64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture

Ruby Lee and Jerry Huck
Hewlett-Packard Company
19410 Homestead Road
Cupertino, CA 95014
rblee@cup.hp.com, huck@cup.hp.com

Abstract

This paper describes the architectural extensions to the PA-RISC 1.1 architecture to enable 64-bit processing of integers and pointers. It also describes MAX, the Multimedia Acceleration eXtensions which speed up the processing of multimedia and other applications with parallelism at the intra instruction, or subword, level. Other additions to the PA-RISC 2.0 architecture include performance enhancements with respect to memory hierarchy management, branch penalty reduction, and floating-point performance.

1. PA-RISC architecture evolution

When the original PA-RISC 1.0 Architecture was designed in the early eighties, its goal was to be a single architecture that efficiently spans Hewlett-Packard's three computer lines: the HP3000 commercial minicomputers, the HP9000 technical workstations and servers, and the HP1000 realtime controllers. Before introduction, the program was codenamed SPECTRUM. At introduction in 1986, it was known as HP's Precision Architecture [1,2], HP-PA, or just PA. Subsequently, the architecture was called PA-RISC, with the first version of the architecture known as PA-RISC 1.0.

Since its introduction, the PA-RISC architecture has remained remarkably stable. Only minor changes were made over the next decade, to facilitate higher performance in floating-point and system processing. When PA-RISC 1.0 was designed, floating-point performance was not essential for the majority of the HP computer systems targeted at that time. Hence, the architecture defined floating-point support as optional coprocessor instructions, without emphasizing high performance. In 1989, driven by the performance needs of the HP9000 technical workstation line, PA-RISC 1.1 was introduced. This included additional floating-point capabilities, such as more floating-point registers, doubling the amount of register space for single-precision floating-point numbers, and introducing combined operation floating-point instructions[3]. These floating-point features enabled higher performance in technical computations, including graphics, where single-precision floating-point numbers are extensively used.

In the system area, PA-RISC 1.1 architectural extensions were made to speed up the processing of performance-sensitive abnormal events, such as misses in the address translation cache (also called the TLB). Such architectural changes are only visible to the operating system, and do not affect any applications programs. Minor system changes have been added to the three editions of the PA-RISC 1.1 architecture, known as editions 1, 2 and 3, respectively, of the architecture manual [3,4,5].

PA-RISC 1.1 also added bi-endian support. Previously, PA-RISC 1.0 was a consistently big endian machine, but in PA-RISC 1.1, support for little endian was also provided by means of a mode bit.

The PA-RISC 2.0 architecture represents the first time that user-visible changes have been made to the core integer architecture. In addition to support for 64-bit integer data and 64-bit flat addresses, other user-visible changes have also been added to enhance the performance of new user workloads. For example, Multimedia Acceleration eXtensions (MAX) have been added to speedup multimedia processing by software running on the main processor, rather than on separate optional hardware. Some additional floating-point and system-level changes have also been added. However, the principal of keeping the programming model stable has been carried forward as much as possible in the 64-bit version of the architecture, which we will denote PA2 in the rest of this paper. PA-RISC 1.0 and PA-RISC 1.1 versions of the architecture are denoted PA1 in the rest of this paper.

In section 2 below, the design requirements for PA2 are outlined. In section 3, we describe the 64-bit extensions for both data and addresses. In section 4, we describe the multimedia extensions. In section 5, we describe other performance extensions.

2. PA-RISC 2.0 requirements

PA1 was designed with the basic word size of 32 bits. The word size determines the width of general registers and datapaths for integer and address calculations. The support for virtual addresses and floating-point was already 64 bits, even in the original architecture definition. The basic goal of PA2 is to extend the PA-RISC architecture to a word size of 64 bits, for integers, physical addresses and flat virtual addresses.

32-bit general registers and addresses with a maximum of 2^{32} byte objects become limiters as physical memories larger than 4 Gbytes become practical. Some high-end applications are exceeding the 4 Gbyte working-set size. Enabling future systems with higher capacity and cost-effective use of this extra capacity is required.

Another requirement is to maintain complete binary compatibilty. That is, the binary representation of existing PA1 software programs must run correctly on PA2 processors.

The transition to 64-bit architectures is unlike the previous 32-bit microprocessor transition, which was driven by an application pull. By the time that technology enabled cost-effective 32-bit processors, many applications had already outgrown 16-bit size constraints, and were "coping" with the 16-bit environment by awkward and inefficient means.

With the 64-bit transition, fewer applications need the extra capabilities and many applications will choose to forgo the transition. Due to cache memory effects, if an application does not need the extra capacities of a 64-bit architecture, then it can achieve greater performance by remaining a 32-bit application. 64-bit architectures are still necessary since some crucial applications - e.g. databases and large scale engineering programs - and the operating system itself will need the extra capacities enabled by 64-bit architectures.

This analysis leads to the requirement that 32-bit applications are important and must not be penalized when running on the 64-bit architecture. 32-bit applications will remain a significant portion of the execution profile and should also benefit from the increased capabilities of the 64-bit architecture without being ported to a new environment. Of course, it is also a requirement to provide full performance for 64-bit applications, and extended capabilities that are enabled by a wider machine.

Another requirement is to provide significant performance enhancements for new applications in the workload profile, and new computing environments that will be common during the lifetime of PA2. For example, the shift in the workloads of both technical and business computations to include an increasing amount of multimedia processing led to the Multimedia Acceleration eXtensions (MAX) which are part of the PA2 architecture. Previously, a subset of these multimedia instructions were included in the PA7100LC processor, an implementation of the PA-RISC 1.1 architecture, as implementation-specific features [6,7,8].

A final requirement is to introduce the 64-bit extensions without disrupting the user community's understanding of the architecture. It is valuable to build on how mechanisms work in PA1 processors and naturally extend that definition. This requirement is usually easily satisfied since it is often more awkward to deviate from the current models.

Maintaining compatibility, enabling full 32-bit performance, full 64-bit performance, new workload support and increased performance capability present the architects with several fundamental tradeoffs. The next several sections describe how these tradeoffs were resolved and the resulting definition.

3. 64-bit extensions

3.1. 64-bit computation extensions

The PA-RISC computational model is extended to 64-bits for the general register file, integer computation and address formation in PA2. These 64-bit extensions provide instructions to operate on 64-bit operands with the same capabilities that are provided for 32-bit operands. In a few cases, some functionality is not extended to 64-bit elements (e.g.: **divide step**) and in some cases, new capability is provided (e.g.: multimedia instructions). Addition, subtraction, logical operations, condition formation for branches and nullification, and bit manipulation (**extract** and **deposit** instructions) are extended in the obvious ways.

Not so obvious is how to maintain 32-bit element compatibly in a 64-bit container. There are a few approaches. One strategy is to provide a duplicate set of instructions that operate on 32-bit operands and produce consistent 64-bit results. This requires a modified ALU to compute a different kind of result (e.g., sign-extend across 32 bits), which may increase the cycle time of the processor. In this model, comparisons and condition generation instructions need only look at the 64-bit result. Computing a result from different sized operands will require an additional instruction to sign-extend one of the operands.

Another approach is to include a mode bit that changes the interpretation of all instructions to operate only on one size data producing the same size result, either all 32 bits or all 64 bits. While no new instructions are required, this strategy restricts the mixing of 32-bit and 64-bit data.

A final strategy, and the one chosen by PA-RISC, is to treat all operands and results as 64-bits in size and provide extra instructions when a 32-bit interpretation is needed. This keeps the simplest 64-bit ALU design to enable high frequency. It does require a replication of some shift instructions and conditional branch comparisons that use 32-bit operands.

One additional complication is the PA-RISC conditional skip capability that is present in most computation instructions. The condition needs to know if the operands are 32-bits or 64-bits in size. Fortunately, the PA1 architects had carefully left a free subop encoding bit in the ALU instructions. In PA2, this is called the d-bit, used to distinguish between 32-bit and 64-bit conditions, overflows and carry, for the ALU instructions.

Another important issue is whether 32-bit load data is sign- or zero-extended. Sign extension simplifies the mixing of 32-bit and 64-bit data and addresses. Unfortunately, it also adds sign extension circuitry to the critical cache load path. Getting a higher frequency is worth the one additional instruction needed for sign extension when mixing 32-bit and 64-bit operands.

The floating-point register file is separate from the general register file and is already 64-bits in size. 64-bit signed integer formats already existed in the floating-point register file. Compatibility is easily achieved by replicating the few integer conversion operations. One small complication is the need to introduce an unsigned integer format. Conversions of 32-bit unsigned integers are supported with the existing 64-bit signed conversions (truncated overflows). To support 64-bit unsigned operands, PA2 includes unsigned integer to floating-point conversion instructions. Otherwise, synthesizing an unsigned convert from a signed convert requires several instructions.

A 64-bit integer multiply is not defined since the existing double-precision multipliers are only 53x53-bit multipliers. Widening to 64-bits would create extra delays for floating-point operations. 64-bit integer multiplies can be synthesized from the existing 32-bit multiply instruction.

3.2. 64-bit addressing extensions

The PA1 virtual addressing model combines a space identifier and an offset, by concatenation, to form a global virtual address that is shared by all processes [2]. Space identifiers are resources mostly managed by the operating system. They specify where an address space begins. Extending the offset to 64-bits is straightforward. This gives each process 2^{64} bytes of flat virtual address space. The Space identifier will determine where in the much larger global address space this region is. Protection is provided by using keys, called protection identifiers.

Another consideration in the PA2 definition is the interaction between the computation model for 32-bit compatibility and address formation. If the upper bits of a register that contains a 32-bit pointer or index is always a known value, then the entire 64-bits can be used as an address in address computation. If the upper bits can be polluted with overflows and other side-effects, then an extra operation to clear (or maybe sign extend) these bits is needed.

Since PA2 extends most computation instructions to produce all 64-bits of the result as if the operands were 64-bit elements, a special mechanism - an address mode bit - is introduced to maintain compatibility in address formation. This is stored in the processor status word (PSW) as the W(ide) bit. When zero, the upper bits of the address are forced to zeros, and space register selection is from the two most significant bits of the 32 lower bits. When one, all 64-bits participate in address formation, and space register selection is from the two most significant bits of all 64 bits. This mechanism allows existing 32-bit PA1 programs to run, unchanged, on the new PA2 processors. It also means that 32-bit programs do not pay any penalty in extra instructions when compiled for a PA2 processor.

Since the mode bit only affects address formation, 32-bit applications still have access to 64-bit data registers and the full 64-bit computation instructions. For example, a 32-bit program compiled for PA2 processors can use 64-bit registers to support 64-bit objects, and libraries can be written to use the extended capabilities of the PA2 architecture.

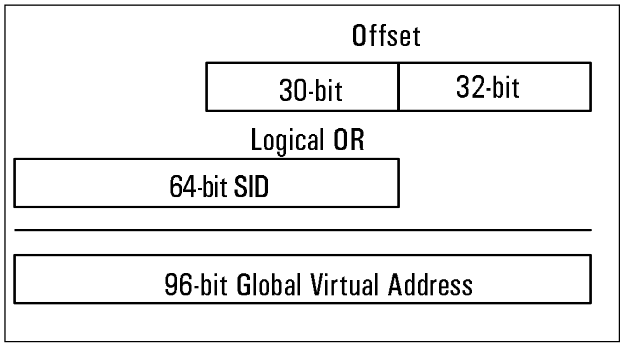

Figure 1: PA2 virtual addressing model

Address formation for 64-bit programs uses the upper two bits of the base register to select a Space register, much like the PA1 "short-pointer" address model[2]. The base register plus an immediate or index register forms the space offset. However, while PA1 concatenates the Space identifier and the offset, in PA2, the Space identifier overlaps the upper bits of the offset and is logically OR'ed together with these upper bits (see figure 1). This leaves the position of the space register in the same relative place to form the global virtual address.

The overlap region is only 30 bits wide. The upper 2-bits of the 64-bit offset are not included in the global virtual address formation since they are used to select the Space identifier. From the hardware perspective, the overlapped 30 bits of the offset are AND'ed with the PSW W-bit. The AND's output is then OR'ed with the lower 30 bits of the Space identifier. The W-bit acts to zero the upper offset bits when in 32-bit address mode and to pass those bits unchanged when in 64-bit address mode. PA2 processors may implement up to 64-bit Space identifiers, which allows 2^{96} bytes of global virtual address space.

While the hardware allows an arbitrary offset and Space identifier to be combined, software conventions will create non-overlapping regions of the global vitrual address space, in which the offset and Space identifier will be disjoint parts of the virtual address. Programs that violate the conventions and address outside their permissible address space will be caught by the protection identifier mechanism.

An alternative to this approach would have been to not overlap the offset and Space identifier, and just use a single AND gate with the W-bit. The AND/OR circuit is slightly more expensive but provides better utilization of the TLB by allowing variable-sized spaces, rather than just fixed-sized spaces. Each TLB entry must include all the address bits used on the lookup.

The PA-8000 processor [9], the first PA2 implementation, has 32-bit Space identifiers which allows 2^{64} bytes of global virtual address space. On the PA8000, the operating system might choose to allocate a single large space for all global shared objects (say 2^{62} bytes in size), allocate 16K spaces for 2^{48} byte "large" processes and reserve the remainder (roughly 2 billion) of the spaces for 32-bit applications. The large processes have space identifiers with the low 16 bits zero and never generate offsets with more than 48 bits of significance.

The management of very large amounts of physical memory also requires consideration of the basic page size. The size of the processor's TLB is not growing as quickly as the physical memory size. In some cases, the TLB size is being trimmed to meet cycle time constraints. Uniformly increasing the page size is one approach, but an alternative is to allow variable-sized pages. The implementation of an associative TLB is easily adapted to allow variable-sized pages. Entries specify their page size and the hardware does the appropriate masking of the virtual address bits. PA2 defines eight page sizes ranging from the existing 4 Kbyte pages to 64 Mbytes by factors of 4. System software can choose to use any or all of these different sizes.

These address mechanisms extend the PA1 architecture to allow complete compatibility, more flexible address space management, and a simple hardware implementation. The width and number of the protection identifier registers has also been increased to enable the full utilization of these features.

Table 1: PA2 features for 64-bit support

New PA2 feature	Motivation
d-bit in ALU instructions (32-bit vs 64-bit conditions, overflow, or carry-in)	mixed-mode (32-bit and 64-bit) computations and data
Some replicated branch and shift instructions with immediates	mixed-mode data and conditions
W-bit in PSW (Wide)	32-bit vs. 64-bit pointers
variable-sized spaces	more flexible inter-space management, and fewer bits per TLB entry
variable-sized pages	more flexible intra-space management, and fewer TLB entries
larger protection identifiers	more flexible protection regions
more protection identifier registers	more efficient management of protection identifiers
load/store double (64 bits)	64-bit memory access
Conversion between unsigned integers and floating-point	Efficient integer handling in floating-point unit
Space crossing branches	Performance in dynamic libraries
longer displacement call type branch instructions	Reduced overhead in large programs (ex. Databases)
longer displacement load and store instructions (both GRs and FRs)	greater reach in base plus displacement addressing

3.3. Mixed-mode execution

Mixed-mode execution can refer to the mixing of 32-bit and 64-bit applications and operating systems, to the mixing of 32-bit and 64-bit data and computations in a single application, or to the mixing of 32-bit and 64-bit data and pointers in a single application. PA2 supports the first two definitions. The third definition requires too many software restrictions to work efficiently under all scenarios. Enabling it architecturally is insignificant compared to the software changes needed to ensure correctness and efficiency.

In the transition from 32-bits to 64-bits, the ability to run 32-bit applications on existing 32-bit operating systems or new 64-bit operating systems is a key compatibility requirement, and is fully supported by the new architecture. 32-bit applications may call a 64-bit operating system. The W-bit is changed from 0 to 1 to enable this transition from 32-bit pointers to 64-bit pointers. Of course, 32-bit applications may call 32-bit operating systems and 64-bit applications may call 64-bit operating systems. The W-bit stays at 0 in the former case, and at 1 in the latter case. It is not meaningful for a true 64-bit application to call a 32-bit operating system, since the 32-bit operating system would not be able to handle the larger 64-bit data and flat addresses used by the 64-bit application.

In the case of mixed-mode data, PA2 supports simultaneous computation on both 32-bit and 64-bit data. A 32-bit application has access to 64-bit data and operations, and a 64-bit application has access to 32-bit data and operations. Note however, that this architectural flexibility could be reduced by other software conventions; for example, if the procedure calling convention does not allow 64-bit callee-save registers in a 32-bit application.

Table 1 summarizes the additions to the architecture for 64-bit data and addresses. The last two entries in Table 1 provide increased reach in base-plus-displacement addressing, especially useful since programs are getting larger and their data regions are also growing.

4. Multimedia acceleration extensions

Workload characteristics have changed to include an increasing amount of digital multimedia processing. This includes the use of digital images, speech, 3-D graphics, audio and video, in addition to the text, numbers and 2-D graphics commonly handled by computers in the past. Since most digital multimedia data is compressed for more economical storage and transmission, merely reading and writing multimedia data can involve compute-intensive decompression and compression processing.

Multimedia Acceleration eXtensions (MAX) are a small set of instructions motivated by the desire to accelerate multimedia processing by software rather than by special-purpose hardware. General-purpose processors are increasing rapidly and steadily in performance so that using software running on these processors to implement functionality previously possible only with specialized add-in boards seems feasible. The goal is to introduce instructions that provide significant performance improvement with insignificant impact on the area, cycle-time and design time of the PA-RISC processor.

Like other word-oriented processors, PA-RISC supports data that is larger or smaller than the word size. For computations on integers larger than words, multi-precision arithmetic is supported with a few simple instructions. For computations on integers smaller than words, the least significant bits of the registers and datapath are used. For example, computation on 16-bit integers in a 64-bit processor essentially wastes three quarters of the 64-bit registers and integer datapaths. Many digital multimedia representations use 8-bit, 12-bit or 16-bit data for the basic pixel component or audio sample. Hence, speeding up the processing of subword data is a promising, cost-effective approach to accelerate multimedia programs.

MAX allows the parallel processing of packed subwords in a word-oriented processor. For example, four 16-bit subwords can be packed into a 64-bit word in PA2 processors. By merely blocking the carries at each 16-bit boundary, a 64-bit adder can generate four 16-bit adds in a single cycle. When the carries are not blocked at these three subword boundaries, the adder performs the usual 64-bit add function. Hence, a parallel 16-bit add instruction uses the same resources as a standard 64-bit add instruction. It also reads two 64-bit general registers, takes a pass through the ALU, and writes one 64-bit result register. The only incremental overhead is the decoding of a new instruction and the equivalent of three AND-gates to block the carries at the appropriate subword boundaries in the ALU. The MAX instructions speed up a multimedia program by reducing its pathlength (or number of instructions executed) [10].

Table 2 summarizes the MAX instructions in the PA2 architecture.

4.1. Parallel subword instructions

The most common arithmetic operations needed to accelerate multimedia processing by a microprocessor, with pre-existing integer and floating-point support, is somewhat different than for a Digital Signal Processor (DSP) or a standalone media processor. In DSPs, the multiply-accumulate operation is often a key primitive instruction. But for many microprocessors, such as PA-RISC, this function is already available in the floating-point unit. Often, the audio processing and graphics transformations that depend on this function are already adequately handled by single-precision floating-point instructions.

Other pixel-oriented computations, like image and video processing and graphics rendering, can be sped up considerably with additional microprocessor instructions. Here, the most frequent operations are often the simpler add and subtract operations. Hence, Parallel Add, and Parallel Subtract instructions are introduced.

The parallel add and subtract each have three variants, which specify what happens on an overflow. The default action is modular arithmetic, where any overflow is discarded. If signed saturation is specified by an instruction completer, an overflow causes the result to be clipped to the largest or smallest signed integer in the result range, depending on the direction of the overflow. Similarly, if unsigned saturation is specified, an overflow causes the result to be clipped to the largest or smallest unsigned integer in the result range[6,7].

Table 2: Multimedia instructions in PA2

Parallel Subword Instruction	**Motivation**
Parallel add with modular arithmetic with signed saturation with unsigned saturation	basic operation, where saturation speeds up and simplifies overflow handling
Parallel subtract with modular arithmetic with signed saturation with unsigned saturation	as above
Parallel shift left & add with signed saturation	multiply by integer constant
Parallel shift right & add with signed saturation	multiply by fractional constant
Parallel average	arithmetic mean; division by 2
Parallel shift right arithmetic (propagates sign)	data alignment; divide signed integer
Parallel shift right logical	data alignment; divide unsigned integer
Parallel shift left	data alignment
Mix	rearrange subwords from two source registers
Permute	rearrange subwords from one source register

Often, the multiplications required are by constants. This is sped up with Parallel Shift Left and Add, and Parallel Shift Right and Add instructions. These two instructions are very effective in implementing multiplication by integer or fractional constants, respectively. They require just a minor modification to the existing preshifter to the integer ALU, rather than a whole new multiplier functional unit on the integer datapath.

The integer divisions are also often very simple ones. For example, divide by two is commonly needed. This is sped up with the Parallel Average instruction, which adds the two operands, then performs a divide by two. This involves a right shift of one bit. In the process, the overflow bit is shifted in as the most significant bit of the result, so the instruction has the added advantage that no overflow can occur.

Divisions by a power of two can also be done in parallel by using the Parallel Shift Right (Arithmetic or Logical) instructions. These instructions may be used for division of signed and unsigned subwords, respectively. They use the existing 64-bit shifter, but *block* any bits shifted out from one subword from being shifted into the adjoining subword. Similarly, division by a constant can be simulated with a combination of the above instructions.

While the Parallel Shift Right instructions may freely be used for integer division, the Parallel Shift Left instruction may only be used for multiplication if it is known that the subword values are small enough so that overflow is not possible during the left shift of each subword. No checking is done for overflow, i.e., for significant bits shifted out on the left of each subword in this instruction. This is because the main use of the parallel shift instructions is for data alignment. Here, each subword represents some fixed-point number that should be pre- or post-aligned after some arithmetic operations.

4.2. Subword rearrangement instructions

Subword data stored sequentially in memory is often ready for parallel processing with no further data rearrangement. Sometimes, the algorithm requires the subwords to be re-arranged within the word, so that further parallel subword arithmetic may be applied. For example, suppose the same algorithm must be applied to both the rows and the columns of a matrix. Figure 2 shows a 4x4 matrix of 16-bit subwords, contained in four 64-bit registers. Starting with the left matrix, Parallel Subword instructions may be applied to four columns in parallel, since these have data in the four separate subword tracks in the registers. Then, in order to apply the same algorithm to four rows in parallel, a 4x4 matrix transpose should be done. This requires that the first subword of four registers be packed into a single register. Similarly, for the second subwords, the third subwords and the fourth subwords. This is shown by the matrix on the right in figure 2.

Figure 2: 4x4 matrix of 16-bit subwords

original matrix					**transposed matrix**				
Ra=	a1	a2	a3	a4	Re=	a1	b1	c1	d1
Rb=	b1	b2	b3	b4	Rf=	a2	b2	c2	d2
Rc=	c1	c2	c3	c4	Rg=	a3	b3	c3	d3

Rd= d1 d2 d3 d4 Rh= a4 b4 c4 d4

The conventional way to achieve this is by storing the subwords back into different locations in memory, then reading them back as a word, with the subwords in the re-arranged positions within the word. Considerable speedup may be achieved if the subwords can be re-arranged within the register file, rather than going through memory with store and load instructions. With PA2, we introduce the versatile and novel Mix and Permute instructions, for such subword rearrangements within the register file.

The Mix instructions take subwords from two registers, and interleave alternate subwords from each register in the result register. Mix Left starts from the leftmost sub-word in each of the two source registers, while Mix Right ends with the rightmost subwords from each source register. Mix is defined for 2-byte (16-bit) and 4-byte (32-bit) subwords as shown in figure 3.

Figure 3: Mix instructions

Let Ra = a1 a2 a3 a4,
and Rb = b1 b2 b3 b4.

Mix Left, *2byte*, Ra,Rb,Rc	Rc = a1 b1 a3 b3
Mix Right, *2byte*, Ra,Rb,Rc	Rc = a2 b2 a4 b4
Mix Left, *4byte*, Ra, Rb, Rc	Rc = a1 a2 b1 b2
Mix Right, *4byte*, Ra,Rb,Rc	Rc = a3 a4 b3 b4

A 4x4 matrix transpose, as shown in figure 2, can be done with just 8 such Mix instructions.

The Permute instruction takes one source register, and produces a permutation of the subwords in that register. With 2-byte subwords, this instruction allows all 256 possible permutations, with and without repetitions, of the four subwords in the source register. Figure 4 shows some possible permutations:

Figure 4: Permute instruction examples

Let source register = a b c d
Then some possible permutations are:

a a a a	permutation with repetition
d c b a	permutation without repetition
b a a d	with repetition
c c a a	with repetition
a d b c	without repetition

The Mix and Permute instructions are useful for inner-loop subword data rearrangement operations. They are also useful for data formatting. For example, the Mix instruction with register zero as one source, can be used to expand 2-byte subwords into 4-byte subwords. After the desired computation has been done with this expanded precision, a Mix instruction can also contract the 4 byte subwords contained in two registers, back into 2-byte subwords.

5. Other performance extensions

Table 3 shows some other features defined by PA2 for added performance and functionality. These were carefully chosen to avoid adding any additional circuitry in critical paths. Most control registers have also been extended to 64-bits in width.

Table 3: Other performance features

New PA2 features	Motivation
Cache hint: Spatial Locality	Prevent cache pollution when data has no reuse
Cache Line Prefetch	Reduce cache miss penalty, and prefetch penalty by disallowing TLB miss
Weakly ordered memory accesses	Enable higher performance memory systems
Conditional Branch Prediction Convention	improve conditional branch performance
Hints for Procedure return stack	improve unconditional branch performance
FMAC	Higher FLOP capability
Multiple FP condition bits	Greater parallelism in floating-point comparisons

5.1. Cache and memory extensions

Another crucial parameter in high performance processors is how effective the memory hierarchy is in reducing memory latencies. Caches are effective in reducing latencies, but additional improvements can be made by explicitly specifying in an instruction the best strategy for handling a memory reference. The PA2 architecture defines an additional cache hint - *spatial locality* - that specifies that an operand and nearby data is needed, but need only be buffered and not placed in the cache. Programs that read and write large quantities of data with little or no re-use are best candidates for this hint.

In one measurement of a database application, the operating system kernel's block copy routine was observed to cache miss on nearly all (approx. 96%) of its sources and targets. Presumably the target is soon needed, but the routine would benefit by specifying spatial locality on the data fetches and reduce cache pollution and memory bandwidth usage.

An additional mechanism that is defined for PA2 processors, is a cache prefetch instruction that signifies not only that the data should be loaded but also the desired cache state (shared or exclusive). This instuction ignores the normal memory TLB exceptions. Ignoring exceptions allows a much more aggressive use of the feature by the compiler. With this definition, the compiler does not need to verify the validity of the address. The compiler will still want to avoid memory bandwidth waste and not prefetch cache lines that will not be used.

By extending the control of the compiler to explicitly manage the memory heirarchy when it has knowledge of the access patterns or feedback from prior executions of the code, additional synergy is achieved between the processor and the compiler. These cache management features enable more effective use of the memory hierarchy.

To enable highly concurrent main memory implementations, memory accesses in PA2 are considered *weakly ordered*, unless the load or store instruction is marked as *strongly ordered*. In a weakly ordered memory system, the order in which memory access instructions appear in the program need not be the order in which they are completed. Since PA1 assumed a strongly ordered model of memory accesses, an O(rdered)-bit in the PSW can force the system to behave as if all memory accesses were strongly ordered for backward compatibility.

5.2. Conventions for branch penalty reduction

The penalties associated with branch execution are becoming major barriers to higher performance parallel issue processors. Conventions have been used by some machines to decide whether a branch should be predicted taken or not. In PA1, simple static prediction, where backward branches are predicted taken and forward branches are predicted not taken, was optimized by the conditional branch delay-slot nullification scheme [1,2]. Augmenting this static prediction with dynamic prediction is a further improvement.

An additional improvement is to allow the compiler more control in specifying the type of prediction that should be used. In the most general case, the compiler would like to specify the prediction outcome under a variety of conditions. Whether this branch predictor is in a dynamic structure, what kind of predictor to use, and static information could all contribute to a prediction. In the case of PA-RISC, no free encodings are available to give this flexibility. Instead, a convention is defined to indicate the compilers best static prediction for a given branch. The conditional branch comparison instructions have redundant encodings that can be exploited by reversing the order of the operands and relation. For example,

```
        comb,<=  r5,r7,target
```
is the same as:
```
        comb,>  r7,r5,target.
```
but allows a different branch prediction hint.

The following *branch prediction convention* is defined for PA2 processors. If the register numbers in the conditional branch instruction are in ascending order, the branch is predicted in one direction, otherwise the branch is predicted the other direction. The instruction set had enough redundancy to allow many branches to be hinted in this way. Conditional branch instructions involving comparisons with immediate operands do not have this flexibility. For those situations, an extra instruction to copy the immediate to a register can be generated to allow the hinted form.

While this approach allows most branches to have prediction indications, using register number order is just a convenience. Having a dedicated bit would be more desirable since it is simpler to decode.

Another area for branch prediction is the management of the return addresses. Using a simple push-down stack of branch addresses, most procedure returns can be correctly predicted. The PA2 architecture includes hints to push and pop procedure return values.

5.3. Floating-Point performance

A few additions are also made to improve floating-point performance. The floating-point Multiply Accumulate instruction takes three floating-point source registers, multiplies two of them, accumulates the product with the third register, and writes the result into another register. Only one rounding function is done to preserve accuracy. This instruction increases the performance of many floating-point intensive computations, including many audio, graphics transformations, and DSP type applications where single-precision floating-point accuracy is desired.

Another feature is to allow the setting and testing of 8 floating-point condition bits, rather than just one floating-point condition bit.

6. Summary

This paper has described the PA-RISC 2.0 architecture. While full support for 64-bit integers, physical addresses, and flat virtual addresses is provided, the instructions themselves have remained at 32 bits. Hence, much of the innovation in the 64-bit extensions has been in encoding the desired new functionality in the remaining unused instruction encodings.

The 64-bit extensions provide full-performance 32-bit compatibility for existing PA-RISC 1.0 and PA-RISC 1.1 programs. 32-bit applications can use 64-bit operations and vice versa, for data computations. Switching between 32-bit and 64-bit pointers is provided by the Wide-bit. Full-performance 64-bit programs and operating systems is also enabled.

PA2 also provides innovative new capabilities for new applications, such as multimedia information processing, that will be important in the lifetime of the 64-bit architecture. Features are also added to improve the performance of the cache and memory system, branching, and floating-point execution.

PA2 maintains the programming models of the PA1 architecture [2,10]. Also, none of the additional features added to the architecture have any significant impact on the basic processor cycle time, which is determined by the cache-hit path, and the 64-bit ALU path. Yet, they extend the PA-RISC architecture for the next generation of software and hardware for full-performance 64-bit programs, compatible, full-performance 32-bit programs, new applications, and performance-aggressive implementations.

7. Acknowledgements

The authors would like to thank the PA-RISC Extensions team, especially Michael Mahon.

8. Bibliography

[1] Mahon M., Lee R., Miller T., Huck J., Bryg W, "Hewlett-Packard Precision Architecture: The Processor", Hewlett-Packard Journal, vol . 37 no. 8, Aug. 1986, pp. 4-21.

[2] Lee R., "Precision Architecture", IEEE Computer, vol. 22 no. 1, Jan 1989, pp. 78-91.

[3] PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 1st. edition, Part Number 09740-90039, Hewlett-Packard, November 1990.

[4] PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 2nd. edition, Part Number 09740-90039, Hewlett-Packard, November 1992.

[5] PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 3rd edition, Part Number 09740-90039, Hewlett-Packard, February 1994.

[6] Lee R.B., "Accelerating Multimedia with Enhanced Microprocessors", IEEE Micro, vol. 15, no. 2, Apr. 1995, pp.22-32.

[7] Lee R., "Multimedia Acceleration with Subword Parallelism in Microprocessors", Distinguished Lecture Series X, recorded March 24, 1995, University Video Communications, P.O.Box 5129, Stanford, CA 94309.

[8] Gwennap L., "New PA-RISC Processor Decodes MPEG Video", Microprocessor Report Vol 8 Num 1, Jan 24, 1994, pp . 16-17.

[9] Hunt D., "Advanced Performance Features of the 64-bit PA8000", Proceedings of IEEE Compcon, March 5-9, 1995, San Francisco, California.

[10] Lee R., Mahon M. and Morris D., "Pathlength Reduction Features in the PA-RISC Architecture", Proceedings of IEEE Compcon, February 24-28, 1992, San Francisco, California, pp. 129-135.

[11] PA-RISC 2.0 Architecture, Kane G., Prentice Hall, IBSN: 013-182-7340, 1995.