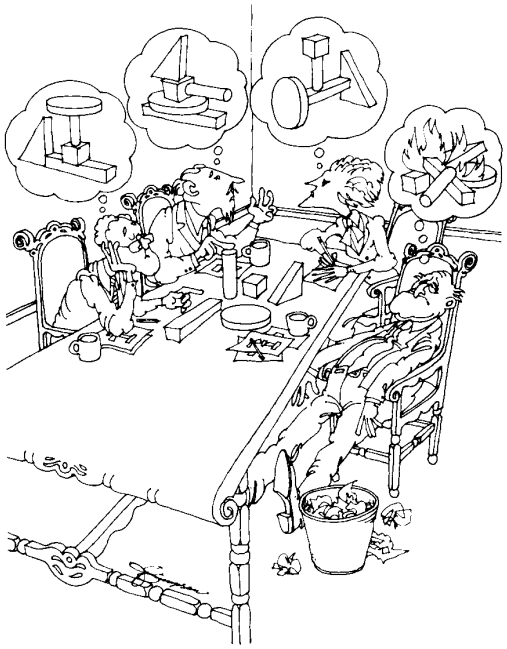


Precision Architecture

Ruby B. Lee

Hewlett-Packard Company

Hewlett-Packard designed Precision Architecture to serve as a common foundation for its computer systems, to enhance software portability, to provide price-performance advantages, and to streamline the company's hardware and software development, manufacturing, and support activities. Prior to this, each of HP's three major computer product lines, the HP3000, HP9000, and HP1000 systems, had different processor architectures, operating systems, and input-output systems.

This article describes the processor component of the Hewlett-Packard Precision Architecture system, henceforth referred to simply as "Precision." It describes the architecture's goals, how the architecture addresses the spectrum of general-purpose user information processing needs, and some architectural design trade-offs.

Goals. When HP charged the original architects with designing the new architecture, it presented us with some high-level, strategic goals. The architecture should be general purpose for use in commercial and technical applications. It should be scalable across technologies, cost ranges, and performance ranges and provide price-performance advantages. It should allow the leveraging of common hardware and software components. It should be designed with architectural longevity in mind, including features that enhance the possibility of a long, useful life for the 1990s and beyond. It should allow growth and extendibility. It should support multiple operating environments, for example, single-user and multiuser, centralized and distributed computing, and conventional and object-oriented environments. It should support the implementation of highly reliable, secure systems and real-time environments.

The Hewlett-Packard Precision Architecture provides a simple, comprehensive foundation for general-purpose computer systems. It is scalable, efficient, and extendible.

A version of this article appeared in *Proc. 22nd Hawaii Int'l Conf. on Systems Sciences*, Jan. 3-6, 1989, Kailua-Kona, Hawaii.

For the processor architecture, the technical mapping of these strategic goals resulted in a simple RISC-like execution model[1-3] with features for code compaction and dynamic path-length reduction, coupled with a more sophisticated set of extendibility and longevity features.

Precision execution model

For the execution model of the architecture, we mapped the scalability and price-performance goals into the following design guidelines:

- Precision instructions should be executable in a single cycle with simple (pipelined) processor hardware.
- Code compaction and dynamic execution time reduction should be considered for frequently executed operation sequences.

These guidelines resulted in an architecture where sometimes more than one operation was performed in one instruction cycle, and other times only a part of a more complex operation was performed by one instruction. We based these design decisions on extensive measurements and studies of the frequency of operations and operation sequences.[4-6] We made the basic assumption that high-level languages would be used for programming and that software and hardware would interact for

0018-9162/89/0100-0078$01.00©1989 IEEE

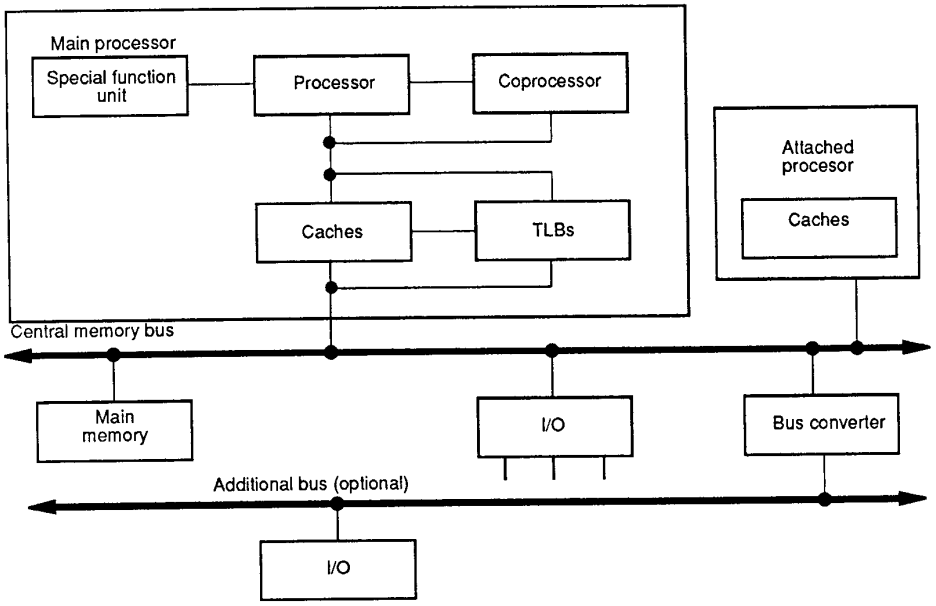


Figure 1. Typical system organization.

the most efficient execution.[1,4,5] For example, we assumed the use of high-level language optimizing compilers for optimizing code generated from user programs.

Figure 1 shows the modules in a typical system organization. Figure 2 shows the simple hardware needed for the execution unit. Figure 3 shows the registers, including the 32 general-purpose registers (GRs); the control registers (CRs), of which 25 are defined; the eight space registers (SRs); and the processor status word (PSW). I will describe the functions of these registers in the course of the article.

Table 1 summarizes the instruction set in terms of the generic operations implemented per instruction. The 53 generic instruction types can be expanded to 140 total instructions when we count all alternatives and options. The data types supported by the basic processor are signed and unsigned word, halfword, byte, packed and unpacked decimal numbers, 8-bit ASCII, and 16-bit international characters.

Simple hardware. The architecture has a general-register-based, load-store execution model with a simple execution engine comprising an arithmetic logic unit (ALU) and a shift-merge unit (SMU).

There are 32 general-purpose registers, where GR0 is a constant zero source as well as a bit-bucket destination. While using more than 32 simultaneously addressable

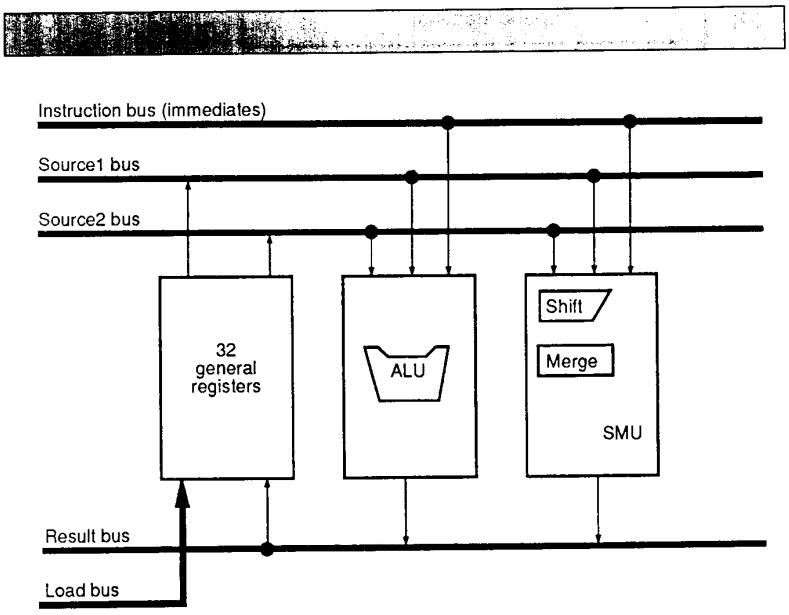


Figure 2. Execution data path.

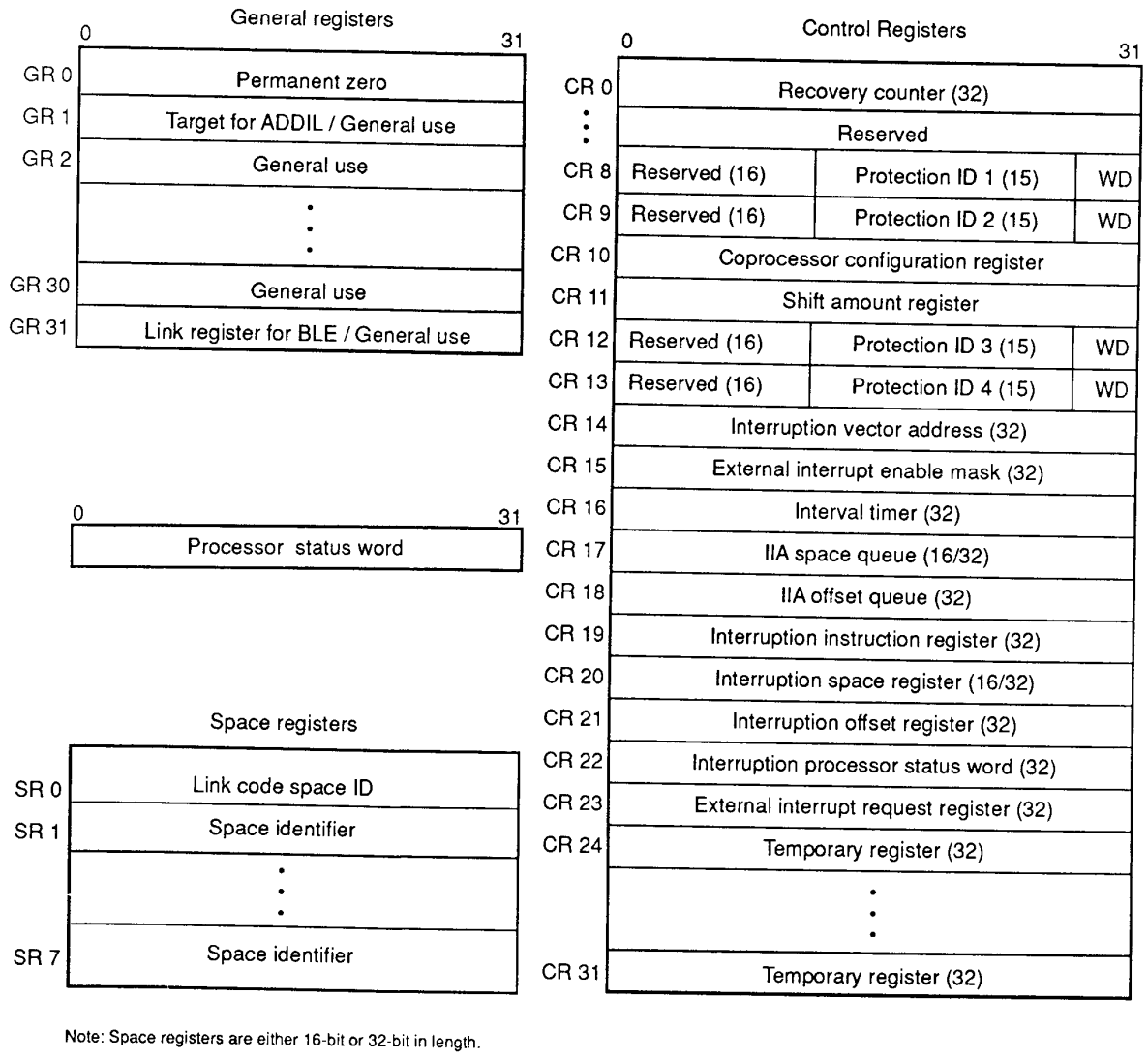


Figure 3. Registers.

general registers sometimes decreases the number of memory accesses, the trade-off yields an increase in the process swap time, in the number of bits needed to specify register addresses in an instruction, and in the area and access time needed for a larger bank of registers. Specialized register structures to improve procedure calling[2,7] often have many hidden registers, incurring complexity without the advantage of making an increased number of simultaneously accessible registers available to a good register allocator.[8]

Minimal decode instructions. To simplify instruction fetching and decoding, all Precision instructions are fixed-length 32-bit words. This eliminates, for example, the complexity of handling page faults during the fetching of a single instruction, which can happen for variable-length instructions.

Fixed-length instructions also enhance the use of fixed bit positions for time-critical operations, without waiting for decoding of the instruction. For example, since general register operands are always

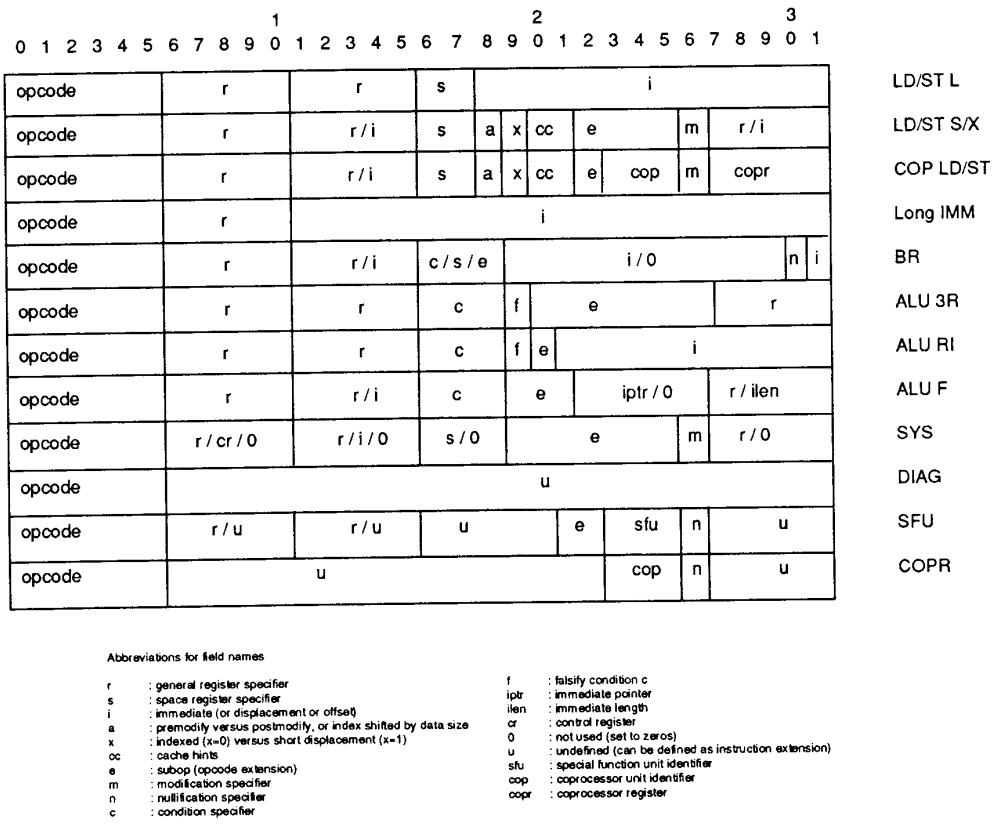


Figure 4. Instruction formats.

specified by the two leftmost register fields in any register format (see Figure 4), the reading of general registers can occur in parallel with instruction decoding. The target register, however, can be in any one of the three register fields in different instruction formats. This is an acceptable trade-off since the processor has ample time to decode the target register specification.

Combined operations. Many Precision instructions combine two operations into a single 32-bit instruction word. For example, in functional instructions (see Table 1) each instruction implicitly specifies an optional "conditional nullify" or "skip" feature in addition to the main arithmetic, logical, unit, or bit operation. In a single cycle, as the data transformation operation is performed, the condition specified in the instruction is also evaluated. If the condition evaluates to true, then the following instruction is nullified. A nullified instruction has the effect of a NOP (no operation), with no changes to any architecturally visible state, including memory, and no side-effects like causing traps or nullification.

Conditional branch instructions also combine two operations into a single 32-bit instruction by allowing simultaneously a functional operation to be performed on two registers, a condition to be evaluated, and a PC-relative branch target to be calculated, with the branch taken only if the condition evaluates to true. This again achieves code compaction, eliminates the need for storing condition codes in the processor, and enhances possibilities for reordering code in optimizing compilers.

Combined operations reduce both static code size and dynamic execution time, since only one instruction is needed rather than two or more.

Zero-cycle addressing and loading. The architecture makes extensive use of immediate data embedded within the instruction itself as one of the sources of operands. An immediate operand saves a memory load operation, provides effectively zero-cycle addressing, and does not require the use of a general register. Precision immediates are unusual in that they are "maximal length," that is, they fill up all unused bits in the fixed-length instruc-

Table 1. Instruction set.*

Memory Reference Instructions

Load {Word/Halfword/Byte} {Long/Indexed/Short} [Modified]
Store {Word/Halfword/Byte} {Long/Short} [Modified]
Load Word Absolute {Indexed/Short}
Store Word Absolute Short
Load Offset
Load And Clear Word {Indexed/Short}
Store Bytes Short

Branch Instructions

(a) Unconditional

Branch And Link {Displacement/Reg}
Branch Vectored
Branch External [and Link]
Gateway

(b) Conditional

Add {Reg/Immed} And Branch if {True/False}
Compare {Reg/Immed} And Branch if {True/False}
Move {Reg/Immed} And Branch if {True/False}
Branch On Bit {Variable/Constant}

System Instructions

(a) System Control

System Mask {Set/Reset/Move to}
Move {to/from} Control Register
Move {to/from} Space Register
Load Space ID
Break
Return From Interrupt
Diagnose

(b) Memory Management

Insert TLB {Instruction/Data} {Address/Protection}
Purge TLB {Instruction/Data} [Entry]
Probe Access {Read/Write} {Reg/Immed}
Load Physical Address
Load Hash Address

(c) Cache Management

Flush {Instruction/Data} Cache [Entry]
Purge Data Cache
Sync

Functional Instructions

(a) Arithmetic

Add {Reg/Immed} [with carry] [and Trap on {overflow/cond/overflow or cond}]
Sub {Reg/Immed} [with borrow] [and Trap on {borrow/cond/borrow or cond}]
Shift {One/Two/Three} And Add [and Trap on Overflow]
Divide Step

(b) Logical

Or {Inclusive/Exclusive}
And {True/Complement}
Compare {Reg/Immed} And Clear
Add Logical
Shift {One/Two/Three} And Add Logical

(c) Unit and Decimal

Unit Xor
Unit Add Complement [and Trap on Condition]
Decimal Correct
Intermediate Decimal Correct

(d) Bit Manipulation

Extract {Variable Pos/Constant Pos} {Signed/Unsigned}
Deposit {Variable Pos/Constant Pos} {Reg/Immed}
Zero and Deposit {Variable Pos/Constant Pos} {Reg/Immed}
Shift Double {Variable Pos/Constant Pos}

(e) Long Immediate

Add Immediate Left
Load Immediate Left

Assist Instructions

(a) Special Function Unit Interface

Spop {Zero/One/Two/Three}

(b) Coprocessor Interface

Copr Load {Word/Doubleword} {Indexed/Short}
Copr Store {Word/Doubleword} {Indexed/Short}
Copr Operation

Key
Reg = register
Immed = immediate
Pos = position
cond = condition

*Curly brackets indicate that one alternative within the curly brackets is selected for a given instruction, while square brackets indicate an optional feature that can be specified in the instruction.

tion and hence maximize the probability that a constant can be represented as immediate data within the instruction. Usually, this would imply that the sign position, in its traditional encoding as the leftmost bit of an integer value, would occur in variable positions. Precision solved this problem by encoding the sign position of these variable-length immediates as the rightmost bit, simplifying decoding and sign extension.

Full-word immediates. Sometimes, even maximal-length immediates in an instruction are not long enough, since a 32-bit immediate or displacement is needed. Precision introduces 32-bit immediates in the instruction stream by using two fixed-length 32-bit instructions. For example, Load Immediate Left loads into a general register, GR*i*, a 21-bit immediate padded on the right with 11 zeros. A Load instruction executed later, with this GR*i* as the base register, supplies the low-order bits of the 32-bit displacement value.

This method has the advantage that each instruction can still be a fixed-length

32-bit word, simplifying instruction fetching and decoding. The alternative—variable-length instructions—requires either instruction alignment provisions with attendant memory wastage, or handling the complexity of a page-fault potentially occurring during an instruction fetch.

A trade-off in the use of immediates arises in encoding space versus operation orthogonality, that is, the size of the value that can be represented by the immediate versus the other options that can be specified in the fixed-length instruction. For instructions with a long immediate field, we chose to include only those instruction variants most frequently used rather than achieve full operation orthogonality with instructions where both operands come from registers.

Memory reference instructions. Effective address calculation for Precision load and store instructions uses the same execution unit (Figure 2) as add instructions and is based on the same guideline of single-cycle execution.

Static and dynamic displacements. All address calculations for load and store instructions are based on the base plus displacement, or base plus (shifted) index addressing modes, the most frequently used addressing modes.[9,10] Static displacements of 14 bits can be accomplished in one instruction, and 32-bit static displacements can be done with two instructions using a long immediate instruction, as described earlier. Using an index register, 32-bit dynamic displacements are possible.

Byte addressing. One reason Precision implements byte addressing rather than just word addressing is to allow the efficient movement of unaligned strings of bytes or characters, common in commercial computations.

A unique Store_bytes instruction simplifies such moves by allowing storage of any sequence of one to four bytes starting at any byte location within a 32-bit word. This includes *tribytes*, defined as three consecutive bytes in a word. Storing of tribytes comes free with byte addressing. In other architectures, unaligned byte moves would have required loading and masking of the destination word.

Address stride mechanisms. For indexed load instructions, the value in the index register can also be shifted by the data size to index bytes, halfwords, or words. Moreover, the instruction can specify address modification of the base register, with support of both premodification and postmodification. A load or store operation combined with address modification is another example of combining two operations in a single instruction word.

A hardware-software trade-off resulted in the absence of indexed store instructions in the basic architecture. We chose to do this because achieving single-cycle execution would require a register file with three read ports rather than two. Coprocessor indexed store instructions exist, however, since the data register being stored comes from the coprocessor rather than the basic processor.

Another interesting encoding trade-off is that, in long-displacement load and store instructions, the timing of address modification (pre or post) is encoded by the same bit that encodes the sign of the displacement (increment or decrement). This prevented cutting in half the range of the 14-bit displacement while still allowing efficient accessing of stacks with the predecrement and postincrement options.

Delayed load effect. Optimizing compilers for Precision processors try to insert one or more instructions after a load instruction to prevent interlocked pipeline cycles. However, Precision processors will interlock if an instruction following a load instruction uses a register with a pending load. This hardware-software trade-off incurs insignificant additional hardware complexity while preserving code compaction by not requiring the insertion of NOPs after load instructions, as in some other architectures without such interlocks.[3,11] More significantly, the provision of hardware interlocks gives implementors the freedom to design different pipelines while guaranteeing object code compatibility.

Branch instructions. Precision implements delayed branching with some extra optimization features. In some architectures,[2,3,7,11] if a common instruction cannot be found, NOPs have to be inserted in the delay slot of a conditional branch, which can be executed for both paths of the branch. Precision achieves delayed branching with both static and dynamic code size reductions by enhancing the usage of the delay slot instruction following a conditional branch instruction. Conditional nullification is performed for backward branches only if the condition is false and for forward branches only if the condition is true. For example, by closing loops with backward branches, compilers can always move the first instruction of the loop to the delay slot of the loop-closing backward branch, decreasing the loop size by one. By using forward conditional branches to rarely used code, software can again optimize the use of the delay slot instruction for the more frequently used fall-through path. If code is arranged so that backward branches are more likely to be taken than forward branches, then hardware can use the sign of the branch displacement as a static branch prediction bit.

Simple Branch And Link instructions are used as procedure call primitives, with the return address saved in a general register. A base-relative branch using this general register is used for subroutine return.

Functional instructions. Functional instructions execute a data transformation operation in a single pass through the ALU or SMU (see Figure 2).

Arithmetic instructions. The Add and Subtract arithmetic instructions come with the widest range of options (see Table 1). The Add And Trap on condition option allows range checking, often required by high-level languages, to be accomplished with minimal instructions.

Multiply and divide primitives. The Shift And Add instructions implement a simple integer multiply and accumulate function, using the standard ALU hardware (Figure 2) with a wider multiplexer on one port. Multiplication by small constants can be accomplished in a few cycles, and multiplication by a variable can be done typically by breaking the multiplier into 4-bit pieces.[5] The Divide Step instruction implements a single-bit nonrestoring division operation and can be used in a sequence to perform integer division.

To implement full fixed-point integer multiply and divide in a single cycle would have required special hardware. We did not consider this cost-effective for a basic, general-purpose Precision processor because our studies of large collections of programs show that integer multiply and divide operations are rarely used, and multiplication usually occurs with a constant known at compile time.[5,6] Hence, we included only simple multiply and divide primitives in the basic instruction set, with floating-point instructions and integer

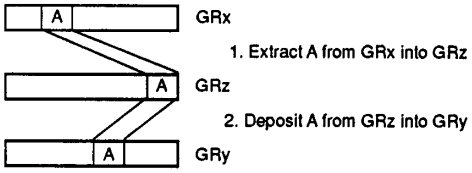


Figure 5. Arbitrary bit field movement.

multiply and divide instructions added as instruction-set extensions supported by optional hardware assists (described in the section on the Precision assists architecture).

Logical operations. The logical instructions allow efficient implementation of arbitrary Boolean conditions. For example, the Compare And Clear instruction first assumes a Boolean value of false by storing zero in the target register. The negation of the desired Boolean condition is used to conditionally nullify the following instruction. This instruction, if not nullified by the Compare And Clear instruction, will set the target register to true. Other architectures usually require a branch instruction to implement the equivalent Boolean function.

Unit and decimal primitives. Since a strategic goal for Precision Architecture is to support commercial applications, it must handle decimal operations in Cobol-like languages as well as alphanumeric code manipulation. The instruction set includes five instructions for parallel processing of small units (digits, bytes, and halfwords) within a word.[4,5] They are used for word-parallel string search and decimal arithmetic. The halfword units support the processing of 16-bit international character sets.

Unlike floating-point arithmetic, these instructions do not require significant additional hardware. Hardware support consists only of condition logic on the carry bits of each 4-bit group of the ALU. Cobol applications—important on HP3000 machines—have been found to run many times faster on Precision machines than on the previous non-Precision-based HP3000 machines.

Bit field manipulation. Although the main unit of transfer and operation is the 32-bit word, often it is desirable to be able to manipulate arbitrary bit fields within a word or across a word boundary. Examples include the efficient emulation of other instruction sets, bit-block transfers, unaligned byte moves, and field extraction from records.

The shift-merge unit implements efficient bit-field manipulation instructions (see Table 1). For example, Extract takes an arbitrary-length field from any portion of a word and creates a result with this field right justified, with optional sign extension. Deposit does the reverse operation, inserting a right-justified field into any portion of a target word, optionally clearing the rest of the target. Hence, in two instruction cycles Precision can perform an arbitrary bit field movement (see Figure 5). Other architectures usually simulate these instructions by a sequence of shifting and masking.

Extendibility and longevity features

Beyond the simple execution model described above, Precision also includes features designed to give the architecture a potentially longer useful life by allowing growth and extendibility of the architecture. Below, I will describe some of these aspects: the virtual memory model, access protection, the assists architecture, the interrupt system, and the input-output system and multiple processor support.

Virtual memory model. The Precision architects felt that the longevity of an architecture lies in the range of its addressing capabilities rather than in the size of its words or the specific operations implemented. While processing 64-bit integers rather than 32-bit integers might increase accuracy, we did not consider the trade-off in the hardware required for 64-bit datapaths throughout the processor to be cost effective for general-purpose computers.

However, computer usage has clearly tended towards the processing of larger programs and more data. Hence, the key to next-generation architectures is not as much the increase in data size from 32 to 64 bits as the increase in addressing range. Precision provides a 64-bit virtual address range, which is four billion times more virtual storage than in current architectures with 32-bit virtual addresses.[7,9,11]

The large virtual address space allows virtual addresses to be defined globally across processes. This contrasts with architectures where the same address can be used for different objects by different processes. An advantage is that address translation information does not have to change on a process switch. Global virtual addressing allows interacting processes to accumulate a stable working set of address translations despite frequent process switching.

Virtual address manipulation in the processor. Manipulating 64-bit virtual addresses efficiently with a 32-bit data path requires some ingenuity. Using the standard 32-bit ALU, effective address calculation in memory reference instructions is performed for 32-bit quantities to determine the byte offset within a virtual space. The virtual space, selected from one of the eight space registers or the implicit program space register, is then concatenated with the byte offset to give the full virtual address (see Figure 6a). Software conventions are commonly observed for the use of space registers.[4]

Different levels of the architecture are defined with respect to the size of the virtual space implemented: level-0 architecture with no space registers, level-1 architecture with 16-bit space registers, and level-2 architecture with the full 32-bit space registers. This allows the virtual memory to be scaled down for a lower cost Precision processor by reducing the width of each entry in its translation look-aside buffer and attendant data paths.

The architecture also incorporates a concept called "short pointers" to allow handling of 48-bit or 64-bit virtual

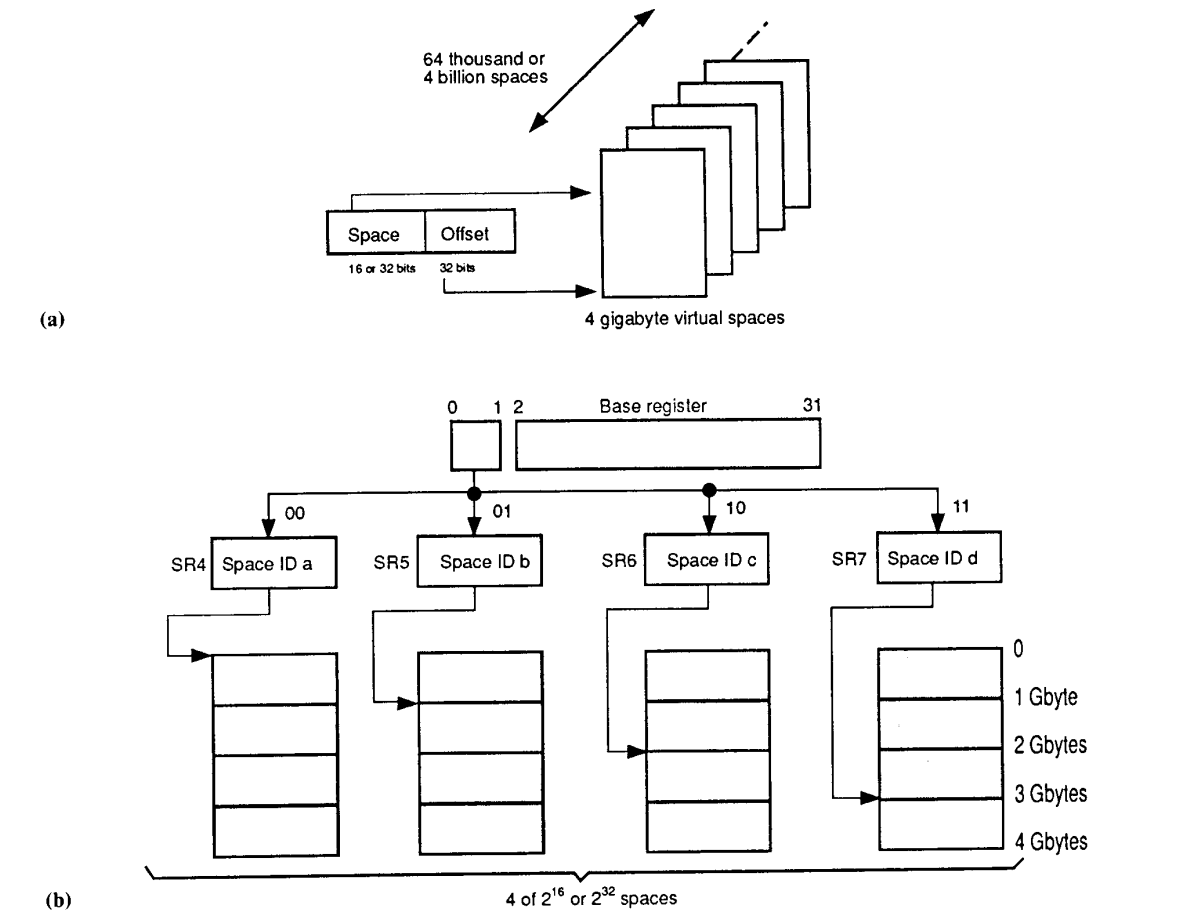


Figure 6. (a) Virtual memory organization and (b) short-pointer addressing.

addresses with short 32-bit pointers (see Figure 6b). It allows, at a given time, data access to four distinct virtual spaces, each space being one gigabyte in size. Long-pointer addressing provides access to four billion virtual spaces, each space being four billion bytes in size (Figure 6a). Short-pointer addressing allows pointers to be the same size—32 bits—as the standard integer data type, a situation often assumed by existing high-level languages like C. It also allows efficient passing of pointers via the 32-bit general registers.

Virtual space management in the memory-disk system. The virtual address is further partitioned into the space identifier, the virtual page number (VPN), and the page offset. Each page has a fixed size of 2 kilobytes. The space identifier and the VPN are translated into a 21-bit physical page number (PPN), which is then used to access physical memory. If the physical page is not in memory, a page-fault occurs, and the missing page is brought in from the disk.

Two software tables are used: a hash table to index into a page directory (Pdir) table, which contains one entry for each physical page in the main memory. Each entry in the Pdir is either empty or contains the VPN of the virtual page mapped to that physical page slot. This has the advantage of reducing the size of the page tables to correspond to the size of the physical memory, rather than to the size of the much larger virtual memory. Both the hash table and the page directory table permanently reside in physical memory for performance reasons.

To speed up the virtual to physical address translation process, a translation look-aside buffer (TLB) is defined as the processor's interface to the virtual memory system. This TLB acts as a cache for virtual to physical address translations. If an address translation is not in the TLB, a TLB miss occurs, handled either by a software interrupt routine or by a hard-wired sequence of operations. The architecture defines memory management instructions for inserting, changing, querying, and deleting entries in the TLB (see Table 1).

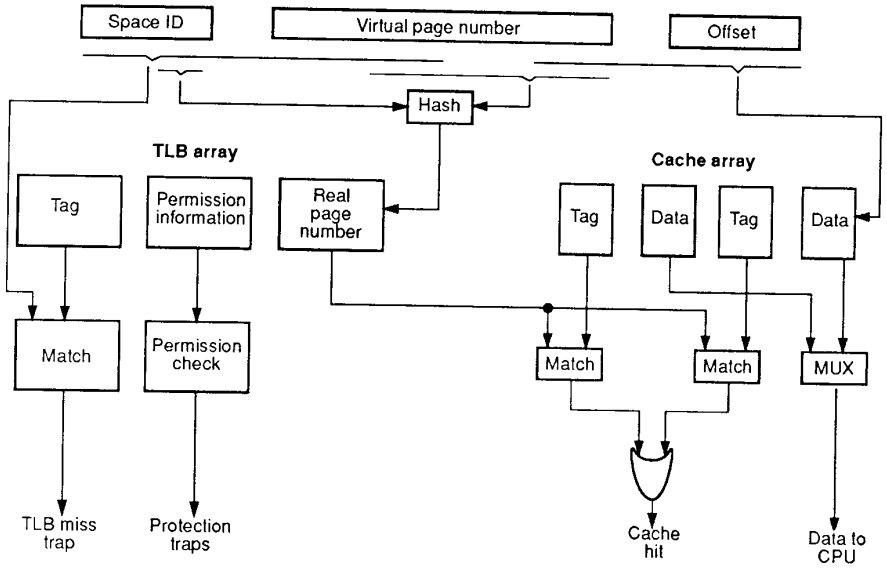


Figure 7. Virtual address translation, protection checking, and cache accessing.

Minimizing paging traffic. A dirty bit defined for every Pdir and TLB entry indicates if the page now differs from its disk image. This dirty bit is cleared to zero when the page is first brought in and when the page is written to disk, and remains clear as long as no writes to the page occur. The first time a program tries to write to that page, a dirty bit update trap occurs, which changes the dirty bit in both the Pdir and the TLB entry from zero to one. This allows the operating system to avoid writing out unmodified pages to the disk. The increase in system performance is well worth the slight overhead in TLB and Pdir management.

Address aliasing. A hardware-software optimization allows virtual cache indexing, which facilitates single-cycle loads from virtual memory. It does this by not allowing software to do address aliasing or mapping of different virtual pages to the same physical page. While address aliasing is of some use to software, it imposes significant performance degradations on hardware because it precludes the use of the virtual page as part of the index into the cache memory.

For example, a virtual access could put data into the cache based on its index, and a later virtual access, using a different (aliased) address, would not find the data in the cache because the index was different in the virtual page portion. The second access would then go to memory, where it might get an inconsistent or stale copy of the data.

By effectively disallowing address aliasing, caches can use the virtual page number as part of the index without causing the stale data problem. This allows the cache to be accessed in parallel with the virtual address translation being done by the TLB (see Figure 7), without restrictions on the size of the cache. If address aliasing were allowed, either virtual address translation and cache accessing would have to be serialized, or the cache size would have to be restricted to that of the page size multiplied by the cache set-associativity.

Access protection. The architecture provides hardware support for access protection to be built into the storage unit and performed in the same cycle as virtual address translation and cache access (see Figure 7).

Precision protection checking is defined at the page level, to control access to the page in three dimensions: the type of access allowed (read, write, or execute), the privilege level at which access is allowed, and the group of processes allowed access to the page. One reason for the choice of a 2-kilobyte page size rather than a larger one is so that access control can be defined at a finer granularity (useful in object-oriented environments, for example).

Privilege levels. For access rights checking, the architecture defines four hierarchical protection rings. The current privilege level of a process is checked against the privilege level for the read, write, or execute access being made to that page by this process.

Generalized supervisor/user transfers. This privilege-level mechanism allows a process to have different access rights over time without the overhead of changing TLB entries when access rights change 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 rather than an interrupt or process switch. This can be done by a procedure call to a Gateway instruction, which branches to the body of the more privileged routine. The

Gateway instruction can promote the privilege level while saving the caller's privilege level in the return address register so that it cannot be "forged" by the caller. On returning to the caller, a privilege-demoting branch instruction is used.

Access identifiers. The currently executing process can claim membership in up to four groups of pages simultaneously, each group having its own access identifier and write-disable bit, saved in four control registers. The access identifier allows each process sharing memory to access different domains in memory without the overhead of changing the TLB on process switch. Four access identifiers are provided to facilitate the controlled transfer of information between logical environments. These four access identifiers are checked against the protection identifier attached to the virtual page being accessed. A protection identifier of all zeros attached to a page allows public access to that page.

When set, the write-disable bit disallows writing for all privilege levels to the pages protected by the associated access identifier. This allows, for example, a single writer and multiple readers for a group of processes accessing a common protected domain of pages.

These protection features built into the architecture allow implementation of very secure, flexible environments. They might not be necessary for single-user or dedicated-controller environments, but they are necessary for efficient implementation of secure multiuser systems.

Assists architecture. One of the goals of Precision Architecture is to define a general-purpose, basic instruction set and allow future instruction set extensions. These future "assist" instructions could then be executed on optional hardware assists to speed up the processing of specialized computations, such as floating-point or graphics. An assist instruction defines the architectural interface between the processor, memory, and any future assist in terms of data movement operations, but we left specific functions performed by an assist for future definition.

Software compatibility. While other architectures define backward software compatibility with a previous instruction set,[10] the Precision assists architecture defines forward software compatibility with future assist instruction sets. In addition, software portability among Precision processors with different configurations of hardware assists is also achieved. These compatibility and portability goals are achieved by means of a transparent assist emulation trapping mechanism that automatically causes an interruption on detecting an assist instruction not supported by a hardware assist. This allows a software trap handler to perform the function required by the assist instruction, using the basic Precision instructions. Critical information needed for emulation is present in the interruption parameter registers, considerably speeding up the emulation routines.

SFUs and coprocessors. The architecture recognizes two classes of assists: special function units (SFUs) and coprocessors. SFUs are viewed as very tightly coupled to the main processor buses, serving as alternate functional units to the ALU or SMU in the execution unit of a basic Precision processor. As such, an SFU receives its operands from the general registers and places its result into a general register, like a basic ALU instruction. A 3-bit SFU identifier is attached to each SFU instruction, allowing up to eight SFUs simultaneously in a system.

We view a coprocessor as a hardware assist coupled to the processor at the level of the data cache or memory. As such, it has its own set of coprocessor registers, loaded from or stored to memory using the same virtual address translation and protection mechanism as the basic processor.

Coprocessor load/store instructions are like processor load/store instructions, except that the target/source registers are coprocessor registers rather than the processor's general registers. Coprocessor registers can be of a different size than processor registers; for example, the floating-point coprocessor registers are 64 bits wide.

Other than the coprocessor load/store instructions, there is only one other coprocessor instruction, where the operations to be performed by the coprocessor can be defined as an instruction extension. As for SFUs, a 3-bit coprocessor identifier is attached to each coprocessor instruction, allowing up to eight coprocessors or 16 logically different assists in a Precision configuration.

While an SFU provides execution-unit extendibility, a coprocessor also provides register-set extendibility.

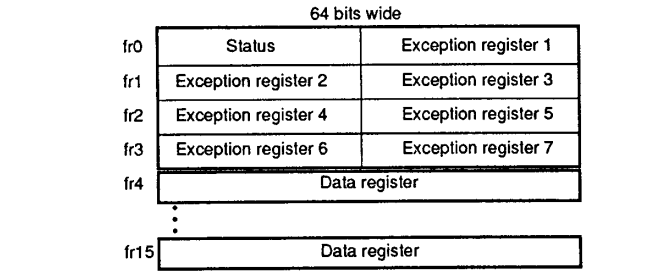
An example of an assist is the floating-point coprocessor (see Figure 8). The Precision floating-point architecture allows highly pipelined implementations. It complies with the ANSI/IEEE 754-1985 floating-point standard, although not all operations and exceptions need to be supported by hardware, since an assists exception trap can be used for software support of unimplemented features.

For complex operations not frequent enough to justify the addition of assists hardware, a software call to a streamlined subroutine—called *millicode*—is used.

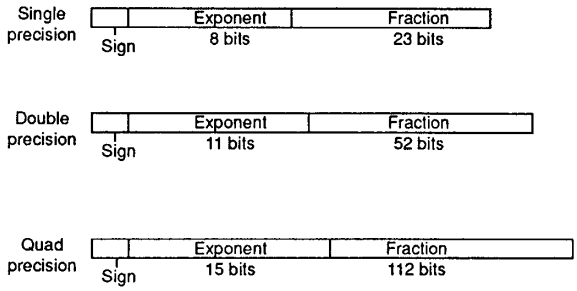
Interrupt system. In Precision Architecture, the term "interruptions" includes all abnormal events like memory faults, protection violations, computation exceptions, hardware malfunction, power failure, timer interrupts, and external interrupts. Synchronous interruptions (those caused by instruction execution) are precise interruptions across all Precision processors, allowing predictability and the leverage of software interruption handlers. Asynchronous interruptions (external to the instruction stream, like machine checks, power failure, and external interrupts) provide a standardized way of reporting malfunctions, saving state, and giving rapid real-time response to external conditions and requests.

Interruption registers. The interesting aspects of the Precision interrupt system are probably the ways in which the interruption registers are used for fast context switching, expediting interruption processing, and implementing precise interruptions even with delayed branching. There are six control registers (Figure 3) used to save state, such as the processor status word (PSW) of the interrupted program, the instruction causing an interruption, the virtual space and offset for data memory reference instructions, and the virtual spaces and offsets of the first two instructions processed upon returning from interruption servicing.

Fast context switch. Interruption servicing is implemented as a fast single-cycle context switch rather than a complete process swap. The information in the interruption registers is usually continuously updated by a Precision processor during normal instruction processing so that, on detecting an interruption, the processor only has to save the current PSW in the interruption PSW register, clear the current PSW, and pass the control flow to a vectored location in a dynamically relocat-



(a)



(b)

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

(c)

Figure 8. Floating-point coprocessor, including (a) floating-point registers, (b) floating-point data types, and (c) floating-point coprocessor instructions.

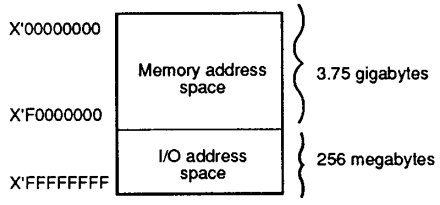


Figure 9. Physical address partitioning.

able interruption vector table. Clearing the PSW has the effect of disabling other interruptions, freezing the interruption registers, and enabling real mode addressing.

Eight control registers can be used as scratchpad registers for quick general-register saves under privileged software control. Upon completion of interruption processing, a Return From Interrupt instruction is executed, which restores the saved processor state and restarts execution at the interrupted instruction.

Precise interruptions with delayed branching. Delayed branching has been known to cause difficulties in interruption processing. Precision has easily solved this with the interruption instruction address (IIA) queue. The IIA queue consists of two instruction-return addresses, which are the first two instructions processed upon returning from the interruption. Interruptions caused by branch instructions are always taken after the branch instruction completes.

The hardware automatically collects in the IIA queue the addresses of the delay slot instruction, followed by the target instruction of the branch (generally non-contiguous addresses). Since the IIA queue saves the return addresses of the time-sequential instructions following an interruption, there is no restriction on a branch instruction occurring in the delay slot of another branch instruction.

Flexible external interrupts. There are 32 external interrupt classes, each of which can be individually masked by privileged software. When an external interrupt occurs, its corresponding interrupt pending bit is set in the external interrupt request register. If the corresponding mask bit in the external interrupt mask register is also set, an external interrupt is taken.

Debugging and diagnostic hooks. The architecture provides debugging support traps to aid in software development. A Break instruction can be used to insert software checkpoints anywhere in the code, causing a break trap when executed. This instruction allows software encoding of bits within the instruction, which will be ignored by the hardware but interpreted by the software in the Break trap handler.

Pages can also be tagged by two trap-enable bits that cause a trap whenever any reference is made to that page, or only whenever a store is made to that page. Traps can also be enabled whenever a

branch is taken, or whenever the privilege level of the running process is promoted or demoted.

A recovery counter is defined to facilitate the implementation of fault recovery in software rollback schemes and for single-step debugging. It can be enabled to cause an interrupt after the execution of a predetermined number of instructions.

Precision Architecture also includes a Diagnostic instruction, whose only defined field is the 6-bit major opcode field. The rest of the instruction can be defined for implementation-specific operations, like accessing pipeline registers or implementation-specific mode bits, not otherwise directly accessible by software. This instruction has proven very useful in boot-up, self-test, and diagnostic routines.

I/O system and multiple processor support. The architecture defines a memory-mapped input-output system, with I/O devices mapped to the top sixteenth of the four-gigabyte physical address space (see Figure 9). A Precision I/O module can be interrogated and controlled by software via load and store instructions. I/O addresses are not cached, and software maintains cache coherency for direct memory access by means of explicit cache control instructions.

A simple semaphore operation, Load And Clear Word, resembles the test-and-set indivisible operation in earlier architectures.[10] Instructions for purging and flushing the translation look-aside buffers and caches allow software to maintain TLB and cache coherency when necessary.

Bus standards have also been defined for hardware-managed TLB and cache coherency, in which case software sees only a single cache and a single TLB. The architecture does not constrain the type of asymmetric multiple processor support implemented by the total hardware-software system.

Precision processor implementations

While describing the range of Precision products is beyond the scope of this article, I will give some processor references and describe a typical pipeline.

First-generation Precision processors have been implemented in a variety of technologies, including transistor-transistor logic,[12] n-type metal-oxide semiconductor,[13] complementary metal-oxide semiconductor,[14] and emitter-coupled logic (in a prototype), with a variety of clock speeds, cache support, and TLB support, over a range of performance and cost. These processors are used in both the HP3000 series 900 business computer line and the HP9000 series 800 technical computer line.

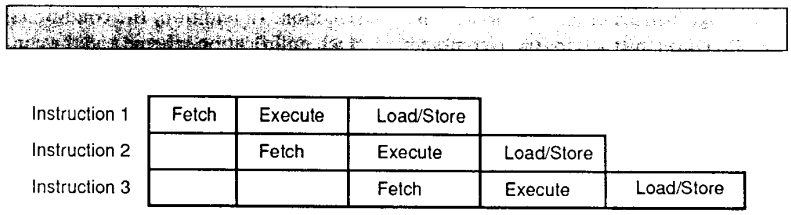


Figure 10. Typical pipeline.

Figure 10 shows a typical pipeline for a Precision processor.[12] In the Fetch stage, the fetched instruction is decoded at the same time as the reading of the general registers. During the Execute stage, the operands are routed through the ALU or SMU, where a functional operation or address calculation occurs, and the condition is also evaluated if necessary. At the end of the Execute stage, the result is stored in the general registers and also bypassed to the next Execute stage if necessary.

There are no pipeline penalties for delayed branching, since the target address is calculated during the Execute stage at the same time the condition is evaluated for a conditional branch. This is done in time to fetch either the target instruction or the sequential instruction in the next cycle. Similarly, there are no pipeline penalties for load or store instructions, except when the register being loaded is used in the immediately following instruction. This situation is minimized by Precision optimizing compilers using code reordering.

Attainment of the SPECTRUM goals

In the spirit of architectural acronyms,[2,3,7,9,11] I will take the liberty of defining a SPECTRUM architecture as one with the following goals:

- Scalable implementations
- Price-performance advantages
- Extendible architecture
- Commercial applications
- Technical applications
- Reusable components
- Unconstrained lifetime
- Multiple environments

This makes Precision a SPECTRUM architecture, since the above goals include most of the major ones enunciated for its design. These goals are more similar to those addressed in the design of architectures like the IBM 360/370 architecture[10] and the DEC VAX architecture[9] rather than to those of RISC microprocessor architectures.[2,3,7,11]

However, the Precision execution model shares many features common to these RISC architectures.[2,3,7,8,11] These include features like register-based execution, simple load-store interface to memory, delayed branching, simple addressing modes, fixed-length instructions, and three-register nondestructive functional instructions. Such architectural features can usually be implemented with simple, pipelined processor hardware, where single-cycle execution is achievable for most instructions. Since the hardware requirements are simple, these architectures are scalable in the sense that they can be implemented by low-cost hardware or by higher cost, higher performance processors, which have very fast processor clock frequencies. A variety of process technologies with different densities, speeds, and costs can be used.

Since Precision instructions are executable in a single cycle by simple hardware, we can say that the architecture has price-performance advantages—more instructions can generally be executed in a given amount of time by less costly processors than in architectures with large, complex instruction sets.[9,10] However, simply executing more instructions in a given amount of time does not necessarily imply that more useful work is accomplished, especially if very little is accomplished in

an average instruction.[2,3,7,11] For this reason, Precision instructions try to combine frequent instruction pairs, like Compare and Branch, into one instruction. This saves both static code space and dynamic execution time, since one instruction replaces two and only one execution cycle is needed for both operations.

In fact, all Precision functional instructions have a built-in skip operation; memory reference instructions can have base register address modification operations; and conditional branch instructions combine both the condition generating operation and the branch operation in one instruction. In addition, the conditional branch nullification scheme and the conditional trapping scheme allow both static and dynamic code optimizations for looping, jumps to error routines, and range checking. The use of maximal-length immediates also helps to reduce the number of load instructions and the execution time involved.

Note that, in computer systems with disks, tapes, graphics accelerators, and other I/O devices, the cost of the processor subsystem is important, although not necessarily the dominating factor in system cost. Similarly, the performance of the processor subsystem is important, but not necessarily the dominating factor in system performance.

The Precision assists architecture allows flexible instruction-set extendibility without sacrificing software compatibility. In fact, the built-in assists emulation trap allows software to be compatible among Precision systems with different configurations of hardware assists and even with future, as yet undefined, assists. The Diagnose instruction also allows implementation-specific instruction-set extensions. This is useful for implementing reliable and serviceable systems.

The fact that Precision processors have been used in both the commercial HP3000 and the technical HP9000 product lines attests to the general-purpose nature of the architecture. Decimal operations are supported for Cobol applications, while efficient coprocessor integration contributes to high performance for floating-point applications. The most frequently used Precision instructions are quite different for Cobol, Fortran, or C applications.

The Precision machine models have leveraged or reused key components like hardware VLSI processors, floating-point processors, cache and TLB controllers, and standard bus controllers. Both the HPUX (Unix) and MPE-XL operating systems can run, unmodified, on all Precision processor systems. By defining not only user-visible architecture, but also system-visible architecture, Precision Architecture has defined not just an applications binary interface,[7] but also a systems binary interface. Naturally, when object code is compatible at the most privileged systems level, it is also compatible at the least privileged user-applications level. Precision Architecture, together with Precision bus standards, has provided the potential for streamlining software, hardware, and input-output developments.

The longevity of Precision Architecture is certainly unconstrained by its large 64-bit virtual address space, since this is four billion times larger than current 32-bit virtual address architectures.

The virtual memory structure and the built-in access protection features allow the implemention of multiple operating environments, since a large amount of addressability is provided, with provisions for various kinds of access control and protection for different domains of pages. The flexible bit-manipulation features enhance the emulation of older instruction sets, contributing to easy migration from

older HP machines. The external interrupt system allows environments requiring fast, real-time response to asynchronous events. The access protection features, machine checks, power-fail interrupt, and diagnostic features provide hooks for implementing secure, highly reliable, and serviceable systems.

Precision Architecture has a simple execution model, where each instruction can be executed in a single cycle by a simple, scalable processor. This is enhanced by code compaction and execution time reduction features for the efficient processing of frequent operation sequences. The architecture provides a 64-bit virtual address space to support growing user needs and flexible protection mechanisms to implement secure multiuser systems. Moreover, the assists architecture provides forward compatibility with new instructions and register sets that can be added, with these instructions executed transparently by either hardware assists or software emulation.

Precision Architecture provides a systems binary interface for software compatibility at both applications and systems levels. It forms the basis for the consolidation of hardware and software production. The architecture has been refined through extensive performance measurements and analysis and tested against various hardware implementations and software environments. It combines successful architectural ideas evolved from the past with several innovative features to support both current computing needs and future cooperative computing environments. □

Acknowledgments

The Precision program, initially called "Spectrum," was started by Joel Birnbaum at Hewlett-Packard Laboratories, Palo Alto. The original architecture design team consisted of Allen Baum, Hans Jeans, Russell Kao, Michael Mahon, Terence Miller, Steve Muchnick, William Worley, and myself. This team surveyed the ground, set all the major architectural directions, and produced the first version of the HP Precision Architecture in 1982. Subsequently, many individuals contributed to the refinement of the architecture, including Steve Boettner, Bill Bryg, Mike Fremont, Dave Fotland, Carol Thompson, Craig Hansen, Jerry Huck, Dave James, Dan Magenheimer, and too many others to list here completely. The architecture would not have existed in its present form without their efforts.

The program flourished with the continuing support of John Young and Dean Morton. I would also like to acknowledge the dedicated Hewlett-Packard implementation team members who worked on the Precision processors, coprocessors, operating systems, compilers, I/O systems, bus standards, performance measurement and analysis, debuggers, databases, networks, graphics enhancements, VLSI technology, CAD tools, manufacturing, marketing, sales, support, and management. It is through their efforts that the Precision Architecture has been instantiated in a spectrum of real-world machines, environments, and applications.

References

1. J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard J.*, Vol. 36, No. 8, Aug. 1985.
2. D. Patterson, "Reduced Instruction Set Computers," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 8-21.
3. J. Hennessy et al., "MIPS: A Microprocessor Architecture," *Proc. Micro-15*, IEEE, Oct. 1982.
4. M.J. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard J.*, Vol. 37, No. 8, Aug. 1986, pp. 4-21.
5. K.W. Pettis and W.B. Buzbee, "Hewlett-Packard Precision Architecture Compiler Performance," *Hewlett-Packard J.*, Vol. 38, No. 3, March 1987, pp. 29-35.
6. J.A. Lukes, "HP Precision Architecture Performance Analysis," *Hewlett-Packard J.*, Vol. 37, No. 8, Aug. 1986, pp. 30-39.
7. R.B. Garner et al., "The Scalable Processor Architecture (Sparc)," *Proc. 26th Compcon*, 1988, pp. 278-283.
8. G. Radin, "The 801 Minicomputer," *Proc. SIGArch/SIGPlan Symp. Architectural Support for Programming Languages and Operating Systems*, ACM, Palo Alto, Calif., March 1982, pp. 39-47.
9. W.D. Strecker, "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proc. NCCC*, June 1978, pp. 967-980.
10. *IBM System/370 Principles of Operation*, Form No. GA22-7000, IBM, Poughkeepsie, N.Y., 1970.
11. J. Moussouris et al., "A CMOS RISC Processor with Integrated System Functions," *Proc. 31st Compcon*, March 1986, pp. 126-131.
12. D. Fotland et al., "Hardware Design of the First HP Precision Architecture Computers," *Hewlett-Packard J.*, Vol. 38, No. 3, March 1987, pp. 4-17.
13. S. Mangelsdorf et al., "A VLSI Processor for HP Precision Architecture," *Hewlett-Packard J.*, Vol. 38, No. 9, Sept. 1987, pp. 4-11.
14. A. Marston et al., "A 32b CMOS Single-Chip RISC Type Processor," *Proc. IEEE Int'l Solid-State Circuits Conf.*, Feb. 1987, pp. 28-29.

Ruby B. Lee is a manager of a VLSI processor design team at Hewlett-Packard. She is a founding member of the Precision Architecture program and a principal designer of the processor architecture, the assists architecture, and a VLSI processor implementing Precision Architecture. She is the primary inventor of four patents on Precision Architecture, with more patent applications in progress.

Lee has a BA from Cornell University, an MS in computer science from Stanford University, and a PhD in electrical engineering from Stanford. She has written several papers on parallel processor organizations, performance analysis, computer architecture and design, novel VLSI architectures, testing, and testability.

Readers can contact the author at Hewlett-Packard Company, Bldg. 42U7, 19447 Pruneridge Ave., Cupertino, CA 95014.