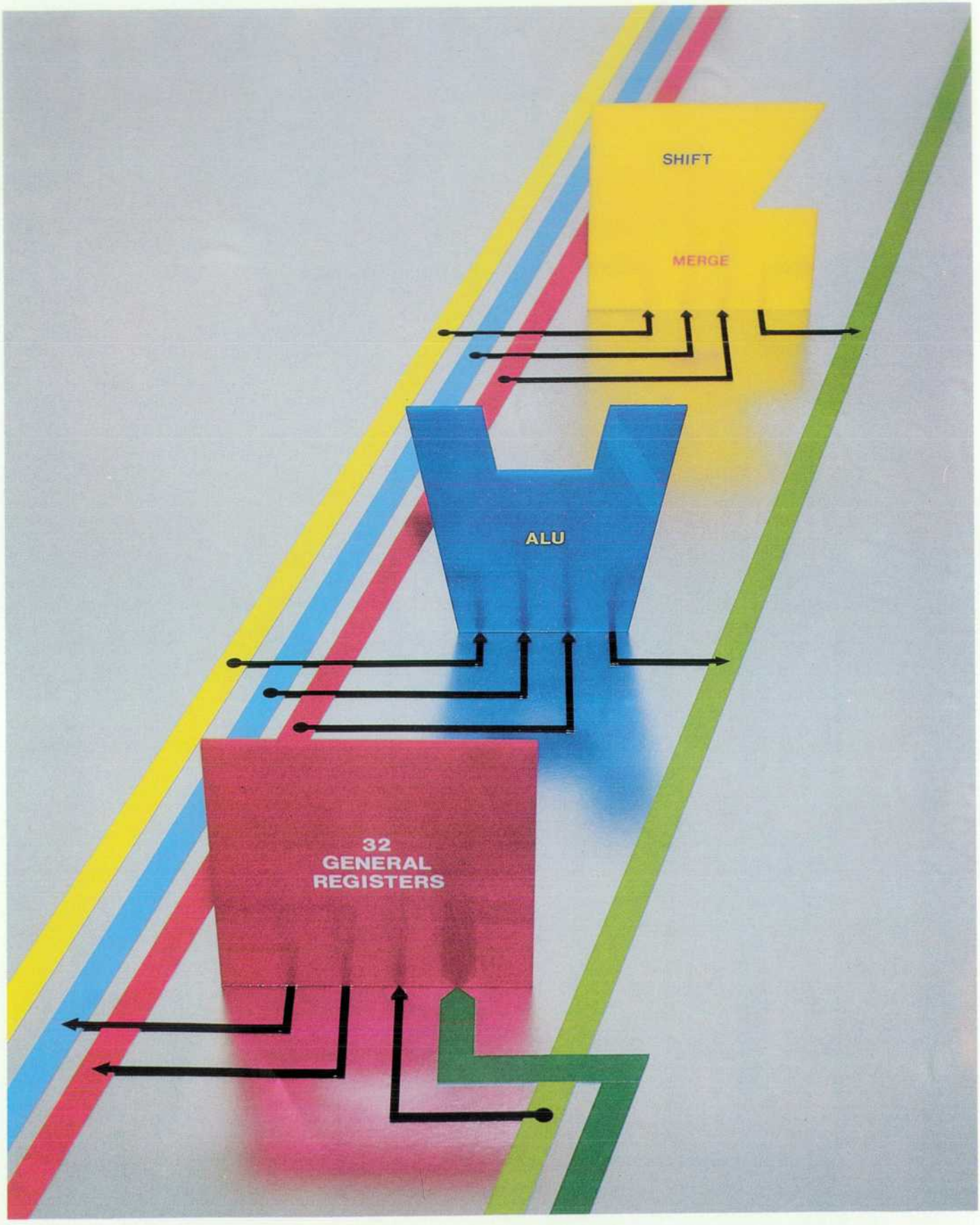
HEWLETT-PACKARD
JOURNAL

AUGUST 1986



HEWLETT-PACKARD JOURNAL

August 1986 Volume 37 • Number 8

Articles

Departments

In this Issue

The HP Precision Architecture development program, known within HP as the Spectrum program, is the largest system development program ever undertaken by the Hewlett-Packard Company. The program developed not only a new system architecture, but also all hardware and software components necessary to constitute an entirely new computer system family. It encompassed architecture, VLSI technology, the MPE XL commercial operating system, the HP-UX real-time standard UNIX operating system, a new family of optimizing compilers, a new data base facility, and integration with the HP AdvanceNet networking strategy.

The papers published in the August 1985 and January 1986 issues of the HP Journal outline the reasons for the development of HP Precision Architecture and describe the structure of the next generation compiler family. In this issue of the HP Journal, we are happy to be able to present the first of a planned set of papers that explain key program elements in greater levels of detail. We intend these papers to be tutorial in nature, describing and explaining program elements and presenting the basic research and measurement results that were achieved.

In this issue we begin with papers covering an overview of the processor architecture (page 4), a summary of the I/O architecture (page 23), a description of the performance analysis activities used throughout the program (page 30), and a description of the simulator tools that grew into our general software diagnostic tools (page 40). In subsequent issues, we plan to present papers describing hardware components, software system components, software engineering practices, and performance results. We expect that the collected set of papers will then constitute a good technical overview of the Spectrum program and the key research results that emerged from it.

-William S. Worley, Jr.
Guest Editor

Cover

The cover photograph shows a "block diagram" representing the HP Precision Architecture execution engine, which is shown more conventionally in Fig. 3 on page 7.

What's Ahead

Next month's issue will have a series of articles on the design of the HP 9000 Series 300 modular engineering workstations, and a part historical, part tutorial treatise on implementing a worldwide electronic mail system, based on HP's experience with its own HP DeskManager product.

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, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A.

Hewlett-Packard Precision Architecture: The Processor

This article describes the architecture's basic organization, execution model, control flow model, addressing and protection model, functional operations, and instruction formats and encoding.

by Michael J. Mahon, Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck, and William R. Bryg

"Everything should be made as simple as possible, but not simpler." Albert Einstein

THE HP PRECISION ARCHITECTURE development program had the objective of designing a computer architecture capable enough and versatile enough to excel in all of Hewlett Packard's computer markets: commercial, engineering and scientific, and manufacturing. Such an architecture would have to scale easily across a broad performance range, provide for straightforward migration of applications from existing systems, and serve as the architectural foundation for at least the next decade of product development.

To address this problem, an unusual group of people was brought together, from within and outside Hewlett-Packard, possessing unusually diverse experience and training. Under the leadership of Bill Worley, this small group of compiler designers, operating system designers, performance analysts, hardware designers, microcoders, and system architects was forged into a team. The intent was to bring together many different perspectives, so that the team could deal effectively with design trade-offs that cross the traditional boundaries between disciplines.

The design methodology was as unusual as the team. It was an iterative, closed-loop, measurement-oriented approach to computer architecture. The process began with data collection and analysis of what computers—Hewlett-Packard's and others'—were actually doing during application execution. Early results validated the suggestions of some RISC architecture researchers that simpler designs were a better match to the actual behavior of machines, and could substantially improve cost/performance.[1] The scalability and generality requirements provided further incentives to reduce system complexity.

After a simple "core" architecture was postulated, the team examined it intensively through simulation and measurement. We evaluated its suitability as a target for compilation and optimization, and as a host for modern operating systems. Logic designs were done simultaneously in several circuit and packaging technologies to evaluate the implications of the architectural decisions on hardware realizations.

After a round of evaluation, the results became the basis for a series of proposed refinements to the architecture. After critical study, the best proposals were incorporated into the architecture, the simulator was updated, and the evaluation process began again.

This process continued for four major (and many minor) iterations over a period of 18 months. At each successive iteration, the architecture and all proposed changes were published internally for review by key technical people in product divisions. As the project progressed, an increasing proportion of the proposals and evaluations came from divisional participants.

The iterative design sometimes resulted in adding a function. For example, the frequent requirement to shift index registers to index to half words, words, or double words in a byte-addressed machine led to the addition of a zero-to-three-bit preshifter to scale one of the inputs to the adder.

More frequently, iteration resulted in deleting mechanisms revealed as too onerous or too little used. An example is the deletion of the STORE INDEXED instruction, because it was the only instruction that would have required a register file capable of reading three registers simultaneously. Compiler strategies were found that all but eliminated the need for the STORE INDEXED instruction, which in any case could be simulated in two instructions. Another example was the deletion of a rather irregular MULTIPLY STEP instruction, when it was discovered that virtually all integer multiplications could be performed efficiently using SHIFT AND ADD instructions, which were a natural byproduct of the index preshifter described above.

The result of this process is an architecture honed by data, tested against various implementation technologies, and broadly tuned to a wide variety of system and application tasks.

Overview

An HP Precision processor is one element of a complete system. The system also includes memory arrays, I/O devices, attached processors, and interconnection structures such as buses and networks. Fig. 1 shows a typical system. The processor interfaces to a central bus like any other module and uses the bus to reference main memory and I/O devices. External interrupts are also transmitted over the buses.

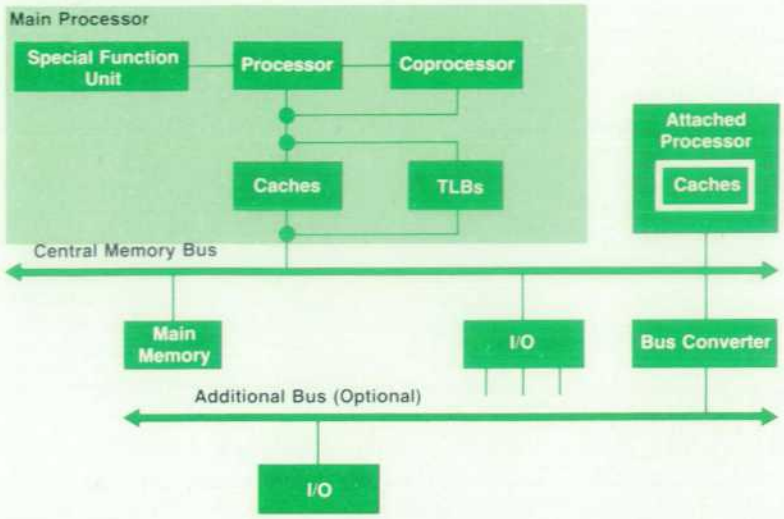


Fig. 1. *HP Precision Architecture system organization.*

Processor Overview

The processor module is organized as instruction fetch and execute units with a tightly coupled high-speed cache system. While a cache is optional, it is such a cost-effective component that nearly all processors will incorporate this hardware. The processor module may also include a hardware address translation table called a *translation lookaside buffer* or *TLB*, and *assist hardware* for extra functions such as floating-point operations. The main data paths are 32 bits wide, and the memory system is byte addressed.

The execution unit performs data transformations on local registers and generates addresses to reference the cache and main memory. It has a memory system interface for moving data operands between the memory system and the registers. The execution unit may be supplemented by assist hardware—coprocessors or special function units—to augment its capabilities for application-specific operations or data types. This is discussed further in the sections on the execution model.

The fetch unit calculates the instruction address, fetches the instruction, decodes it, and sends information to the execution unit. The fetch unit greatly benefits from a reduced-complexity instruction set. Instructions are all fixed-width 32-bit objects, simplifying decoding and calculation of the next instruction address. The fetch unit is responsible for the control flow in the processing of instructions. This is discussed further in later sections on the control model.

HP Precision Architecture uses a memory hierarchy as a cost-effective means of achieving nearly the speed of the fastest (highest) memory level, with the capacity of the largest (lowest) memory level. The highest level of the hierarchy is the registers, followed by the caches. Main memory is the next level and the I/O system provides the largest and slowest level of storage. In HP Precision Architecture, the cache system is architecturally visible in the sense that there are cache control instructions for cache management. A virtual memory system is a characteristic feature on all but the smallest HP Precision processors. Virtual address protection and translation provide security and a large, flat, global address space for all processes. This is discussed further in the sections on the addressing and protection model.

Provisions are made for attached processors, which interface to the system hierarchy at the memory bus level, and typically have their own registers and local cache system. Attached processors can provide such functions as I/O or vector processing. Clustered and tightly coupled multiprocessing are also supported for modular expansion of the system.

Processing Resources

The processing resources are organized around three register arrays and a few specialized registers (see Fig. 2). The general register array contains general-purpose registers used for all computations. The space register array is used to build virtual addresses. The control register array is a collection of registers used for virtual address protection, interruption processing, and other miscellaneous functions.

The general register array contains thirty-two 32-bit general-purpose registers. Register zero is special: it always returns zero when read and discards any result when used as a target register. This specialization is easily implemented in hardware and eliminates the need for instructions for unary or condition-testing operations. For example, a copy operation is a logical OR with register zero and unary SUBTRACT also uses register zero as a source. Registers 1 and 31 are also specialized as implied targets for a few instructions that have no space in the instruction for target register specifiers.

The space register array contains eight registers. When

Fig. 2. *HP Precision Architecture processing resources are organized around three register arrays and a few specialized registers.*

one of these is concatenated to a 32-bit address offset, a virtual address is formed. Three levels of the architecture are defined, according to the amount and degree of virtual addressing supported. The level-zero HP Precision processor does not support any virtual addressing and need not implement the space registers. When building a processor for a highly integrated, dedicated system, it is a considerable savings in hardware cost to eliminate the virtual address hardware. General-purpose computers, however, require virtual addressing. A level-one processor supports 16-bit space registers for a 48-bit virtual address space and a level-two processor implements 32-bit space registers to allow the full 64-bit virtual address space.

The control register array consists of twenty-five registers which contain system state information. Four of these control registers are used by the virtual address system to identify protection groups for the current process. The shift amount for instructions that perform variable-length shifts is stored in a control register. An interval timer is included as a control register. The configuration of coprocessors in a system is also stored in a control register. The remaining control registers are used as temporary registers and to record the state of the machine at the time of an interruption.

An HP Precision processor also maintains registers for the current instruction address, the current instruction, and the processor status word (PSW). The current instruction address is divided into its virtual space identifier (IAS) and its offset (IAO) within the space. The instruction register (IR) contains the current instruction. The PSW holds various flags for enabling virtual addressing, protection, interruptions, and other status information.

Fig. 2 shows the processing resources. A complete context switch only involves the saving of the general registers, the space registers, and several of the control registers. The instruction address registers and PSW are saved in control registers by the hardware at the time of any interruption. Since the process state is small and no extra manipulation of cache or TLB (translation lookaside buffer) structures is necessary, fast context switching is obtained. No additional resources are needed to save intermediate machine states, since interruptions are always taken at instruction boundaries.

Data Types

HP Precision Architecture supports data types for arithmetic, logical, and field manipulation operations. All data objects must be stored on their naturally aligned addresses, that is, 32-bit data objects must start on word-aligned (four-byte) addresses, 16-bit data objects must start on half-word-aligned addresses, and 8-bit data objects must start on byte-aligned addresses. This general alignment rule is easily obeyed by software and significantly improves the cost and speed of cache memory hardware. It also eliminates the possibility of a cache miss or address translation fault in the middle of a data or instruction reference, thereby simplifying the processor control.

Signed and unsigned integers may be 8, 16, or 32 bits long. Signed integers are represented in two's complement form. Characters are 8 bits long and conform to the ASCII standard. While bits are not directly addressable, efficient support is provided to manipulate and test individual bits and bit fields in general registers. Both packed and unpacked representations of decimal numbers are supported by software. Packed data is always aligned on a word boundary and consists of 7, 15, 23, or 31 BCD digits, followed by a sign digit.

Floating-point numbers are addressed as 32-bit (single-precision) or 64-bit (double-precision) quantities. The coprocessor interface allows this wider data path for loading and storing double-precision floating-point operands. The floating-point data format conforms to the ANSI/IEEE 754-1985 standard.

Execution Model

HP Precision Architecture assumes a register-based execution model, with all operands coming from registers and all results going back into registers. The thirty-two general-purpose registers are used for local storage of operands, intermediate results, and addresses.

The execution engine for the basic HP Precision instruction set consists of a simple arithmetic logic unit (ALU) and a shift-merge unit (SMU), as shown in Fig. 3. The ALU has a preshifter on one port and a complementer on the other port. The SMU consists of a shifter and a mask-merger. It is used for implementing field manipulation operations. The shifter concatenates two 32-bit operands and performs a right shift. The mask-merger selects a contiguous field of bits from the output of the shifter and merges this with the other bits from its second input source, forming a 32-bit result. The second input source to the mask-merger may be a mask of all zeros or all sign bits, or may come from a general register.

The typical execution data flow consists of reading two operands from general-purpose registers, routing these two operands through the ALU or the SMU with the proper function selected, and storing the result back into a general register. This is the data flow for the basic three-register model of execution, which facilitates single-cycle execution, since no memory references are required.

Single-Cycle Execution

A primary design goal was that all functional computations in the basic instruction set could execute in one machine cycle in a pipelined implementation of the processor architecture. Operations were selected for inclusion in the basic instruction set only if they could be implemented in a reasonably small number of logic levels, to guarantee a short cycle time. This does not necessarily mean that the operation performed had to be primitive in function. In fact, rather sophisticated operations were allowed in the architecture if they proved useful to the compilers, and were implementable in a short machine cycle with relatively simple hardware.

Complex operations that are necessary to support required software functions but cannot be implemented in a single execution cycle are broken down into primitive operations, each of which can be executed in a single cycle. Examples are the DECIMAL CORRECT operations which are primitive operations for performing arithmetic on BCD data, the SHIFT AND ADD operations which are primitives

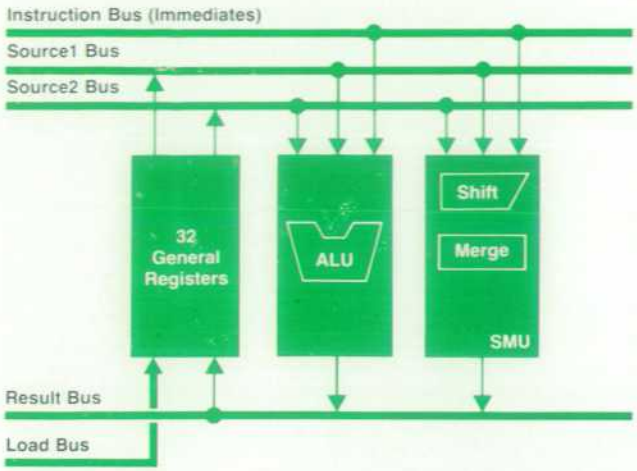


Fig. 3. *The execution data path consists of a simple arithmetic logic unit (ALU) and a shift-merge unit (SMU).*

for integer multiplication, and the DIVIDE STEP operation which is a primitive for integer division.

Single-cycle execution was a design goal of the architecture, but is not a constraint on the implementations. For example, an HP Precision microprocessor may operate with slower memories, performing a load instruction in more than one cycle.

Immediates

A notable aspect of HP Precision Architecture's register-based model of execution is its heavy use of the instruction register as a source for operands, in addition to the thirty-two general-purpose registers. Many HP Precision instructions have an immediate field embedded in the 32-bit fixed-length instruction. These immediates are made maximal-length, in the sense that they fill up all unassigned bits in the given instruction. This maximizes the probability that a constant can be represented in the instruction as immediate data. Although immediates come in various sizes in different instruction classes, their sign bit is always in a fixed position. An immediate operand is advantageous since it does not have to be loaded to a general register and therefore saves both a memory access and the use of a general register.

Although maximal-length immediates in an instruction are capable of representing most of the constant values that are needed, it is desirable to have the capability of embedding full-length 32-bit immediates in the instruction stream. HP Precision Architecture does this by means of a pair of instructions. First, a long-immediate instruction is used to load or add the most significant twenty-one bits of the immediate value, padded on the right with eleven zeros, into a general register. A subsequent instruction, using this register as the base register, supplies the low-order bits to complete the 32-bit immediate. In this way, a 32-bit constant value can be placed in a general register, or a load or store instruction can be performed with a full 32-bit static displacement. An alternative approach—creating a double-word instruction—would have introduced the more complex possibility of a page fault occurring in the middle of an instruction fetch.

Load and Store Operations

The general register array is the only level of the memory hierarchy that interacts with the execution engine. The general registers interact with the rest of the memory hierarchy via the LOAD and STORE instructions.

The LOAD and STORE instructions are designed to execute in a single cycle in a pipelined implementation of the architecture that includes a data cache memory that operates at the speed of the processor. This immediately excludes the specification of multiple loads and stores or levels of address indirection in a single instruction.

Even with a fast cache memory, data may not be available until one cycle after the memory access is initiated. Therefore, following a load instruction, the software tries to schedule one or more instructions that do not use the target register being loaded. However, the hardware must be able to interlock the pipe if an instruction following a load instruction uses the target register that has not yet been loaded.

The size of the data item loaded or stored can be a byte, a half word, or a full word. It is possible to store any contiguous sequence of bytes within a word, either starting from the leftmost byte or ending with the rightmost byte, using the STORE BYTES instruction. For example, it is possible to store the leftmost three bytes or the rightmost three bytes of a register into three contiguous bytes of memory. This instruction is a useful primitive for moving unaligned strings of bytes from one memory location to another.

All address calculation in the LOAD and STORE instructions is based on the base register plus displacement addressing mode. The displacement can be a long 14-bit signed displacement, a short 5-bit signed displacement, or an index register. An index register, if used, may optionally be shifted left by 1, 2, or 3 bits to permit integer addressing to half words, words, or double words, respectively. Both the base register and the index register used in address calculation can come from any of the general registers.

Flexible Address Modification Mechanisms. Automatic address modification mechanisms allow one to walk through a data structure more efficiently, by updating the address register to the next item in the data structure to be referenced while fetching the current item.

Flexible address modification mechanisms are included in HP Precision Architecture, providing high-performance functionality in a single cycle. For example, it is possible to modify the base register for a subsequent load or store instruction by adding to it the long or the short displacement value specified in the instruction itself, or the value of an index register, optionally shifted to multiply by the size of the object to be loaded or stored.

If address modification is specified, either premodification or postmodification can be performed. *Premodification* means that the address calculation is performed and the result used as the address to initiate the memory access. *Postmodification* means that the original content of the base register is used as the address to initiate the memory access.

An unusual feature of this premodify or postmodify addressing mode is that in the long-displacement instructions, the sign bit of the displacement is also used as the bit to select premodification or postmodification. This al-

Floating-Point Coprocessor

HP Precision Architecture generally conforms to the concept of a simple instruction set realizable in cost-effective hardware. However, certain algorithms like floating-point operations realize substantial performance gains when implemented on specialized hardware. The floating-point instruction set is an example of HP Precision Architecture's instruction extension capabilities.

Floating-point instructions are supported through an assist coprocessor to provide high-performance numeric processing. As a coprocessor, the floating-point unit contains its own register file and executes concurrently with the basic processor. Operands from the caches are loaded or stored from any of twelve floating-point registers. The data format, all operations, and exceptions fully conform to the ANSI/IEEE 754-1985 standard. Very high-performance coprocessors can be implemented by combining hardware pipelining with the HP Precision high-level language optimizer.

The floating-point coprocessor is organized like the basic processor. All operands from main memory are referenced using coprocessor load and store instructions. Normal virtual address translation and protection checks are made and data is transferred between the cache (or memory) and the floating-point register file. Both single-precision (4-byte) and double-precision (8-byte) operands can be referenced with a single instruction. Quad-precision (16-byte) operands are referenced using a pair of double-precision coprocessor memory reference instructions.

The basic processor performs index and short-displacement address calculations for the coprocessor load and store instructions. While STORE INDEXED instructions are not provided for the basic processor, COPROCESSOR STORE INDEXED instructions are provided since only two general register reads and a nonconflicting coprocessor register read are required.

Floating-Point Register File

The register file contains twelve 64-bit data registers, a 32-bit status register, and seven 32-bit registers for reporting exceptional conditions, as shown in Fig. 1. The twelve data registers also form six 128-bit quad-precision registers. The data registers are numbered from 4 through 15. Register 0 holds the status register. When register zero is used as the target or source of a coprocessor load or store, the status register is referenced. But when used as the source of an operation, register zero returns a floating-point zero. This is used for simple assignments, arithmetic negation, and comparisons with zero.

Fig. 1. *Floating-point register file.*

The status register holds information on the current rounding mode, the exception flags, and exception trap enables for the five IEEE exceptions: overflow, underflow, divide by zero, invalid operation, and inexact. If the exception trap is not enabled, then a default result is returned and the corresponding exception flag is set in the status register. If the exception trap is enabled, an interruption to the main processor occurs, with the exception and the instruction causing it recorded in an exception register. On overflow, underflow, and inexact exceptions, the correctly rounded result is delivered to the destination register. On invalid operation and divide-by-zero exceptions, the source registers are preserved. Users can specify a trap handler for any of the five IEEE exceptions, using the information preserved.

The coprocessor uses an additional nonmaskable exception, called *unimplemented*, to pass off to software those operations not implemented by the coprocessor hardware. The unimplemented trap triggers a software emulation of the desired operation with the original operands.

The Boolean result of a floating-point comparison is stored in a bit in the status word. This bit can conditionally nullify the next instruction when tested. No conditional branch is allowed. A conditional branch would have increased the critical path for branch determination.

Floating-Point Operations

The floating-point coprocessor defines eleven fundamental operations in three precisions. All of the operations, except for conversions to fixed-point formats, produce floating-point results. Source and destination formats are the same except for conversions that have explicit source and destination formats. Rounding is specified by a mode field in the status register. The COPY and ABSOLUTE VALUE operations are nonarithmetic and do not cause exceptions. The following table summarizes the defined arithmetic operations for single, double, and quad formats.

FADD	Addition
FSUB	Subtraction
FMPY	Multiplication
FDIV	Division
FREM	Remainder
FSQRT	Square Root
FRND	Round
FCMP	Compare

CONVERSION instructions from floating-point formats to fixed-point formats and between floating-point formats are also included. When converting from floating-point to fixed-point format, the current rounding mode can be temporarily changed to round-to-zero. Many programming languages define conversion to integer as rounding to zero. In accordance with the standard, the default rounding mode is rounding to the nearest integer.

Scalability and Performance

HP Precision Architecture is designed to adhere strictly to the IEEE floating-point standard. The standard does not, however, require that all floating-point operations be performed in high-performance hardware, and does not specify the instruction set level presentation of the hardware. Whenever there is little performance advantage to be gained by performing an operation in hardware, consideration should be given to simplifying the hardware and performing the operation in software. The unim-

plemented exception trap mechanism is employed to avoid handling operations and exceptional conditions in hardware.

The simplest HP Precision systems may completely exclude a floating-point unit. Each floating-point instruction causes an assist emulation trap and system software completely simulates the function. Special control registers speed the simulation of load and store instructions. Some implementations can reduce the complexity of hardware control by supporting only those operations that use available floating-point hardware. In this case, exceptional conditions arise that can require additional processing or software assistance. For example, the unimplemented exception trap mechanism can be used to handle the square root operation and corner case operands like infinities, NaNs (not a number), and denormalized numbers.

The floating-point coprocessor is architected to be pipelined to allow very high-performance numeric processing. Fundamental to this is the delaying of exception reporting. If the coprocessor must inform the basic processor immediately that the current instruction overflows, then little concurrent processing and pipelining is possible. In HP Precision Architecture, the coprocessor can freely accept a non-load/store operation independent of any earlier operations, provided space exists in the exception registers to report exceptions. This allows seven instructions to be in execution simultaneously while the basic processor continues. Load and store instructions to independent data registers can also be fully overlapped. The coprocessor need only complete pipelined instructions when the result is being requested. References to the status register are special and require all operations to be completed.

A minimally pipelined machine might perform only a single floating-point operation at a time, but permit load and store operations to execute concurrently. This requires an interlock against stores of the single result register specified in the executing operation, and an interlock on the source registers during the period that the source exceptions are tested in the operations. The second interlock may never occur in some implementations.

The floating-point instruction set is designed to allow software the option of performing pipelined operations without the need for complex hardware control. The high-level language optimizer places instructions in a sequence to avoid the most common interlocks. The use of results is delayed as long as possible and effective overlap with other integer operations is obtained.

lows the specification of premodification or postmodification without using up a bit of the long displacement field. Memory accesses with long displacement fields perform predecrement or postincrement, depending on the sign of their displacements. In theory, this is less general than allowing the specification of premodification or postmodification to be orthogonal to the sign of the displacement, as is true for the short-displacement load and store instructions. In practice, however, the feature works very well for maintaining stacks stored in the memory. For example, for a stack growing in the direction of decreasing memory addresses, pushing onto the stack from a register is done by a store with predecrement and popping off the stack is done by a load with postincrement.

Combined Instructions

The basic types of operations in most instruction sets fall into three categories: data transformation operations, data movement operations, and control operations. In general, one instruction performs one of these operations. A *combined instruction* performs more than one of these operations in one instruction. In HP Precision Architecture, almost every instruction performs a combination of two of these operations in a single cycle, with relatively simple hardware.

HP Precision Architecture has two types of data transformation and control operation combinations. The first type has a more general transformation operation combined with a restricted control operation, whereas the reverse is true for the second type. Examples of the first type are ADD instructions that can conditionally skip the execution of the following instruction. Examples of the second type are COMPARE AND BRANCH instructions.

The LOAD and STORE instructions combine a data movement operation (moving data between a general register and the memory system) with a transformation operation (the accompanying address calculation and modification).

HP Precision Architecture's combined instructions allow the execution engine to be used efficiently, since the data transformation portion of a combined instruction is performed in the simple execution engine shown in Fig. 3.

Assist Instructions

The architecture allows for flexible instruction set extensions by means of assist instructions. *Assist instructions* are instructions in which the data movement functions are defined between the processor or the memory and the assist hardware, but the data transformation functions are left unspecified. An *extension instruction* is defined by specifying in an assist instruction the data transformation operations to be performed by the assist hardware. *Assist hardware* is optional hardware that accelerates the execution of a set of assist instructions. In the absence of the assist hardware, an extension instruction is emulated by software, using a transparent assist emulation trap mechanism. Critical information required for emulation is saved in control registers, substantially reducing the emulation time.

HP Precision Architecture allows up to sixteen assists in a system configuration, supporting sixteen logically differentiated sets of instruction set extensions. These are divided into two generic types of assists: the special function units (SFUs) and the coprocessors (COPs).

Special function units use the general registers as sources and targets of operations. They are coupled very closely to the basic processor and its register buses.

Coprocessors provide functions that use either memory locations or coprocessor registers as operands and targets of operations. They are coupled less closely to the basic processor. Coprocessors may also directly pass doubleword quantities between the coprocessor and the memory. This is suited to the manipulation of quantities that are too large to be handled directly in the general registers.

The HP Precision instruction set can be extended by defining a set of assist instructions in applications where specialized hardware is justified by its frequency of use or by the resulting performance improvement. The architecture allows such instruction set extensions without compromising software compatibility. An example of such an

instruction set extension is the instruction set for the floating-point coprocessor (see box, page 8).

Control Flow Model

HP Precision Architecture defines a computer in which the flow of control passes to the next sequential instruction in the memory unless directed otherwise by branch instructions, nullification of instructions, or interruptions. These three mechanisms can potentially alter the sequential flow of control in instruction processing.

Branching

The architecture has both unconditional and conditional branch instructions. All branch instructions exhibit the delayed branch feature.

In a pipelined processor, it is difficult to execute a branch instruction in one cycle, since the branch target address has to be calculated before the target instruction can be fetched. Hence, taken branches frequently result in pipeline interlocks, in the absence of other prefetch mechanisms.

To minimize such pipeline interlocks, HP Precision Architecture defines a one-instruction delayed branch. This means that a *delay instruction*, which is the instruction following a branch instruction, is executed before the program control flow passes to the target instruction of the branch. The delay instruction is not executed when it is explicitly nullified by its preceding branch instruction. This branch nullification feature is explained later.

The delayed branch mechanism allows compilers to schedule a useful instruction in the cycle during which the branch target address is calculated. For example, this might be an instruction that preceded the branch instruction.

Unconditional Branching. HP Precision Architecture defines *local* branches, where the control flow passes to another location within the current virtual space and *external* branches, where instruction processing continues at a location that may be in a different virtual space.

The design of high-speed pipelines is simplified if branch target address calculations can be made before the execution of the branch instruction itself. In HP Precision Architecture, the most common branch instructions have branch targets calculated relative to the address of the branch instruction itself, with displacements given in the branch instruction. These are called *relative* branches with *static displacements*. Unconditional branch instructions have a 17-bit signed displacement field, and the conditional branches have a 12-bit signed displacement field.

Although a 17-bit displacement will cover almost all branch distances, it is insufficient in certain situations. Furthermore, it is not always possible or convenient to generate a static displacement at compile time for some branches. Hence, the architecture includes branch instructions with 32-bit *dynamic displacements* specified by the contents of a general register.

Branches are also needed to locations that have no relation to the address of the branch instruction—for example, to independent relocatable modules. This is called *absolute* branching, since the address of the target instruction can be anywhere in the address space. HP Precision Architecture also allows absolute branches: the branch displacement is added to the contents of a general register called the base register.

Subroutine Calls. The subroutine call primitives are BRANCH AND LINK instructions, which save the return address of the calling routine in a general register before transferring the control flow to the subroutine. Both local (intraspace) and external (interspace) subroutine calls are defined. The external subroutine calls must save a larger return pointer, indicating also the virtual space of the caller.

The external BRANCH AND LINK instruction uses implicit link registers for saving both the caller's space identifier and the offset within that space. Space register zero (SR 0) is used for saving the space identifier and general register thirty-one (GR 31) is used for saving the offset address. This permits the maximum number of bits to be used for encoding the static branch displacement.

Subroutine returns are accomplished by using an absolute branch instruction, specifying the general register used to save the link address in the BRANCH AND LINK calling instruction. If appropriate software conventions are used, a uniform subroutine return sequence can be used for both local and external calls.

Inter-Ring Branches. Four hierarchical protection rings are implemented in HP Precision Architecture. Each ring has a privilege level associated with it, the innermost ring (privilege level 0) being the most privileged ring and the outermost ring (privilege level 3) being the least privileged ring.

The architecture defines unconditional branch instructions that perform inter-ring crossings in one instruction. Three of these are outward branches, causing a decrease in the process privilege level. Only one branch instruction (GATEWAY) is an inward branch, causing an increase in privilege level.

Conditional Branching. In many architectures, conditional branching is accomplished by two separate instructions. The first instruction calculates a condition, and saves the result of this condition calculation in state flip-flops in the processor called a *condition code*. A subsequent conditional branch instruction may alter the program's control flow depending on the value of the condition code.

Statistics of instruction sequences show that in an overwhelming majority of cases, a conditional branch instruction is immediately preceded by the instruction that sets the condition tested by the branch. HP Precision Architecture capitalizes on that fact by combining the two instructions into one instruction, thus achieving code compaction, reduction of execution time, and elimination of condition code flip-flops in the processor state. Each conditional branch instruction includes a data transformation operation, which generates a condition that is used immediately to determine whether the branch is taken or not. Such conditional branch instructions also provide greater opportunities for an optimizer to reorder instructions, with less bookkeeping.

There are four kinds of operations that can be executed with a conditional branch instruction. The ADD AND BRANCH instruction is useful for closing loops. The COMPARE AND BRANCH instruction is useful for closing loops and for if-then-else control structures. The BRANCH ON BIT instruction

allows branching on the value of any bit in a general register. The MOVE AND BRANCH instruction is useful for reinitializing a register before branching away.

HP Precision Architecture also implements a special nullification scheme to optimize the use of the delay instruction following a conditional branch instruction.

Nullification

HP Precision Architecture defines a control flow feature called the *nullification* of the immediately following instruction. When an instruction is *nullified*, it executes as a no-operation (NOP), and the effect is as if it had never been in the instruction stream. This means that no change in any architecturally visible state, like general registers, memory, control registers, or space registers occurs because of a nullified instruction. A nullified instruction does not cause any traps to be generated, and it does not cause its successor instruction to be nullified. All branch instructions and data transformation instructions have the ability to nullify the instruction to be executed next.

All branch instructions have a single-bit nullification field. An unconditional branch instruction can "always nullify" or "never nullify" the execution of its delay instruction by setting the value of the nullification field to one or zero, respectively. A conditional branch instruction can "conditionally nullify" or "never nullify" the execution of its delay instruction in the same manner. The never nullify feature is used whenever a delay instruction can be found that can always be executed, regardless of whether the branch is taken or not.

A conditional branch is taken when the condition it specifies evaluates true. To optimize the use of the delay instruction following the conditional branch, the delay instruction is nullified for backward branches only if the condition is false, and for forward branches only if the condition is true. Since the compilers use the convention that loops are closed with backward branches, the delay instruction of this branch can now be "inside" the loop, saving a cycle on each iteration. The following example illustrates this.

```
        .
        .
        .
        XXX
LOOPB:  YYY
        .
        .        } Loop body
        .
        ZZZ
        COMBT,C,n,    r1,r2, LOOPB;
        XXX
        WWW
```

As shown, the first instruction (XXX) of the loop body can always be duplicated following the loop-closing branch, COMBT. When the COMBT instruction is executed, if condition C is true, then the XXX instruction is executed and control passes back to LOOPB. Otherwise, the next instruction (XXX) is nullified and processing continues with instruction WWW.

For forward branches, the nullification definition allows shorter code sequences for if-then constructs, as shown in the following example.

```
                .
                .
                .
                RRR
"If"            .
Code            .
                .
                SSS
                COMBT,C,n    rx,ry,THRU;
                TTT
"Then"          .
Code            .
                .
                UUU
        THRU:   VVV
                .
                .
                .
```

When the conditional branch instruction, COMBT, is executed, if condition C is true, the next instruction is nullified and the branch is taken around the "Then" code to the location THRU. Otherwise, the next instruction (TTT) is executed.

Every data transformation instruction has an implicit conditional skip operation built into it. In a single cycle, the function specified by the transformation instruction is performed by the execution engine, and a condition specified in the instruction is evaluated. If the condition evaluates true, then the next instruction to be executed is nullified. If the condition evaluates false, then the next instruction is executed, or not nullified.

The following example shows the use of nullification in an ALU instruction to implement a compact control sequence for a high-level language construct.

High-level language:

if (a < b) then b = b + 1;

Equivalent HP Precision assembly language:

```
SUB,>=   a,b,r0;   Subtract [GR b] from [GR a], discarding
                   the result, and nullify next instruction if
                   [GR a] ≥ [GR b].
ADDI     1,b,b;    Add the immediate value 1 to [GR b],
                   writing the result back to GR b.
```

Conditional Trap. In some instructions, the condition specified in the instruction is used to cause a conditional trap, rather than the nullification of the next instruction. An advantage of taking a conditional trap rather than conditionally nullifying a branch to a trap routine is that the majority of instructions do not incur the penalty of a nullified instruction. For example, when an add or subtract instruction is used to perform range checking, the penalty of a conditional trap is taken only in the rare cases where the range check fails.

While it is a common feature of other architectures to have an ALU instruction trap on arithmetic error conditions like overflow, it is a special feature of HP Precision Architecture to allow trapping on defined conditions that are not arithmetic errors.

Assist Nullification. In assist nullification, the condition upon which nullification is performed is generated by the assist hardware rather than by the basic processor. Instead of defining assist branch instructions, the processor's unconditional branch instructions are used for control flow changes in assist programs. The equivalent of conditional branching is achieved using a pair of instructions: a data transformation assist instruction with its nullification field set to one, followed by an unconditional branch instruction. The assist instruction generates a condition that determines whether the following branch instruction should be nullified.

An assist can be defined with the nullification operation dependent upon the condition generated either in the current assist instruction or in the previous assist instruction. The latter is called *delayed nullification*. Delayed nullification allows other instructions, executed by the basic processor or other assists, to be scheduled during the time the assist hardware is performing a lengthy computation that generates the condition for determining nullification.

Interruptions

Interruptions are anomalies that occur during instruction processing, causing the control flow to be passed to an interruption handling routine. In the process, certain processor state saves and changes are made automatically by the hardware. Upon completion of interruption processing, a RETURN FROM INTERRUPT instruction is executed, which restores the saved processor state, and execution proceeds with the interrupted instruction.

Traps, faults, checks, and interrupts are different anomalies that may happen during instruction processing on a computer. In HP Precision Architecture, they are all handled by the same basic mechanism. The term interruptions is used in discussing these anomalies as a group.

The architecture implements a *single-level* interruption system. This means that once an interruption is chosen for service, it cannot be preempted for service by a higher-priority interruption. It also implies that only one interruption is serviced at a time. If an instruction raises multiple interruptions, the highest-priority interruption is serviced, and then the instruction is reexecuted, which causes the other interruptions to be raised again. Then the next highest-priority interruption is serviced, and so on.

The nesting of interruptions is not excluded, since the interruption handling routine can choose to reenable other interruptions once it has saved the appropriate state. Since the machine state is saved in registers rather than in memory when an interruption is serviced, interruption handlers must leave interruptions disabled until they have saved the machine state in memory.

In certain pipelined processors, interruptions are often not precise, in the sense that they may not be serviced immediately after the instruction that caused the interruption. This is because in overlapped instruction processing, several successive instructions may already have been partially or fully processed by the time the interruption caused by an instruction is generated. This imprecision adds considerable complexity to interrupt handling routines.

In a nonoverlapped processor, precise interruptions are easy to implement, since an instruction is fetched and completely executed before the next instruction is fetched. Hence, interruptions can be serviced between instructions, that is, after the instruction causing the interruption and before the next instruction's processing starts.

HP Precision Architecture requires that interruption servicing appear the same for both overlapped and nonoverlapped processors. Hence, all implementations must provide precise interruptions, and resume execution at the same instruction as a nonoverlapped implementation.

Traps and Faults. Traps and faults are *synchronous* interruptions, meaning that they are caused by the processing of an instruction or a sequence of instructions. A *trap* occurs when the function requested by the current instruction cannot or should not be carried out, or system intervention is desired by the user before or after the instruction is executed. A *fault* occurs when the current instruction requests a legitimate action that cannot be carried out because of a system problem, such as the absence of a page from main memory. After the system problem has been corrected the faulting instruction will execute normally.

In HP Precision Architecture, the overflow trap and the conditional trap occur for arithmetic instructions. The privileged operation or privileged register traps occur when certain system management instructions or control registers are accessed by a process with insufficient privilege. An illegal instruction trap is generated for undefined operation codes, or illegal instruction sequences which could otherwise cause security breaches. The assist exception and emulation traps allow assist hardware to request the processor to service assist-generated traps, or to emulate assist instructions not supported by hardware.

Virtual memory faults and traps may also be generated for instruction fetches or data fetches in virtual mode. For example, if the virtual-to-physical address translation is not found in the hardware translation lookaside buffer, a TLB miss fault is generated. If a virtual memory access fails the protection checking required for the access, then a memory protection trap is generated. These traps are generated independently for instruction and data virtual accesses. The first time a page is written, a TLB dirty-bit trap occurs, which is used by the system to distinguish unmodified pages from modified (dirty) pages at page replacement time.

HP Precision Architecture also has a rich set of debugging support traps. A BREAK instruction is defined in the architecture to allow the insertion of software breakpoints. Whenever such an instruction is executed, a break trap occurs. Any store instruction to a virtual address may also generate a data memory break trap, if this trap is enabled by a bit in the TLB entry. This allows the tracing of all data updates to a given page. A similar facility traps on any reference whatsoever to a given virtual page. Traps may also be generated, if enabled, after a branch is taken, or when the privilege level of the running process is promoted or demoted. Architectural support for software rollback

schemes is also implemented by means of a recovery counter trap. A 32-bit control register, the recovery counter, can be initialized to any integer value. If enabled, the counter is decremented for every nonnullified instruction that is executed, and a recovery counter trap is generated when a zero value is reached. The recovery counter can be used in fault recovery, to permit an exact reexecution of the instruction stream since the last checkpoint.

Checks and Interrupts. A *check* occurs when a hardware malfunction is detected. Depending on the nature of the malfunction, checks may be synchronous or asynchronous with respect to the instruction stream. HP Precision Architecture defines two types of machine checks: a high-priority machine check and a low-priority machine check.

An *interrupt* occurs when an external entity, like an I/O device or the power supply, requires attention. Interrupts are asynchronous with respect to the instruction stream.

There are thirty-two external interrupt classes, each of which can be individually masked by privileged software. The architecture defines two control registers specifically for handling these external interrupts. The external interrupt request (EIR) register and the external interrupt enable mask (EIEM) register each have thirty-two bits, one for each external interrupt class. A privileged instruction allows the writing of any set of mask bits to the EIEM register and the clearing of any selected bits in the EIR register. When an external interrupt of any class occurs, its corresponding interrupt pending bit is set in the EIR register. If the corresponding mask bit in the EIEM register is also one, then an external interrupt is taken. An EIR register bit remains set, leaving the external interrupt pending, until explicitly reset by an interruption handler.

Relative priority of these thirty-two external interrupt classes is not assigned by the architecture or by the hardware. When multiple unmasked external interrupts occur simultaneously, or when there are multiple external interrupts pending in the EIR register, the external interrupt handler selects the order of service.

Interruption Parameters and Servicing. Six control registers are defined to save interruption parameters and expedite the processing of interruptions. The collection of information in these interruption parameter registers occurs only when the interruption state collection enable flag (Q bit) in the processor status word (PSW) is set.

These interruption parameter registers save the processor status word of the interrupted process, the instruction that is interrupted, and the data address (space and offset portions) for memory reference instructions. Two other register pairs form two queues, saving the space and offset portions of the addresses of the first two instructions to be processed upon returning from the interruption.

The two queues are necessary because in an architecture with delayed branching, at least two return addresses must be saved before jumping to the interruption handler. Two are necessary because the last instruction to be completed before the interruption may be a taken branch. In this case the next two instructions to be executed may not be contiguous, since one is the delay instruction and the other is the target instruction. These queues are constantly updated by the hardware whenever interruption parameter collection is enabled. When an interruption is taken, the queues and other interruption parameters are preserved by disabling further interruption collection.

Interruption servicing is implemented as a fast context switch, which is much simpler than a complete process swap. When an interruption occurs, the current processor status, represented by the PSW, is saved. Then, the PSW is cleared to zeros to disable further interruptions, to enable real-mode addressing, and to freeze the information collected in the interruption parameter registers. The current privilege level is set to the highest privilege level. The control flow then passes to a vectored location in an interrupt vector table, which is dynamically relocatable. This simple set of architecturally defined operations facilitates a fast and uniform switch to interruption servicing for all implementations.

Addressing and Protection Model

HP Precision processors access memory using byte addresses. Larger addressable units include half words, words, and double words. An address is either physical or virtual. All load and store instructions can be used in either virtual or physical mode. Virtual mode is enabled separately for instruction fetches and data accesses by two flags in the processor status word.

A pointer to physical memory is a 32-bit unsigned integer whose value is the address of the first byte of the operand it designates. Physical addresses are used directly, with no protection or access rights checking performed. Virtual addresses are translated to physical addresses and undergo protection and access rights checking as part of the translation. This allows the hardware support for access control to be built into the storage unit.

The input/output (I/O) architecture is memory mapped. That is, complete control of all system components (of which I/O attachments are a special case) is exercised by the execution of load and store instructions to virtual or physical addresses. This approach permits I/O drivers to be written in high-level languages. Furthermore, since the usual page-level protection mechanism is applied during virtual-to-physical address translation, user programs can be granted direct control over particular I/O devices without compromising system integrity.

Virtual Memory Addressing

A virtual address is defined globally and has the same meaning when used by any process. This is in contrast to other architectures, which permit use of the same address for different objects by different processes. The virtual address space is so large that processes can be assigned separate address ranges for private data. Address translation information does not need to change upon a process switch and the information needed for address translation can be represented more compactly. Global virtual addressing therefore allows closely coupled processes to accumulate a stable working set of address translations in spite of frequent process switching.

Virtual memory is structured as a set of address spaces, each containing 2^{32} bytes. A level-one processor implements 2^{16} spaces (16-bit space registers), and a level-two

processor implements 2^{32} spaces (32-bit space registers). A space is specified by a space identifier, and is divided into pages, each 2048 bytes in length.

For a level-two processor, the concatenation of a 32-bit space identifier and a 32-bit offset within the space forms a virtual address. Alternatively, a virtual address may be viewed as the concatenation of a 53-bit virtual page number and an 11-bit offset within the page.

For virtual addressing, space identifiers are specified in space addressing registers. These include the space portion of the instruction address register and the eight space registers SR 0 through SR 7 (see Fig. 4). One such register is implicitly or explicitly selected by every instruction that generates a virtual address.

SR 0 is used as an implied target by the interspace procedure call instruction. SR 1 through SR 7 have no architecturally defined functions, but it is expected that their use will be constrained by the following software conventions. SR 1 through SR 3 are used as scratch registers for the manipulation of 64-bit virtual pointers. SR 4 tracks the current program's space and provides access to literal data contained in the current code space. SR 5 points to a space containing process private data, SR 6 to a space containing data shared by a group of processes, and SR 7 to a space containing the operating system's code, literals, and data. The conventions for SR 4 through SR 7 were chosen to permit use of 32-bit virtual address pointers (see below) for almost all data references.

SR 5 through SR 7 can be modified only by code executing at the most privileged level. SR 0 through SR 4 can be changed by an unprivileged user. Shared libraries or subsystems will be assigned individual code spaces, and branching into those other spaces will involve changing SR 4.

Instruction and Data Addressing. Instruction addresses are computed for instruction fetch, instruction cache flush instructions, instruction TLB instructions, and branch target calculations. Instructions that explicitly reference a space register use the 3-bit S field, located in the instruction, to designate one of the eight space registers.

Data addresses are computed for load, store, semaphore, probe, data cache, and data TLB instructions. Data addresses specify one of the eight space registers in an interesting way: only a 2-bit S field in the instruction is used. When the 2-bit S field is nonzero, it selects the corresponding space registers 1, 2, or 3. When the S field is zero, the space register is designated by adding four to the two high-order bits of the base register specified in the instruction. This allows the selection of space registers 4 through 7.

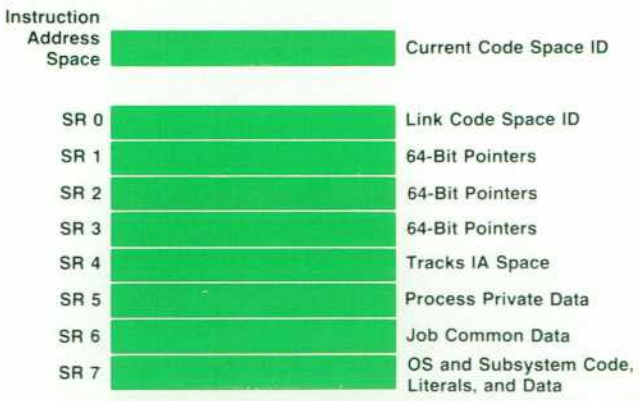


Fig. 4. *Space register conventions.*

Data references with the S field equal to zero allow addressing of four distinct spaces selected by the high-order bits of a 32-bit pointer. This is called *short-pointer* addressing (Fig. 5), since a 32-bit value both specifies an offset and selects a space register. Only one fourth of each space is directly addressable with short pointers. This region corresponds to the quadrant selected by the upper two bits. For example, if a base register contains the hex value 80001000, the content of space register 6 is the space identifier and the third quadrant of the space is directly addressable.

Short-pointer addressing allows the pointer data type of conventional languages to be 32 bits in length. Therefore, such pointers can be handled efficiently in the general-purpose registers. Also, pointers are the same length as the standard integer data type, a situation assumed by a number of existing high-level langauge programs. Long pointers are 48 bits or 64 bits in length, consisting of a 16-bit or 32-bit space identifier together with a 32-bit byte offset within the space, for level-one and level-two processors, respectively.

Software Virtual Address Translation

TLBs (see box, page 16) do not contain the translations for all pages in memory simultaneously. When they do not have the desired translation, a TLB miss occurs. In many architectures, TLB misses are handled in microcode. In HP Precision Architecture, they may be handled in software. When a TLB miss is detected, the hardware does not have sufficient information to complete the instruction being executed. Instead, an interruption is generated to invoke the appropriate TLB miss handler. One miss handler handles misses during instruction fetch, and another handles misses during data access. The virtual address causing the miss is directly available to the TLB miss handler in interruption parameter control registers to expedite miss handling.

Because of the critical effect on system performance of the speed of address translation, all information required to translate the virtual address of a page that is actually present in physical memory must be permanently resident in memory. Because of the size of the virtual address space, tables describing all virtual pages cannot be kept permanently in memory. Thus the data structures used to translate valid virtual addresses (no page fault) describe only physically present pages and have a size proportional to the size of physical memory, consuming less than 2% of the available memory. The information represents a one-to-one mapping between physical and virtual pages. Thus it cannot support memory aliasing (see box, page 16) or process-specific address translation. A desire to use these efficient structures was an important motivation for disallowing both features.

This address translation information resides in a physical page directory (PDIR). The physical-to-virtual address translation is obtained by using the physical address as a direct index into the PDIR. The translation of a virtual address to a physical address is accomplished using two tables, the hash table and the PDIR. Each table is located

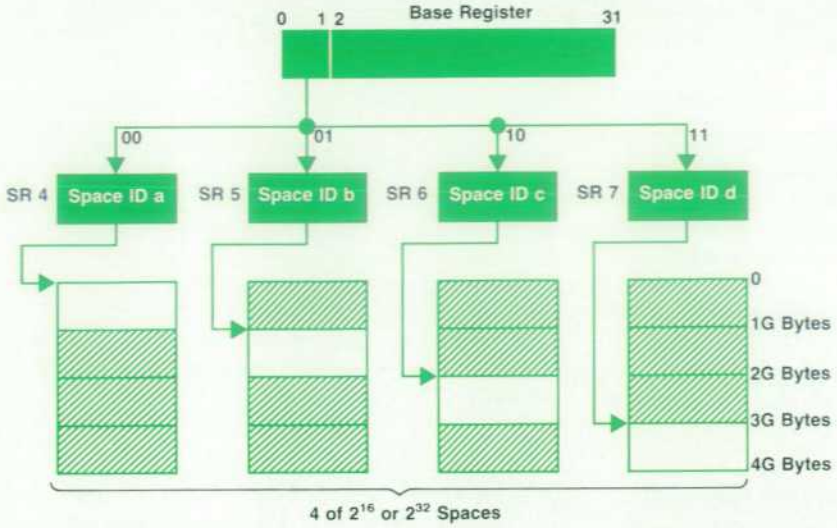


Fig. 5. *Short-pointer space selection allows addressing of four distinct spaces selected by program data.*

by a pointer which defines its absolute starting address. For efficiency, these pointers are kept in control registers (assumed to be CR 24 for the PDIR address and CR 25 for the hash table address).

The purpose of the hash algorithm is to map virtual addresses to a smaller, denser name space. The number of entries in the hash table is typically a multiple of the number of entries in the PDIR, rounded up to the nearest power of two. Since multiple virtual addresses can map into the same hash table entry, they are linked together as a chain of PDIR entries. The TLB miss handler hashes the virtual address, looks up the start of the chain in the hash table, and looks through the chain in the PDIR until it finds either the match or the end of the chain. If it finds the match, it puts the information from the PDIR into the TLB and retries the instruction. If it finds the end of the chain, the page is not in memory and a page fault is signaled by the software.

The physical page directory (PDIR) contains one entry for each page of physical memory, plus one for each physical or virtual I/O device. The entries for physical pages are at nonnegative offsets from the location pointed to by CR 24, and the I/O entries are at negative offsets. This arrangement corresponds to the layout of the 32-bit physical address space which places physical memory at the lower end of the space and memory mapped I/O devices at the upper end.

The design of the hash table and PDIR are such that later implementations can service TLB misses in hardware, with a reduction in the time spent servicing TLB misses. Control registers have been reserved to contain the hash table address and PDIR address.

Paging Management. One function of an operating system is to swap out pages that have not been accessed recently, to make room for pages being accessed that are still on disc. To help implement this, there is a reference bit for each page, within the PDIR entry, even though there is no hardware bit corresponding to it in the TLB. Instead, the entry is only allowed to be in the TLB if the reference bit is set. When the reference bit is cleared, the TLB entry is also purged by software. The next time there is a TLB miss, the miss handler will also set the reference bit in the PDIR. Thus, the operating system can clear the reference bit, and if the bit is still clear sometime later when it examines it again, it knows that the page has not been accessed in the meantime.

Each entry of the PDIR (and the TLB) has a dirty bit that tells whether the page has been modified since it was brought in from disc. When the page is first brought in, the dirty bit is clear. As long as only reads are done to the page, the bit will remain clear. However, the first time a program tries to store data to that page, the TLB causes a dirty bit update trap, which sets the bit to one in both the PDIR and the TLB. This provides information to the operating system so that it can avoid writing out unmodified pages, since the copy on disc is still valid.

Access Control

Access rights checking is based on the access rights and access ID fields in the TLB entry used to perform the translation. Access rights checking occurs with virtual address translation, unless disabled by the P flag in the PSW. There is no access control when using physical addressing.

Fields in the TLB entry for a particular page permit control of access to the page in three dimensions:

- Which of data read, data write, instruction execute, and the privilege level change function of the GATEWAY instruction are permitted (What)
- The privilege level at which the process must be executing (When)
- The process or group of processes allowed to access the page (Who).

These three dimensions are provided by two independent, simple mechanisms that combine to provide the required protection which can be evaluated in parallel to provide efficient access control. The combination is designed to support both conventional and virtual machine operating systems.

Access Rights. The first two dimensions of access control are provided using the access rights field of the TLB entry and the process privilege level. There are four levels (0 to 3), with 0 being the most privileged. Associated with each

HP Precision Architecture Caches and TLBs

An HP Precision processor typically interfaces to the memory system via the translation lookaside buffer (TLB) and the cache memory. The architecture is designed to allow simple, high-speed implementations by making the TLB and cache visible to software, and by placing constraints on software. The architecture also explicitly separates instruction and data caches, and instruction and data TLBs, although this is not a restriction on hardware implementations.

A *cache* is a small, high-speed memory that shortens main memory access times by keeping copies of the most recently accessed data. The cache is divided into blocks of data, and each block has an address tag that specifies the corresponding block of memory. When the processor accesses data, the block is copied from main memory into the cache. If the processor modifies the data (by doing stores), the copy in the cache will be more up-to-date than the copy in memory. The stale data in the memory at the place specified by the tag is eventually updated to correspond to the new data in the cache, using either the copy-back or the write-through update strategies.

Similarly, a *TLB* speeds up virtual address translations by acting as a cache for recent translations. When the processor accesses memory with a virtual address, the TLB checks for an entry with that virtual page number. If it is present, the corresponding physical page number is used to generate the physical address. Otherwise, there is a TLB miss, which must be serviced before the virtual memory access can be finished.

To allow the implementation of large, high-speed caches, the architecture disallows *address aliasing*, the capability of having two different virtual pages mapped to the same physical page. While address aliasing is of some use to software, it has severe impact on cache design. Normally, a portion of the address called the index is used to specify a block or a small group of blocks to be examined for a matching tag, instead of examining all blocks in the cache. Address aliasing precludes using the virtual page as part of the index. Otherwise, a virtual access could put data into the cache based on its index, and a later virtual access, using the other (aliased) address, would not find it in the cache because the index was different in the virtual page portion. The second access would then go to main memory, where it would get an inconsistent or stale copy.

Since HP Precision Architecture prohibits the use of address aliasing, the cache can use the virtual page portion of the address as part of the index, without causing the stale data problem described above. This allows the cache to be accessed in parallel with the TLB without restricting the size of the cache to that of the page size multiplied by the set-associativity of the cache organization.

If an object is to be referenced by both its virtual address and its corresponding physical address, software must flush the cache before accessing the data in the other mode. The one exception is if the physical and virtual addresses are identical, namely, the virtual address is in space zero and the offset within the space is the same as the physical address. Since the addresses are identical, the index chosen by the cache would be identical, thus avoiding the above stale data problem. This case is called equivalent mapping.

Uniprocessor Cache Management

HP Precision Architecture makes caches visible to software, and supports separate instruction and data caches when desirable for extra bandwidth, or a unified cache for reduced expense. It will also support very low-cost systems without caches, where

process is a current privilege level.

The access rights information is encoded in seven bits divided into three fields: type, first privilege level (PL1), and second privilege level (PL2) fields. The type field defines the use of the page (data or code) and, for privilege promotion instructions, the privilege level to which the process will be promoted. PL1 and PL2 define the privilege levels required for read, write, or execute access to the page. The meaning of the type field and the interpretation of PL1 and PL2 are given in Fig. 6. Read and write fields specify the least privileged levels allowed to read or write the page, respectively. Xleast gives the least privileged level allowed to execute instructions from that page. Xmost gives the most privileged level allowed to execute instructions from the page and is used to prevent privileged code from inadvertently branching onto a page that cannot be trusted.

The privilege level mechanism allows a process to have different access rights over time without the overhead of changing TLB entries when access changes or at process switch. Thus user programs (privilege level 3) can invoke the services of an operating system supervisor (privilege level 1) or kernel (privilege level 0) using an efficient procedure call and no interruption or process switch is required.

The entry to a more privileged routine can be implemented as a procedure call to a GATEWAY instruction that branches to the body of the routine. If a GATEWAY instruction is fetched from a proprietary code page, then when it executes it changes the privilege level to that specified by the low-order two bits of the type field for that page (if that level is more privileged than the current level). The GATEWAY instruction stores the caller's privilege level in the return address register so that it cannot be "forged" by the caller.

The architecture defines two trap conditions (higher and lower privilege transfer traps) that can be enabled to allow an operating system to intercept privilege level changes. These are provided to support languages that allow multiple processes to share a single stack with different access rights.

Access ID. A second field in the TLB entry, the 15-bit access ID, provides the third dimension of access control. It allows each process sharing memory to access different domains in memory without the overhead of changing fields in the TLB (and associated data in memory) on process switch.

Type	PL1	PL2	Use
0	Read	-	Read-Only Data Page
1	Read	Write	Normal Data Page
2	Read/Xleast	Xmost	Normal Code Page
3	Read/Xleast	Write/Xmost	Dynamic Code Page
4-7	Xleast	Xmost	Proprietary Code Page

Fig. 6. *Interpretation of access rights fields.*

the cache control instructions are treated as NOPs.

FLUSH DATA CACHE and FLUSH INSTRUCTION CACHE instructions remove a cache block and update memory if necessary. PURGE DATA CACHE removes a cache block without update. The latter is used only when the data can be destroyed, for example when the page is removed at the conclusion of a program.

The architecture puts the responsibility of uniprocessor cache consistency on software, based on the assumption that the software knows when special action is needed to ensure consistency. Software must take special action when it is changing a page's virtual address, when it is modifying the instruction stream (self-modifying code), and when it is performing I/O.

When the operating system changes a page's virtual address, it must flush the range of addresses for that page, to ensure that there are no blocks in the cache using the old virtual address.

If software stores into the instruction stream, the modification would occur in the data cache, while instructions are fetched out of the instruction cache. Rather than have the cache somehow figure out that software is doing this, software is required to flush the data from both the data cache (to update main memory) and the instruction cache (to force the next fetch to go to memory) after modification. Since self-modifying code is so infrequent, the extra time required is negligible.

From the standpoint of the cache, I/O is like another processor reading or modifying memory. If the I/O system is reading data from memory that is currently in the cache, it is reading a stale, out-of-date copy. Other architectures have solved this problem either by having I/O go through the cache, or by having all I/O transactions interrogate the cache to see whether it has a more up-to-date copy. This either uses up available cache bandwidth, depriving the processor, or lengthens the cache cycle time, slowing down the entire computer. HP Precision Architecture requires software to flush the address range involved in the I/O transfer before it occurs, so that the cache does not need to do any checking. The overhead of flushing for I/O is a very small amount and less than the impact on performance incurred by the other schemes.

HP Precision instructions include a nondivisible load and store zero instruction, LOAD AND CLEAR WORD, which is similar to the test and set operation in other architectures. This instruction reads a word from main memory, flushing the cache first if it is present, then clears the word in memory, in one indivisible operation. It is used to implement semaphores to synchonize access to data structures that are shared between the processors and the I/O modules, or for data structures that can be modified by two or more processes operating asynchronously.

Multiprocessor Cache Management

For HP Precision uniprocessors, software is responsible for cache consistency. For multiprocessors, however, hardware is responsible for cache consistency since the model presented to software is one in which all the processors share a single instruction cache, a single data cache, a single instruction TLB, and a single data TLB. This is because it may be difficult for software to recognize all data consistency situations in a multiprocessor and handle these situations efficiently for both uniprocessor and multiprocessor systems. Software is still responsible for maintaining consistency for I/O, for instruction modification, and for virtual address mapping.

In an actual multiprocessor system, each processor may have its own cache and TLB. To maintain the model of a single shared cache and TLB among processors, standard cache consistency methods are used. In addition, the explicit cache and TLB flush and purge instructions are broadcast to all processors, so that a flush instruction executed by one processor will affect all processor caches or TLBs in the system. The broadcast flushes and purges still do not affect I/O modules, allowing them to remain simple.

An access ID of zero defines a page with public access allowed, subject only to access rights checking. A nonzero access ID permits access to the corresponding page only when one of the four protection IDs in control registers matches the access ID.

The four protection IDs designate up to four groups of pages that are accessible to the currently executing process. Four are provided to facilitate the controlled transfer of information between logical environments. The low-order bit of each of the four protection IDs is the write disable (WD) bit. When the WD bit is set to 1, writing is disallowed for all privilege levels to the pages so protected. For example, the WD bit allows a single writer and multiple readers for a group of processes.

Privileged software needs a mechanism by which it can avoid performing, on behalf of a less privileged caller, actions not permitted the caller. This is provided by the PROBE instructions, which test the caller's ability to read or write a particular page of memory.

Functional Operations

The data transformation instructions provide all of the common arithmetic and logical functions. There are also several uncommon functions that provide building block instructions for complex operations and functions for efficient high-level language optimizations. The transformation instructions form a powerful resource for compilers to generate efficient code while defining an easily implemented hardware execution engine.

Each transformation instruction also specifies the conditional occurrence of either a skip or a trap, based on its opcode and the condition field. An immediate source can also be specified. The arithmetic/logical instructions are not completely orthogonal. Only those operations and options considered useful were defined.

Arithmetic Operations

Addition and subtraction instructions offer the widest flexibility in operand specification, condition formation, and testing. The two operands can come from two general registers, or from one general register and an 11-bit signed immediate. The SUBTRACT IMMEDIATE instruction is a reverse subtraction to allow subtraction of a variable from an immediate. Subtraction of an immediate from a variable is performed with an ADD IMMEDIATE instruction. The carry or borrow bit can be included in the addition or subtraction.

Software will be able to construct any often needed function in a single instruction. Since a conditional trap or an overflow trap can optionally be specified, many range violations and overflow checks required by high-level languages can be performed without extra instructions. For some checks an additional instruction might be needed, but generally the architecture provides for the optimization

of the high-frequency execution path.

Studies of large collections of programs show that integer multiply and divide operations are infrequently used. Furthermore, when multiply is used, one of the operands is usually a constant known at compile time. Hence, instead of implementing a general multiply or divide instruction, HP Precision Architecture implements multiply and divide primitives, which do not require additional execution hardware.

The SHIFT AND ADD instructions are used as building block multiply instructions. They specify a one, two, or three-bit shift of one of the source registers before adding it to the other source. By combining a short sequence of these instructions, multiplication by a constant can be done quickly. The SHIFT AND ADD instructions are performed by the basic execution engine, and share the use of the preshifter multiplexers in the ALU data path with the address calculation for the load and store instructions. These easily implemented multiply primitives are used effectively by the software for a variety of constructs.

Multiplication by the constants 3, 5, 9, and any power of 2 can be done in one instruction. Multiplication by other small constants can be performed in two or three instructions. When it is necessary to perform multiplication by a variable, a specialized subroutine breaks the multiplier into four-bit pieces and forms the complete product in an average of twenty instructions.[2]

Division by small constants is handled as special cases by the compilers, while for general cases, the DIVIDE STEP instruction implements a single-bit nonrestoring division operation. A specialized subroutine uses thirty-two of these instructions, in combination with SHIFT DOUBLE instructions, to produce the quotient and remainder.

The added hardware cost and potential increase in basic machine cycle time, coupled with infrequent use, ruled out the inclusion of division and multiplication in the basic instruction set. The architected assist instruction extensions include integer multiply and divide functions for applications requiring higher frequencies of multiplication and division.

Logical and Field Operations

Logical operations are fundamental instructions for data manipulation. OR, XOR, AND, and AND COMPLEMENT instructions provide a full range of logical operations. The AND COMPLEMENT instruction ANDs a register with the complement of a second register. This operation reduces the number of masks required for carrying out bit manipulation.

Boolean values are easily generated using the COMPARE AND CLEAR instructions. This instruction first assumes a Boolean value of false by always storing a zero in the target register, and specifies the negation of the desired Boolean condition for the conditional nullification of the following instruction. The following instruction, if not nullified, will set the target register to true. Other architectures often require branch instructions to implement an equivalent function.

The field manipulation instructions, like EXTRACT, DEPOSIT, SHIFT DOUBLE, and BRANCH ON BIT, are implemented by the shift-merge unit of the basic execution engine (Fig. 3).

An EXTRACT instruction takes a field from any portion of a word and creates a result with the field right-justified. The remainder of the target register is filled with zeros or sign-extended, supporting both logical and arithmetic right shifts as special cases.

A DEPOSIT instruction takes a right-justified field and puts it into any portion of the target word, thus merging the selected field with data in the rest of the word. DEPOSIT IMMEDIATE deposits a sign-extended five-bit immediate into the target register, which is perfect for setting or clearing a small number of bits in a register. ZERO AND DEPOSIT clears the remainder of the target, which is useful when the original target information is not wanted. DEPOSIT instructions can easily implement left shift operations and multiplications by a power of two.

Fig. 7 illustrates the movement of an arbitrary field, A, from general register x to another arbitrary field position in general register y, using a pair of extract and deposit instructions. General register z is used as a temporary register for this operation.

SHIFT DOUBLE instructions concatenate two registers, shift them 0 to 31 bits, and store the 32 rightmost bits into the target. If one of the source registers is general register zero, a left shift or right shift is performed. If both source registers are the same, a rotate operation is performed. SHIFT DOUBLE instructions are useful for unaligned byte moves or bit-block transfers, and for extracting data fields spanning word boundaries from packed records.

The fields for these operations are specified by position and length. The length is always an immediate in the instruction, but the position may be either an immediate or the contents of a control register called the shift amount register. This allows dynamically generated shift amounts. Unlike other architectures that specify the field position by encoding the leftmost bit in the field, HP Precision Architecture specifies the rightmost bit position. This was done to simplify the control logic for the shifter by making the number of bits of right shift depend only on field position, not on both position and length.

Unit Operations

HP Precision Architecture includes a set of five instructions designed to support the parallel processing of small units (digits, bytes, and half words) within a word. These instructions make use of the seven low-order PSW carry/borrow bits. They are included in the architecture primarily to facilitate string search (byte and half word units) and decimal arithmetic (digit units). The half word units support the processing of 16-bit international character sets.

The UNIT XOR and UNIT ADD COMPLEMENT instructions

Fig. 7. *Movement of an abritrary field using extract and deposit instructions.*

can be used to compare corresponding subunits of two words for equality or a less-than relationship. These operations are particularly useful for scanning for byte or halfword values a full word at a time.

Packed decimal numbers represent each decimal digit in a 4-bit field. When these numbers are to be added together, 6 must be added to each digit of one operand so that carries will propagate properly during binary addition. After addition, each result digit must have 6 subtracted from it unless the addition for that digit generated a carry. Also, when a repeated sequence of additions is to be performed, the bias must be restored to the result by adding 6 to each digit from which a carry was generated. These correction steps are performed by the DECIMAL CORRECT and INTERMEDIATE DECIMAL CORRECT instructions, respectively.

Assuming that the bias value and operands are in general registers, BCD additions and subtractions require three instructions to retire each 8-digit word.

Instruction Formats and Encoding

In HP Precision Architecture, all instructions have a fixed length of thirty-two bits, which is one word of memory. Time-critical functions are placed in fixed-position fields, so that they can proceed with minimal or no decoding. Since all instructions are word-aligned, an instruction never crosses a page boundary.

The addresses of the two general register source operands for the execution engine are placed in fixed-position fields (bits <6:10> and bits <11:15>), so that registers can be read before or during the decode phase of the instruction. If an immediate operand is required rather than a general register operand, the selection is done by a multiplexer in front of the appropriate port of the ALU or shift-merge unit.

In instructions with three register specifiers, the third register specifier is placed in the last five bits of the instruction, bits <27:31>. However, any registers to be used as source operands must be specified in the first two register specifier fields. A register used as the target register for a data transformation or data movement operation can be specified in any of the three register specifier fields. Decoding the address of a target register is not time critical, since the writing of a result occurs later than the reading of operands.

The space register specifier field is also placed in a fixed-position field, since it is also used to supply an operand for virtual memory addressing.

The major operation code field (opcode) is placed in a 6-bit fixed-position field. The operations are divided into subclasses, each subclass occupying one point in the code space of the major opcode. Each operation in a subclass occupies one point in its suboperation (subop) code space. The size of the subop field depends on the particular subclass of operations. The placement of the subop field is done to minimize the impact on the fixed fields of more time-critical operations. The encoding of the subop field is done to minimize decoding within a subclass. Often, bits in the subop field can be wired directly to control points in the particular portion of the processor implementing this subclass of instructions.

In the case of a subclass of operations with a relatively long immediate field in the instruction format, a subop field would take away bits from the long immediate field. So, each of these long-immediate instructions is assigned a point in the major opcode space. Examples are the load and store instructions with long displacements and the ALU instructions with long immediates.

Immediates embedded in an instruction are sometimes broken up into different fields so as not to impact the placement of fixed fields, and to minimize the multiplexing required for assembling immediates of different lengths.

Although immediates come in various sizes, their sign bit is always in a fixed position: the rightmost bit position of the immediate. This aspect of the instruction encoding enables immediate sign extension to proceed without lengthy decoding and selection from various bit positions, which would happen if the sign bit were placed in the customary leftmost position of the variable-length immediate fields.

Formats

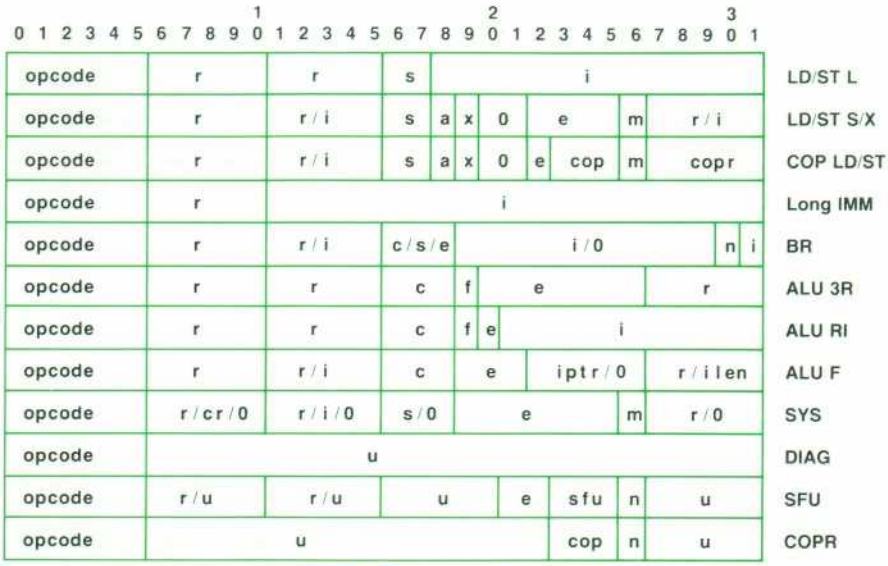
Fig. 8 shows the instruction formats used to encode all HP Precision Architecture instructions. The first three formats are for load and store instructions, followed by the instruction formats for long immediate instructions, branch instructions, three types of ALU instructions, system management instructions, the DIAGNOSE instruction, special function unit instructions, and coprocessor instructions.

The first format, for the long-displacement load and store instructions, essentially determined the positions of most of the major fixed-position fields like the opcode, the two source register specifier fields, and the space register specifier field. It also determined the right alignment of an immediate field, with the sign bit occupying the rightmost instruction bit. The ALU 3R format, for the basic three-register data transformation operations, determined the positions of other fixed-position fields like the third register specifier field, the condition field, and the falsify (condition negation) field.

The last three formats show the instruction extension capabilities in the architecture. One major opcode is reserved for the DIAGNOSE instruction, which can be used to define implementation dependent instructions. Only the major opcode of this instruction is defined. The next two are assist instruction formats, for the special function unit and coprocessor types of assists, respectively. For example, the floating-point coprocessor uses coprocessor unit identifier "zero" and encodes all its operations in the u fields. While DIAGNOSE instructions are not portable between implementations, the assist instructions are fully portable, with transparent software emulation of these instructions in the absence of hardware support.

Conclusion

HP Precision Architecture is frequently referred to as a reduced instruction set computer (RISC) architecture. Indeed, the execution model of the architecture is RISC-based, since it exhibits the features of single-cycle execu-



The Abbreviations for Field Names are:

r	: General Register Specifier	c	: Condition Specifier
s	: Space Register Specifier	f	: Falsify Condition c
i	: Immediate (or Displacement or Offset)	iptr	: Immediate Pointer
a	: Premodify versus Postmodify, or Index Shifted by Data Size	ilen	: Immediate Length
		cr	: Control Register
x	: Indexed (x=0) versus Short Displacement (x=1)	0	: Not Used (Set to Zeros)
cc	: Cache Hints	u	: Undefined (May Be Defined as Instruction Extension)
e	: Subop (Opcode Extension)		
m	: Modification Specifier	sfu	: Special Function Unit Identifier
n	: Nullification Specifier	cop	: Coprocessor Unit Identifier
		copr	: Coprocessor Register

Fig. 8. *HP Precision Architecture instruction formats.*

tion and register-based execution, where load and store instructions are the only instructions for accessing the memory system. The architecture also uses the RISC concept of cooperation between software and hardware to achieve simpler implementations with better overall performance.

HP Precision Architecture, however, goes beyond RISC in many ways, even in its execution model. For example, RISC machines emphasize reducing the number of instructions in the instruction set to simplify the implementation and improve execution time. Only the most frequently used, basic operations are encoded into instructions. However, frequency alone is not sufficient, since some instructions may occur frequently because of inefficient code generation, arbitrary software conventions, or an inefficient architecture.

In designing the next-generation architecture for Hewlett-Packard computers, the intrinsic functions needed in different computing environments like data base, computation intensive, real-time, network, program development, and artificial intelligence environments were determined. These intrinsic functions are supported efficiently in the architecture. Minimizing the actual number of instructions is not as important as choosing instructions that can be executed in a single cycle with relatively simple hardware. Complex, but necessary, operations that take more than one cycle to execute are broken down into more primitive operations, each operation to be executed in one instruction. If it is not practical to break these complex operations into more primitive operations, they are defined as assist instructions, by means of the architecture's instruction extension capabilities. If more than one useful operation can be executed in one cycle, HP Precision Architecture defines combined operations in a single instruction, resulting in a more efficient use of the execution resources and in improved code compaction.

HP Precision Architecture's execution model has other noteworthy features like its heavy use of maximal-length immediates as operands for the execution engine, and its efficient address modification mechanisms for the rapid access of data structures. The architecture also includes some uncommon functions for efficiently supporting the movement and manipulation of unaligned strings of bytes or bits, and primitives for the optimization of high-level language programs.

HP Precision Architecture has gone beyond RISC in its control flow model with its conditional branch optimization features, its ring-crossing branch instructions, its nullification features, its conditional trap feature, its debugging support, and its efficient interruption mechanisms.

The architecture's virtual memory addressing and protection mechanisms support a wide range of system needs, from the smallest controller to the largest multinetwork environment. Indeed, the HP Precision program was internally codenamed Spectrum, since its objective was to serve the full spectrum of HP customers' information processing needs.

In summary, HP Precision Architecture represents an evolution of the more successful ideas in past computer architectures, combined with support for the anticipated needs of future computer systems.

Acknowledgments

Many people made valuable contributions to HP Precision Architecture. They are too many to list here, but the architecture would not exist without their efforts. The original eight-person design team, which surveyed the ground and laid the foundation for what was to come, included the first three authors listed on the first page of this paper and William Worley, Allen Baum, Hans Jeans, Russell Kao, and Steve Muchnick.

References

1. J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, Vol. 36, no. 8, August 1985.
2. D.S. Coutant, C.L. Hammond, and J.W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, Vol. 37, no. 1, January 1986.

Authors

August 1986

4 Processor

William R. Bryg

With HP since 1979, Bill Bryg has been a logic designer for the HP 3000 Series 64 Computer and has worked on the architecture and logic for HP Precision Architecture and for the HP-UX system. His BS and MS degrees in electrical engineering were both awarded in 1979 by Stanford University. Born in Chicago, Illinois, he now lives in Saratoga, California. He's married and enjoys dancing, gardening, skiing, and reading science fiction.

Ruby Bei-Loh Lee

Ruby Lee is one of the original members of the HP Precision Architecture design team. With HP Laboratories since 1981, she worked on all aspects of the processor architecture, did the initial performance analysis, and designed the assist architecture. She also did the systems design, performance analysis, control implementation, and testability design for a VLSI microprocessor for the program. She is the holder of a BA degree from Cornell University and an MS degree in computer science and a PhD degree in electrical engineering from Stanford University (1980). She was also an assistant professor at Stanford. She has written several papers on parallel processors, performance analysis, computer architecture, and VLSI structures and testing and is named inventor on several patent applications related to HP Precision Architecture. Ruby and her husband and two children live in Cupertino, California.

Michael J. Mahon

Michael Mahon was born in Mt. Carmel, Illinois and studied physics at St. Louis University. After receiving his BS degree in 1963, he continued his studies at the California Institute of Technology (MS 1965). He has been with HP since 1981 and has contributed to the development of HP Precision Architecture, the HP-UX system, and various compilers. His other professional experience includes work on computer operating systems, interactive graphics, compilers, language design, and computer architecture. He is the author of two papers on real-time data collection and interactive graphics and is named inventor on three patents related to wafer-scale integration and an associative store. Michael lives in San Jose, California, is married, and has five daughters. He likes flying, photography, and microcomputers.

Jerome C. Huck

Jerome Huck has been with HP since 1983 and is the R&D section manager responsible for processor and assist architecture in the Information Technology Group. He first worked with hardware and software implementation teams at HP Labs on floating-point coprocessor definition. He also worked on the floating-point emulation package for HP-UX and MPE XL. Jerome attended Marquette University, receiving his BSEE degree in 1975. He continued his studies at Stanford University and completed work for his MSEE degree in 1977 and for his PhD degree in 1983. His professional interests include floating-point processors, parallel and pipelined processors, and optimizing compilers.

Terrence C. Miller

Born in New Rochelle, New York, Terrence Miller attended Yale University. His BS degree in engineering and applied science was awarded in 1969 and his PhD degree in computer science was awarded in 1978. He has been with HP Laboratories since 1979 and is currently the manager of the programming environments department. He was a member of the original HP Precision Architecture design team and was a project manager for the HP Labs work on the C compiler and code optimizer for the program. He is named coinventor on several patent applications related to HP Precision Architecture and is the author or coauthor of three papers on compiler design and code optimization. Before joining HP he was a lieutenant in the U.S. Navy and an assistant professor at the University of California at San Diego. Terrence lives in Menlo Park, California.

23 Input/Output System

David V. James

An alumnus of the Massachusetts Institute of Technology, Dave James earned BS and MS degrees in electrical engineering and computer science in 1973 and a PhD degree in electrical engineering in 1978. He has been with HP Laboratories since 1980

and has worked on speech signal processing, graphics and workstation hardware design, and the I/O system for HP Precision Architecture. His previous professional experience was in digital audio signal processing. He has written several papers on FFT quantization errors and digital VLSI circuits for music synthesis and is named inventor on five patent applications related to the HP Precision Architecture I/O system. His specialty is application of digital signal processing to high-quality computer image generation. A native Californian, Dave was born in Oakland and lives in Palo Alto. He's married and has two children. When not working on a house-building project, he enjoys swimming and running marathons.

Robert D. Odineal

Bob Odineal is a Stanford University graduate with a BS degree in biology and an MSEE degree, both received in 1982. With HP since 1981, he's an IC design engineer and project manager. He has worked on a number of ICs for HP Precision Architecture, including an I/O channel adapter, a bus converter, and a memory controller. A resident of Saratoga, California, he's on the board of directors for a theater organization and is on the board of trustees for his church. His outside interests include waterskiing, hang gliding, photography, and Shakespeare.

30 Performance Analysis

Stephen G. Burger

Stephen Burger has been with HP's Computer Systems Division since 1981. He has developed software for an I/O project and contributed to the design of the MPE XL system and I/O software definition for HP Precision Architecture. He is presently a project manager for MPE XL. Stephen completed his BS degree in computer science from the University of Utah in 1981.

Joseph A. Lukes

Tony Lukes first came to HP in 1965 from SCM Corporation, and worked on the HP 9100A Computing Calculator, leaving HP two years later. He then worked for the IBM Corporation on digital circuit design, logic design of large computer systems, design automation, file and memory systems, data base management systems, offload engines for data bases, and file systems. He rejoined HP in 1982 and managed performance analysis of HP Precision Architecture. He's now an R&D laboratory manager for data bases in the Information Technology Group. Tony studied electrical engineering at Oregon State University (BSEE 1961), at the University of California at Berkeley (MSEE 1963), and at Stanford University (PhD 1972). He's a specialist in performance analysis and data base management subsystems and is interested in artificial intelligence, distributed data bases, and operating systems. Tony is married and is the father of two daughters.

40 Simulator

Daniel J. Magenheimer

Dan Magenheimer has worked on the HP Precision Architecture project since its inception. He has contributed to the design of the instruction set, the simulator and remote debugger, the object code emulator/compiler, the HP-UX system linker, and millicode. He's now a project manager for software architecture at HP's Information Technology Group. Born in Milwaukee, Wisconsin, he attended the University of California at Santa Barbara, completing work for a BA degree in computer science in 1981. He earned his MSEE degree from Stanford University in 1985. Dan and his wife and daughter live in Union City, California. He sings in the choir at his church and enjoys playing volleyball.

CORRECTION

In printing the June 1986 issue, the photographs in Fig. 3 on page 22, Fig. 4 on page 23, and Fig. 5 on page 24 were reproduced without any gray tones, and therefore are not representative of the display quality of HP's Ultrasound Imaging System. Fig. 1 on page 45 is much closer to what the display really looks like.

HP Precision Architecture: The Input/Output System

A simple, uniform architecture satisfies the I/O needs of large and small systems, and provides flexibility for future enhancements.

by David V. James, Stephen G. Burger, and Robert D. Odineal

THE HP PRECISION I/O SYSTEM was defined to provide a flexible framework to leverage existing I/O card designs without restricting the capabilities of low-cost or high-performance I/O cards in the future. The HP Precision Architecture development program provided an opportunity to incorporate and achieve global objectives of scalability, leverageability, and flexibility in a corporate I/O strategy. These objectives have been met by basing the I/O system on the design strategies of simplicity and uniformity.

Scalability is provided by a unified family of compatible buses. A basic single-bus configuration can be extended to include higher-performance or lower-performance buses, or expanded to include additional buses of the same performance.

Leverageability requires interchangeable parts. Hardware interchangeability is achieved by using one physical component in systems having similar requirements for function and performance. Software interchangeability is achieved by using one version of I/O driver software for functionally equivalent hardware components that differ only in performance and capacity.

Flexibility is more than the use of leveraged components. A system is flexible when it is implemented to meet existing needs and is alterable to match the changing needs of the future with minimal perturbation of a customer's existing system. A flexible I/O system allows the existing I/O card designs to be leveraged for the initial product shipment, while also allowing the I/O system to be upgraded to support more demanding I/O requirements in the future (e.g., multiprocessors, shared peripherals, and memory mapped graphics). Flexibility is also provided by minimizing configuration restrictions in the I/O system.

Levels of Design

The definition process for the I/O system included rigorous documentation at all levels of the design. These design levels included the I/O architecture, the connect protocol, and multiple definitions of bus standards. The I/O architecture defines the types of modules that connect to an HP Precision bus (including processors, memory, and I/O) and defines the memory mapped registers used by other modules to control or observe the module's activity. This architectural interface is defined in sufficient detail to allow the hardware and software to be developed independently. HP Precision I/O Architecture includes the definition of simple instructions fetched from memory and executed by I/O modules with direct memory access (DMA) capabilities, but does not include the definition of instructions executed by the more general-purpose processor module.

The connect protocol defines the standard set of bus transactions used to communicate between modules defined by HP Precision I/O Architecture. This includes the definition of transaction functionality, transfer sizes, alignment restrictions, and returned status information. In addition to implementing the connect protocol, each HP Precision system bus definition includes the timing of signal transitions, voltage thresholds of transceivers, power requirements, and other physical parameters.

The HP Precision program provided a unique opportunity to upgrade all levels of the I/O system definition simultaneously. The method used to develop the system was top-down definition coupled to bottom-up verification.

The steps in top-down definition are architecture, protocol, standards, and design. The I/O architecture is defined around a model established to meet the objectives. The architectural concepts define the required connect protocol. The bus standards are defined based on that connect protocol, and the bus standards are used in the design of I/O cards. Fig. 1 illustrates the process.

The simultaneous activity in the architecture and design phases of the definition were coordinated to provide constant feedback between the intermediate levels. The initial designs revealed flaws or incompletely specified portions of the bus standards. These were corrected in the bus standards and the corrections were propagated up to the appropriate higher level. Feedback also occurred between the bus standards and the connect protocol, and between the connect protocol and the I/O architecture. This controlled feedback process provided the design evaluations required to update the initial drafts of the I/O architecture, connect protocol, and bus standards documents. These documents are the basis for the design of the system components, or modules.

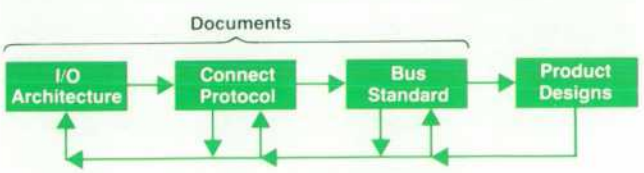


Fig. 1. *Feedback paths in the definition process for HP Precision I/O Architecture.*

Bus Options

HP Precision I/O Architecture is based on functional entities called modules. The minimal system consists of a processor, memory, and I/O modules attached to a single system bus, as shown in Fig. 2. The single-bus configuration is sufficient to support low-range and midrange products.

For high-end products, multiple buses are required, as shown in Fig. 3. The processor and memory are connected to a higher-performance HP Precision bus and the I/O modules are connected to other low-cost buses. In this example, I/O connections to a "foreign" bus and a "native" system bus are illustrated. The native system bus, or simply system bus, implements the HP Precision connect protocol; the foreign bus does not. The native and foreign buses are connected through a *bus adapter* module, and special software is required to support the connection. The bus adapter architecture allows I/O cards developed for other buses to be leveraged in the HP Precision system products, as discussed later.

Two native system buses can also be connected through a *bus converter* module. This connection is transparent to normal software operation.

Based on the destination address of a transaction, the bus converter forwards the transaction to remote modules attached to a physically separate bus. The bus converter is not involved in local transactions between modules attached to the same bus. Unless bus errors occur, the forwarding of a remote transaction is transparent to the module that originates the transaction. This allows the I/O driver software developed for a local module to be leveraged when the module is moved to a remote bus. Software changes are limited to the optional recovery of errors detected on the remote bus (the bus converter logs and isolates system bus errors).

The bus converter is implemented as a module pair; one module is attached to each of the two system buses. The module pair can be physically separated and connected with a high-speed link (e.g., fiber optics), as shown in Fig. 3. This separation is required when the buses cannot be physically adjacent because of mechanical packaging constraints or customer requirements to support remotely located peripherals. This would be the case for large I/O configurations, processor clusters, or remotely located graphics and data collection peripherals.

Module Addressing

When a system bus is initialized, each module initially responds to a 4K-byte "hard" physical address range. The module's 4K-byte address space is divided into 1024 32-bit I/O registers. Access to these I/O registers is provided by the read or write transactions defined by the connect protocol. For example, write transactions are used to reset the system or a card, interrupt the processor, and initiate I/O operations. The more common I/O registers, such as those used for module identification and initialization, are standardized to support autoconfiguration and simplify operating system software.

A 256K-byte address space, aligned to begin at a multiple of 256K bytes, is provided for each system bus; this is sufficient to support 64 modules. The physical properties of card connectors, backplanes, and transceivers normally limit the number of card slots on a bus to 16. Thus, to provide a complete set of 64 modules on a system bus, hardware designers would be required to implement four modules on each card. For example, a multifunction card might consist of a processor, memory, and two I/O modules. In general, not all cards have four modules and the bus address space is only partially used.

The initial address space allocated to memory and I/O modules is not generally sufficient to support normal module operation. For these modules, one of the registers in the initial address space is used for dynamically assigning an extended address space, as shown in Fig. 4. The extended address is always a power of two in size, and is aligned to a physical address that is a multiple of its size. To simplify configuration firmware and software, the extended address space can be assigned independently of the module's initial hard address space.

The initial 4K-byte address space of an I/O module maps to the supervisor element. Additional register sets, or I/O elements, are required to communicate directly with the attached devices. These I/O element registers are typically located in an extended address space, which is dynamically assigned by a writing to a supervisor element I/O register.

To simplify the I/O driver software, a single I/O element (register set) is allocated for each device to be controlled by the software. Multiple devices are supported through multiple I/O elements. The architecture provides the design freedom needed to achieve a good match between physical hardware implementation and logical software interfacing. For example, a disc controller implemented as a single physical device can interface to software through the address space of a single I/O element. A full-duplex terminal controller can be assigned two I/O elements, one for data input and one for data output. The software can thus service the inbound and outbound data streams independently. A terminal multiplexer with eight full-duplex ports can be implemented as 16 I/O elements, allowing software to perform independent I/O operations on each data stream.

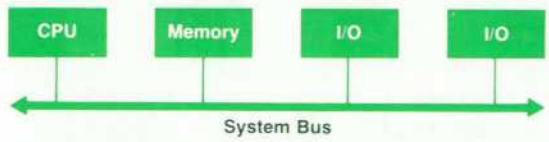


Fig. 2. *Small HP Precision system configuration (up to 64 modules).*

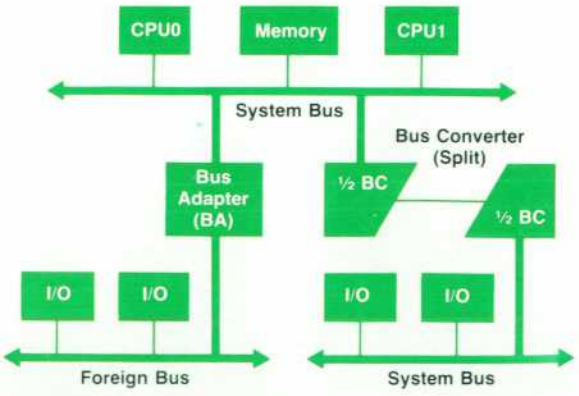


Fig. 3. *Large HP Precision system configuration.*

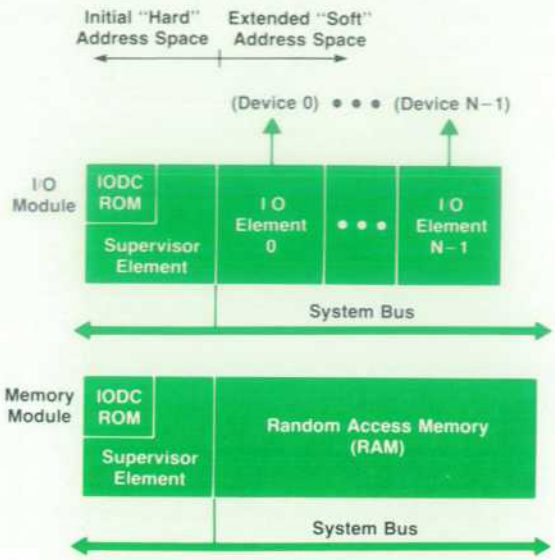


Fig. 4. *When the initial address space allocated to a memory module or an I/O module is too small, one of the registers in the initial address space can be used for dynamic assignment of an extended address space. A special ROM, IODC (I/O dependent code), supports autoconfiguration.*

During normal operation, an I/O device is controlled by accessing I/O element registers directly. When DMA or similar hardware on the I/O module is shared by multiple devices, the use of this shared resource is scheduled by the I/O module hardware, not the I/O driver software. This simplifies software and generally provides a more efficient mechanism for scheduling shared hardware resources. The I/O registers in the supervisor element are only used for module identification, initialization, and error recovery.

Two sizes of I/O elements are defined, 128 bytes and 4K bytes. The packed version (128 bytes) allows up to 16 I/O elements to be packed into a single 2K-byte page. In the unpacked 4K-byte version, two pages are provided for the support of privileged and unprivileged I/O registers. Unprivileged registers are accessible through both pages; privileged registers are accessible only through the lower-addressed page. The higher-addressed page can be mapped directly into the user's virtual address space without compromising system security. This allows many of the I/O element registers to be accessed directly, without the overhead of calling operating system software.

On a memory module, the extended address space maps to the module's RAM. Because the extended address space is automatically assigned, hardware switches are not required to configure memory addresses. This improves the reliability of the card, and eliminates service calls caused by improperly selected switch settings. After initial configuration, the supervisor element registers are read periodically to update the system's memory error log.

I/O Dependent Code

As illustrated in Fig. 4 for I/O and memory modules, each module contains card specific ROM called I/O dependent code, or IODC, which is accessible through standardized I/O registers. The content of the IODC is sufficient to identify the proper diagnostic and I/O driver software for the module. This is provided to support autoconfigurable operating system software. Operator intervention is not required to configure a new physical card.

System initialization, or boot, involves the execution of firmware code to initiate an I/O operation on one of the boot devices, such as a disc. To minimize updates of processor ROMs, this firmware is split between the processor and the I/O modules. The portion of the code shared by all I/O modules is located on the processor module. The primitive I/O drivers are provided by the I/O modules, and are called to initialize, test, and read data from the selected boot device. A stable HP Precision instruction set simplifies the support of IODC on I/O modules; new ROMs are not required for each upgrade of the processor hardware.

In addition to assisting system initialization, the IODC ROM is used to distribute module self-test code, and can be used to insulate standard I/O driver software from the implementation dependent features of module identification, configuration, and error recovery.

Address Space Allocation

HP Precision I/O Architecture uses a single 32-bit physical address space. When a physical module is accessed through a virtual address, the translation to a physical address is performed by the processor, and a physical address is used in the bus transaction. The physical address space is partitioned into two distinct spaces, the I/O address space and the memory address space, as shown in Fig. 5.

Address space is dynamically assigned. I/O addresses are assigned from the high end of the physical address space and memory addresses are assigned from the low end of the physical address space. This generates a compact address space assignment that minimizes the page table resources required to map virtual memory accesses.

Initially, only the broadcast address space is defined. A broadcast write transaction is used by a processor to initialize the 256K bytes of address space for its bus. Additional address space is assigned to other buses and extended module address spaces as required. The extended address space for I/O modules and memory modules is allocated from the available I/O address space and memory address space, respectively.

The words in the I/O address space correspond to I/O registers. Software references to these registers are processed differently from memory transactions; the load or store instruction triggers a bus transaction rather than a data cache access. The fixed partitioning of I/O and memory addresses simplifies the processor hardware required to identify the I/O register accesses, which bypass the data cache.

The dynamic allocation of the address space allows the address space to be assigned to additional buses or I/O elements as required to support the selected hardware configuration. Although the total physical address space is limited, the number and size of modules that can be supported are quite large, as shown in the table on the next page.

HP Precision Architecture Configuration Limitations (Approximate)

I/O Address Space

Total I/O Address Space	256M Bytes
System Buses (256K bytes each)	1024
Processor Modules (4K bytes each)	64K
Packed I/O Elements (128 bytes each)	2M

Memory Address Space

Total RAM Configured	3.75G Bytes

Connect Protocol

HP Precision I/O Architecture defines a standard software interface to module registers, independent of the physical bus standard. To implement this interface, and to support transparent forwarding of transactions through bus converters, a single connect protocol is defined for all system buses.

The connect protocol defines the required and optional transactions for all system buses. These transactions are initiated by a master, and invoke a response from one or more slaves. For a read transaction, data is transferred from the slave to the master. For a write transaction, data is transferred from the master to the slave. For a broadcast transaction, data is transferred from the master to all slaves.

Although the data transfer sizes are different for I/O and memory transactions, the basic format of the transactions is maintained, as shown in Fig. 6.

During the address phase, the address of the transaction is asserted on the bus. The bus address of the master (master ID) follows, and is sufficient to identify the module initiating the transaction. The master ID is transmitted while the slave is decoding its address, and generally has a minimal impact on system performance.

The master ID field is required to resolve potential deadlock conflicts when transactions are forwarded through bus converters. The field is also used by the smart-cache protocols to maintain consistent copies of data in the cache lines of processors attached to separate buses.

Only a small set of data transfer sizes is defined. The basic 4-byte and 16-byte sizes, and the larger optional transfer sizes (32, 64, ...) are all powers of two in size, and are described later in this article. Support of additional transfer size options in the memory address space would have increased the cost of memory modules, since they support all of the options.

At the conclusion of the transfer, status is transferred from the slave (or slaves) to the transaction master. If the slave detects an error (such as a double-bit memory error), an error condition is reported to the master, to prevent the use of corrupted data. For transactions that are correctly specified, but cannot be completed immediately, a busy status is returned and the transaction is automatically retried by the master. The busy status is required to avoid deadlocks in bus converters and is also used by the special transactions provided to maintain cache consistency in a multiprocessor environment.

Parity or alternative forms of error checking protect the transaction and slave addresses, master ID, data, and status signals. When control signals cannot be parity protected, their values and timing are designed to simplify the detection of faults through alternative mechanisms (bus timeouts, for example).

Separate transaction types are provided in the I/O and memory address spaces. This allows the data transfer size to be optimized for its intended use. The read and write transactions in the I/O address space are designed to access I/O registers, which are words (four bytes in size and alignment). Simple cost-sensitive cards may implement only the least-significant byte of each I/O register.

Transaction Types

Based on the requirements of processors and DMA-based I/O modules, transactions in the memory address space are optimized for burst data transfers. The CPUs use burst transfers to read or write cache lines. The DMA-based I/O modules use burst transfers to process buffered data packets efficiently. Nibble-mode and static column RAM technologies have minimized the cost of supporting the high-performance burst-mode transfers on memory modules.

All buses support the smallest (16-byte) memory address space transaction. This quad-word transfer typically uses 50% of the peak bus bandwidth. Larger burst transfers (e.g., 32 and 64 data bytes) are options, and are not defined for all system buses. If the transfer size is defined in the bus standard, it is supported on all modules responding as slaves in the memory address space, and is optionally used by the transaction masters (processors, the DMA-based I/O modules, and bus converters). In general, the low-cost DMA-based I/O module designs use 16-byte transfers, and

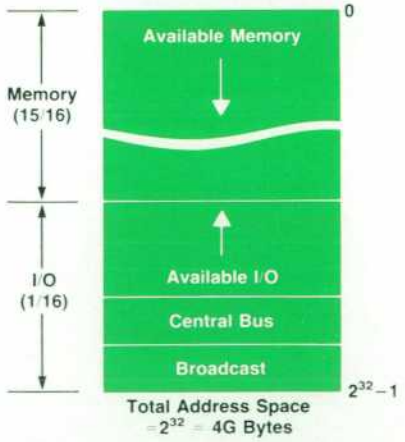


Fig. 5. *Partitioning of the physical address space.*

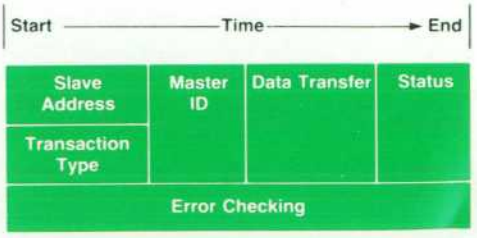


Fig. 6. *Basic format of I/O and memory transactions.*

high-performance DMA I/O module designs use the largest transfer size defined by the bus standard.

Two special types of transactions are defined by the connect protocol: broadcast and semaphore. To simplify their implementation, the architecture constrains the use of these nonstandard transactions. Broadcast transactions are only used to update I/O registers in the supervisor element. A write to a register offset in the broadcast portion of the I/O address space is equivalent to a sequential set of writes to the same register offset in each of the supervisor elements. Broadcast transactions to I/O elements or to the memory address space are not defined. These and other generalized uses were not required; supporting them would have needlessly complicated the module designs.

On many of the industry standard buses, semaphore operations are implemented by the processor, which requires the definition of an indivisible read and write transaction pair. Although this transaction pair has been used successfully in previous designs, it is difficult to forward through bus converters, and increases the complexity of high-bandwidth pipelined bus standards.

In the HP Precision connect protocol, the semaphore operation is implemented by memory module hardware, and has minimal impact on the complexity of bus standards. The semaphore transaction has a unique command code, but is otherwise identical to the quad-word read transaction defined in the memory address space. Like the read, the semaphore is recognized by the memory controller, and four words of data are returned from RAM. The semaphore transaction is distinguished by an important side effect—the first word at the quad address in RAM is cleared as the transaction completes. This is sufficient to implement the semaphores defined by the HP Precision instruction set.

Module Interrupts

In any computer, when a module such as an I/O device requires special service from a processor, the other tasks must be interrupted. The interruption mechanism enables the processor to respond quickly to high-priority interrupts while queuing and eventually servicing large numbers of low-priority interrupts, all with minimal performance overhead on the processor.

HP Precision I/O Architecture defines a very simple interrupt system that requires little special hardware and allows great flexibility in the processor's response to each interrupt. A key aspect of this interrupt system is the assignment of interrupt control to software. The architecture gives software the power to assign arbitrary interrupt priorities to all modules, direct each module's interrupts to any processor in the system, and selectively process or queue individual interrupts or priority levels.

When a module needs attention or service from a processor, the module communicates its need to the processor's external interrupt request register by using the same single-word, memory mapped write transaction used for all other intermodule communication. This ensures interrupt requests can be passed from any module in the system to any processor in the system without requiring specialized interrupt hardware. Also, since the connect protocol defines broadcast transactions to be a special case of single-word write transactions, a module can broadcast its interrupt request simultaneously to all processors in the system. Like the other transactions defined by the connect protocol, the interrupts propagate transparently through bus converters, and can be sent to a processor on any system bus.

Interrupts in HP Precision I/O Architecture differ from most other designs, which interlock the low-priority devices while the high-priority tasks are being executed. This interlock was discovered to be inefficient for uniprocessors and unreliable for multiprocessors. For uniprocessor configurations, this interlock would require that the interrupting module retry the write to the processor's interrupt register until it is completed successfully. The repeated transactions are an inefficient use of bus bandwidth. For a two-processor configuration, this interlock generates a potential hardware deadlock. For example, when two processors are executing separate high-priority tasks, and software on each processor sends a lower-priority interrupt to the other, both processors become deadlocked.

Interrupt Groups Hardware

HP Precision processor interrupts are based on hardware support of 32 interrupt groups. Software assigns one of these groups to an I/O element before an I/O operation is initiated. The value of the interrupt group is returned to the processor when an interrupt occurs. Software can independently disable any one or more of the interrupt groups, delaying their processing to a more convenient time. This is simpler and more flexible than architectures that set the interrupt priority in special-purpose hardware, restricting the ability of software to modify the order in which interrupts are processed.

Fig. 7 shows the functionality of the interrupt hardware that supports the interrupt groups. The interrupt system hardware consists of one register (the external interrupt message or EIM register) on each I/O element that generates interrupts, and two registers (the external interrupt enable mask or EIEM register and the external interrupt request or EIR register) on each processor. Before an I/O operation is initiated, software writes a 32-bit value to the EIM register

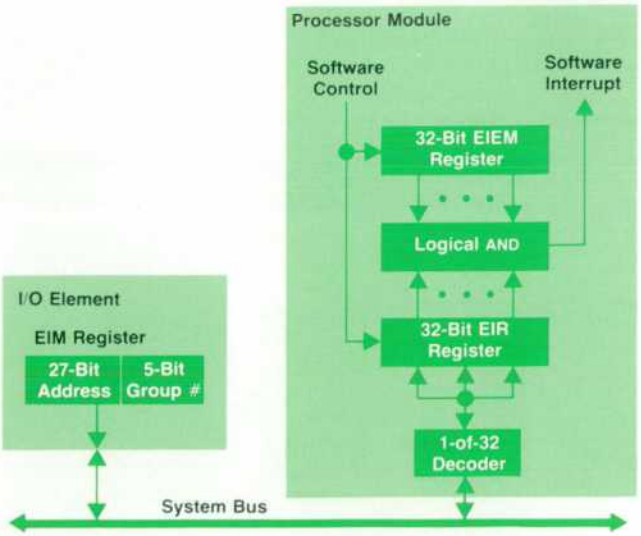


Fig. 7. *HP Precision interrupt hardware.*

of the I/O element. This value includes both the address of the processor to be interrupted and a five-bit encoding of the interrupt group assigned to the I/O element. When an element needs service, the single word in its EIM register provides both address and data for a single-word write transaction. The address determines the processor to be interrupted, and five bits of the data specify the interrupt group bit to be set in the processor's 32-bit EIR register. Each bit of the EIR register is continuously ANDed with the corresponding bit of the EIEM register, and if any bit of the result is true, the processor is interrupted. Software has complete control of the EIEM register to specify the interrupt groups that are recognized. As software services each interrupt, it clears the associated bit of the EIR register to prepare for the next interrupt. Software running on one processor is able to interrupt another processor simply by writing the appropriate data value to the processor's EIR register.

Although the architected interrupt system is fast and flexible, the information provided to software is minimal (only the interrupt group is known). In a system with many I/O elements, each of which must interrupt the processor to signal its completion of an assigned task, many of the interrupt group bits in the EIR register are shared. Unless an alternative mechanism is provided, the processor software would be burdened by the overhead of polling the I/O registers on I/O elements to resolve the source of interrupts that map to a shared interrupt group bit. A more efficient mechanism is the status chain feature, which is associated with DMA modules and is described below.

DMA Module Capabilities

Direct memory access, or DMA, is defined as an optional feature of an I/O element in HP Precision I/O Architecture. DMA is simply the transfer of data between the I/O element and system memory without intervention by a processor. The primary objective of DMA is to minimize the effort required of the processor to support I/O transfers. A high-performance DMA model allows the data to be transferred efficiently to system memory, minimizing the need to provide operating system specific data processing hardware or firmware on the I/O card.

A uniform DMA model is defined by the I/O architecture and supported by the connect protocol. The DMA modules access system memory using the same bus transactions that processors use. All DMA elements present the same memory mapped register interface to software, and software communication to initiate DMA activity uses the single-word memory mapped transactions defined for communication with other I/O registers. A uniform definition of the I/O registers in the DMA hardware interface simplifies the software interface, since many of the DMA software utilities can be shared by all of the DMA-based I/O software drivers.

To simplify the connect protocol and processor cache designs, all DMA transfers are performed directly to memory, and are not affected by the contents of the instruction or data caches. Shared utilities in the I/O driver software use the cache flush and purge instructions to maintain consistent copies of data in the processor caches and memory module RAM.

The DMA element activity is initiated by writes to two of the I/O registers on the DMA element. The first I/O register holds the address of the DMA command data structure in memory. The write to the second I/O register triggers the fetching of these DMA commands from memory for execution by the DMA element. The command data structures in memory are organized as a sequence of DMA requests, as shown in Fig. 8. Each DMA request is organized as a sequence of four-word data structures, or quads. The quads are aligned to an address that is a multiple of their size.

The data structures are based on linked lists of quads, rather than a less flexible set of sequential table entries. The four words of the quad include a pointer to the next quad in the chain, a command for the DMA element and two arguments for the command. The DMA element executes the successive commands in the chain of quads, automatically advancing from one quad to the next without processor intervention. The quad chains can be of arbitrary length, and can be dynamically extended as required to queue additional DMA requests. This is accomplished by changing the pointer in the final quad from its previously null value to the address of the first quad in the chain to be appended.

Each quad chain consists of one or more DMA requests. In general, each request corresponds to a separate call of the I/O driver software. For shared I/O devices, such as a file system disc, the I/O driver software is expected to append multiple requests for sequential processing by the DMA element. Each request is typically partitioned into

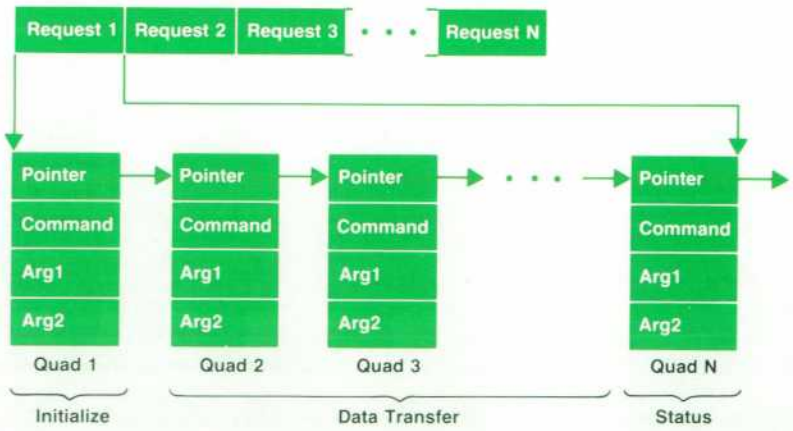


Fig. 8. *Data structures for DMA commands are organized as a sequence of four-word quads.*

three phases. The quads in the initialize phase provide the parameters for the following data transfer, such as the media address and length of the total disc transfer. The quads in the data transfer phase define the physical memory addresses involved in the DMA data transfer. The quad in the status phase is used to inform the processor when the request completes.

The chaining of quads in the data transfer phase allows the data to be transferred to noncontiguous ranges of physical addresses in the memory address space. This is useful in the virtual memory environment provided by HP Precision Architecture. An I/O request processed by the I/O driver software typically involves a transfer to or from a contiguous range of virtual addresses. Software converts the virtual address range into a set of noncontiguous physical addresses, and generates the corresponding chain of quads for use in the data transfer phase of an operation. Only data transfers to or from the memory address space are defined, and DMA input is aligned to a multiple of 64 bytes. Arbitrarily aligned DMA data transfers and transfers to the I/O address space are not required, and would have complicated the hardware design of DMA modules.

The status phase of a DMA request returns a summary of the DMA element's status to an entry stored in memory. The status information is sufficient for software to complete the processing of successful DMA requests. Unless errors occur, this summary is sufficient to allow a DMA element to continue processing additional requests without software intervention. The status phase generally consists of a single quad, called the link status quad. The link status quad instructs the DMA module to write its own status to a completion entry in system memory (the entry address is specified by Arg2). The completion entry is inserted into a linked list in memory (the value of Arg1 specifies the address of the list head), and a processor interrupt is optionally generated. The linked list of completion entries is called the completion list. The ordering of entries in the completion list is LIFO (last in, first out) to minimize the complexity of the hardware implementation. By software convention, each of the completion lists is assigned to a unique processor interrupt group.

Hardware updates four words of the completion entry, and software conventions define additional words of data in the entry, such as the address and arguments for the interrupt service routine. Before the DMA request is initiated, these parameters are saved in the space reserved for the completion entry. When an interrupt is received, software decodes the interrupt group to select the completion list to be processed. The data saved by software in the completion entry is used to dispatch quickly to the proper interrupt service routine.

Bus Adapters

Foreign buses are buses that do not conform to the specification of the HP Precision connect protocol. They can be connected to a system bus through a bus adapter. The bus adapter allows HP to preserve the investment in previously developed products when migrating to the HP Precision I/O system. Cards developed for the proprietary HP-CIO backplane are used in the first HP Precision products. By allowing the use of proven I/O technologies and systems, the bus adapter has accelerated the design cycle of the initial HP Precision Architecture implementations.

The first bus adapter to be developed for the HP Precision I/O system is the HP-CIO channel adapter (see Fig. 9). This adapter is fully compatible with all existing HP-CIO I/O cards, and with HP-CIO cards currently in development. Although the HP-CIO protocol differs from the HP Precision connect protocol in many ways, the bus adapter maps all of the necessary HP-CIO functions into the standard register interface through which it communicates with the HP Precision I/O system. In accordance with the HP-CIO protocol, the channel adapter serves as a central time-shared DMA controller on the HP-CIO bus. The adapter is the initiator of all HP-CIO bus transactions, and it is the arbitrator that manages the allocation of the HP-CIO bus bandwidth. As a bus adapter, the HP-CIO channel adapter provides data buffering and address generation as it transfers data between the I/O modules on the HP-CIO bus and the memory modules on other buses within the HP Precision I/O system. The adapter also translates interrupts and error messages into the protocol used by the HP Precision I/O system. By handling all normal DMA transfers and the majority of error conditions in complete autonomy, the HP-CIO channel adapter can greatly reduce the processor overhead required to operate the HP-CIO bus. Except in the rare error case that requires software intervention, the HP-CIO channel adapter appears to the HP Precision I/O system as a set of DMA I/O elements that conform to most of the specifications of HP Precision I/O Architecture.

In the future, the bus adapter module can also be used to support other foreign buses, such as VME. To support these cards, bus adapter hardware and I/O driver software are required to convert between the HP Precision I/O Architecture and connect protocol and the conventions of the foreign bus. For example, interrupts and DMA transfer protocols are usually different, and need to be converted. Although other foreign buses share many properties, their features require special considerations in the design of each bus adapter.

The leverage of foreign I/O card designs is not achieved without cost. Special bus adapter hardware is required, autoconfiguration capabilities are reduced, and software complexity is increased. Autoconfiguration features are generally not available on foreign buses. This typically limits the assignment of boot devices to preallocated slots on the bus, or requires a bus adapter ROM update to support

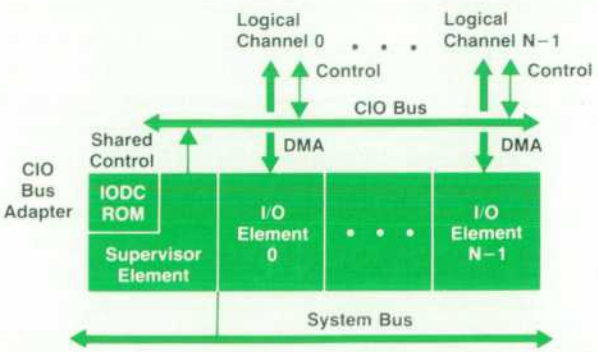


Fig. 9. *HP-CIO bus adapter with centralized DMA protocols.*

new boot devices. Access to foreign I/O cards is indirect, and involves software interactions with shared bus adapter resources to initiate an I/O operation, implement the DMA transfer to memory, or convert between interrupt protocols. This overhead increases the complexity of I/O driver software.

I/O System Summary

By adhering to the strategies of simplicity and uniformity, many benefits were realized.

Simplicity is illustrated by the alignment of addresses and address ranges, the minimal number of transactions defined in a single connect protocol, and the implementation of processor interrupts (the EIR and EIEM registers). Uniformity is illustrated by the transfer of interrupts between modules (an existing word write transaction is used), the definition of standard module I/O registers for identification and configuration of modules (including processors and memory), and the use of a single connect protocol for all bus standards.

The verification of the architecture through actual designs has shown the benefits of meeting the original objectives. Scalability is achieved through simplicity, and the architecture makes things "as simple as possible, but not simpler." "Not simpler" means that concepts and components are designed to meet the global objectives, rather than only the needs of a local design center. Components in the system are interchangeable, so the cost of developing them is amortized by their use on many different systems.

The biggest benefit HP's customers will see comes from the flexibility and the identical support of common components. Identical support for common components provides transparent migration to faster components, and more or faster buses. This migration can be accomplished with minimal perturbation of the customer's software and/or working environment.

Acknowledgments

As one of the initial designers for the I/O architecture, Mike Fremont provided invaluable guidance in the development of the initial drafts of the I/O architecture document. Major proposals and feedback on the initial versions of the bus standards were provided by Dave Fotland and Steve Chorak. We would like to thank Bill Worley and Michael Mahon, who provided the initial directions, resources, and design freedom required to develop a new I/O system.

This is only a small portion of the contributors to the I/O system design. Many hardware and software engineers participated in the process.

HP Precision Architecture Performance Analysis

Performance analysis was crucial to instruction set selection, CPU design, MIPS determination, and system performance measurement.

by Joseph A. Lukes

HEWLETT-PACKARD PRECISION ARCHITECTURE is a key component in Hewlett-Packard's computer strategy for systems well into the next decade. This article is intended to be a brief overview of the contributions of a collection of people from HP's performance evaluation community in the evolution of this strategy. It describes the role of these performance groups in the design and measurement of the architecture, and in the CPU design and systems measurement techniques that have led to the computer systems based on this architecture. Presentation of measured performance data will not be done in this article, but will be left to later papers in this and other journals.

Selection of the Instruction Set

The creation of HP Precision Architecture combined the expertise of highly experienced specialists in computer hardware design, compilers, operating systems, architecture, and performance analysis. The architecture team had investigated a number of papers on reduced instruction set computers[1] and the general conclusion was that a reduced instruction set computer was a feasible vehicle with which to migrate HP from its HP 1000, HP 3000, and HP 9000 Computers to a common architecture. The purpose of this section is to describe the efforts of the HP Laboratories performance analysis team in creating the data used to select the instruction set of the new architecture.

A team of performance analysts was chartered to extend the studies described in reference 2. To achieve these objectives, an Amdahl V6 computer was acquired and an interpretive instruction tracer program similar to that described in reference 2 was created. This program operated

Distribution of IBM 370 Instructions by Frequency

Instruction	Language COBOL	Fortran	Pascal
Branch	24.2%	18.0%	18.4%
Logical Operation	14.6	8.1	9.9
Load/Store	40.2	48.7	54.0
Storage-Storage Move	12.4	2.1	3.8
Integer Math	6.4	11.0	7.0
Floating-Point Math	0.0	11.9	6.8
Decimal Math	1.6	0.0	0.0
Other	0.6	0.2	0.1

Fig. 1. *An example of the type of data gathered to aid HP Precision Architecture instruction set selection.*

under the IBM VM/CMS operating system and gathered raw data from a variety of benchmark programs run on the V6 by interpreting and simulating each instruction. Specifically, we gathered the following data:

- Instructions executed and sequence in which executed
- Virtual address of instructions
- Virtual address of each operand
- Identification of registers used by the operation
- Contents of each operand for certain operations.

This basic data allowed us to derive valuable statistics, among which the following were of greatest value:

- Dynamic frequency of instruction occurrences
- Address traces for instruction sequences and for data referenced by these instructions
- Characteristics of operations such as the number of characters used in a move operation, operand values in arithmetic operations, distances branched, etc.
- Frequency of operation pairs.

Fig.1 illustrates the type of data gathered. Here the distribution of classes of instructions for COBOL, Fortran, and Pascal benchmarks is shown. Fig. 2 shows the distribution of time spent in these benchmarks per instruction class. Note that the bulk of the operations are simple loads, stores, and branches. Other operations occur relatively infrequently but can take a much larger amount of time. By distribution in time, floating-point operations for Fortran and Pascal, and storage-to-storage move operations for COBOL are important instructions. (Storage-to-storage moves can be simulated by a sequence of load/store instructions).

Most of the programs we tested exhibited these characteristics. The bulk of operations, both in frequency of use and time, are simple and dominated by the load, store, and branch operations. Since simple operations appear to dominate the frequency of instructions in a computer, the concept of cycle-per-instruction architectures has arisen.

Distribution of IBM 370 Instructions by Time

Instruction	Language COBOL	Fortran	Pascal
Branch	15.3%	21.4%	18.4%
Logical Operations	5.6	4.3	9.9
Load/Store	15.5	32.7	54.0
Storage-Storage Move	52.4	3.0	3.8
Integer Math	2.3	9.3	7.0
Floating-Point Math	0.0	27.4	6.8
Decimal Math	5.9	0.0	0.0
Other	3.0	1.9	0.1

Fig. 2. *More data gathered to support instruction set selection. Load, store, and branch instructions dominate.*

Such information is just beginning to appear in computer science literature.[1] Reference 3 points out that the fairly complex instruction sets of most computers are really not as helpful to the compiler writer as might be thought. Instructions such as the IBM 370 MVC (move character) and MVCL (move character long) are examples of instructions that might profitably have been left to a simple set of load and store operations. These instructions move any number of contiguous bytes (from one to sixteen million). However, Fig. 3, derived from our benchmark studies, shows that storage-to-storage moves really only move a small quantity of data. Reference 2 found the same central truth. Why bother with really sophisticated movers of characters like MVC and MVCL when a simple load/store combination in a small loop can outperform the more sophisticated move instructions?

Another interesting observation made from looking at the instruction mixes of a variety of benchmarks is that the typical mix does not seem to be much a function of the type of work that is being done. For example, technical work (such as large Fortran simulations or CAD/CAM) and commercial work (such as old master in, new master out or data base work) show the same characteristics: loads,

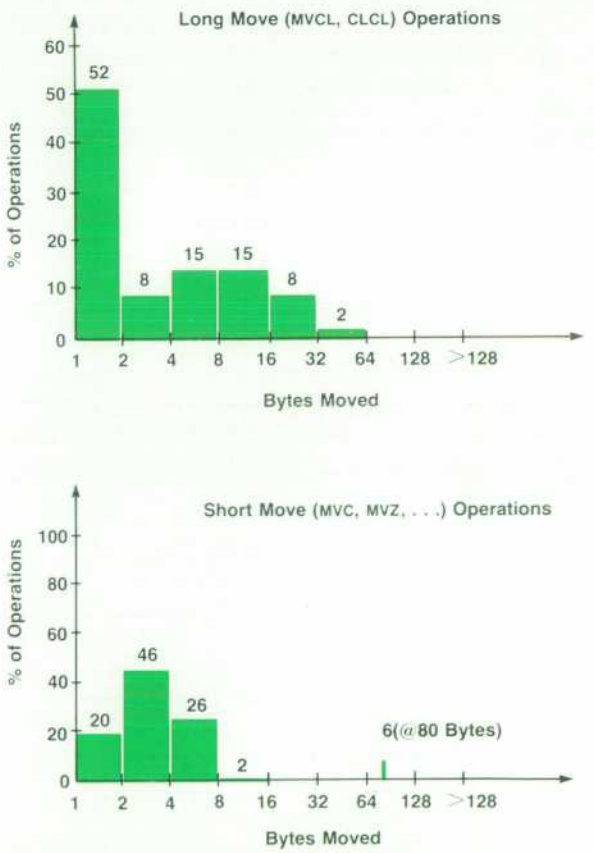


Fig. 3. *Storage-to-storage moves were found to move only a few bytes in most cases.*

stores and branches make up the bulk of operations.

Fig. 4 shows the instruction mix from an HP 3000 Computer during a peak period (predominantly data base intensive). Compare this with Fig. 2 and you will find that, other than the extensive use of floating-point operations in scientific work, they are very similar. One myth that did not seem to be borne out by the benchmark work that we did is that commercial (COBOL) jobstreams require a large amount of decimal arithmetic. We cannot find any evidence to back this assertion.

Another rather nonintuitive result of the early benchmark measurements is that a specialized integer multiply/divide coprocessor, unlike the results for floating-point, is probably not worth the extra expense. Fig. 5 shows the results of the measurement of a large Fortran program's use of multiply operations and the associated operand value distributions. At least one operand in the vast majority of cases is small (less than 500), making the techniques mentioned in reference 3 quite feasible with a net improvement in performance.

As a result of these studies and a variety of others, the compiler, architecture, hardware, and operating system teams established and refined the architecture to a set of instructions for HP Precision Architecture. The architecture, since it was based on measurements of a large sample of workloads, evolved from a simple RISC machine to the far more sophisticated operation set and computer organization outlined in reference 1. In summary, the conclusions drawn by the team selecting the instructions for HP Precision Architecture were:

- Simple instructions are most of what is executed in a wide variety of work.
- There are complex instructions that occur frequently enough (e.g., floating-point) to justify a special set of hardware to execute them. In HP Precision Architecture CPUs these are known as coprocessors or special functional units.[4]
- Load/store (move a 32-bit word) architectures make sense since they permit high-speed general registers to be used effectively as the first level of the storage hierarchy.
- Simulate complex but infrequent operations so that the underlying instruction set can be as simple as practicable.

The next section describes the efforts involved in selecting the parameters associated with the family of central processing units based on HP Precision Architecture.

Distribution of HP 3000 Instructions by Time

Instruction	Percent of Work
Branch	19.9%
Logic Operations	18.9
Load/Store	45.0
Storage-Storage Move	4.9
Integer Math	8.7
Floating-Point Math	0.0
Decimal Math	0.0
Other	2.9

Fig. 4. *Instruction mix for an HP 3000 Computer during a peak period.*

HP Precision Architecture Computers

The previous section outlined how a set of instructions was chosen, each of which, with few exceptions, executes in one major cycle of the central processing unit's clock. For example, a central processing unit (CPU) with a 10-MHz clock would be a 10-million-instruction-per-second (MIPS) processor with a cycle-per-instruction architecture. This, of course, assumes no delays as a result of cache or TLB (translation lookaside buffer) misses. Complex instruction set computers (CISC), on the other hand, sacrifice cycles of the CPU to execute more functionally complex instructions sets, generally through the aid of microcode.

Before describing how the CPU family associated with HP Precision Architecture was designed, a few words need to be said about computer system performance. A very popular measure of the power of a computer system is to specify the number of MIPS (millions of instructions per second) that the system's central processing unit(s) can execute. This measure is an estimate of the capacity of the CPU to execute the work asked of it. The higher the MIPS value, the faster the work is done by the CPU. However, computer systems are not just CPUs. Indeed, they consist of peripherals, interconnections, main storage, applications, operating systems, data communications subsystems, data base subsystems, compilers, and an entire set of policies for scheduling and billing that affect the performance of the computer system. Because of this plethora of variables in the equation that determines the performance of a given computer system, people have tended to concentrate on the relative simplicity of MIPS.

The customer who purchases a computer system is generally not really interested in the capacity of any one component of that system, such as the CPU, to do work. The customer is concerned with the response time with which work is completed, the throughput in jobs per unit of time, the number of active terminals connected to the system,

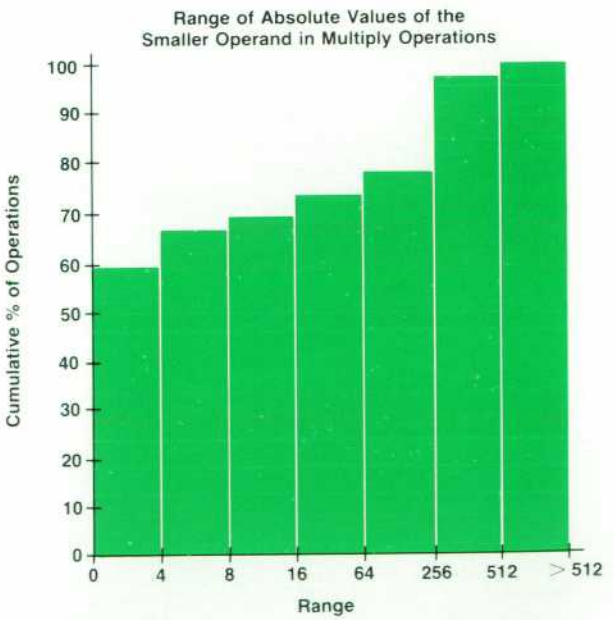


Fig. 5. *Distribution of operand values for multiply operations in a large Fortran program.*

and so on.

The next section will describe how we extended the metrics used to evaluate the HP Precision Architecture CPU family to systems performance. The rest of this section will concentrate on how we determined the MIPS performance of the first members of the HP Precision Architecture CPU family.

In creating a new architecture, the engineers and scientists charged with its development are faced with a large problem—how do you measure the effects of the architecture on MIPS, system throughput, response time, working set size, etc., when you do not have a CPU built from the architecture? To be sure, one can prototype the architecture, but the cost becomes prohibitive for all but a small number of prototypes. What follows in this section is a description of the hierarchical approach taken to develop the HP Precision Architecture processor family with minimum possible cost and maximum performance.

Let us first examine the MIPS value. It is relatively easy to calculate:

$$\text{MIPS} = [(\text{cycles per instruction}) \times (\text{CPU cycle time})]^{-1}$$

The CPU cycle time is the period of the major clock in the CPU. This value varies from 100 to 200 nanoseconds for transistor-transistor (TTL) logic, and from 20 to 80 ns for emitter-coupled (ECL) or HP NMOS-III logic. The MIPS capacity of a CPU can be increased by reducing the CPU clock time by means of new technologies. Many examples of this trend are seen in the current offerings of many computer vendors, including HP.

The other variable in the MIPS equation is the number of cycles per CPU instruction. The lower this number, the higher the MIPS value will be. As obvious as this seems, there is still raging controversy over the efficacy of the cycle-per-instruction architectures (i.e., RISC architectures), since critics of the reduced-complexity approach claim that numerous things can happen that tend to lower systems performance when one attempts to reduce the cycles per instruction (CPI) to one. It is not the purpose of this paper to argue either side. All of our work at HP so far, however, has pointed out that the HP Precision Architecture does not appear to limit the ability to make very high-MIPS computers from existing technologies through reductions in the CPI (cycles per instruction) and that such CPUs are capable of offering systems performance comparable with CISC machines of the same MIPS rating. Details of this work will accompany specific product announcements.

Simulator and Prototype

An earlier section described in part how the instruction set was chosen. At first, a proposed set of instructions was chosen from the collection of written data on cycle-per-instruction architectures. Then experiments were run on instruction mixes derived from other architectures that could be measured (i.e., the IBM 370 set on the Amdahl V6 and the HP 3000). Finally, analyses were done to select the instruction and register sets.

Since analysis alone could not take into account the effects of various design alternatives of the architecture and the CPUs designed to implement it, a simulator was written (see article, page 40). The simulator extended our ability to evaluate design trade-offs by allowing the user to program simple kernel programs, either by hand or through the use of a portable C compiler. These kernels were chosen for their lack of I/O (no operating system existed) and for their simplicity of compilation (only primitive compilers existed). A number of invaluable experiments were run on the simulator and continue to be run on it even today.

However, a simulator is relatively slow and expensive to use and the number of experiments involved in choosing the parameters of the CPU family became too much for the simulator alone. Another problem with simulators is that they do not convince the skeptical that a revolutionary new architecture can actually be implemented in existing technologies such as CMOS, TTL, ECL, or NMOS. As a consequence, a prototype HP Precision Architecture CPU was built. It was named the LESS (low-end Spectrum system) and, although very simple compared to the products recently announced, it was a fully functioning HP Precision Architecture CPU that achieved about 0.8 MIPS.

The LESS prototype gave software developers very early access to the architecture, and served as a vehicle for experimentation for the architecture, hardware, compiler, operating system, and performance teams. An interesting and useful tool that came out of the LESS prototype was an analyzer board that could be connected to the HP 64000 Logic Development System (more about this later).

As useful as the simulator and the LESS prototype were, there were parameters of the CPU designs that these techniques could not determine without untoward expense and time. Only very simple jobs could be run through the simulator and prototype since there were no operating systems or product-level compilers available. Consequently, the technique used in selecting the instruction set for HP Precision Architecture was used again, that is, traces on the Amdahl V6 and on HP 9000 and HP 3000 Computers. The instruction mixes for these computer systems were measured and used to simplify the CPU designs for minimum cycles per instruction. In addition, address traces were used to generate families of cache and translation lookaside buffer (TLB) statistics.[5] These measurements were then used to calculate the cycles per instruction for a proposed CPU design.

The cycles per instruction (CPI) value is, in simple terms, a function of the instruction mix, the parameters of the cache and TLB, and the CPU design:

$$\text{CPI} = \text{basic instruction time} + f_1(\text{cache,TLB}) + f_2(\text{interlocks})$$

where the basic instruction time is 1 cycle for most HP Precision Architecture instructions, f_1(cache,TLB) is the contribution to CPI of cache and TLB misses, and f_2(interlocks) is the contribution to CPI of the CPU design. CISC machines tend to have basic instruction times of 4 to 10 cycles for the average instruction. The cache and TLB penalties and interlock penalties are not really affected by the architecture to any great extent.

MIPS Model

We have developed a relatively simple model of the MIPS

performance of HP Precision Architecture CPU implementations based upon the mixture of instructions and miss rates of the different cache and TLB sizes and organizations. Fig. 6 illustrates the instruction mix and Fig. 7 the cache and TLB curves for one benchmark measured on the Amdahl V6. Before fully functioning HP Precision Architecture systems were available, such curves were used to design the CPUs.

For example, let the cache and TLB designs be such that the cost of a cache miss is 20 cycles and that of a TLB miss is 100 cycles. Assume that we are calculating the MIPS of a processor whose basic instruction times are 10 cycles for floating-point operations and 1 cycle for all other operations. Moreover, the pipeline design dictates a load/use penalty of 1 cycle per occurrence (here a load/use pair consists of a load followed by either a load or a store operation).

Using the workload characteristics of the Fortran program depicted in Fig. 6, the basic instruction time is $10(0.126) + 1(1-0.126)$ or 2.13 cycles per instruction for instructions alone.

From Fig. 7, the cache and TLB misses, assuming the memory system designs depicted in Fig. 7, are

$$\text{Cache miss rate} = 3.5\% \text{ (for an 8K-byte cache)}$$

$$\text{TLB miss rate} = 0.2\% \text{ (for a 512-entry TLB)}$$

Consequently, the cache contribution to f_1 (cache, TLB) is $(0.035)(1+0.348+0.154)(20)$, where the first factor is the miss ratio of this particular cache, the second is the number of instruction and data references per instruction (one for the instruction itself and a probability of 0.348 of the instruction's being a load or 0.154 of its being a store), and the third factor is the penalty of a miss, or 20 cycles. Thus the contribution of cache misses to the number of cycles per instruction is 1.05 cycles per instruction.

In like manner, the contribution to the CPI of the TLB misses is 0.3 cycles per instruction.

Finally, for this very simple model of the components of MIPS, the contribution of interlocks to the CPI consists of the above-mentioned load/use interlock, (0.06)(1 cycle per instruction), plus the penalty paid for no-op (no operation) instructions,[4] or (0.02)(1 cycle per instruction), for a total contribution of f_2 (interlocks) = 0.08.

The value of CPI, the cycles per instruction for this machine design and this benchmark, is CPI = $2.13+1.05+0.30+0.08 = 3.56$ cycles per instruction. Note that this workload has a very high-floating point content, so the CPI value is larger than for most workloads.

Instruction	% Occurrence
Load	34.8
Store	15.4
Floating-Point	12.6
Load/Use	6.0
Branch	12.3
No-Op	2.0

Fig. 6. *Instruction mix for a large Fortran benchmark run on the Amdahl V6 with HP Precision instructions.*

The MIPS value for a 100-ns clock implementation of this design and benchmark would be (100 ns per cycle × 3.56 cycles per instruction)$^{-1}$ = 2.81 million instructions per second.

If one could replace the floating-point operations with integer operations, and if each integer operation took 1 cycle per instruction, the MIPS value of this design would be 4.10 million instructions per second, since the CPI in this case would be 2.44 cycles per instruction instead of 3.56.

The ideal RISC-design processor would have a MIPS rating of 10 or more, assuming one cycle or less per instruction.

Measurements on Actual Processors

Today we base our instruction mixes, cache and TLB curves, and other CPU parameters on measurements made on the HP 3000 Series 930 and HP 9000 Model 840 processors. Figs. 8 and 9 are examples of these measurements. Future HP Precision Architecture machines are being designed with this data. The logic analyzer board mentioned above and the HP 64000 Logic Development System have been invaluable tools in debugging, analyzing, and tuning the new HP 3000 and HP 9000 processors. Fig. 10 shows a curve of MIPS versus time for the HP 9000 Model 840 derived using the logic analyzer board. Figs. 8 and 9 were also measured by the analyzer board.

In practice, the MIPS performance of a CPU varies with time and workload. HP has used heavily loaded values, as

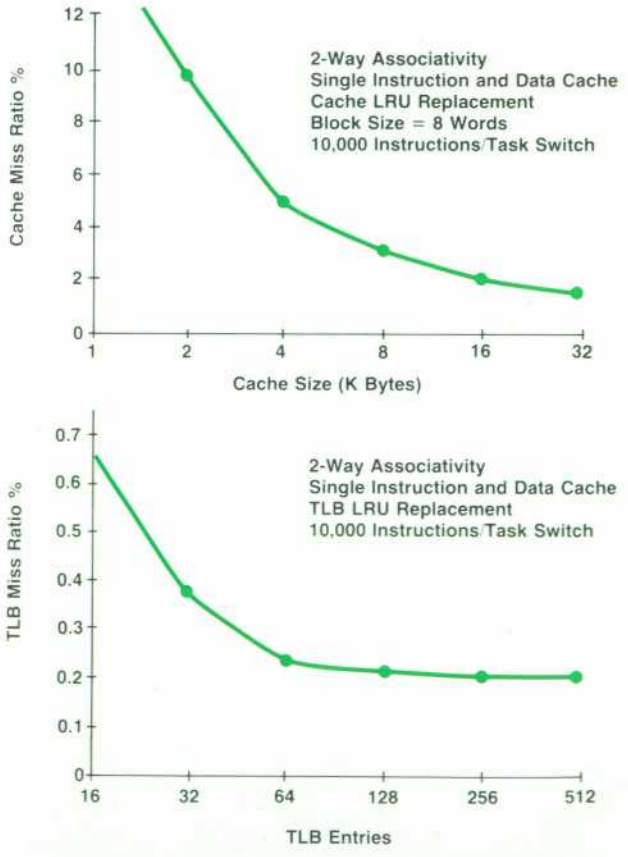


Fig. 7. *Cache and TLB miss ratios for the benchmark of Fig. 6.*

measured with the analyzer, for the specifications for the new HP 3000 and HP 9000 CPUs. However, a computer system consists of far more than the central processing unit. The next section will describe how the entire computer system is tracked through its design with extensions of the techniques used in designing the architecture and processors.

Systems Performance

The previous section emphasized the estimation and op-

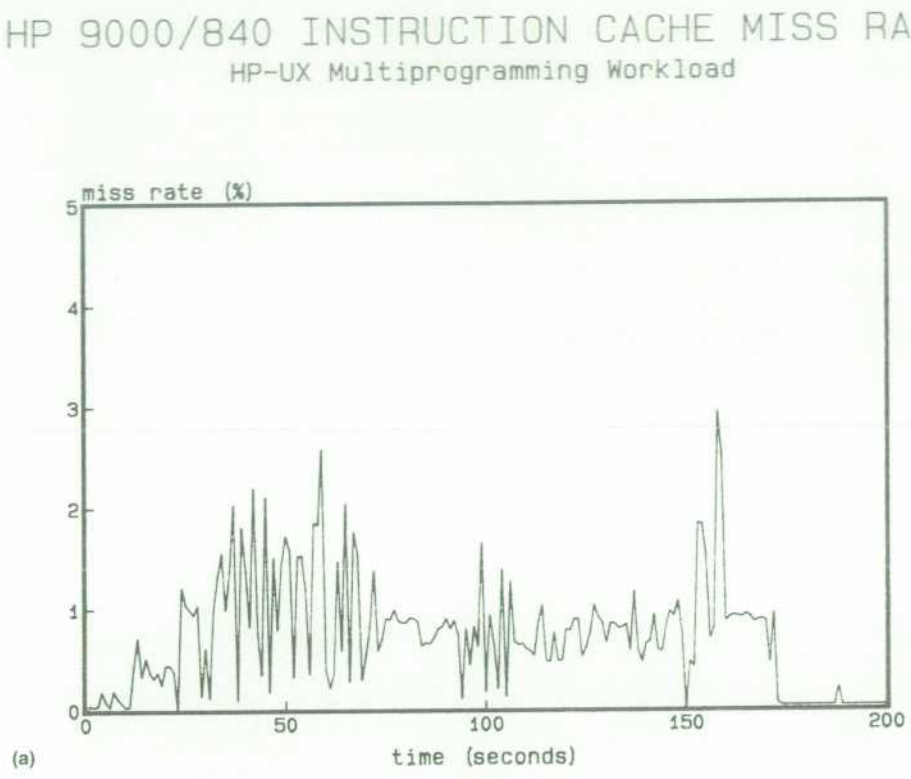
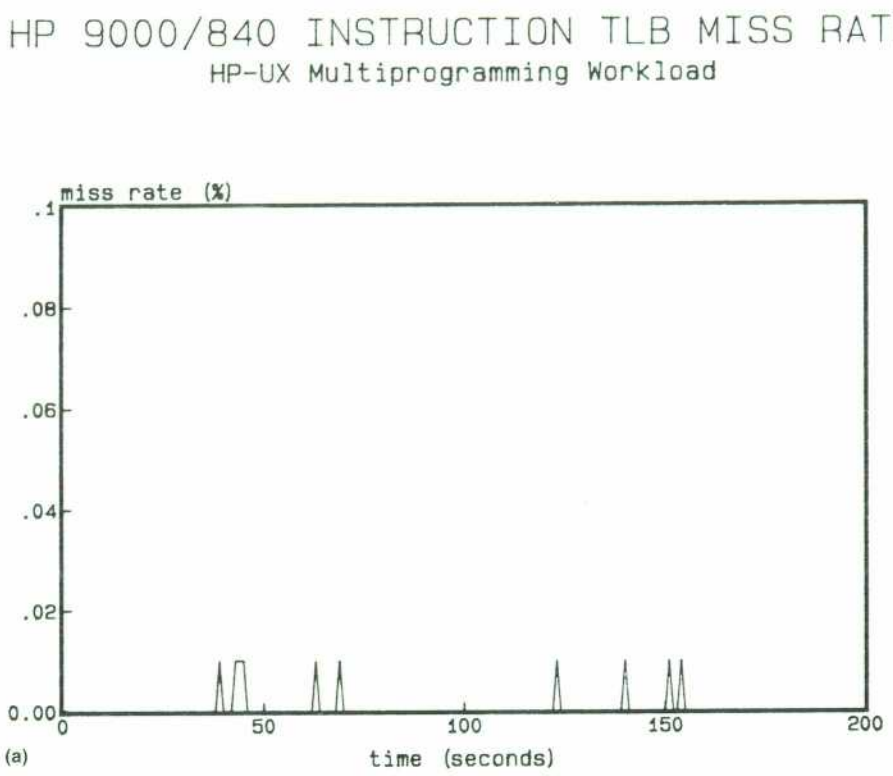


Fig. 8. *Example of a cache miss rate measurement for the HP 9000 Model 840, an HP Precision processor. (a) Instructions. (b) Data.*

timization of the CPU capacity as measured in millions of instructions executed per second. As pointed out in that section, an increase in MIPS does not necessarily guarantee a similar increase in system performance measures, such as an increase in system throughput or a decrease in user response time. Examples of reasons why system performance may not increase proportionally to MIPS are serialization on software queues created to guarantee data consistency (locking or latching), or an input/output subsystem not designed to support the increased number of users that



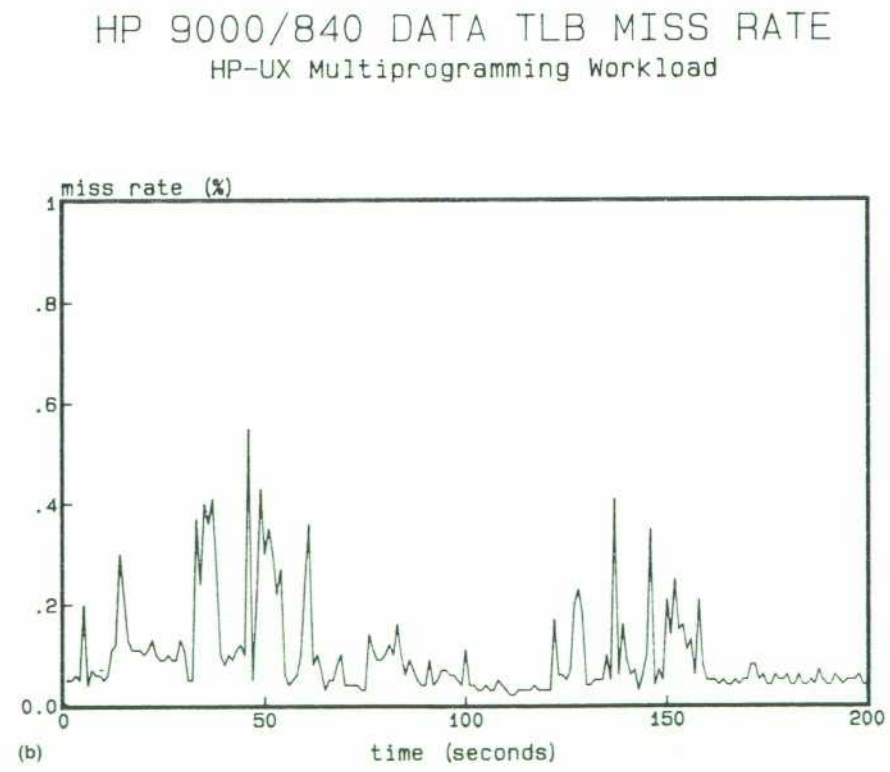


Fig. 9. *Example of a TLB miss rate measurement for the HP 9000 Model 840 processor. (a) Instructions. (b) Data.*

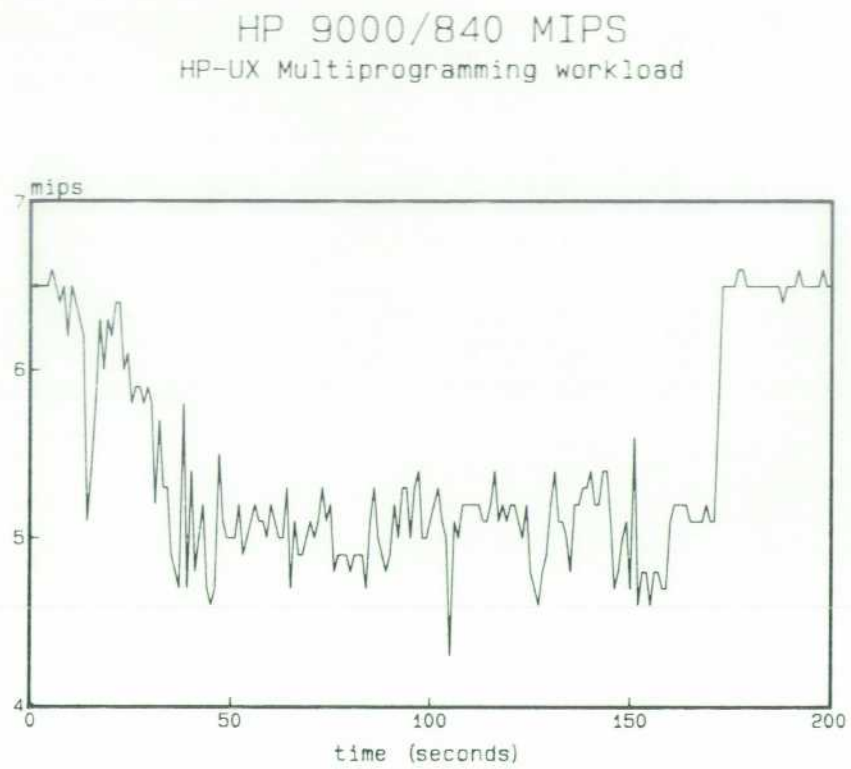


Fig. 10. *An HP 9000 Model 840 MIPS performance measurement.*

an increase in the capacity of the central processing unit would dictate.

It has been pointed out that something in a computer system has to be a bottleneck since a perfectly balanced computer system is probably impossible to achieve.[6] It would seem to be common sense that the best choice of the component of the computer system selected to be the bottleneck would be the most expensive component. To choose any other would be to waste the most expensive component and would not maximize performance and minimize cost. Usually, the most expensive component of the computer system is the CPU, hence the choice of MIPS as the metric of computer system performance. But, it should be realized that it takes a lot of hard work to make the CPU the bottleneck in all but the simplest systems. This situation presumes a highly-tuned operating system, data communication system, data base management system, and set of applications. Therefore, although it can be understood why MIPS is so often used as the performance metric of choice, this measure must be tempered by other measures.

An effort was begun in HP Laboratories in 1982 to characterize the environment in which HP computer systems were operating. This study concentrated on the characterization of HP 3000 and HP 1000 installations, since the HP 9000 was too new at the time to characterize clearly. Fig. 11 is a synopsis of the type of data gathered by measurements of actual HP customer installations (in this case an HP 3000 installation). From this data, HP engineers have created a set of workloads used to characterize the high end of the HP 3000 and HP 1000 environments. From these model workloads, benchmarks have been created to study system performance by measurements of the new software components of the computer systems based on HP Precision Architecture. Fig. 12 is a high-level view of this process.

It must be mentioned that by "workloads" is meant written descriptions of snapshots of actual installations. Fig. 11 is such a snapshot. This form of description of a computer system is useful for analytic and simulative modeling of possible future alternatives based on this computing environment. Reference 6 further describes the process of gathering and using workloads in systems performance studies. Reference 7 is excellent in its depiction of system models.

Estimation Using Workload Data

A very simple example of how systems throughput can be estimated using data from workloads is as follows.

Let us predict the throughput of a proposed computer system with a new COBOL compiler and a data base management and file system redesigned for increased system throughput. Assume that the current DBMS (data base man-

Site: Sample		Period of Observation	3600 seconds
		Total CPU	1971 seconds
		Transactions	3105
Percent of Dynamic Path Length			
Data Base:	10.43	OS Kernel:	26.48
File System:	12.82	Misc:	6.11
I/O:	33.76	User:	10.4
I/O Information			
Total I/O:	134940	Data Base I/O:	39611
Disc I/O:	91238	Non Data Base I/O:	51627
Non-Disc:	43702		

Fig. 11. *Data gathered from an actual HP 3000 installation.*

agement system) has an emerging bottleneck for systems with increased CPU power because of serialization on its buffer pool, and that the new design alleviates this bottleneck, but costs more in instructions. Assume that the computer system that this system is to replace has the workload characteristics shown in Fig. 11 and that the central processing unit of the current system has one fourth the processing ability (MIPS) of the new system's CPU. Estimates show that the new DBMS and file system must execute approximately twice the number of instructions (dynamic path length) to achieve the same transaction rate as the current DBMS and file system. Also, the new COBOL compiler is assumed to have a 10% reduction in dynamic path length. What is the throughput of the new system relative to the current system?

The following simple model is indicative of the techniques HP design engineers are using to evaluate the kinds of options depicted above. Using Fig. 11, let us define several terms (using the techniques outlined in reference 6):

P = time of observation of the system under measurement
C = seconds the CPU is active during the measurement period P
T = number of completed transactions observed during period P
λ = transactions per second = T/P
θ = seconds of CPU time per transaction = C/T.

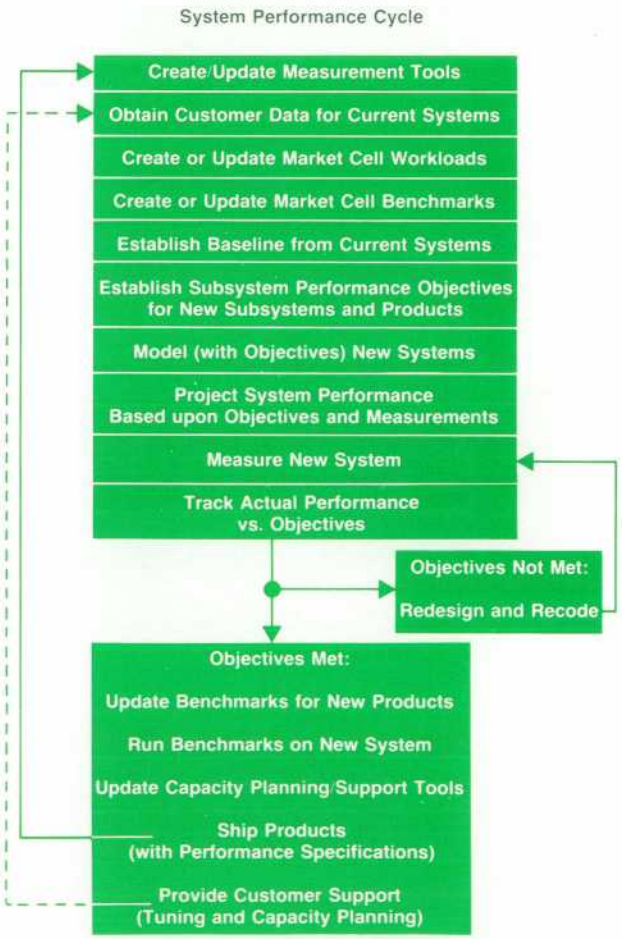


Fig. 12. *The process used to determine system performance.*

If we assume that we have a uniprocessor CPU system, then $\rho_{CPU} = \lambda\theta$ = C/P = CPU utilization.

If we assume that the new system is operated at the same CPU utilization as the old one (to keep response times roughly the same, for example) then

$$\lambda_{old}\ \theta_{old} = \lambda_{new}\ \theta_{new}.$$

That is, the ratio of transaction throughput of the new to the old system is equal to the ratio of the old CPU seconds to execute T transactions to the new CPU seconds to execute T transactions, or the throughput ratio of the new system to the old (for approximately the same response time) is inversely related to the ratio of CPU times to execute the same number of transactions. The value of the original CPU seconds to execute the observed T transactions in P seconds is made up of the components shown in Fig. 11.

For example, the data base component from Fig. 11 is 205.6 seconds, or 10.43% of the 1971 seconds that the CPU is active during the observation period P. In like manner, the file system component in the current system of Fig. 11 is 252.7 seconds. The 1971 seconds of active CPU time during the sample period of 3600 seconds recorded in Fig. 11 is therefore made up of six component parts: the DBMS subsystem, the file system, the low-level I/O subsystem, the operating system kernel, the user application code (which we know is written in COBOL), and the effect of direct terminal connection, which, although not shown in Fig. 11, is represented in the I/O counts shown in Fig. 11 as "non data base."

If the system software making up the current system were perfect and allowed increasing levels of multiprogramming and multiprocessing without penalty, then the new computer system under consideration would have four times the throughput for a comparable response time as the old, since the only limiting factor in this "perfect" computer system is the power (MIPS) of the system CPU. However, an increase in CPU power of a factor of four from the current system would, for this example, allow an increase in data base throughput of only 10% because of the serious serialization mentioned above. The factor of two increases in dynamic path length for the new DBMS and file system seem to be a high price to pay for increased throughput, however. Let us use the data in Fig. 11 to test the sensitivity to this supposition.

For the new CPU and software, the 1971 seconds is reduced to 1971 seconds ÷ 4 = 492.8 seconds because of the increased processing power of the new CPU. However, the COBOL compiler costs less in computer time and the DBMS and file system cost more. So, the actual figure is 492.8 + (205.6 + 252.7)/4 − (0.1)(205)/4 = 612.5 seconds.

Consequently, the new system has a throughput relative to the old of 1971 seconds/612.5 seconds or 3.22:1 instead of the expected 4.0:1. However, the current design of the DBMS and file system would evolve into an increase of only 10% for an increase of 400% in CPU power, whereas the new system design allows 3.22/4.0 or 81% of the raw CPU power to be realized. This particular example points out some not so obvious factors in computer system designs:

- Transaction throughput may not track linearly with the

power of the system CPU.

- One may have to execute more instructions in some components of the software system to realize the power of an enhanced processor's capacity.
- Don't spend disproportionate time in tuning a component that doesn't have much affect on system performance (such as the COBOL compiler in the above example).

Estimation Using Benchmarks

The benchmark, as distinguished from the workload is a set of programs, data, and user interactions with the system that simulates an actual computing environment. The simpler benchmarks, such as Whetstones or Linpacks,[8] attempt with one batch program to depict a diverse multiuser environment with possibly hundreds of active users. We have felt that such benchmarks are unrealistic and have extended benchmarking to include realistic benchmarks that model actual computer installations. These benchmarks are driven by test setups that use terminal simulators, and a system executing interactive and batch benchmarks.

Two benefits have been derived from the HP Laboratories study of customers' use of HP computer systems. One is a data base of customer measurements like those shown in Fig. 11 upon which we can base workloads and benchmarks, and the other is the formalization of the tools gathered to create this data base into tools and services that are being sold today, such as HP CapPlan, HP Snapshot, and HP Trend. The invaluable information gathered by this effort has allowed us to profile a large portion of HP's customer set. Our workloads and benchmarks, as a consequence, are much more complex than the industry standards such as Whetstones and Linpacks. To measure system performance, HP development engineers have had to integrate performance measurement tools into the software and hardware of the HP Precision Architecture family. The analyzer board and the HP 64000 are examples of such instrumentation. Further examples include software instrumentation that parallels but extends that familiar to the users of HP 3000 systems and new instrumentation for the HP 1000 and HP 9000 user.

Acknowledgments

A number of people who contributed to the processes described in this article need special mention. The first performance team at HP Labs consisted of Mike Gardner (who also ran the hardware group), Lillian Yee, and Dan O'Brien. Others who joined later were Ron Kliesch, Paul Dembry, Paul Primmer, Jeff Dielle, Tom Gefell, Tony Engberg, Tom Spuhler, Dave Hood, and a host of others, too numerous to mention, from HP product divisions.

References

1. J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High Precision Architecture," *Hewlett-Packard Journal*, Vol. 36, no. 8, August 1985.
2. L.J. Shustek, *Analysis and Performance of Computer Instruction Sets*, PhD thesis, Stanford University, January 1978.
3. D.S. Coutant, C.L. Hammond, and J.W. Kelly, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, Vol. 37, no. 1, January 1986.
4. M.J. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor," *this issue*.
5. A.J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, no. 3, September 1982.
6. P.J. Denning and J.P. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys*, Vol. 10, no. 3, September 1978.
7. H. Kobayashi, *Modeling and Analysis*, Addison-Wesley, 1981.
8. J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," *ACM SIGARC*, January 1985.

The HP Precision Simulator

Designed for flexibility, portability, speed, and accuracy, the simulator is useful for both hardware and software development.

by Daniel J. Magenheimer

THE HP PRECISION SIMULATOR is an internal tool used heavily throughout the research, design, and development stages of HP Precision Architecture and its systems software. In addition to functionally simulating the architecture at the instruction set level, the simulator provides an interactive screen-oriented machine state display and user interface, a combination that makes it particularly useful in compiler and operating system development. The simulator can also be used as a performance evaluation tool, since it can easily model different hardware implementations and record statistical results for comparison. Finally, the ability to set code and data breakpoints, change registers and memory locations, and record branch histories makes it an effective assembly-level debugger.

Development

The goals in the development of the simulator were four: flexibility, portability, speed, and accuracy. Flexibility was truly a requirement—in the early research phases of the architecture, instructions were added and deleted nearly on a weekly basis and bit fields were shuffled frequently. Timely results were necessary to evaluate the new instruction sets, so changes to the simulator were frequent. Portability was also needed, because simulations were done by design engineers in their daily work environment as well as batched on powerful mainframes. To this end, all coding was done in the high-level C language and use of library routines was limited to those present in the portable C library. This design choice allowed ports to several HP machines, two Digital Equipment Corporation machines, and an Amdahl mainframe, and even allowed the simulator to simulate itself.

Speed and accuracy were the most important requirements. Simulation speeds of thousands of instructions per second were necessary to provide timely feedback for performance measurement, but since the simulator was the only implementation of the instruction set before hardware prototypes became available, complete and accurate simulation of each instruction was mandatory. Of course, these goals often conflicted. For example, top speed can be obtained by coding in assembly language, but then portability is lost. Nonetheless, a reasonable simulator was created which has all of the desired characteristics and satisifies the needs of a large class of design and development engineers.

User Interface

The simulator presents a screen containing four nonoverlapping windows (see Fig. 1). One of these windows is used for user command entry and message reporting. The three other windows contain useful machine state information as follows:

- The register window shows the contents of either of two sets of registers—the general registers or the space and system control registers—all in eight-digit hexadecimal format.
- The program window provides a nine-instruction view into the program space. Each line of the window contains the address of a word, both in hex and symbolically, the word itself (in hex), and a symbolic disassembly of the instruction. The first two columns indicate whether a breakpoint is set at that instruction and where the pro-

```
Register General
 r0  / 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
 r8  / 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
 r16 / 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
 r24 / 00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
 pc  = 00000000.00000000   priv = 0    psw = 0004000e    sar = 0
Program (Space: 00000000)
 > 00000800 = $START$      / 23600800   LDIL    0x40000000,27
   00000804 = $START$+0004 / 377b0008   LDO     4(27),27
   00000808 = $START$+0008 / 34020002   LDO     1(0),2
   0000080c = $START$+000c / 00027820   MTSP    2,5
   00000810 = $START$+0010 / 23c00800   LDIL    0x40000000,30
   00000814 = $START$+0014 / 37de0428   LDO     532(30),30
   00000818 = $START$+0018 / b7de000e   ADDI    7,30,30
   0000081c = $START$+001c / d7c01c1d   DEPI    0,31,3,30
   00000820 = $START$+0020 / 0fc012a8   STWS,MA 0,4(0,30)
Data (Space: 00000001)
40000000/00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
40000020/00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Command

* pj $START$
*
```

Fig. 1. *The HP Precision simulator screen has four windows showing commands and messages, registers, program lines, and data.*

gram counter is currently positioned.
- Sixteen words are presented in hexadecimal format in the data window. Each line is preceded by an address, possibly symbolic.

When the four-window format is too restrictive, for example when displaying lists or tables of information, the screen is cleared and information is presented a screen at a time, prompting the user to hit a carriage return to see the next screenful.

The screen and windows can be manipulated for fine adjustments or major changes of context with simple commands given in the simulator command language. The command language is simple enough that a beginner can pick up the essentials quickly, but extensible so that the advanced user can accomplish tasks with a minimum of keystrokes. Each command is given with a set of arguments (sometimes optional). Frequent commands can be abbreviated, and a command can be repeated easily. For example,

```
* sr
```

displays the space and system control registers in the register window, while

```
* pj @(%main+0.4)
```

repositions (jumps) the program window to the first word (four bytes) following the symbol main.

A help facility is also provided for quick reference (see Fig. 2).

```
=       assign value to register/data address
ar      display assist registers
bs      set breakpoint
bD      delete all breakpoints
bd      delete breakpoint n
blist   list breakpoints
contin  continue program
cd      get indirect files from given directory
dj      jump to specified address in data window
db      move data window backward n words
df      move data window forward n words
dbD     delete all data breakpoints
dbd     delete data breakpoint n
dblist  list data breakpoints
dbs     set data breakpoint
disasm  dump memory disassembled as instructions
do      execute indirect command file
dascii  enter/exit ascii mode in data window
dstack  enter/exit stack mode in data window
gr      display general registers
grclr   clear all general registers
jnl     open journal file
load    load executable file from dis
macro   define macro
macdel  delete (pop) macro definition
maclist list macros
memdump dump memory in hex format to file
pj      jump to specified address in program window
pb      move program window backward n words
pf      move program window forward n words
page    create/change access protection for page
quit    quit (?!)
run     run program
redraw  redraw screen
reglist list registers to file
step    execute single (or n) instruction(s)
save    save simulator status in file
space   create/change bounds/protect of specified space
sr      display special registers
stack   display stack trace
stats   print statistics for most recent run
stop    stop execution of program
symbol  define symbol
symdel  delete (pop) symbol definition
symlist list symbols
tblist  list last few taken branches
trace   generate address trace to file
update  update screen
!       pass command string to system
#       convert hex to decimal
<       take application input from file
>       write application output to file
.goto   goto label
.if     conditionally execute rest of line
:       target of goto (and else in .if-else)
;       ignore line (comment)
```

Fig. 2. *The simulator provides a help facility for quick reference.*

Program Simulation

As the name implies, the primary function of the simulator is to simulate HP Precision Architecture instructions and programs built from these instructions. To accomplish this function, several commands and features provide the ability to load and execute programs. These capabilities are complemented by a full statistics gathering mechanism.

Although small programs can be entered entirely by hand, it is much more efficient to be able to load a binary program from the underlying file system. The binary object file contains sufficient information to allow the simulator not only to load the program and its data, but also to build a symbol table and initialize the screen and program counter properly.

There is an additional problem: when an operating system loads a program and prepares it to run, it must map the virtual addresses of the program to physical memory locations, determine program protection, and enter this information in internal data structures. On the simulator, there is no operating system to perform these tasks. It must do the mapping and information storage itself by making assumptions about the program it is loading. These assumptions can be overridden or completely determined by the user, but reasonable defaults are selected which are generally sufficient.

Programs can be run from start to finish without interruption, stopped at appropriate places and continued, or single-stepped for debugging or educational purposes. In any case, the effect of each instruction is completely and accurately simulated and statistics are gathered. For example,

```
* run
```

invalidates entries in the cache and TLB (translation lookaside buffer), resets statistical counters, and starts execution of the program, and

```
* contin
```

starts the program without any initialization (for example, after encountering a breakpoint).

```
* step 100 update
```

single-steps 100 instructions, updating the screen following each, while

```
* step 0 .if !%r1 stop
```

continues stepping forever until general register 1 is equal to 0.

In addition to virtual memory mapping, the simulator must also provide other functionality normally associated with an operating system. All interruptions must be appropriately handled. For some, such as the TLB miss faults, reasonable action takes place to correct the problem and program execution continues. For many others, such as insufficient privilege traps, a program bug is indicated. The simulator notifies the user with a descriptive message and stops the program.

Another example of required functionality beyond direct instruction simulation is I/O. Even the simplest benchmarks require reading from a file or writing to the screen. To support this, the simulator recognizes certain pseudoinstructions as HP-UX system calls. By recoding the system library to use these pseudoinstructions, it is possible to run many large programs (including the simulator itself) and measure the user component of their performance.

Debugging Features

To be able to observe changes in machine state as one steps through a program is often sufficient to debug a program, but more sophisticated features are always helpful. The simulator allows assignment to any register or memory location at any point in the program execution. This can be used to patch or skip portions of code, change data or parameters to procedures, and simulate external events (e.g., interrupts). In addition, a large set of internal simulator variables can be changed to modify the behavior of the simulator or remember important values.

Another useful feature is the ability to set code and data breakpoints. Code breakpoints are marks within the executable code of a program that cause execution to be halted when they are encountered in the normal flow of a running program. When a breakpoint is hit, the program stops and control is returned to the user at command level. Data breakpoints can be viewed as temporary access restrictions on a region of data. Access of data within the region causes a running program to halt at the instruction that attempted the access. The region can vary in size from one byte to an entire space and can be specified to cause a break either on writes or on both reads and writes. Commands are provided to set, delete, and list current code and data breakpoints. For example,

```
* bs %print
```

sets a code breakpoint at the memory location associated with the symbol print.

```
* bs %foo 50 .if !%r2 = %u0 %u0+1
```

sets a code breakpoint such that the program will stop only on the 50th time that the instruction at the beginning of the foo procedure is executed and will count how many times (out of 50) that general register 2 is equal to 0 at that point, recording the result in a user temporary register. Another common bug that can be detected with the help of the simulator is the wild branch, a branch that has a false target far outside the program, or worse, a target at a random place within the program. When this happens, the simulator can provide a list of up to the last twenty taken branches to assist in determining what went awry (see Fig. 3). The simulator can also display the current stack trace, a list showing what procedures called the current procedure, along with parameters and local variables. Finally, a command can be given to dump a region of memory to a file, either in hex or in disassembled instruction format.

```
        -- Last 10 taken branches ---
   Delay slot address          Branch target address
   ------------------          ---------------------
   00000000.00000854            00000000.00000858
   00000000.00000858            00000000.00000873
   00000000.000008ab            00000000.000008bb
   00000000.00000923            00000000.00000933
   00000000.00000957            00000000.00000927
   00000000.0000092f            00000000.000008af
   00000000.000008b7            00000000.0000085b
   00000000.00000863            00000000.0000095b
   00000000.00000973            00000000.00000873
   00000000.000008ab            00000000.000008bb
RETURN to continue...
```

Fig. 3. *Wild branches can be detected with the help of the taken branch list, a list of up to twenty of the last taken branches.*

Performance Analysis

Before hardware became available, the simulator was the only tool capable of analyzing performance on a native instruction stream. As mentioned above, statistics are collected during program execution to provide cycle count and instruction distributions. Often, however, performance issues go well beyond the instruction set. To this end, the simulator is equipped with a large set of parameters and flags which allow a performance engineer to analyze different cache and TLB sizes. Operational characteristics such as cache and TLB miss overhead and replacement algorithms can be analyzed, and various memory delays and interlocks can be modeled.

Using these parameters, studies were done to estimate

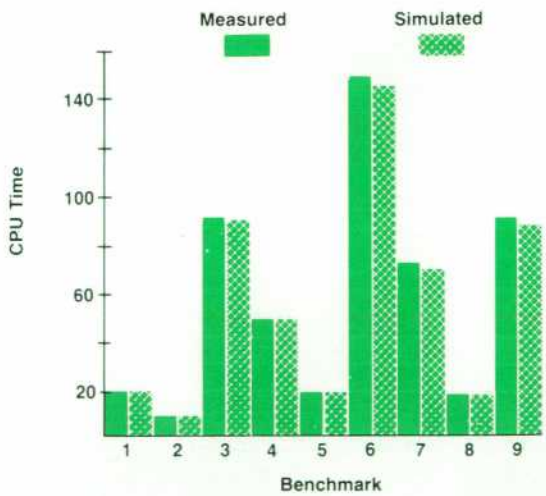


Fig. 4. *Simulator estimates compared with actual machine measurements.*

the benefits of cache control hints, critical-word-first cache algorithms, and two-level TLBs and to compare the performance of a small instruction cache against an instruction lookahead buffer. Other studies accurately estimated the performance of different implementations before they were built. Fig. 4 compares simulator estimates with actual machine measurements.

Lastly, the simulator can provide instruction execution traces to feed into other present and future performance analysis tools.

Miscellaneous Features

Many features are provided to assist the advanced simulator user. Indirect command files can be executed to avoid repetitive command sequences. These files can be nested, can be commented, and can contain if statements and goto statements which raise the command language to the power of any high-level programming language (Fig. 5). Other commands provide for saving and restoring of the simulator's state (so a session can be resumed at a later time), recording of command sequences in a journal file, macro definition and use, and an HP-UX shell escape.

```
= $u29 0 ; Put stack marker unwind back to Q.
.if $u28^d0d0caca .goto trans
= $u5 ((r12-r13)/2) ; Delta P.
= $u31 $u5 ; P for display.
.goto stat
: trans
= $u5 pc ; translator pc.
: stat
= $u4 r5&ffff ; STATUS.
= $u30 $u4 ; STATUS for display.
dj --sr5.(r3-6) db ; Show stack around Q.
.if $u28^d0d0caca .goto m2
= $cmdfecho$ 1 ; Emulator
do cmcurm.ss
.goto showem
: m2
= $cmdfecho$ 1 ; Translator
do cmcurtm.ss
: showem
# $u30
.if $u28^d0d0caca .goto done : # $u31
: done
```

Fig. 5. *An example of a simulator command program. (Courtesy of Tony Hunt.)*

Progeny

Work on the simulator influenced the development of several other tools. Foremost among these is RDB (see box), a remote debugger, which is still being used for low-level operating system and I/O development and booting of newly developed hardware implementations. The instruction disassembler component of the simulator has been extracted and used in HP-UX's assembly language debugger adb and in the high-level language debugger xdb. The user interface has been borrowed for a hardware support monitor and for the MPE-XL native and compatibility mode debuggers. Other work is in progress to extend the simulator to handle multiprocessing configurations.

Acknowledgments

I am indebted to Richard Steiger, whose design principles and work on a microprocessor debugger called MUD (actually μD), heavily influenced the design and user interface of the simulator. Many thanks to Carol Hammond Thompson for her patience in debugging the simulator with her architectural verification programs. Thanks also to Michael Mahon, Steve Muchnick, Terry Miller, Steve Boettner, and Jerry Huck for their help and suggestions in designing the simulator.

Remote Debugger

RDB is a remote debugger, a tool that runs on an HP 9000 Model 220 Computer (the host machine) but allows manipulation of programs on a different (remote) machine. This capability is especially important in the early stages of testing a new machine implementation and developing and bringing up an operating system on it. The tool has been used extensively for these purposes. RDB consists of three major components: a user interface, an intermachine interface (including hardware and software), and a small software monitor which runs on the remote machine.

The user interface was extracted from the HP Precision simulator and, except for a few minor differences in the list of commands, an inexperienced user would be hard-pressed to tell them apart. Besides leveraging thousands of lines of code, this approach minimized the learning effort for engineers who used both tools. As with the simulator, registers and memory locations can be changed and code breakpoints can be set (data breakpoints are not supported). The same interruption handling and HP-UX system call support as in the simulator are used, thus allowing large programs to be run and measured on new hardware without operating system support.

The intermachine interface consists of two I/O drivers, one on the host system and one on the remote system, that communicate through a GPIO 16-bit parallel card. The host controls the communication by issuing a small set of commands: READ a variable amount of data from a physical address, WRITE data to an address, and lock and release semaphores. The remote processor is started by writing to a continuation flag at a fixed memory location, and is stopped by asserting an external interrupt.

The monitor is a small (less than 1K bytes) operating system subset that catches interrupts, traps, and faults and notifies the host processor of the type of interruption. Since the host processor can only read from and write to memory, not registers, the monitor is also responsible for saving the machine state in memory and restoring it on continuation.

Acknowledgments

RDB was truly a team effort. Martin Liu designed and coded the software portion of the intermachine component and the remote monitor. Jim Hull designed and built the parallel card. Dan Hachigian wrote the device driver for the card. Thanks also to Bruce Thompson, Lamont Jones, Tom McNeal, and Chris Mayo who extended the tool to handle new machines and environments.

Dan Magenheimer
Project Manager
Information Technology Group

Reader Forum

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, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A.

Editor:

I read about the Spectrum program in a recent issue of the *HP Journal* ("Compilers for the New Generation of Hewlett-Packard Computers," January 1986). In that article you talked about the primitive instructions that will be accepted. You wrote that the instructions that reference only registers are faster than those that imply access to memory.

You said that among the instructions that take more than one cycle in their execution are the loads. The order of execution of an operation, such as addition, might be:

1. Load first operand, a, in register p;
2. Load second operand, b, in register r;
3. Perform the operation between registers p and r;
4. Store the result if necessary.

Why is it not possible to have an instruction that loads both operands at the same time, in one step? This implies a dual access to memory in reading only. (The cases that would imply dual writing to memory are very rare.) With such an instruction, the order of execution now is:

1. Load, by dual access, the two operands a and b in registers p and r;
2. Perform the operation between registers p and r;
3. Store the result if necessary.

This way the computer saves a step, since it takes only a single step to load the operands in the two registers.

D. ing. Dejan Claud
Maramures, Romania

As you noted, some instructions are permitted to take longer than one cycle to complete. In most implementations, however, LOAD executes in a single cycle. Additional cycles may be required if the next sequential instruction refers to the register loaded by the LOAD. This is an "interlock" situation that could take between one cycle (if the data is already in the cache) and many cycles (if the data must be fetched from main memory). The faster case is usually scheduled by the optimizing compilers in such a way that the register loaded is not referenced by the immediately following instruction. In this way, the interlock is avoided, and useful computation is performed in parallel with completion of the LOAD.

You ask why it is not possible to implement an instruction that loads two operands to two registers at the same time. In fact, LOADs are less frequent than your example suggests. Frequently one or more operands are already present in registers, and do not require access to memory.

To carry out simultaneously all of the actions required to implement a "double load," considerable additional hardware would be required. Providing for dual access to the cache would require the duplication of essentially the entire cache and virtual address translation hardware. Memory systems are inherently serial devices, because all memory elements share common addressing, checking, and control hardware. It is possible to design memory systems that are truly dual-ported (not just a "multiplexed" single port), but the cost in decreased speed or capacity is considerable.

Of course, to fully support "double load," the address and data buses, the effective address adder, and the data-addressing and register-specifying content of the instruction itself would all have to be duplicated. In short, the architecture supports the highest-performance LOAD that is commensurate with a high-speed implementation. Any additional function associated with LOAD would increase cost more than it increased performance, and so would be inadvisable.

A key characteristic of HP Precision Architecture is that, unlike most microcoded machines, the performance of implementations is limited by the hardware's ability to perform the requested operation, not by the control unit's ability to decode instructions, specify registers, and sequence signals. The best that one can hope for is that all (or much) of the machine's hardware can be kept busy on each cycle by most programs. In general, this objective is better served by simpler hardware configurations than by complex ones.

Michael J. Mahon
Manager, Computer Languages Laboratory
Information Technology Group

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

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

HEWLETT-PACKARD JOURNAL
August 1986 Volume 37 • Number 8

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3000 Hanover Street
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
P.O. Box 529, Startbaan 16
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8551