip forms the basis of this circuitry. During a memory write operation, 32 bits of data are sent through the EDC logic, which generates seven checkbits. These are merged with 32 bits of data in the proper RAM bank. When a memory read operation occurs, these 39 bits are sent through the EDC logic, which internally regenerates what the seven checkbits should be and compares them to the checkbits that it actually got from the RAM bank. The result of this comparison is called a syndrome. The checkbits for each 32-bit pattern are chosen so that the syndrome reveals useful information about any errors that are detected. If the error is a single-bit error, the syndrome can be decoded to see which bit is wrong. The EDC does this and corrects the error. Multiple-bit errors are not correctable. The best that can be done is to detect their presence and interrupt the processor by pulling on the error signal on the MidBus.

System Monitor Module

The power system consists of several major components:

- Ac front-end power distribution unit
- 5-kVA isolation transformer
- Fan tray with four ac fans and backup battery
- Three 300W power supplies
- System monitor module
- Internal and external control panels.

The relationship among these components is represented in the system block diagram, Fig. 9. The system monitor module serves as an interface between the power supply and the SPU boards. It generates secondary power (+5V S1 and +5V S2) to the CPU and memory boards, processes power-on and powerfail warning signals to the MidBus, terminates and arbitrates MidBus signals, monitors system temperature, and interfaces with the control panel and access port. It also includes some miscellaneous processor dependent hardware consisting of the time-of-day clock, stable storage, diagnostic switches, and an EPROM for processor dependent code.

The secondary power is generated by two dc-to-dc converters. In the normal mode of operation, the system monitor module converts +28V from two of the power supplies to +5.1V to supply up to six memory boards, or 24M bytes of memory. During the backup mode, the dc-to-dc converters take power from the 10V 10A backup battery. The battery can supply up to six memory boards for at least 15 minutes during a powerfail. An external battery connec-

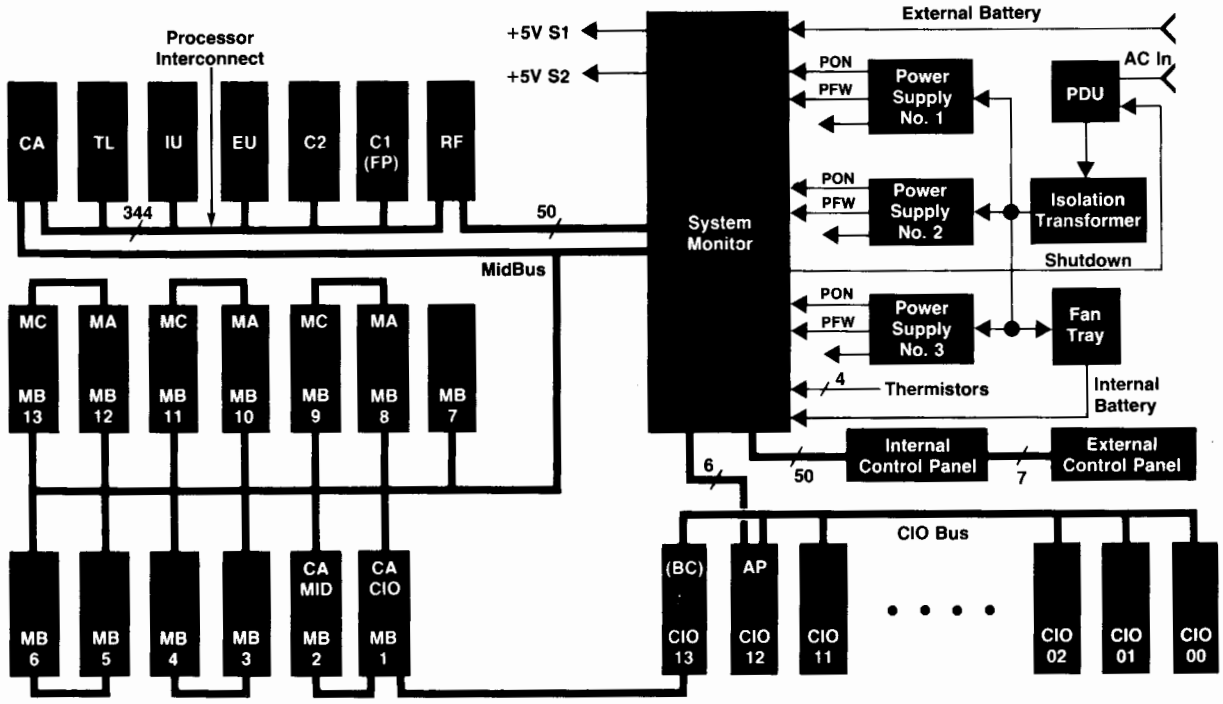


Fig. 9. *System diagram of the HP 9000 Model 840 and HP 3000 Series 930 Computers.*

tor is provided if additional backup time is required. The battery enable switch is built into the ac circuit breaker on the power distribution unit. Another battery test switch on the power distribution unit can bypass the battery enable switch for powerfail testing.

The system temperature is monitored by four thermistors on the backplane. When an overtemperature situation occurs, the system monitor module turns on the yellow warning light and sets a flag to the CPU at 45°C inside the cabinet. At 60°C, the system monitor module trips the ac circuit breaker and shuts off all the power including battery backup.

The processor dependent hardware on the system monitor module communicates with the CPU via diagnostic instructions. The diagnostic instructions have access to the control-panel hexadecimal status displays, the time-of-day clock chip, the stable-storage CMOS RAM, and the diagnostic switches. The software can read from and write to these components via entry points in the processor dependent code. The clock chip and the stable-storage RAM are backed up by two lithium batteries on the system monitor module during powerfail. A parallel-to-serial interface converts 16 bits of hexadecimal display data serially to the access port to facilitate remote diagnosis.

Acknowledgments

Many people contributed to the hardware development of the Model 840/Series 930. The authors would particularly like to thank Lee Moncton for his guidance as our original project manager, Joe Mixsell for helping us keep the faith during the dark times, Paul Bliley and Chris Livingston for their technical assistance during all phases of the project, Darlene Harrell for prototyping work, Rosemary Kingsley for production documentation and making sure we always had parts when we needed them, Ken Robertson, Luann Piccard, and Bea Netter for the mechanical and industrial design, Al Hum, Randy Teegarden, and Tom Wylegala for production engineering, Jess Pawlak for supporting those first 36 systems, Jerry Everett and his people for supporting over 400 lab prototype and production prototype systems, Jim Finnell for the original cache design, Don Cross for the I/O channel design, Don Williamson for the original E-Unit design, Chuck Gebber, Tom Harms, and Albert Chun for their contributions to the floating-point board, Julie Wu, Tom Alexander, and Carl Woodard for taking over and maintaining the majority of the original CPU design, Bill Shellooe for his invaluable help in creating and maintaining our tools, and Cheryl Gressman and all of the people in the printed circuit design shop.

References

1. M.J. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pp. 4-21.

2. D.V. James, et al, "Hewlett-Packard Precision Architecture: The Input/Output System," *ibid*, pp. 23-30.

3. F.W. Clegg, et al, "The HP-UX Operating System on HP Precision Architecture Computers," *Hewlett-Packard Journal*, Vol. 37, no. 12, December 1986, pp. 4-22.

4. J.R. Busch, et al, "MPE XL: The Operating System for HP's Next Generation of Commercial Computer Systems," *Hewlett-Packard Journal*, to be published.

5. G. Buchanan, et al, "A Distributed Terminal Controller for HP Precision Architecture Computers Running the MPE XL Operating System," *this issue*.

An Automated Test System for the First HP Precision Architecture Computers

Besides testing for proper operation, the system gathers specific failure information and generates summary statistics to be used in improving the manufacturing process.

by Thomas B. Wylegala, Long C. Chow, and Randy J. Teegarden

THE AUTOMATED TEST SYSTEM for the first computers of the HP Precision Architecture family can test up to ten HP 9000 Model 840 or HP 3000 Series 930 Computers simultaneously. Fig. 1 is a block diagram of the test system.

A Model 840/Series 930 Computer configured with two special boards can be connected to the test system via a cable. The test system then has the ability to load diagnostic programs into the Model 840/Series 930 and monitor the results of those tests. The host for the test system is an HP 9000 Model 220, but any HP 9000 machine that runs the HP-UX 5.1 operating system could serve as well.

Key Features

The Model 840/Series 930 computer contains 32K bytes of test code resident in ROM. This self-test code is executed whenever the computer is powered on or reset. There is a need to supplement this test code with additional specialized test programs. Also, it is expensive to modify firmware based code, but easy to add a new test to the test system. Therefore, the test system provides the capability to download test programs into the memory of the computer under test and to initiate their execution.

The test system monitors the results of test execution and writes the status to a log file. This eliminates the need to have a human operator constantly observing the unit under test to judge whether the unit has passed. The test system collects the data elements that are critical to the success of the quality control program.

Little peripheral equipment is required to support the testing process. Without the test system, the minimum configuration to run diagnostics on Model 840/Series 930 processors includes a console and a disc for each unit under test. The peripherals for the testing of ten units would be

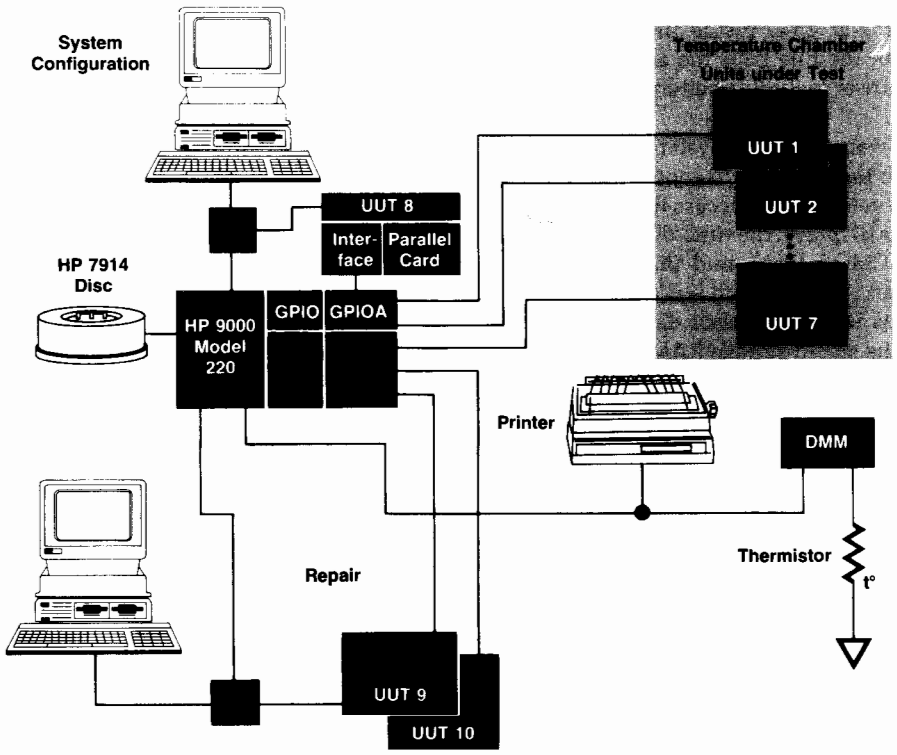


Fig. 1. *Block diagram of the test system for HP 9000 Model 840 and HP 3000 Series 930 processors.*

expensive and would consume valuable factory space.

Finally, the test system requires minimal cooperation from the unit under test. Even catastrophic self-test failures can be detected properly by the test system. It can even download test programs into a unit whose normal input/output channels are inoperable.

Hardware Developed

The communications interface between the test system and the unit under test is based on the link used to support the remote debugger (RDB),[1] so the test system is fully compatible with RDB. That link involves a 16-bit path between a GPIO card in the HP 9000 host and a parallel card in the Model 840/Series 930 under test. For the test system, two custom boards (called the test system interface and the GPIOA adapter) were designed to enhance the RDB link. The connections are shown in Fig. 2.

Many advantages accrue from inserting these two custom cards into the path. Signals can be transmitted reliably over distances up to 300 feet (versus 3 feet for the original link) using an EIA RS-422 differential interface. The system will work with both the new differential drive and the original single-ended drive versions of the parallel card. The test system interface also has access to signals on the access port card slot. These signals allow the test system to give the unit a hard reset and to receive the 16 bits of serial data sent to the access port card. These 16 bits of data, which on Model 840/Series 930 processors also appear on the LED status display, enable the unit under test to send messages to the test system even if a failure in the CIO channel adapter prevents communication via the console. Finally, the test system can determine when the unit under test has been powered on or off.

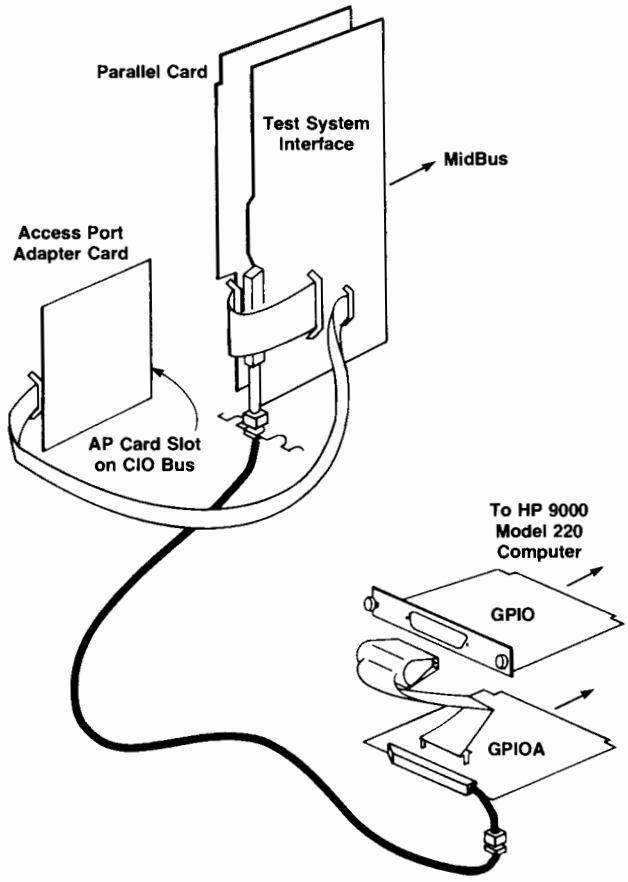


Fig. 2. *Parallel link configuration.*

Firmware Developed

The I/O dependent code (IODC)[2] for the parallel card was written to make communications with the test system possible. The IODC contains a communications server which enables the Model 840/Series 930 to interpret the data being transmitted to it by the test system through the parallel card. When the boot information in the Model 840/Series 930's stable storage area indicates that a parallel card is to serve as the boot device, the communications server is launched after the self-test has completed.

Software Developed

The test system control software has a multiprocess structure. The main process, which is scheduled when the test system is initiated, contains the user interface. It maintains the status windows, updates the softkeys, and executes user commands. There is one p_monitor process for each computer under test. These processes manage the communications with the units under test. Finally, there is a background process that performs periodic and intermittent tasks. The test system control software allows easy reconfiguration of each test station. Parameters that can be configured include the list of tests to be executed and the number of temperature cycles needed for test completion.

The test system allows diagnostic programs to be loaded into the computers under test. Developing these diagnostics was another challenge. Several sets of test programs were written. For example, there is an exerciser for the translation lookaside buffer (TLB) that verifies the proper operation of each field in every entry of the TLB. A total of 49 of the original architecture verification programs were adapted for use with the test system. These programs perform extensive testing of the arithmetic, logical, and branch instructions. The test programs obey some conventions to make the test system's job easier. For instance, the programs use the access port interface path to transmit error codes to the test system to indicate failures or a pass code to indicate when the test has completed successfully. All of the tests written for the Model 840/Series 930 use the data FFFF in hexadecimal to signal success and preface a failure message with the data DEAD. The Model 840/Series 930 tests also compute a checksum on their instruction text to ensure that the program was downloaded correctly.

Relation to the Manufacturing Process

Model 840/Series 930 boards are subjected to three levels of testing during their manufacture. First is a board-level in-circuit test on the HP 3065 Board Test System.[3] This test screens out most process related problems, such as bent pins and solder bridges. Next, boards are grouped into sets, installed into backplanes, and subjected to functional tests in a temperature chamber. The temperature chamber continuously cycles from 0 to 55°C with a period of two hours. This test detects temperature-sensitive component

failures and precipitates failures in marginal parts. The final step is to perform an operating system level verification test on the completely assembled unit. Verification test suites have been written for both the HP-UX and MPE XL operating systems.

The test system plays its biggest role in the temperature chamber phase of testing. Model 840/Series 930 boards spend between ten and twenty-four hours in the chamber. During this time, the test system supervises the execution of the prescribed set of test programs.

The temperature chamber test sequence has six phases.

1. The operator installs a board set into a backplane in the chamber and applies power to the unit. The test system automatically detects that the unit has powered on and spawns a p_monitor process for it.
2. The test system sends the unit a hard reset signal. The unit will perform its self-test and initiate the boot sequence.
3. The test system waits until the unit signals successful completion of the boot from the parallel card. Anything other than a boot completed message is considered an error.
4. The test system gets the next test to run from the test list. (The list is circular; the first test is rerun after the last test completes.) The test system downloads the test and directs the unit to execute it.
5. The test system allows time for the test to execute and checks on the result. If the unit passed, the test system continues with step 2.
6. In the event of an error, the test system records the error code, signals the operator that a failure has occurred, and attempts to rerun the test that failed. The system also generates a failure record to be added to the data base.

The test system is also a valuable aid in repairing defective boards. Repair technicians have access to the same set of diagnostic programs that first indicated the presence of the failure. The technicians can query the test system data base to determine the exact circumstances of the failure and any prior attempts at repair. This obviates the need to have paper tracking forms accompany the boards to the repair area.

To integrate the entire manufacturing operation, the test system is also used in the final configuration area. The full set of diagnostics is available, and the data base can be accessed. The test system also provides special utilities to initialize the units before shipment.

Results Observed

Before installation of the test system, the only test programs that the Model 840/Series 930 could execute in the temperature chamber were those contained in ROM, that is, the power-on self-test. The test system supplements the self-test with a variety of other tests. Moreover, the test system forces the Model 840/Series 930 to complete the entire boot sequence, in itself a good test. The effectiveness of the tests can be measured by the failure rate experienced in the next level of testing, that is, the operating system tests. In the fourteen-week period before installation of the test system, the failure rate experienced during the operating system tests was 19%. That is, of 228 sets of Model 840/Series 930 boards that successfully executed self-test in the chamber, 44 were subsequently shown to have hardware failures by the next level of testing. By contrast, during the first seven weeks in which the test system was operational, the failure rate dropped to 2.5% (only one failure out of 41 sets tested).

Summary

The test system is designed to fill two critical needs in computer manufacturing. The first is to subject the computers to stringent functional tests so that defects can be revealed at the earliest possible time. The second is to keep accurate records of the results of those tests so that the manufacturing process can be constantly improved. With respect to the first point, the test system has already been a great success. A number of failures were detected by testing in the temperature chamber that would otherwise have gone unnoticed until the operating system tests. Moreover, whenever better diagnostics are written, the test system will be ready to download them. The test system automatically generates a template for each failure occurrence; the operator need only input the serial number of the board that was defective. This data base can be scanned to reveal the weak points of the process.

Acknowledgments

The authors owe special thanks to Dave Campbell, who provided inspiration and support to the project. Other contributions were made by Craig Chatterton (interface to PDC), Tom Taylor (GPIO driver), and George Winski (temperature measurement).

References

1. D. Magenheimer, "Remote Debugger," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, p. 43.
2. D.V. James, et al, "Hewlett-Packard Precision Architecture: The Input/Output System," ibid, pp. 23-30.
3. T.R. Fay and J.E. McDermid, "The HP 3065 Board Test Family: A System Overview," *Hewlett-Packard Journal*, Vol. 35, no. 10, October 1984, pp. 4-9.

A Distributed Terminal Controller for HP Precision Architecture Computers Running the MPE XL Operating System

Up to 48 terminals or printers connected to each controller communicate with HP 3000 Series 930 or 950 Computers over an IEEE 802.3 local area network.

by Gregory F. Buchanan, François Gaullier, Olivier Krumeich, Eric Lecesne, Jean-Pierre Picq, and Heng V. Te

WITH THE EVER INCREASING capacity of computer systems comes the demand for more terminal connections. On the other hand, SPU (system processing unit) cabinets are becoming smaller with each generation. Finding space in these smaller SPU cabinets for the increasing number of terminal connections is a challenge.

For HP Precision Architecture computer systems using the commercial operating system, MPE XL, the solution is the HP 2345A Distributed Terminal Controller (DTC). The approach taken was to move the terminal connections out of the SPU cabinet and into the DTC.

The HP 2345A is designed for the HP 3000 Series 930 and Series 950 Computers. It enables up to 48 asynchronous devices (terminals or serial printers) to be connected to these systems over an IEEE 802.3 local area network (LAN), thereby greatly simplifying the cabling and lowering the associated costs. Future releases will allow a terminal user connected to the DTC to establish a session with an MPE XL system, and then by a simple command, to switch to another MPE XL system on the same LAN. This switching capability, combined with the possibility of distributing the DTC in a building, is a major contribution of this new commercial computer system family.

Software Architecture

Fig. 1 shows the overall DTC software architecture. To achieve the objectives of performance and simplicity, the DTC has a special operating system, AOS, which is a straightforward dispatcher with associated services like memory and timer management. An added benefit of this dedicated operating system was easy integration of the DTC software into a Pascal workstation along with debugging tools. This approach proved very efficient in the design, testing, and integration phases.

The stack of protocols in the DTC has been reduced to a minimum, and some layers of the ISO OSI model have been combined. Layers one and two are the standard IEEE 802.3 and IEEE 802.2 Class I. These standard protocols were chosen so that the DTC could share the LAN (the same physical cable, medium attachment unit, and LAN controller) with HP 3000 Network Services (NS/3000) or any IEEE 802.3 compatible devices.

For the upper OSI layers, no protocols existed that met the objectives of performance and simplicity, so proprietary protocols are used instead of standard protocols such as TCP/IP and TELNET. It was also felt that additional functionality was needed in the standard protocols to ensure satisfactory support of terminals in the MPE environment. A goal was to offload the character-oriented tasks, like character backspace or line delete processing, from the host and do it in the DTC to save processing power in the host and at the same time provide real-time feedback to the user's keystrokes at the terminal. This results in greater overall system efficiency and more friendliness for the cus-

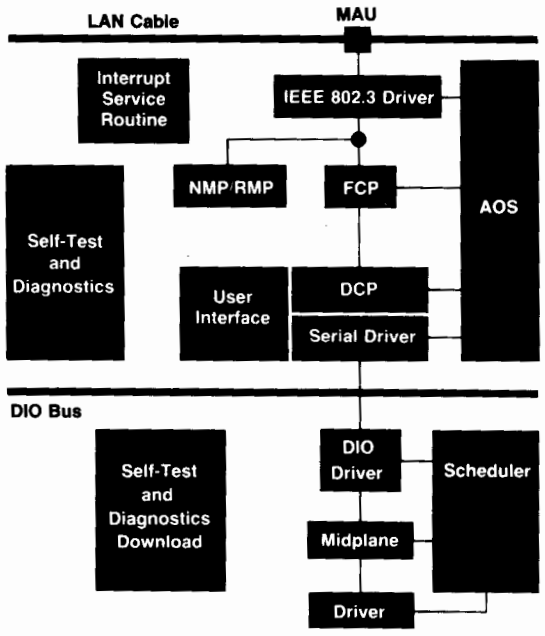


Fig. 1. *Distributed terminal controller (DTC) software structure. AOS is the DTC operating system. FCP is the flow control protocol. DCP is the device control protocol. NMP and RMP are network management and remote maintenance protocols.*

tomer, but it also means that the DTC needs a lot of intelligence and an elaborate software structure (Fig. 1) to organize this processing.

It was also apparent that some sort of transport mechanism is necessary to prevent flooding the DTC with data. Since the processing power of the host is at least an order of magnitude greater than the processing power of the DTC, the DTC needs some way to tell the host to stop sending data so the DTC can process what it has received. This functionality is also needed because a user at a terminal can always stop the flow of data and resume later, a printer can run out of paper, and so on. This requirement resulted in the definition of a special DTC flow control protocol.

Flow Control Protocol

As implied by the name, the most important feature of this protocol is to make sure that the DTC is not overflowed by the host. There are other protocols that provide this feature, but no single protocol that offers all of the features that are needed. The DTC flow control protocol (FCP) is based on a study done by Hewlett-Packard Laboratories in 1984 of a protocol known as Fast Path, a higher-performance, stripped-down version of the ARPA TCP/IP protocols. The DTC FCP is derived from Fast Path and from CCITT Recommendation X.25 Level III.

The features of the DTC FCP are:

- Simplicity
- Flow control
- Reliability
- Connection-oriented
- Connection assurance
- Fragmentation and reassembly.

To maintain DTC efficiency, the FCP has been kept very simple. Since it is layered on top of IEEE 802.3, which provides error detection, error detection is not part of the FCP; only error recovery is needed. The FCP was also designed knowing that errors on a LAN are infrequent, so the mainstream functions are optimized while the error recovery processing is more cumbersome.

The FCP is a reliable transport because it guarantees that all packets of data are delivered in the same order they were received, and that no packets are lost or duplicated. Reliable transport is needed because control information is exchanged between the DTC and the host, and algorithms that manage that exchange are simpler if they are based on a reliable transport protocol.

The FCP is a connection-oriented protocol. There is a simple procedure for establishing and breaking connections between the DTC and the host. With this scheme, each of the 48 ports of the DTC has its own connection for exchanging data and control information without interfering with the traffic on the other ports. This feature will be very helpful in the future to provide access to more than one host from the same port of a DTC.

To ensure reliable exchange of information on a port basis, a connection assurance mechanism was defined so that each end, the DTC or the host, can make sure that the other entity is still alive and functioning.

While MPE does not limit the size of a single write, the request has to be passed over the LAN, which restricts the size of the frames to 1518 bytes. Therefore, the FCP provides a way to fragment and reassemble the user data. The more-data bit provides this functionality.

The protocol is fully symmetrical. There is no master/slave relationship.

FCP Operation

To ease the implementation, the header of an FCP packet is of fixed length, as shown in Fig. 2. Seven types of packets are used to establish and break connections, to ensure reliable transport of information, and to provide connection assurance. To offer sustained throughput without a request/reply scheme, which is less efficient under heavy load, a window mechanism is used.

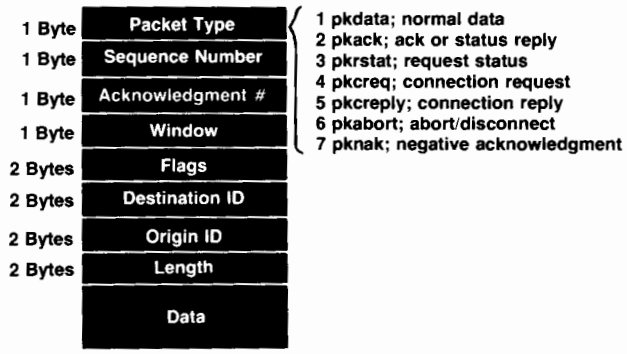


Fig. 2. *DTC flow control protocol packet format.*

Fig. 3 shows how flow control is performed using the window mechanism. Three fields are used in this algorithm: 1) the sequence number field, which identifies the packet sent, 2) the acknowledge number, which indicates the next sequence number that the receiver is waiting for, signaling to the sender that all packets sent previously have been received, and 3) the window field, which indicates the last sequence number the receiving end is willing to accept. In the example of Fig. 3, the left side opens a

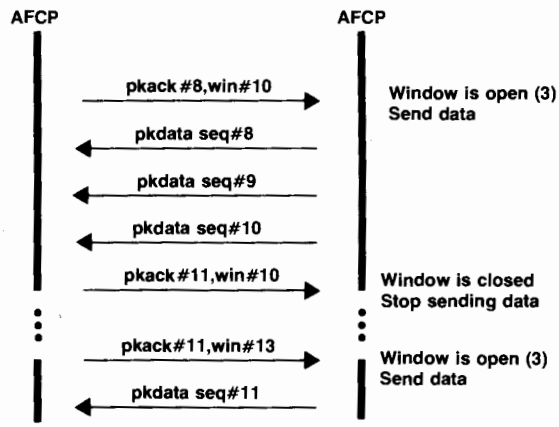


Fig. 3. *Flow control mechanism window field.*

window for three packets at the beginning, and the window closes automatically when the right side sends seq#10. The left side does not reopen the window with ack#11,win#10, but only confirms the window number and the last sequence number received. When memory becomes available, the left side reopens the window for three more packets. In the meantime, the right side waits for the window to reopen before sending any more data.

Device Control Protocol

As mentioned above, the DTC performs all the byte-oriented tasks. The host controls the behavior of the DTC using the DTC device control protocol (DCP). Fig. 4 gives an overview of DCP functionality. Similar protocols, for example CCITT Recommendation X.29 and the virtual terminal protocol used by HP network services products,[1] already existed but either could not be used for performance reasons or would have had to be modified so extensively that it seemed easier to define a new protocol. The DCP is designed to be simple, efficient, and compatible with the HP CIO (channel input/output) bus.

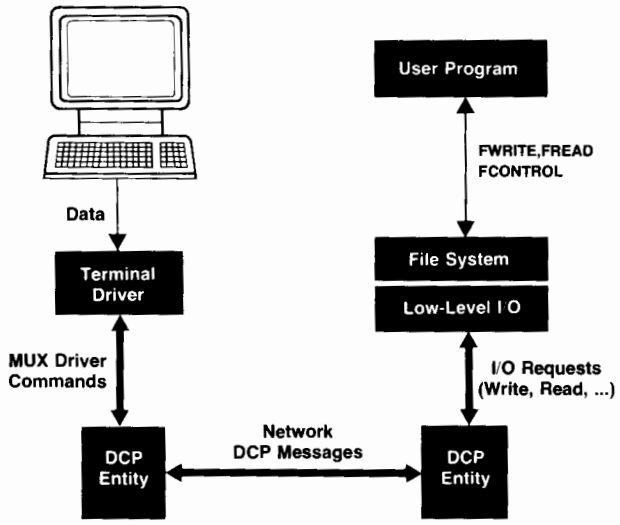


Fig. 4. *DTC device control protocol overview.*

Simplicity is achieved through a small number of message types and a fixed-length header. Fig. 5 shows the message types defined by the DCP. It is fairly easy to decode a request since the format is fixed and relevant information (request code, data length, etc.) is always in the same place. Also, when changing the value of a parameter to alter the behavior of the DTC, the parameter can always be found at the same place in the packet.

Because the protocol is compatible with the CIO bus, the same pieces of code can be shared to control devices, either over the internal bus of the host or remotely over a network. Sharing the same code and using the same protocol not only save resources, but also ensure that a terminal will always work the same way, regardless of the physical connection to the system. This important feature will guarantee

Request from the Host 5 Message Types	• Write Device Data • Write/Read Device Data • Write Port Configuration • Read Port Information • Control Port (abort)
Reply from the DTC 2 Message Types	• Write/Read Device Data Reply • Read Port Information Reply
Asynchronous Event to the Host 6 Message Types	• Break Detected • Subsystem Break Detected • Flow Control Timer Expired • Link Level Connected (Modem Port) • Link Level Disconnected (Modem Port) • Console Attention Character

Fig. 5. *Device control protocol message types.*

that in the future, even if new ways of connecting terminals are invented, the functionality and the behavior of those terminals will be the same, thus greatly simplifying migration to the new means of connection.

The device control protocol is implemented on a multiplexer board to use the hardware resources, such as microprocessor power and RAM space, efficiently. One example of this efficiency is in the design of the reply sent by the multiplexer card. The reply has a trailer instead of a header. This allows the multiplexer to start sending data before the request is fully completed, thus saving RAM space.

The DTC device control protocol is not symmetrical, the host being the master and the DTC being the slave.

The host configures a port of the DTC using the write port configuration message. Fig. 6 shows the format of this message. It contains all the configuration information for read, write, and modem control operations. It also specifies the speed and parity for the port's operation (data bytes 6 and 7). All MPE intrinsics are mapped into a set of values in this message, allowing the user program full control of the terminal.

The DTC returns user data from a read request with the message shown in Fig. 7. The format of this message allows the DTC to forward data to the host as soon as it is received from the terminal. This greatly reduces the amount of memory required to handle a read request, since there is no need to buffer the complete message before sending it. In addition, the protocol uses the fact that the LAN and the host can take data from the DTC faster than the terminal can deliver it. When the read is completed, the DTC sends the completion code, along with the time of the read, to the host in a fixed-length trailer in the last eight bytes of the complete message.

The asynchronous event message allows the DTC to get attention from the host, or to signal that conditions enabled through the write port configuration message have occurred. Examples of asynchronous events are the establishment of a modem connection, a user calling for attention (break or subsystem break), or a modem disconnection.

(continued on next page)

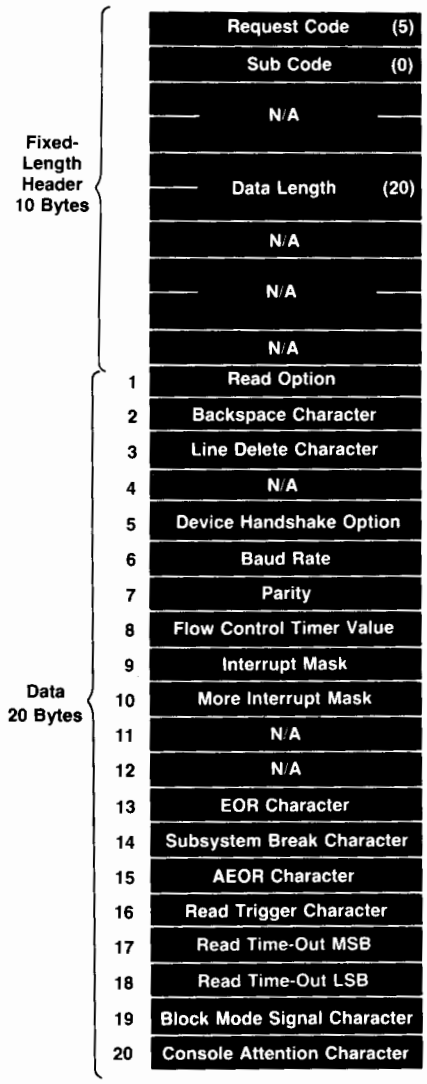


Fig. 6. *Device control protocol write port configuration message.*

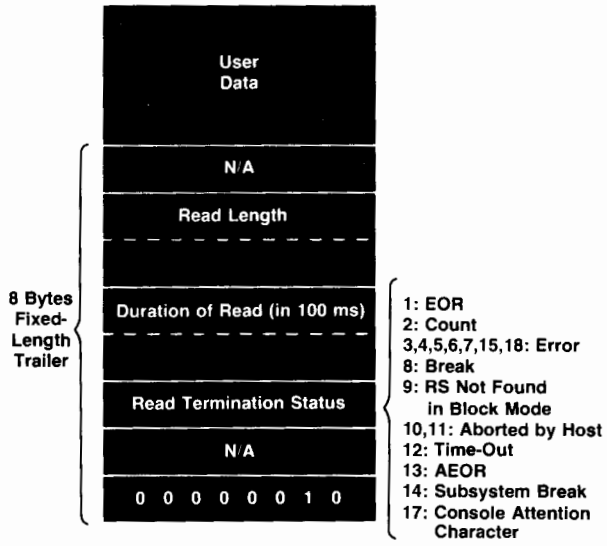


Fig. 7. *Device control protocol read/write data reply message format.*

Network Management and Remote Maintenance Protocols

The DTC can be looked at from two points of view. In the first, the DTC is treated as a local terminal multiplexer, except that it is connected to the SPU by an extended bus. Like a local multiplexer, the DTC belongs to a single SPU. The difference in managing the DTC is that commands are sent on the bus instead of the SPU having direct access to the multiplexer hardware. That the bus is an IEEE 802.3 LAN is irrevelant. It could just as easily be a fiber optic link, or some type of point-to-point connection.

From the second point of view, the DTC is treated as a service provider on a LAN. The service it provides is terminal and printer access to MPE XL systems. From this point of view, the DTC is not restricted to a single MPE XL system. Instead, it can provide its terminal access service to several MPE XL systems at once. This point of view multiplies the management problems because the DTC is a shared resource that may need to be managed by multiple MPE XL systems. To allow the DTC to grow, the second viewpoint was taken in developing the design criteria. To reduce the cost of the DTC, it does not have local mass storage. Instead, it depends on the MPE XL system to load its code and configuration files, and it sends memory dumps and logs errors to files on the MPE XL system. The on-line diagnostics for the DTC are controlled by the MPE XL system. All of these functions require exchanges of information between the DTC and MPE XL—in other words, a protocol.

Two protocols designed by other HP networking groups closely met the needs of the DTC. These are the remote maintenance protocol (RMP) and the network management protocol (NMP). The DTC design team helped define these protocols, and the DTC is one of the first products to use them. Using these HP standard protocols will allow the DTC to fit easily into the HP network management strategy.

The DTC uses the remote maintenance protocol to download its code and configuration files, and to upload diagnostic dumps. To begin a download sequence, the DTC sends a boot request packet asking for the location of its code file. The boot request is sent to a LAN multicast address, so all MPE XL systems on the LAN receive it. Each of the MPE XL systems looks in its configuration file to see if it has the code and configuration files for this particular DTC. If so, the MPE XL system responds with a boot reply, identifying itself to the DTC. The DTC picks one of the responding systems as its active loader and starts asking for code segments. The segment loading is carried out by a simple request/reply exchange of packets with a time-out retry. The DTC requests code segments one at a time from the MPE XL system. If it doesn't get a response within two seconds, it asks again. When it gets the reply the DTC loads the code and asks for the next segment. Once the load is done, the DTC sends a boot complete packet telling the

MPE XL system the load is complete. A similar sequence is used to load the DTC-specific configuration file, except that the opening request is sent to the MPE XL system that loaded the code file, and not to a multicast address. The configuration file contains information such as the speed and parity for each of the terminal connections. Once the DTC has loaded its code and configuration, it comes on-line.

The DTC uses the network management protocol to provide on-line diagnostics from the MPE XL system. These functions include resetting ports, loopback testing, status inquiry, and error logging. The starting point for these diagnostics was the current MPE Termdsm diagnostic. However, changes were required to reflect the separation of the DTC from the SPU. Improvements have also been made in presenting the data to the user. New Termdsm commands were added to display the status of the DTC as a whole, and to display the status of a serial port. Both status commands format the data into customer understandable form. For example, state information is displayed in the user's native language (e.g., "read pending") rather than as a number. The goal of these two displays is to allow the customer to do the first level of problem diagnosis, rather than having to call in HP service personnel. An example section of a port status display is shown in Fig. 8.

(IEEE 802.3 type 10 base 2). Only one of these network connections can be used at a time.

The processor card uses an 8-MHz 68000 microprocessor with 512K bytes of RAM and 64K bytes of EPROM. Also on the card are a network interface, timers, and DIO drivers.

The serial interface card handles the processing, storage, and multiplexing of the data to and from eight terminals. It must be connected to a connector card located on the opposite side of the backplane. The serial interface card circuitry mainly consists of a Z80B processor, 16K bytes of EPROM, 32K bytes of shared RAM (accessible by the Z80B and the processor card), 8K bytes of fast RAM (accessible only by the Z80B), and USARTs.

The connector card is used for physical attachment of the terminals. Three types of connector cards are offered for six RS-232-C modem connections, eight RS-232-C direct connections, or eight RS-422 connections with electrical isolation. It is possible to mix the three types of connections in the same DTC.

A display located on the front panel and connected to the processor card shows the DTC status (download of code in progress, errors, self-test results, etc.). The DTC firmware contains a self-test, which is executed automatically at each power-on, and an off-line diagnostic program, which is run from a local terminal connected to DTC port zero. No host

```
SIC State :
            Read Pending

                 Number Of User Characters Received By SIC =     0
                    Last Special Character Received By SIC = $0D

Modem State :
           Clear To Send =  ON              Request To Send =  ON
         Data Set Ready =  ON          Data Terminal Ready =  ON
         Ring Indicator =  ON             Data Rate Select = ON
     Data Carrier Detect =  ON

        Disconnect Timer =   255               Open Timer = 65535
               DCD Timer = 65535             DCD Counter =   255

              Backspace = $08                  Line Delete = $18
           EOR Character = $0D    Subsystem Break Character = $19
  Alternate EOR Character = $00       Read Trigger Character = $11
  Block Mode Signal Char. = $12
Press <RETURN> to continue >>

 LINE    MODIFY   BLOCK    REMOTE          TERMINAL  MEMORY  DISPLAY    AUTO
 MODIFY    ALL     MODE     MODE             TEST     LOCK    FUNCTNS     LF
```

Fig. 8. *A portion of a port status display.*

DTC Hardware

The DTC is designed around a central backplane (based on the DIO bus) using a press-fit technology. Up to six connector cards and serial interface cards can be connected to the processor card through the backplane (see Fig. 9). The processor card has been leveraged from a board designed at the HP Colorado Networks Division. It includes the LAN access interface and the DTC's overall management functions.

Two types of LAN accesses are offered: either Backbone LAN cable (IEEE 802.3 type 10 base 5) or ThinLAN cable connection is needed to use the off-line diagnostic. The DTC hardware is fully tested by the self-test and the off-line diagnostic.

After code is downloaded from the host to the DTC, on-line diagnostics can be started from the system console to cause various diagnostic and control functions to be executed by the DTC (see preceding section).

The design of the DTC hardware is generic in the sense that it can support almost any software architecture. This ensures compatibility with future applications such as support of new peripherals or concurrent stacks of protocols.

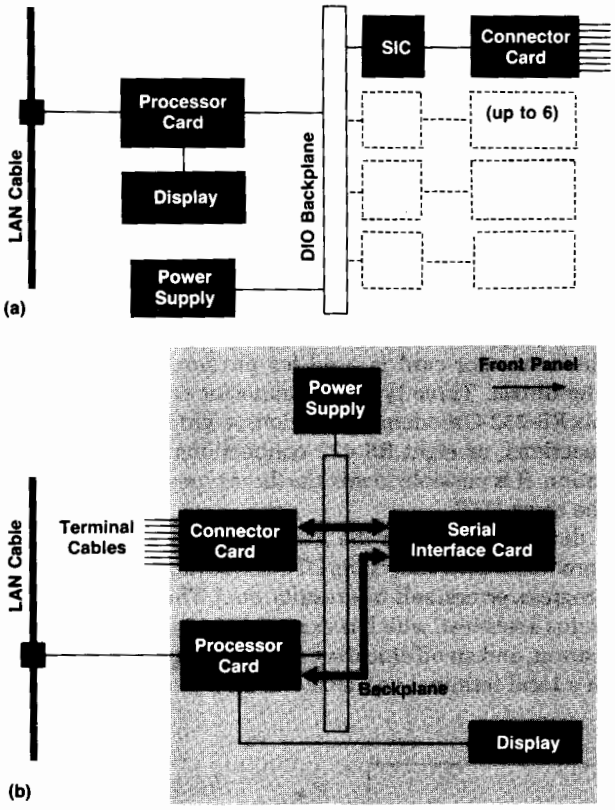


Fig. 9. *(a) DTC hardware architecture. (b) Physical layout.*

Performance Prediction and Measurement

The theory of operation of the DTC is based on queuing theory,[2] specifically the M/M/1 systems, in which customers of a service arrive in a system made up of a queue and a server. The arrivals have a Poisson pattern. The queue discipline is FIFO. The processing time of the server is also a Poisson process. There is only one server for the queue. The average arrival rate in the queue is L. The mean service time of the server is 1/M. Then the mean time spent by the customers in the system (queue + server) is $T = 1/(M - L)$.

In the DTC, each protocol communicates with the others using messages (carrying user data or not). But only one protocol executes at a time. When a protocol sends a message, this message has to be queued by the DTC operating system, AOS. When the currently executing protocol exits, AOS calls the next protocol, which is the destination of the first message of the queue. Thus the DTC can be seen as a queuing system made up of a queue of customers (the messages) and a server (the destination protocol of the first message of the queue).

With this design, a packet that has to be processed by several protocols (IEEE 802.3, DTC FCP, DTC DCP) is queued several times to go across the DTC once. Some of the messages processed are put in the queue again, so that the flow out of the server partially reenters the queue, as shown in Fig. 10.

Let $E = 1/M$ be the mean execution time of the protocols. This is the mean service time of the queuing system. Let L be the incoming flow, that is, the sum of the packets received from the LAN or the terminals. Let N be the number of protocols that must be activated to receive or transmit one packet ($N = 5$). The rate of arrivals in the queuing system is NL, and 1/L is the mean time between message arrivals in the queue.

For the sake of simplicity, assume that the arrival rate of packets and the processing times are Poisson processes (although this is not always the case). To go through the DTC, a packet goes through an M/M/1 system several times (Fig. 10); the DTC is a network of M/M/1 systems. Queuing theory shows that in the case of a network of queues, if the arrival rate in one node is a Poisson process, and if this node partitions this stream into several streams with a given probability of choice, the partitioned streams are also Poisson processes. In the same way, if a node merges several streams into one stream, the resulting stream is also a Poisson process. For this reason, the formulas of M/M/1 systems are still valid (Jackson's theorem).[2]

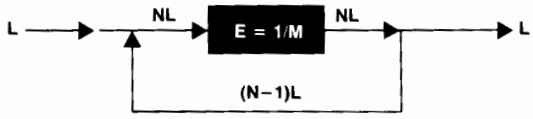


Fig. 10. *DTC queuing model.*

By applying the M/M/1 system model, the time to go through the system once is found to be:

$$\frac{1}{1/E - NL}$$

If an item (packet, etc.) in the DTC undergoes a cycle made up of P tasks, the time to perform this cycle is:

$$\frac{PE}{1 - NLE}$$

Obviously, NLE must be smaller than 1, since:

$$L < \frac{1}{NE} = \text{maximum throughput.}$$

We want to calculate the latency of the DTC and its maximum throughput. Most of the results depend on the traffic and/or on the number of active connections. Results will be given here as functions of the total traffic L, the sum in both directions of all of the flow rates of all of the active connections.

Acknowledgments versus Useful Packets

At the DTC FCP level, two kinds of packets travel on the LAN: data packets, which carry user data and correspond to one I/O operation, and service packets (ACK, NAK). The total throughput is the sum of the data throughput and the number of ACK/NAKs per second. As the processing time of each node limits the total throughput, the data throughput depends on the rate of ACK/NAKs. The LAN is reliable

enough and memory pools are dimensioned so that overruns occur rarely enough to neglect the NAKs and the duplicated packets. To estimate the data throughput we therefore need to know the rate of ACKs.

The number of ACKs per transaction depends on the structure of a transaction, and on the window. According to a real-time analysis of MPE systems conducted at Hewlett-Packard Laboratories, we can consider that each transaction is made up, on the average, of 1.4 reads, 7.2 writes, and 2.2 controls, for a total of 10.8 terminal I/Os. 90% of the user transactions will require 15 terminal I/Os per user transaction or less. This means that the DTC receives, on the average, 10.8 packets when it transmits 1.4 packets (that ratio is a result of the assumption of Poisson processes). The packet transmitted by the DTC in a transaction implies the receipt of one ACK. The packets received by the DTC imply a number of ACKs, depending on the window size and on the value of the stand-alone acknowledgment timer. If this timer is correctly tuned, the number of ACKs transmitted per transaction will be 10.8 divided by the value of the window. Therefore, the traffic will have the following structure:

User packets transmitted per transaction	1.4
User packets received per transaction	10.8
ACKs received per transaction	1

The number of ACKs and therefore the total number of packets exchanged in each transaction depends on the window size:

Window	1	2	3	4	5
ACKs transmitted per transaction	10.8	5.4	3.6	2.7	2.1
Packets per transaction	24	18.6	16.8	15.9	15.3

We can deduce the mean number of ACKs exchanged (received or transmitted) per user packet exchanged:

Window	1	2	3	4	5
ACKs exchanged per packet	0.96	0.52	0.37	0.3	0.25

Let U be the ratio of total traffic to data traffic. Since the total traffic is equal to the data traffic plus the ACKs, U has the following values, depending on the window:

Window	1	2	3	4	5
U	1.96	1.52	1.37	1.3	1.25

Protocol Execution Time

The execution times are different for each protocol. These times were calculated and later confirmed by actual measurements on the DTC. The mean value of E for data packets is E = 1.25 ms, and for ACK packets is E = 0.65 ms.

These values do not depend on the packet length. This is not exactly true, because the DTC DCP copies data to and from the serial interface card. However, since packets are assumed to be shorter than 80 bytes most of the time, this copy takes less than about 16% of the packet processing time and is not accounted for in this analysis.

Maximum Throughput

The maximum throughput is 1/NE, where N is the number of protocol activations for reception or transmission of one packet. NE is in fact the total processing time of a packet. N = 5 for receipt or transmission of user data, and N = 3 for ACKs. The mean rate of ACKs per user data packet is given above as a function of the window size. Using this rate as a weighting factor on the processing times, the table below gives the mean value of NE. T is the data throughput. From the mean value of NE, the maximum total throughput (user packets + ACKs) and the maximum data throughput (data packets only) are deduced.

Window	1	2	3	4	5
ACKs per data packet	0.96	0.52	0.37	0.3	0.25
Mean NE (ms)	4.1	4.7	5.1	5.3	5.4
Maximum total throughput (L)	243	212	196	190	185
Maximum data throughput (T)	124	140	143	146	148

Throughputs are expressed in packets per second, so if the window is 3, the maximum total throughput is 196 packets per second, and the maximum data throughput is 143 packets per second.

Among 48 terminals, approximately 24 will be active (1 I/O per hour), and among these 24 terminals only 12 are doing real work (8 transactions per minute). Therefore the mean data throughput (or nominal traffic) needed is about 19.5 packets/s and the mean total throughput is 26.7 packets/s. Larger flows may occur for printers. Packet concatenation decreases the number of packets per second exchanged, but increases the length of each packet, so that the processing time of one packet may become dependent on the length of this packet.

An estimation of the peak can be obtained by assuming the following conditions: each of 36 terminals does 15 transactions per minute (think time plus response time = 4 seconds) and 15 terminal I/Os are done per transaction. The data traffic is then 135 packets/s. In other words, the DTC supports up to 6 times the expected nominal traffic. The maximum throughput of a serial interface card is about 6×17.2 kilobaud outbound and 8×10 kilobaud inbound.

Latency of the DTC

The latency of the DTC is the sum of the latencies of the serial interface card and the CPU card. The latency of the serial interface card is about 1 ms. The latency D of the CPU card is the time for processing P protocols. As explained earlier, the time to perform P tasks is D = PE/(1 − NLE). We have seen above that U = L/T depends on the window. The latency of the DTC is:

$$\text{Latency} = D + 1\,\text{ms} = \frac{PE}{1 - NEUT} + 1\,\text{ms}.$$

The theoretical curve in Fig. 11 shows the latency when the window is 3, E = 1.25 ms, NE = 5.1 ms, P = 4, U = 1.37, and T varies from 0 to 1/NEU. If T is equal to the mean throughput expected (20 packets/s), the latency is about 8 ms.

There is a saturation effect. The latency cannot become infinite. When the memory pools are empty the incoming flows stop until resources are released. In such a situation, each packet takes about four seconds to process. This can be considered the limit, and it implies a DTC contribution of about 40 seconds in each transaction.

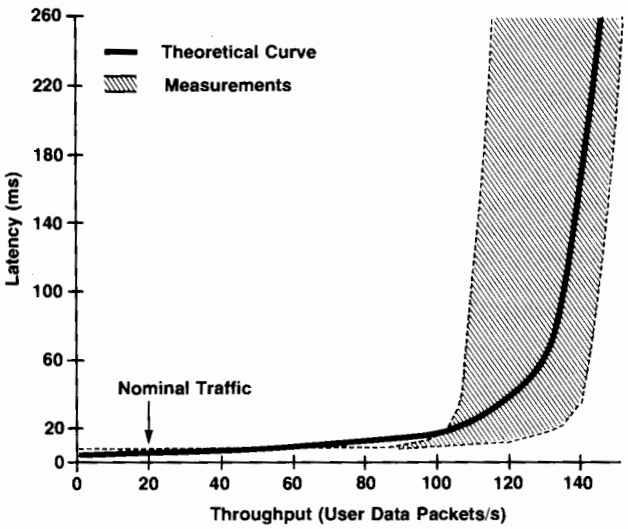


Fig. 11. *DTC latency as a function of throughput.*

Other Results

This model has been used to calculate other parameters of the DTC, including the minimum window that does not decrease the maximum throughput, the influence of overruns on the traffic (creating NAKs and duplicated packets), and the size of the memory pools needed to keep a low level of overruns for received packets without decreasing the maximum throughput for data transmission and ACK transmission. Actual measurements with special equipment have shown a latency of 9 ms for low use, and a maximum throughput of 120 to 150 packets/s (see Fig. 11).

Acknowledgments

The authors would like to acknowledge the following individuals for their contributions. The design teams included Emmanuel Gayet, Christian Gresset, Robert Masson, Jean-Jacques Ozil, Serge Sasyan, and Pierre de la Salle for the software design, Jean-Claude Brun, Stefan Gornisecz, Yves Karcher, Patrice Lovato, Remy Poulailleau, and Pierre-Yves Thoulon for the hardware design, and Francis Allirot, Michel Cauzid, Gilbert Dupont, and Patrick Francois for the mechanical design. Christian Sinet was the hardware quality engineer and Philippe Gascon was the software quality engineer. Gerard Mennetrier and Regis de Poortere assisted in defining all the self-tests and diagnostic procedures. Bernt Kristiansen and Lars Wernberg-Moller served as product managers. Dominique Maurenas did a very efficient job as project coordinator. Daniel Delgado provided support for manufacturing a large number of prototypes. We would also like to thank Jean-Louis Chapuis for his help and dedication to the project, and the many people in the production area, the materials area, and other divisions for their endless hours in making the DTC a product. This list would not be complete without our counterparts in HP's Information Networks Division: Tom Carney, Tom Engleman, Hamid Vazire, and Catherine Smith.

References

1. K.J. Faulkner, et al, "Network Services and Transport for the HP 3000 Computer," *Hewlett-Packard Journal*, Vol. 37, no. 10, October 1986.
2. L. Kleinrock, *Queuing Systems*, John Wiley and Sons, 1976.

Hewlett-Packard Precision Architecture Compiler Performance

Using a combination of simple instructions, optimized in-line code, and highly specialized Millicode routines, HP Precision Architecture machines perform many complex operations faster than CISC machines.

by Karl W. Pettis and William B. Buzbee

THE NEW HEWLETT-PACKARD Precision Architecture[1] is designed to provide a low-level interface to hardware through a small, fast instruction set. With any new architecture, compilers must be developed to provide high-level language interfaces to the machine. This importance of compilers to the new architecture was recognized from the project's inception, and software engineers, including compiler experts, were heavily involved in specifying the architecture.

This paper describes the influence performance criteria had on the implementation of the new compilers and how various problems were overcome. First, the influence that the high-level languages had on the design of the instruction set is described. Specific examples of instructions are given that enable the compilers to implement some high-level constructs efficiently, and to avoid problems that some see as inevitable with a reduced instruction set computer (RISC). Next, the problem of doing truly complex operations is described. These operations are sometimes implemented as instructions on traditional machines. Instead, it was decided to implement a streamlined procedure calling convention and a group of routines known as *Millicode* to solve such problems. Finally, the results for specific examples are presented.

High-Level Language Influence

As described in a previous paper,[2] a team of engineers specializing in various areas of computers designed the new architecture. The design was based on studies that showed what instructions computers actually spend time executing. These studies showed that even computers with very large, complex instruction sets typically spent an overwhelming portion of their time executing very simple instructions, such as memory loads and stores, branching, address calculation, and addition. So the initial design of the instruction set provided for the efficient execution of such frequently used instructions.

A key factor in the design was to make all instructions except branches and loads from memory execute in a single machine cycle. In addition, the cycle time itself was to be very short. Proposed instructions that could not meet these stringent criteria without adding considerably to the complexity of the processor were either eliminated or modified. Nevertheless, the software engineers on the design team made sure that the resulting instruction set was sufficiently rich to allow efficient implementation of most high-level language constructs. Sometimes this was done by adding helper instructions that are simple enough to be executed in a single cycle to make implementation of more complex operations easy. An example is the DIVIDE STEP instruction, which, when combined with an ADDC (add with carry) instruction, computes a single bit of the quotient of a division.

Very early in the project, a prototype C compiler and a processor simulator were developed that enabled the engineers to make sure that high-level constructs could be executed easily using the new instruction set. As the instruction set was implemented and evaluated, changes were suggested and accepted for the definitions of test conditions for various instructions, the way that nullification works for conditional branches, and other aspects of the instruction set.

It is worthwhile to look at some of the instructions that help the compilers produce efficient code.

SHIFT AND ADD. The SHnADD family of instructions includes SH1ADD, SH2ADD, and SH3ADD (along with counterparts that do not set the carry/borrow bits and others that trap on overflow). These instructions shift the contents of the first register argument left by n bits and then add the contents of the second register argument, putting the result into the third register argument. The result is that multiplications by small constants can be performed in a few instructions.

Instruction	Result
SH1ADD r1,r2,r3	r3 ← 2*r1 + r2
SH2ADD r1,r2,r3	r3 ← 4*r1 + r2
SH3ADD r1,r2,r3	r3 ← 8*r1 + r2

For example, to multiply a value in register 8 by the constant 41 with the result being put into register 9, the following code would be produced:

SH2ADD 8,8,9	r9 ← 5*r8
SH3ADD 9,8,9	r9 ← 8*(5*r8) + r8 = 41*r8

These instructions ensure that multiplication by a small constant, which is done fairly often, is very inexpensive. These instructions are also used in the nonconstant case. Even when both operands of a multiplication are not constants, often one of the operands is fairly small. When this

is true it usually takes about 20 cycles to multiply two variable quantities. This fast variable multiply is performed by extracting four-bit "digits" from the smaller operand and using them to select the appropriate shift and add sequence to perform the multiplication.

EXTRACT AND DEPOSIT. A family of instructions is provided to access fields within records. These instructions either extract the value from a field or deposit a value into a field and are used by the compilers when accessing packed or nonaligned data. In some other computers, one has to shift and mask to retrieve values in packed records. As an example, here is a sample of some code from a C program along with the instructions generated. The C code is:

```
struct descriptor {
    unsigned int available     : 1;
    unsigned int locked        : 1;
    unsigned int last_access   :10;
    unsigned int desc_type     : 5;
    unsigned int length        :15;
    };
copy_type(p,q) struct descriptor *p,  *q;
    {
    p → desc_type = q → desc_type;
    }
```

The HP Precision Architecture instructions to do the copy from one field to another are:

```
                    ; The word containing q→desc_type
                    ; is in register 29. The word contain-
                    ; ing p→desc_type is in register 25.
EXTRU 29,16,5,31    ; Extract the five bits of register 29
                    ; corresponding to q→desc_type.
                    ; This is an unsigned field, so use
                    ; EXTRU.
DEP 31,16,5,25      ; Put those five bits into the word
                    ; containing p→desc_type.
```

Being able to use just two instructions to do a transfer of packed data makes it easier for the compilers to generate efficient code when programmers use packed structures. However, these instructions are sufficiently generalized that the compilers can use them to perform left and right shifts (logical and arithmetic), and they can be used in contexts that have nothing to do with fields within a record.

Conditional Branches. On many computers conditional branches are implemented by performing some computation (typically a subtraction) that has the side effect of setting some hardware condition flags. The next instruction conditionally branches based on the settings of these flags. The studies done by the HP Precision Architecture engineers indicated that conditional branches are done very frequently by programs. The decision was therefore made to combine testing of various conditions with branching so that a conditional branch only requires one instruction.

Many of the conditional branches can also be used to modify register contents. Thus, the ADDIBT and ADDIBF instructions (add immediate and branch if true or false) can be used by the compilers to update a loop counter while simultaneously testing to see if the loop is completed. Alternatively, by using the TR condition (TRue = always branch), the compilers can sometimes merge an unconditional branch with a nonbranch instruction to make programs shorter and faster. Branches are implemented in this architecture so that the target of the branch is not executed until two cycles later. On conventional systems, the cycle following the branch is wasted. With HP Precision Architecture, the instruction immediately following the branch instruction is executed in the cycle before the branch takes effect. This enables the compilers to use this instruction slot for useful work. The branch instruction can optionally nullify this following instruction so that it will have no effect. If a compiler cannot always use the cycle after a branch, it will turn on nullification. If nullification is specified for an unconditional branch, the following instruction will always be nullified after the branch instruction. If nullification is specified for a conditional branch, then the following instruction will be nullified if the branch is taken or if the branch is backward, but not both. This nullification scheme is designed so that if a conditional branch is being used in controlling a loop, an instruction from the body of the loop can always be executed in the otherwise wasted cycle after the conditional branch. The effect is that almost all loops can be made one instruction shorter and faster than with a more conventional machine. Since most computer programs spend most of their time in very small loops, even one instruction saved per loop can be significant.

Decimal Correction. There are two instructions, DCOR (decimal correct) and IDCOR (intermediate decimal correct), that provide invaluable assistance for decimal arithmetic. One of the criticisms often leveled at typical RISC architectures is that they provide insufficient support for decimal operations, which are crucial to commercial languages such as COBOL and RPG. Using a biasing scheme and the DCOR and IDCOR instructions, compilers for HP Precision Architecture systems can implement decimal additions and subtractions with the ordinary binary ADD and SUB instructions and minimal extra instructions. Other decimal operations are more complex and are described later in this paper.

STORE BYTES. The STBYS instruction stores zero to four bytes to memory depending on the offset specified in the instruction and the alignment of the base address. It is a great aid in moving data from one location to another, particularly when a filling operation is being performed. This often happens when using strings in Pascal and Fortran and with the MOVE directive in COBOL. This instruction also lets the compiler writer generate simpler code to store data that is inconveniently aligned with respect to word boundaries.

Nullification. Most arithmetic and logical instructions allow conditional nullification of the following instruction. In effect, this is a way for the compilers to generate short if-then sequences in-line without incurring the overhead of actually doing a test and branching. The compiler most often uses this feature in short "canned" code sequences to compute a result. For example, suppose a Boolean flag in Pascal is to contain a value indicating whether a is less than b or not. The Pascal code is:

```
flag := a < b;
```

The HP Precision instructions to perform this statement are:

```
COMCLR,> =      r1,r2,r3    ;  a = r1, b = r2, flag = r3
LDI             1,r3
```

The COMCLR (COMPARE AND CLEAR) instruction sets the value of r3 (flag) to 0. If the value of r1 (a) is greater than or equal to the value of r2 (b), then the following instruction is nullified, and the value of flag remains 0 (false). Otherwise, the LDI (LOAD IMMEDIATE) instruction sets the value of flag to 1 (true).

The ability to nullify the following instruction conditionally is very powerful and can be used in many contexts. For example, to perform a signed integer division by a power of two (such as 8) with truncation towards 0, only three instructions are needed:

```
OR,> =      r1,0,r2     ;  Put a copy of the source
                        ;  (r1) into the destination
                        ;  (r2). Also test to see if the
                        ;  source is positive or 0. If
                        ;  so, then the next instruc-
                        ;  tion is nullified.
ADDI        7,r2,r2     ;  Executed only if the source
                        ;  was negative. If it was,
                        ;  this ensures truncation
                        ;  towards 0.
EXTRS       r2,28,29,r2 ;  Arithmetic right shift the
                        ;  (possibly modified)
                        ;  destination by three bits
                        ;  yielding the final result.
```

An arithmetic right shift by three bits (implemented by the EXTRS instruction) divides a quantity by eight, with truncation towards negative infinity. For positive numbers, this is the same as truncation towards 0. For negative numbers, an adjustment needs to be made. Here we have used nullification so that the adjustment is done only if the dividend is negative. If nullification were not available, we would have had to generate code using branches.

In short, the instruction set provided by the new HP Precision Architecture is very powerful, allowing short instruction sequences to be generated for many high-level operations. Having a simplified instruction set where every instruction executes in a single cycle frees the compilers from having to make complicated analyses of varying instruction sequences to see which is better. With HP Precision Architecture, shorter is better.

Procedure Calling Convention

When the first compiling systems were initially brought up on the new architecture, a traditional procedure calling convention was investigated. This had a frame pointer and a top-of-stack pointer, and parameters were passed to routines by pushing them onto the stack. Upon return, the parameters were popped. However, it was discovered that procedure calls could be performed even more efficiently with a new procedure calling convention.

Under the new convention, the registers are now divided into three classes: the *caller-saves*, the *callee-saves*, and the *linkage* registers. The called routine is free to modify the caller-save registers with no overhead, but must save and restore any registers it uses in the callee-save set. If a routine is simple and does not use many registers, the overhead in making a procedure call is often only a few instructions. There is no pointer to the previous frame, nor are parameters pushed and popped as on conventional machines. The first four parameters are passed in registers. If there are more parameters, these are placed into an area on the stack allocated once at the beginning of the calling procedure.

In addition, a special class of routines has been developed with an even more streamlined calling convention. These routines are called *Millicode* and they are designed to implement more complex operations that are done frequently by programs. Some of these routines, like the multiplication routine, correspond to machine instructions implemented by microcode on other machines. In general, these routines are only allowed to modify a very small number of registers (typically 4 to 6). The compilers know which few registers can be modified by each Millicode call and can arrange to have those registers free, while using other registers (including some caller-save registers) to store temporary intermediate results. Since these routines use so few registers, the compilers can almost always generate code that calls them without having to do any extra saving and restoring of registers. Also, the linkage registers required to call and return from Millicode routines are different than for normal routines. This enables routines that only call Millicode and not other normal routines to have even less overhead, since they are not required to store their linkage information.

Complex Operations

One of the most enduring criticisms of RISC architectures is that they are unable to handle complex operations efficiently. Critics often suggest that applications that rely upon a high percentage of complex operations will execute slowly, suffer from excessive code size, or both. In part because of the RISC extensions provided with HP Precision Architecture, we have experienced the opposite. HP Precision Architecture machines running the most popular COBOL processor benchmarks outperform their CISC counterparts by a factor of 1.3 to 4.2, above and beyond differences in the machines' respective MIPS rates (see page 35). Furthermore, this performance is achieved using 15% to 30% fewer in-line instructions.

The reason for this success lies in the elegant partnership of custom in-line code sequences and Millicode. Together they form a solid foundation for the somewhat paradoxical assertion that complex operations can benefit more from architectures that concentrate on fast simple operations than from those that concentrate on fast complex operations.

To understand the HP Precision Architecture complex operation solution, we must first understand the problems of generating code to perform a complex operation. For the purpose of this discussion, a complex operation is a task that requires three or more of a machine's most basic instructions to complete. Some obvious and very important examples of complex operations are byte moves, string

comparisons, and decimal arithmetic.

In traditional complex instruction set computers (CISC), every machine instruction is performed by the execution of a corresponding microcode program. The microcode programs are made up of microinstructions—the instructions that the actual hardware executes. In a sense, CISC systems are really computers inside of computers. HP Precision Architecture has eliminated the outer simulated computer, providing for the direct execution of its machine instructions. From this point of view, HP Precision Architecture machine instructions can be thought of as roughly equivalent to CISC microinstructions. The reason for elimination of the outer computer is simple. There is a fixed overhead associated with the simulation of the CISC outer computer on the CISC inner computer. For simple instructions this interpretive overhead can account for more than the amount of time actually spent performing the operation. HP Precision Architecture has streamlined simple operations by eliminating this overhead.

For complex operations, however, the story changes. The CISC overhead is relatively fixed. Complex operations often require the execution of dozens or hundreds of microinstructions. Thus, the overhead becomes insignificant. Similarly, the advance HP Precision Architecture gains by eliminating overhead also becomes insignificant for complex operations. Given only this information, it would seem that the critics are correct, and that RISC-like architectures face a serious complex operation performance problem when competing head-to-head with CISC systems.

A Closer Look

The conclusion that RISC-like systems are inherently slower than CISC systems for complex operations rests on the assumption that RISC-to-CISC MIPS ratios are invalid. As the argument goes, it is assumed that more RISC instructions are required to perform a complex operation than CISC instructions, and therefore, a RISC MIPS is worth less than a CISC MIPS. This assumption, however, is not necessarily correct.

Early in the development of HP Precision Architecture complex operation code generation, selected CISC microcode programs were carefully examined. One important characteristic quickly stood out: microcode programs for many types of complex operations spend a large amount of time calculating and interpreting information at run time that compilers knew at compile time. Under a CISC system, this information is lost because a compiler cannot transmit it through the instruction set to microcode. Because microcode program space is scarce and expensive, there is typically only a single CISC instruction per class of complex operations. For example, a CISC system may have one string comparison instruction, one decimal addition instruction, etc. The microcode programs associated with these instructions must therefore be capable of handling all situations and must typically make worst-case assumptions. A compiler may know, for example, that the source and target of a byte move do not overlap, but in a CISC system the microcode would have to determine this at run time.

The obvious solution for the speed problem is to exploit all available compile-time information to eliminate the interpretive overhead suffered by CISC microprograms. The compilers, in effect, become custom microcode program generators. Code generation for simple operations takes place as usual, but when a complex operation is called for, the compiler analyzes all available compile-time information and generates a custom, highly specific "microcode" program that it drops in-line. Using this scheme, an HP Precision Architecture system could perform the operation in fewer cycles. There is, however, a serious problem with this approach—code expansion. Genuinely optimal code sequences for complex operations could require dozens or hundreds of HP Precision Architecture instructions. In some situations, code size could become enormous, perhaps to the point where any speed improvements would be lost in increased cache and page traffic.

If code size were the only concern, then the opposite approach is to pretend that code is being generated for a CISC machine. In other words, develop a run-time library or, ultimately, an interpreter for a compact pseudocode, which contains one procedure for every CISC complex instruction microprogram. Whenever a complex operation needs to be performed, the compiler generates a call to the appropriate routine within the library. This scheme solves the code expansion problem by reducing the in-line cost of a complex operation to that of a procedure call, a cost roughly comparable to the fetching and decoding cost of a CISC instruction. Furthermore, assuming that the run-time library is shared among all processes on a system, this scheme eliminates stress on the memory hierarchy. This code size solution, however, sends us back to square one. Such a run-time library suffers the same interpretive overhead as CISC microcode programs.

HP Precision Architecture Solution

The ideal solution would be to combine the advantages of the two approaches without incurring their disadvantages. The HP Precision Architecture solution comes very close to this ideal. In brief, the HP Precision Architecture compilers examine every complex operation and break it down into steps. The steps in which compile-time information is either unavailable or useless are performed by a call to a routine within a special shared library, keeping the code size compact. For the remaining steps the compilers determine whether the step is interpretive or repetitive. A repetitive step is one that can be performed in a library routine with little or no interpretive overhead, while an interpretive step is one that would suffer from interpretive overhead if performed in a library routine. Interpretive steps are performed using custom in-line code sequences. Repetitive steps are performed by calls to highly specific library routines. A more precise description of the HP Precision Architecture complex code generation solution is given in the pseudocode algorithm of Fig. 1.

The HP Precision Architecture solution is further enhanced by the nature of its special shared library routines, which are known as Millicode. In its simplest form, Millicode is nothing more than a series of instructions packaged to look somewhat like a procedure. A Millicode routine is invoked by a simplified calling mechanism in which almost no state saving is required. Millicode is designed so that a single copy can exist on a system, allowing all processes to share it.

```
IF  [ no useful compile-time information about the operation is available ]
    THEN
    IF  [ the number of instructions required to perform the operation ]
        LESSTHAN [ speed/space factor ] THEN [ Perform the operation
        entirely in-line ]
    ELSE
        [ Generate call to general-purpose Millicode routine ]
ELSE
    IF  [ the operation is fully described by compile-time information ]
        THEN
        IF  [ the number of instructions required to perform the opera-
            tion ] LESSTHAN [ speed/space factor ] THEN [ Perform
            the operation in-line using optimum custom code
            sequence ]
        ELSE
            BEGIN
            [ Separate operation into repetitive and interpretive steps ]
            FOR [ each step ] DO
                IF  [ step is interpretive ] THEN
                    [ Perform the step in-line using optimum custom
                      code squence ]
                ELSE
                    [ Generate call to one of a set of highly specific repeti-
                      tive Millicode routines ]
            END
    ELSE
        BEGIN
        [ Separate operation into steps in which useful compile-
          time information is available and steps in which useful compile-
          time information is not available ]
        FOR [ each step ] DO
            [ Consider the step a separate complex operation and
              reapply this algorithm ]
        END
```

Fig. 1. *HP Precision Architecture algorithm for complex operation code generation.*

Although Millicode serves many of the functions of a CISC machine's microcode, there is an important difference. Millicode doesn't suffer from the severe space restrictions placed upon microcode. This is particularly important for the class of complex operations in which useful information is available at compile time, but whose custom in-line code sequences would be too lengthy. Instead of wasting the compile-time information, a series of Millicode routines can be created, each tailored to a particular combination of compile-time information. Unlike a CISC system's microcode program, these special-purpose Millicode routines can make best-case assumptions. The compiler determines the best one to call based on the available information.

A good example of the HP Precision Architecture complex operation code generation process is an eight-byte move followed by a 22-byte blank fill in which all alignment and length information is known at compile time. This is a reasonably common type of operation when dealing with character strings. Following the algorithm of Fig. 1, the compiler first examines the move operation and determines that it is fully described by compile-time information. Then it finds that the code size to perform the move is small enough, and performs the move entirely in-line with no interpretive overhead (see Fig. 2). Next, the fill portion is examined. Although it also is fully described by

```
soff  = Constant offset to start of source string
toff  = Constant offset to start of target
dp    = Register containing base pointer

LDW   soff+0(dp),reg1          ; Pick up first four bytes
LDW   soff+4(dp),reg2          ; Pick up next four bytes
STW   reg1,toff+0(dp)          ; Store first four bytes
STW   reg2,toff+4(dp)          ; Store next four bytes
```

Fig. 2. *Eight-byte move portion of a move/fill operation.*

compile-time information, the compiler decides that it would require too many in-line instructions. Therefore, the fill is broken down into two steps: full-word fills and partial-word fills. A 22-byte fill is really a five-word fill followed by a two-byte fill. The five-word fill step is a repetitive step, and is performed in Millicode. The two-byte step would require interpretation at run-time if performed in Millicode, so it is performed in-line. The generated code sequence is shown in Fig. 3.

In the above example, only four instructions were devoted to overhead: loading the address of the beginning of the fill, loading the fill character, branching to Millicode, and returning from Millicode.

One extremely important side benefit of the HP Precision Architecture solution is that as more compile-time information is retained and processed, it becomes easier to recognize special cases and take advantage of them. For example, the incrementing of the COBOL unpacked decimal (ASCII display) data type is typically performed in CISC systems

```
soff   = Constant offset to start of source string
toff   = Constant offset to start of target
dp     = Register containing base pointer
blanks = Register containing 0x20202020 (all blanks)
MRP    = Millicode return pointer
sr     = Millicode space register
```

```
{ In-Line Code for Move }

LDW   soff+0(dp),reg1
LDW   soff+4(dp),reg2
STW   reg1,toff+0(dp)
STW   reg2,toff+4(dp)

{ In-Line Code for Fill }

LDO   toff+8(dp),ARG1
BLE   $$fill_5(sr,0)
COPY  blanks,ARG0
STH   ARG0,0(ARG1)
```

```
{ Fill Millicode }

• { entry points }
• { continue to }
• { $$fill_31 }
$$fill_7
   STBYS,b,m    ARG0,4(ARG1)
$$fill_6
   STBYS,b,m    ARG0,4(ARG1)
$$fill_5
   STBYS,b,m    ARG0,4(ARG1)
$$fill_4
   STBYS,b,m    ARG0,4(ARG1)
$$fill_3
   STBYS,b,m    ARG0,4(ARG1)
$$fill_2
   STBYS,b,m    ARG0,4(ARG1)
$$fill_1
   BE           0(0,MRP)
   STBYS,b,m    ARG0,4(ARG1)
```

Fig. 3. *Complete move/fill code sequence.*

by converting to packed decimal, doing the normal decimal addition, and then converting back to unpacked decimal. On HP Precision systems, the compilers recognize that in the typical case no carry will be generated and only the least-significant digit of the number will have to be altered. It then becomes possible to generate extremely fast code sequences to perform the operation (see Fig. 4). In one COBOL processor benchmark, an HP Precision Architecture machine outperformed its CISC counterpart at unpacked decimal incrementing by a factor of 7, above and beyond MIPS rate differences.

Among the special cases recognized by the first generation of HP Precision Architecture optimizing compilers are:

- Unpacked decimal equality comparison. Compare the least-significant digits in-line. If they are not equal, the operation is complete. Otherwise, complete the comparison in a highly specific Millicode routine.
- Unpacked decimal rounding. Compare the digit to be rounded with the constant 5 in-line. If it is less than 5, simply truncate. Otherwise, perform the special display increment on the following digit and then truncate.
- Unpacked decimal multiplication by power of 10. Simply shift the number.
- Unpacked decimal division by power of 10. Simply shift the number.
- Unpacked decimal remainder by power of 10. Truncate or shift the number.
- Unpacked decimal multiplication by a small constant. Turn into a series of possibly scaled additions.
- Unpacked decimal decrement. Same as unpacked decimal increment.
- Unpacked decimal addition of constant. Prebias the constant at compile time and use a special-purpose display addition Millicode routine.
- String comparison. Compare one to four of the leading bytes of the two strings in-line. If they are not equal, the operation is complete. Otherwise, finish the operation using a special-purpose Millicode routine.
- Byte move in which target address = source address + 1. Turn operation into fill.

The exploitation of special cases is really just an extension of the overall HP Precision Architecture complex operation solution. The code sequences for these special cases combine the quest for performing no unnecessary work with the RISC philosophy of tuning for the most common cases.

A legitimate question to ask, however, is "Why couldn't this same strategy be used by CISC systems?" The answer takes us back to the beginning—fast simple operations. The HP Precision Architecture complex operation solution picks up its performance improvements by using compile-time information to generate custom in-line code sequences and to select highly specific Millicode routines. Both the in-line sequences and the Millicode are composed of the fast simple operation building blocks provided by HP Precision Architecture. If a CISC were to attempt to build the same mechanism out of its regular instruction building blocks, the combined overhead incurred would quickly outweigh the advantages of using compile-time information.

```
sboff   =  Constant offset to sign digit of 4-digit unpacked decimal item
table1  =  Register containing pointer to base of set of translation tables
table2  =  Register to hold pointer to increment table
sign    =  Register to hold original sign digit
xsign   =  Register to hold incremented sign digit
sr      =  Millicode space register

.
.
.
LDB         sboff(dp),sign              ;  Load sign digit
LDO         inc_offset(table1),table2   ;  Get pointer to increment table
LDBX        sign(table2),xsign          ;  Translate/increment sign digit
COMIB,<>    0,xsign,all_done            ;  Branch over millicall if no carry
STB         xsign,sboff(dp)             ;  Store incremented sign digit
LDO         sboff(dp),ARG0              ;  Get pointer to sign digit
BLE         $$ginc_cont(sr,0)           ;  Continue increment in Millicode
LDI         4,ARG1                      ;  Pass Millicode length of 4
all_done
.
.
.
```

Fig. 4. *Unpacked decimal increment code sequence.*

Compiler Performance Measurements

To compare the relationship of compilers and computer architectures of different machines accurately, it is first necessary to factor out differences in the machines' raw speed. To compute the following performance ratios, the elapsed time required to perform each benchmark on a particular machine was multiplied by that machine's MIPS rate. For example, if machine A is rated at 2 MIPS and executes a benchmark in 20 seconds, its adjusted time would be 40 seconds. If machine B, rated at 3 MIPS, performs the same benchmark in 10 seconds, its adjusted time would be 30 seconds. We could then say that the compiler/architecture combination of machine B outperforms that of machine A by a factor of 1.33 (40/30), above and beyond differences in the machines' MIPS ratios.

In the following table, performance ratios greater than 1 suggest that the HP Precision Architecture/compiler relationship is more efficient than its CISC counterpart.

Because the HP Precision Architecture code generation strategy calls for custom code sequences which depend on many factors, the performance ratios for a single type of complex operation can vary widely from one specific example to another. Each item in this table represents a class of operations. The compiler performance ratios were generated by comparing an HP Precision Architecture machine with representative CISC machines from HP and other manufacturers.

Moves/Fills	Compiler Performance Ratio
Short move (~10 bytes)	2.7
Long move (~80 bytes)	0.6
Short fill (~10 bytes)	1.2
Long fill (~80 bytes)	0.5
Average move/fill	1.4
Comparisons	
Byte string comparisons	1.1
Unpacked decimal comparisons	2.0
Unpacked decimal comparisons with digit validation	1.0
Packed decimal comparisons	1.1
Packed decimal comparisons with digit validation	0.7
Decimal Addition	
Unpacked decimal addition	3.2
Unpacked decimal addition with digit validation	1.6
Packed decimal addition	2.1
Packed decimal addition with digit validation	1.3
Multiplication	
64-bit integer multiplication	0.7
Packed decimal multiplication	1.5
Packed decimal multiplication with digit validation	1.3
Subscripting	
Small subscripted move; integer subscript	2.9
Small subscripted move; unpacked decimal subscript	3.5
Small subscripted move; packed decimal subscript	1.0

Acknowledgments

Although many HP engineers made significant contributions to the topics covered here, we would like to give special recognition to Richard Campbell, Debbie Coutant, Dennis Handly, Daryl Odnert, and the Computer Language Lab's code quality team.

References

1. M.J. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pp. 4-21.
2. D.S. Coutant, et al, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, Vol. 37, no. 1, January 1986, pp. 4-18.

Authors

March 1987

Edward M. Jacobs

Ed Jacobs is a 1984 graduate of Stanford University (BSEE) and has been with HP since 1985. He worked on the floating-point coprocessor for the HP 9000 Model 840 Computer and is now doing feasibility studies for a future product. He is also working part time on an MSEE degree from Stanford. A native of St. Louis, Missouri, Ed lives in Sunnyvale, California. He's active in HP golf and softball recreational leagues and enjoys horse racing.

Ross V. La Fetra

A California native, Ross La Fetra was born in Los Angeles and attended Harvey Mudd College. He received his BS and MS degrees, both in general engineering, in 1984 and 1985. After joining HP in 1985 he worked on the development of the HP 9000 Model 840 Computer. His contributions include analyzer card software, parts of the cache, and measurement tools. For recreation, Ross likes silkscreening, bicycling, photography, hiking, and camping. He lives in Cupertino.

Allan S. Yeh

Allan Yeh joined HP Laboratories in 1980 and worked on the automation of IC process equipment before moving to a new position that involved designing fiber optics data communication hardware. He designed the system monitor for the HP 9000 Model 840 and HP 3000 Series 930 Computers and did the I/O expansion module for the Series 930. His professional interests focus on signal processing and communication systems. Allan was born in Taichung, Taiwan and attended the National Taiwan University (BSEE 1976) and the University of California at Berkeley (MSEE 1980). A resident of Palo Alto, he likes basketball, jogging, swimming, and listening to classical music.

Simin I. Boschma

Born in Tehran, Iran, Simin Boschma studied electrical engineering at Utah State University (BSEE 1977) and came to HP in 1978. She's a memory design specialist and has designed I/O boards for several series of the HP 1000 Computer and memory boards for the HP 1000 A900 Computer. She also worked on the memory system for the HP 9000 Model 840 Computer. A resident of San Jose, California, Simin is working toward her MSEE degree from Santa Clara University. She's married and enjoys bicycling, softball, and sailing catamarans.

Randy J. Teegarden

Born in McMinnville, Oregon, Randy Teegarden is a graduate of the De Vry Institute of Technology of Phoenix (BSEET 1977). He designed digital signal processing hardware and software at GTE Government Systems before coming to HP in 1984. He was a manufacturing development engineer for the HP 9000 Model 840 and HP 3000 Series 930 Computers, designing the test system interface and GPIO adapter for the automated test system. He's now an R&D engineer and is designing a portion of the CPU board for a future product. Randy and his wife and daughter live in Sunnyvale, California. He's active in his church and enjoys volleyball, bicycling, and woodworking.

William R. Bryg

A graduate of Stanford University, Bill Bryg was awarded concurrent BSEE and MSEE degrees in 1979. He joined HP the same year, working first on processor design for the HP 3000 Series 64 Computer and later on HP Precision Architecture. He designed the TLB for the HP 9000 Model 840 and HP 3000 Series 930 Computers and worked on power-fail recovery for the HP-UX operating system. He's now designing I/O drivers for HP-UX. He's coauthor of a 1986 *HP Journal* article on the processor for HP Precision Architecture. Born in Chicago, Illinois, he now lives in Saratoga, California. He's married and likes games, dancing, skiing, gardening, and reading science fiction.

David A. Fotland

A project manager at the Information Technology Group, Dave Fotland has been with HP since 1979. He's responsible for hardware development for the HP 9000 Model 840 Computer. As an R&D design engineer, he worked on Model 840 hardware, including processor and I/O architecture and the register file board, backplane, and parallel I/O card. Earlier, he contributed to the development of the A700 processor and the RTE-A operating system for HP 1000 A-Series Computers. He's named an inventor on three patent applications related to processor architecture. Born in Cleveland, Ohio, Dave attended Case Western Reserve University, completing work in 1979 for both of his degrees (BSEE and MS computer engineering). He and his wife live in San Jose, California. For a change of pace from his job he plays volleyball, racquetball, and go and is a Dungeons and Dragons fan.

Thomas B. Wylegala

With HP since 1984, Tom Wylegala was responsible for manufacturing test strategy and CPU board repair for the HP 9000 Model 840 and HP 3000 Series 930 Computers. He's now working on I/O architecture in the Information Technology Group. Born in Buffalo, New York, he studied computer science at the Massachusetts Institute of Technology (BS 1977) and then served as a captain in the U.S. Army. He continued his studies at Purdue University, completing work for an MS degree in computer science in 1982 and for an MSEE degree in 1984. Tom and his wife now live in San Jose, California. He's a chess devotee and likes all kinds of racquet sports, especially tennis.

John F. Shelton

With HP since 1981, John Shelton wrote microcode for the HP 1000 A900 Computer and designed the I-unit for the HP 3000 Series 930 and HP 9000 Model 840 Computers. He's currently the project leader for a processor for a future product. He's named co-inventor on a patent related to a decoding technique used for the HP 1000 A900 and is coauthor of an *HP Journal* article on the A900. He has also written a conference paper on the Series 930 and Model 840. Born in Washington, D.C., John attended the Massachusetts Institute of Technology (BS literature 1976) and the University of California at Berkeley (MSEE 1981). He lives in Aptos, California, is married, and has two children.

18 Test System

Long C. Chow

With HP since 1984, Long Chow has been a production engineer for the Data Systems Division. He worked on test systems for HP 1000 A-Series Computers and for HP 9000 Model 840 and HP 3000 Series 930 Computers. He's currently working on an MSEE degree at Stanford University through the HP Resident Fellowship program. He was born in Bandar Seri Begawan, Brunei and completed work for his BSEE degree from Iowa State University in 1983. Now a resident of San Jose, California, Long likes sports, especially badminton, and reads Chinese novels for relaxation.

21 Terminal Controller

Eric Lecesne

Eric Lecesne joined HP in 1981 and is a project leader for X.25 software at the company's Information Networks Division. He was also a project leader for the HP 2345A Terminal Controller and contributed to the development of the HP 2334A Cluster Controller. Born in Cherbourg, France, he attended the École Supérieure d'Électricité et de Radioélectricité of Grenoble and earned a BS degree in nuclear physics in 1978 and an engineering diploma in 1981. Eric and his wife and two children are residents of Sunnyvale, California. He lists astronomy, tennis, and history as outside interests.

Heng V. Te

Born in Phnom Penh, Cambodia, Heng Te studied computer science at the doctoral level at the Université de Technologie de Compiègne and at the University of Pennsylvania. He worked on real-time computer systems and parallel processing before coming to HP in 1983. He was project manager for the multiplexer for the HP 2345A Terminal Controller and other products and is now the section manager for multiplexers for terminal I/O and ISDN. He has published two papers on computer networking topics and is named inventor on two patents related to pattern-matching and to an accelerator for signal processing. Heng lives in Meylan, France, is married, and has a daughter. His outside interests include skiing, windsurfing, reading, and photography.

Jean-Pierre Picq

Born in Paris, Jean-Pierre Picq earned his master's degree in electrical engineering from École Supérieure d'Électricité in 1980 and came to HP the same year. He was project manager for hardware on the HP 2345A Terminal Controller and earlier contributed to the development of the HP 2333A Multipoint Controller and to HP PMF/1000 software. Jean-Pierre, his wife, and daughter live in Grenoble. His leisure activities include skiing and hiking.

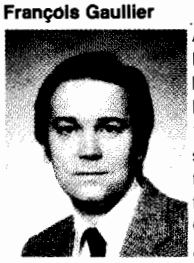
Gregory F. Buchanan

Born in Flint, Michigan, Greg Buchanan served in the U.S. Navy before completing work in 1977 for concurrent degrees from the University of Michigan (BSE computer engineering and BA physical sciences). He joined HP in 1982 and worked on local area networking software before moving to the Grenoble Networks Division, where he was responsible for network management and the boot for the HP 2345A Terminal Controller. He has recently returned to the Information Networks Division in California. His other professional experience was at the Harris Corporation, where he worked on communication protocols. Greg lives in Cupertino, is married, and has one child. He enjoys skiing and camping in his spare time.

Olivier Krumeich

With HP since 1982, Olivier Krumeich was a project manager for the HP 2345A Terminal Controller and has also contributed to the development of HP PCIF/1000 software. Born in Paris, he studied at the École des Mines and earned the equivalent of a PhD degree in 1980. He also served in the scientific branch of the French Air Force. He did research work in solid-state technology before coming to HP. Olivier is married and lives in Meylan. He enjoys skiing, mountain climbing, and tennis.

François Gaullier



An R&D project manager at HP's Grenoble Networks Division, François Gaullier has been with HP since 1971. He headed the design team that developed the software and firmware for the HP 2345A Terminal Controller. Before assuming his current position he worked as an HP systems engineer and as an R&D design engineer. His diploma, equivalent to a master's degree, was awarded by the École Spéciale de Mécanique et d'Électricité in 1971. He's a member of the European Computer Manufacturer's Association. Born in Paris, François is a resident of St. Egreve, is married, and has three children. When he's not working on the remodeling of his home, he enjoys hiking, skiing, and listening to classical or country music.

William B. Buzbee

With HP since 1984, Bill Buzbee has worked on code generation and optimization for HP Precision Architecture. Before joining HP he was a journalist and held positions ranging from sports writer to managing editor of a small daily newspaper. He is named inventor for a patent application related to a new method for generating code to perform complex operations. Bill was born in Chanute, Kansas and educated at the University of Kansas (BS journalism 1980 and MS computer science 1984). He's married and lives in Milpitas, California.

Karl W. Pettis

Born in Gainesville, Florida, Karl Pettis attended Michigan State University and completed work for a BS degree in mathematics in 1975 and an MS degree in computer science in 1977. He continued his studies at Yale University and was awarded another MS computer science degree in 1978. He also did PhD-level work at the University of Arizona before joining HP in 1981. In addition to his work on the optimizer for HP Precision Architecture, he has contributed to the development of HP Business BASIC and HP MemoMaker. He's coauthor of a technical paper on pattern recognition. A resident of San Jose, California, Karl likes games, comic books, and music by Gilbert and Sullivan.

Zvonko Fazarinc

Zvonko Fazarinc was born in Celje, Yugoslavia. His interest in electronics led him to Ljubljana University, where he received an electrical engineering degree in 1952, and to Stanford University, where he received his PhD in 1964. From 1951 to 1960 he worked at the Communications Institute in Ljubljana where he developed microwave radio link equipment. When he arrived in the United States in 1960 he found in the Stanford Radioastronomy Institute a happy match between the needs of the Institute, his professional interests, and his hobby, astronomy. His research on galactic radio sources was the subject of his PhD thesis. Zvonko has been with HP Laboratories since 1965, and has conducted research in selected areas of analog circuits and systems, solid-state devices, and their analysis on digital computers. He was involved in the initial stages of integrated circuit development at HP and made early contributions to the development of circuit design tools. As a laboratory director, he has managed departments in the areas of instrumentation, integrated circuit research, ultrahigh-speed digital/analog interfaces, and signal processing. He has also managed research on distributed systems and their communication needs as well as on software design tools and communication protocols. Many products found in the HP catalog have their origin in these activities. Zvonko is currently HP's Senior Scientific Advisor for Europe and is associated with Stanford University, where he holds a consulting professorship.