SUBWORD PARALLELISM WITH MAX-2

Ruby B. Lee

Hewlett-Packard

MAX-2 illustrates how a small set of instruction extensions can provide subword parallelism to accelerate media processing and other data-parallel programs.

The general-purpose computing workload is changing to include more processing of multimedia information. We define media processing as the processing of digital multimedia information, such as images, video, audio, 2D and 3D graphics, animation, and text. This multimedia data, at the lowest component level, tend to be 16 bits or less. However, general-purpose microprocessors are generally optimized for processing data in units of words, where a word is currently at least 32 or 64 bits.

This article proposes that subword parallelism—parallel computation on lower precision data packed into a word—is an efficient and effective solution for accelerating media processing. As an example, it describes MAX-2, a very lean, RISC-like set of media acceleration primitives included in the 64-bit PA-RISC 2.0 architecture.[1] Because MAX-2 strives to be a minimal set of instructions, the article discusses both instructions included and excluded. Several examples illustrate the use of MAX-2 instructions, which provide subword parallelism in a word-oriented general-purpose processor at essentially no incremental cost.

Subword parallelism

A subword is a lower precision unit of data contained within a word. In subword parallelism, we pack multiple subwords into a word and then process whole words. With the appropriate subword boundaries, this technique results in parallel processing of subwords. Since the same instruction applies to all the subwords within the word, this is a form of small-scale SIMD (single-instruction, multiple-data) processing.[2]

It is possible to apply subword parallelism to noncontiguous subwords of different sizes within a word. In practice, however, implementations are much simpler if we allow only a few subword sizes and if a single instruction operates on contiguous subwords that are all the same size. Furthermore, data-parallel programs that benefit from subword parallelism tend to process data that are of the same size. For example, if the word size is 64 bits, some useful subword sizes are 8, 16, and 32 bits. Hence, an instruction operates on eight 8-bit subwords, four 16-bit subwords, two 32-bit subwords, or one 64-bit subword (a word) in parallel. The degree of SIMD parallelism within an instruction, then, depends upon the size of the subwords.

Data parallelism refers to an algorithm's execution of the same program module on different sets of data. Subword parallelism is an efficient and flexible solution for media processing, because the algorithms exhibit a great deal of data parallelism on lower precision data. The basic components of multimedia objects are usually simple integers with 8, 12, or 16 bits of precision. Subword parallelism is also useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data.

One key advantage of subword parallelism is that it allows general-purpose processors to exploit wider word sizes even when not processing high-precision data. The processor can achieve more subword parallelism on lower precision data rather than wasting much of the word-oriented data paths and registers. Media processors—processors specially designed for media rather than general-purpose processing—and DSPs allow more flexibility in data path widths. Even for these processors, however, organizing the data paths for words that support subword parallelism provides low-overhead parallelism with less duplication of control. A more efficient use of memory also results, since a single load- or store-word instruction moves multiple packed subwords between memory and processor registers. Hence, subword parallelism is an efficient organization for media processing, whether on a general-purpose microprocessor or a specially designed media processor or DSP.

0272-1732/96/$5.00 © 1996 IEEE

Support for subword parallelism

Data-parallel algorithms with lower precision data map well into subword-parallel programs. The support required for such subword-parallel computations then mirrors the needs of the data-parallel algorithms.

To exploit data parallelism, we need subword parallel compute primitives, which perform the same operation simultaneously on subwords packed into a word. These may include basic arithmetic operations like add, subtract, and multiply, as well as some form of divide, logical, and other compute operations. Data-parallel computations also need

- data alignment before or after certain operations for subwords representing fixed-point numbers or fractions;
- subword rearrangement within a register so that algorithms can continue parallel processing at full clip;
- a way to expand data into larger containers for more precision in intermediate computations; similarly, a way to contract it to a fewer number of bits after the computation's completion and before its output;
- conditional execution;
- reduction operations that combine the packed subwords in a register into a single value or a smaller set of values; and
- a way to clip higher precision numbers to fewer bits for storage or transmission;
- the ability to move data between processor registers and memory, as well as the ability to loop and branch to an arbitrary program location. (General-purpose processors require no new instructions for these functions, since the load-word, store-word, and branch instructions already work. For example, the load-word instruction loads multiple packed subwords into a register from memory, in a single instruction.)

Efficient support of inner-loop operations is more important than one-time operations. For example, the data expansion, contraction, clipping, and reduction functions just mentioned tend to occur outside the inner loop, so their support is not as performance critical. There are obviously many possibilities for selecting instructions and features that meet these needs. The next section presents an example of a very small set of instructions supporting subword parallelism that we added to a general-purpose RISC processor architecture.

MAX-2 instructions

MAX is a small set of Multimedia Acceleration eXtensions implemented in PA-RISC processors. For a general-purpose processor architecture like PA-RISC, the trend of both technical and business computations to include an increasing amount of multimedia information motivated the inclusion of such instructions.

We first implemented MAX-1[3] in the PA-7100LC microprocessor, a 32-bit PA-RISC 1.1 architecture.[4] MAX-2 is the second generation of multimedia instructions[1] and an integral part of the 64-bit PA-RISC 2.0 architecture.[5] Hence, all 64-bit PA-RISC processors will contain MAX-2.

Table 1 summarizes MAX-2. The first five parallel subword arithmetic instructions are the same as the MAX-1 instructions, while the rest are new. In addition to four new instruction types, MAX-2 has twice the subword parallelism per instruction, since it assumes 64-bit words; MAX-1 operated on processors with 32-bit words. Since MAX-1 is a proper subset of MAX-2, PA-RISC programs including MAX-1 instructions will run unchanged on PA-RISC 2.0 processors. (When something is common to both MAX-1 and MAX-2, we will use the notation MAX.)

Table 1. MAX-2 instructions in PA-RISC 2.0.

Parallel subword instruction*	Description	Cycles
Parallel add	Adds corresponding subwords	
with modulo arithmetic, HADD	(saturation arithmetic speeds up	1
with signed saturation, HADD,ss	and simplifies overflow handling)	1
with unsigned saturation, HADD,us		1
Parallel subtract	Subtracts corresponding subwords	
with modulo arithmetic, HSUB		1
with signed saturation, HSUB,ss		1
with unsigned saturation, HSUB,us		1
Parallel shift left and add, with signed saturation, HSLADD	Multiplies subwords by integer constant	1
Parallel shift right and add, with signed saturation, HSHRADD	Multiplies subwords by fractional constant	1
Parallel average, HAVG	Calculates arithmetic means of subwords	1
Parallel shift right	Aligns data	
signed, HSHR	divides signed integer; extends sign	1
unsigned, HSHR,u	divides unsigned integer; zero fill on left	1
Parallel shift left, HSHL	Aligns data; zero fill on right	1
Mix, MIXH, MIXW	Rearranges subwords from two source registers	1
Permute, PERMH	Rearranges subwords within a register	1

Although MAX-2 is the second generation of multimedia instructions for PA-RISC, it is still much smaller than the sets proposed for other microprocessors. (See Kohn et al.[6] and the articles by Tremblay et al. and Peleg and Weiser in this issue.) Our design goal was to leverage existing microprocessor capabilities as much as possible. We only added new features that have both significant performance acceleration for media processing and potential general-purpose utility. In addition, the new instructions do not have significant impact on the cycle time, area, and design time of the PA-RISC processor.

We considered supporting 8- and 32-bit subwords in addition to 16-bit subwords, but rejected this for our workstation and server markets. We

rejected 8-bits subwords for insufficient precision, and 32-bits subwords for insufficient parallelism compared to 32-bit (single-precision) floating point.

Although pixel components may be input and output as 8 bits, using 16-bit subwords in intermediate calculations is usually desirable. Color components of less than 8 bits represent very low-end graphics, which may be less important in the future. For medical imaging, pixel components are already 12 rather than 8 bits, again requiring at least 16-bit computational precision.

Using 32-bit integer subwords provides a parallelism of only two operations per subword instruction. Media computations that need this level of precision often work better with single-precision floating-point operations such as FMAC (floating-point multiply and accumulate), which already combine two operations into one instruction.

Parallel subword compute instructions

We found that the most common compute operations in media processing programs are the basic integer add and subtract and simple variations of these.[7,8] The first five entries in Table 1 represent parallel subword versions of such arithmetic operations.

The parallel-add and parallel-subtract instructions each have three variants that describe what happens on an overflow. The default action is modulo arithmetic, which discards any overflow. An instruction that specifies signed saturation clips the result to the largest or smallest signed integer in the result range, depending on the overflow direction. Similarly, an instruction specifying unsigned saturation clips a result that overflows to the largest or smallest unsigned integer in the result range.[3,7]

Often, the multiplications required in media processing are by constants. MAX speeds this up with parallel shift left and add, and parallel shift right and add. These two instructions very effectively implement multiplication by integer and fractional constants.[3,9] They require just a minor modification to the existing preshifter in the integer arithmetic logic unit, rather than a whole new multiplier functional unit on the integer data path.

The parallel-average instruction performs a very common and useful function in image and video processing: the arithmetic mean (Figure 1). It adds two source operands then performs a divide by two. This is a combined-operation instruction, involving an add and a right shift of one bit. In the process, the instruction shifts in the carry-out bit as the most significant bit of the result, so the instruction has the added advantage that no overflow can occur. In addition, it rounds the least significant bit to conserve precision in cascaded average operations.

MAX-2 includes new parallel (subword) shift instructions, which use the existing 64-bit shifter but block any bits shifted out of one subword from being shifted into the adjoining subword. These instructions are useful for data alignment. Here, each subword represents some fixed-point (integer or fractional) number that must be pre- or post-aligned after certain arithmetic operations. Parallel shift right (signed or unsigned) can also speed up division of signed and unsigned subwords by powers of two. It is also useful for sign or zero extension. (While the parallel shift and add instructions allow shifting of only one, two, or three bits, the parallel shift instructions allow shifting of any number of bits.)

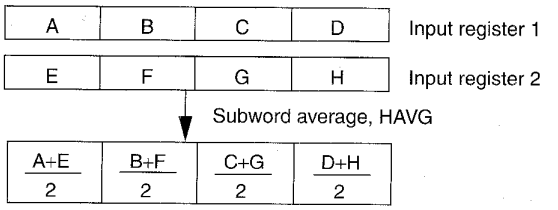

Figure 1. Parallel subword average instruction.

We can use the parallel shift-right instructions freely for integer division. However, we can use the parallel shift-left instruction for multiplication only when we know that the subword values are small enough to not overflow during each subword's left shift. These instructions do no checking for overflow—that is, for significant bits shifted out on the left of each subword. This is because we use parallel shift instructions mainly for data alignment and data rearrangement.

We considered adding subword integer-multiply hardware, but decided against it for the following reasons. Media-processing computations like audio and graphics transformations, which use multiply-accumulate or multiply-by-a-variable operations extensively, usually require intermediate calculation precision greater than 16 bits. When audio samples come as 16-bit linear audio input, internal computational precision greater than 16 bits is desirable.

Given the choice of 32-bit integer versus 32-bit floating-point precision, most audio and graphics programmers prefer the latter for its automatic alignment features, accuracy, and greater dynamic range. FMAC instructions exist in PA-RISC 2.0 processors in both single and double precision. These already provide the combined-operation parallelism of two operations (a multiply and an accumulate) per pipeline cycle. Furthermore, PA-RISC 2.0 processors usually have two such floating-point multiply-accumulate units, giving a parallelism of four operations per cycle. By using the existing floating-point FMAC units, we save considerable area otherwise needed for new integer multiply units in the integer data path. We also save pipeline complexity, since integer multiply is a multicycle operation while all existing integer instructions are single cycle.

A bonus of using the floating-point register file rather than the integer register file is that it makes available twice as many addressable registers (64 rather than 32). This is because in PA-RISC, though each integer and floating-point register file has 32 registers, we can address each 64-bit floating-point register as two 32-bit registers. Another advantage is that this arrangement sometimes allows video and audio processing to occur simultaneously: one on the integer side and the other on the floating-point side, each with its own register file and computation hardware.

Hence, we implement graphics transformations and audio computations as single-precision floating-point programs. Data is loaded directly to one of the sixty-four addressable 32-bit floating-point registers. These programs do not need to move data back and forth between the integer and floating-point register files. Alternative solutions to multiply-by-a-variable involve either new, costly subword multiply hardware or some form

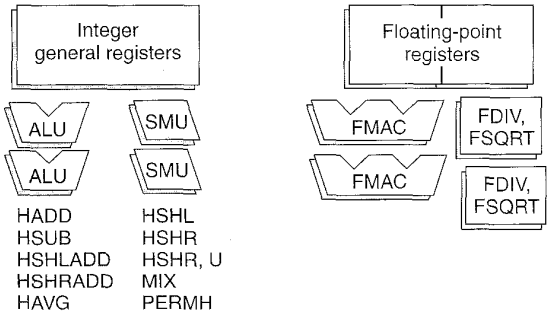

Figure 2. MAX instructions, listed under the units that execute them, use existing processor resources.

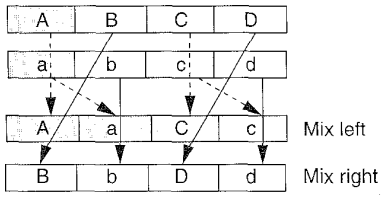

Figure 3. Mix interleaves alternate subwords from two registers.

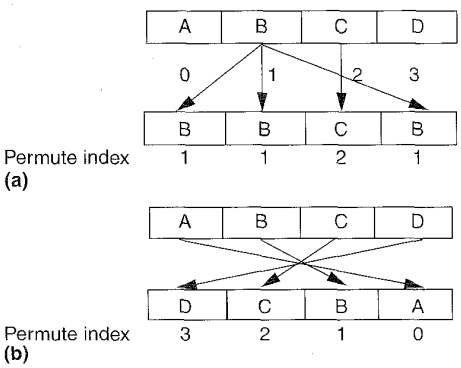

Figure 4. Permute allows rearrangement and repetition (a) and reversal (b) of subwords.

of less costly but less capable multiply hardware.[6]

Figure 2 shows some of the resources of the PA-8000.[10] The existing integer ALUs (arithmetic logic units) and SMUs (shift merge units) implement the MAX-2 instructions. MAX-1 used only the ALUs. However, different types of media-processing computations use the entire processor, including integer and floating-point register files, integer and floating-point functional units, and the enhanced cache memory system.

Subword rearrangement instructions

Many algorithms require subword rearrangement within registers. The challenge is to find a small set of primitives that fulfills most frequent subword regrouping or rearrangement needs. In MAX-2, we introduce just two new data rearrangement instructions: mix and permute.

Mix. These instructions take subwords from two registers and interleave alternate subwords from each register in the result register. Mix left starts from the leftmost subword in each of the two source registers, while mix right ends with the rightmost subwords from each source register. Figure 3 illustrates this for 16-bit subwords.

Mix implements an even-odd paradigm for parallel processing: Even elements are processed in parallel, then odd elements, or vice versa. (We use the names mix-left and mix-right instead of mix-even and mix-odd because whether an element is odd or even depends on whether elements are numbered from zero or one, starting from the left or right.) Mix is more powerful than interleaving sequential subwords from each source register (AaBb rather than AaCc) because it can combine subwords from both halves of each source register in the result register. It also has a lower implementation cost, requiring only very minor changes to the existing two-input, unidirectional shifter.

Permute. The permute instruction takes one source register and produces a permutation of that register's subwords. With 16-bit subwords, this instruction can generate all possible permutations, with and without repetitions, of the four subwords in the source register. Figure 4 shows some possible permutations. To specify a particular permutation, we use a permute index. The instruction numbers subwords in the source register starting with zero for the leftmost subword. A permute index identifies which subword in the source register the instruction places in each subword of the destination register.

Integer or floating-point data path?

We implemented the MAX-2 instructions in the integer data path for several reasons. First, they require only very minor modifications to the integer ALU and SMU units; implementation on the floating-point side would require new subword integer functional units using the floating-point registers.

In addition, to implement MAX on the floating-point side, we would have to replicate many useful integer functions or not use them. These include all field manipulation instructions like shift-pair, extract, and deposit, and logical instructions like AND, AND-complement, OR, and exclusive-OR (see Table 2). The shift-pair instruction is particularly useful, since it allows a 0- to 63-bit shift on any two source registers. Common shift instructions work on only one source register. For example, shift-pair can properly align an unaligned sequence of subwords to a 64-bit word boundary.

The area savings of not adding new integer units on the floating-point side outweighed the relatively minor disadvantage of sharing general-purpose registers with address and loop counter variables. Load and store instructions do not necessarily compete for superscalar issue slots with MAX instructions. For example, the PA-8000 allows two load or store instructions to issue with two MAX-2 instructions, even though they all use the general registers. Furthermore, the floating-point registers and functional units can operate simultaneously with MAX-2 instructions for possible parallel audio

processing or graphics transformations, making a total of 96 registers available for simultaneous processing of different media streams.

Other PA-RISC features

Several other PA-RISC features that enhance floating-point, cache memory, and branching performance[1] also contribute to higher performance media processing (see Table 2). We have already mentioned the usefulness of the FMAC instructions. Multiple floating-point condition bits allow simultaneous testing of several conditions. They are also used to perform faster boundary box "trivial" accept and reject tests in graphics computations.

Cache prefetch instructions allow prefetching cache lines from memory into the cache to reduce cache miss penalties when the processor actually needs the data. This is very useful for the predictable streaming data of many multimedia memory accesses.[11] If a TLB miss occurs for such a cache prefetch instruction, it is ignored, and the instruction reduces to a one-cycle NOP (no operation). This reduces the downside of prefetching a cache line that is not actually used, allowing more aggressive prefetch policies.

PA-RISC processors also have a cache hint in load and store instructions that indicates that the data has spatial locality but no temporal locality. Thus, a processor implementation can fetch the cache line containing the desired data into a buffer without displacing useful data in the cache. This can reduce conflict misses in the cache due to streaming data that the processor does not reuse.

Table 2. Other useful PA-RISC features.

Feature	Description/motivation
Shift right a pair of registers, SHRPD, SHRPW	Concatenate and shift 64-bit (SHRPD) or rightmost 32-bit (SHRPW) contents of two registers into one result register
Extract a field, EXTRD, EXTRW	Select any field in the source register and places it right-aligned in the target register
Deposit a field into a register, DEPD, DEPDI, DEPW, DEPWI	Select a right-aligned field from the source register or an immediate, and places it anywhere in the target register
Logical operations, AND, ANDCM, OR, XOR	Existing logical operations in integer ALUs
FMAC	Floating-point multiply accumulate combined-operation instruction
Multiple floating-point condition bits	Enable concurrency in floating-point comparisons and tests
Cache line prefetch, prefetch for read, LDD R0 prefetch for write, LDW R0	Reduce cache miss penalty and cache prefetch penalty by disallowing TLB miss
Cache hint: spatial locality	Prevent cache pollution when data has no reuse

Mapping data parallelism

A data-parallel computation includes a piece of code that it must execute many times for different sets of data. The goal is to map data-parallel computations that operate on lower precision data onto subword-parallel instructions. A basic technique is to execute multiple iterations of a loop in parallel, rather than to find opportunities for using subword-parallel instructions within the loop itself.

For example, for an 8×8 discrete cosine transform (DCT), the processor must perform the same loop on eight rows and eight columns. If we apply our basic technique, a subword parallel instruction would work on four rows or four columns at a time—that is, on four loop iterations at a time—rather than trying to restructure the code for a single DCT loop using subword-parallel instructions.

While it is beyond this article's scope to describe comprehensive techniques for exploiting data parallelism in general and mapping to subword-parallel instructions in particular, the following example illustrates the process.

High-level programming languages (for example, C) often capture data-parallel computations as for- or while-loops. A high-level-language loop may look like this:

```
short x[200], y[200], z[200], w[200];
int i:
for (i = 0, i < 200, i++){
        z[i] = x[i] + y[i];
        w[i] = x[i] – y[i]; }
```

Figure 5a (next page) shows a PA-RISC assembler version of the loop only, generated by a C compiler. Figure 5b shows the desired subword-parallel assembly version, which is very similar. The programmer has the choice of either modifying compiler-generated assembly code to get the subword-parallel version, or modifying the C code to assist the compiler in generating the subword-parallel version directly. One difficulty for the compiler is data alignment—it cannot be sure that the 16-bit shorts align on the 64-bit boundaries. We propose that the programmer specifically indicate that such alignment is necessary, using C's existing "union" feature, which superimposes the elements in the union onto the same storage locations. The programmer can use the feature to force four 16-bit shorts to be 64-bit aligned (where "long" is 64 bits).

The compiler could recognize statements in for-loops that use variables declared in earlier "union" constructs as hints to generate subword-parallel code. However, for the programmer, it is not necessarily easier to write such stylized C code, since it usually takes multiple C statements to express a single MAX instruction.

Consider the C statements specifying saturation arithmetic for each subword operation, for example. It is often easier for the programmer to just write the MAX-2 assembly instruction itself, or a C macro that corresponds to this assembly instruction. Using macro calls M_HADD and M_HSUB to generate the corresponding MAX-2 instruction, the following C code

```
      ldi       199, Ri          ;set loop count to 200-1
loop: ldhs,ma   2(Raddrx), Rx    ;bring in 1 halfword from address x, postincrement by 2 bytes
      ldhs,ma   2(Raddry), Ry    ;bring in 1 halfword from address y, postincrement by 2 bytes
      add       Rx, Ry, Rz       ;1 add
      sub       Rx, Ry, Rw       ;1 subtract
      sths,ma   Rz, 2(Raddrz)    ;store 1 result in z, and postincrement
      sths,ma   Rw,2(Raddrw)     ;store 1 result in w, and postincrement
      addibf,<  -1,Ri,loop       ;decrement loop count and loop back if not < 0
(a)

      ldi       49, Ri           ;set loop count to (200/4) -1
loop: ldds,ma   8(Raddrx), Rx    ;bring in 4 halfwords from x, postincrement by 8 bytes
      ldds,ma   8(Raddry), Ry    ;bring in 4 halfwords from y, postincrement by 8 bytes
      hadd      Rx, Ry, Rz       ;4 parallel subword adds
      hsub      Rx, Ry, Rz       ;4 parallel subword subtracts
      stds,ma   Rz, 8(Raddrz)    ;store 4 results in z, and postincrement
      stds,ma   Rw, 8(Raddrw)    ;store 4 results in w, and postincrement
      addibf,<  -1,Ri,loop       ;decrement loop count and loop back if not < 0
(b)
```

Figure 5. High-level-language loop: PA-RISC assembler version without (a) and with (b) subword-parallel instructions. In PA-RISC three-register assembly instructions, the order of the fields is opcode, source1, source2, destination. For load instructions, the order is opcode, displacement (base), target. A halfword (h) denotes 16 bits, and a doubleword (d) denotes 64 bits in PA-RISC load and store instructions.

can generate the desired subword-parallel assembly code shown in Figure 5b:

```
Union {long a[50]; short x[200]} e;
Union {long b[50]; short y[200]} f;
Union {long c[50]; short z[200]} g;
Union {long d[50]; short w[200]} h;

int i;
for (i = 0, i<50, i++){
                    M_HADD (e.a[i], f.b[i], g.c[i]);
                    M_HSUB (e.a[i], f.b[i], h.d[i]); }
```

Since the parameters to the macro calls M_HADD and M_HSUB are 64-bit longs, the compiler will issue load-double (64-bit) and store-double instructions. The programmer may freely intermix such in-line assembly macros with C statements. For the compiler, macro expansion is much easier and faster than using pattern matching for code generation. The compiler performs register allocation and renaming, loop unrolling, scheduling, and other optimizations. Hence, PA-RISC compilers support MAX-2 instructions (and other PA-RISC assembly instructions) through macros. Programmers can also include in header files the macro calls for optimized MAX-2 code sequences in Table 3 and any larger macros of their choice, such as the DCT.

Programming examples

This section gives very short code examples, pulled from larger programs, that illustrate how MAX-2 instructions support key needs of data-parallel computations (as enumerated earlier in the section on support for subword parallelism).

Subword rearrangement. Since mix and permute are new MAX-2 instructions not covered in earlier work,[3] we give several examples of their use. Often, an algorithm must rearrange the subwords packed into registers to fully utilize subsequent subword-parallel instructions.

Matrix transpose. Suppose that an algorithm performs a certain sequence of operations on all the rows and columns of a matrix. The first four rows of Table 4 show a 4×4 matrix of 16-bit subwords contained in four 64-bit registers. We can apply parallel-subword instructions to the elements of four columns in parallel, since these are in the four separate subword tracks in the registers. Then, to apply the same algorithm to the elements of four rows in parallel, we must transpose the 4×4 matrix. The last four rows in Table 4 show the transposed matrix.

The conventional way to achieve this is to store the subwords into different memory locations, then read them back as a word with the subwords in the rearranged positions within the word. This requires 16 store (halfword) and four load (doubleword) instructions. We can achieve considerable speedup if we rearrange the subwords within the register file rather than going through memory, which can incur potential cache misses. Using PA-RISC extract and deposit instructions for the rearrangement would require more than 20 instructions.

Table 4 shows such a 4×4 matrix transform done with just eight mix instructions. To transpose each row of the matrix, the processor must read four registers. Hence, for a processor with two register reads and one register write per cycle and no intermediate storage, eight is the minimum number of instructions for 4×4 matrix transpose. Table 4 shows the transformation using four temporary registers. Though we can accomplish the task with only two temporary registers, this does not permit the maximum superscalar use of two SMUs processing two mix instructions per cycle. This matrix transpose takes four cycles with two SMUs implementing mix instructions, and only two cycles with four such SMUs.

Transposing an 8×8 matrix would require four such 4×4 matrix transposes, taking 16 cycles with two SMUs. Sixteen 64-bit registers would be needed to house an 8×8 matrix of 16-bit subwords.

Expanding and contracting. Mix instructions are also useful for data formatting. For example, the mix instruction with register zero as one source can expand 2-byte subwords in Ra into 4-byte subwords in Rx and Ry. After the desired computation with this expanded precision, the mix instruction can also contract the 4-byte subwords contained in Rx and Ry back into packed 2-byte subwords in a single register (see Table 3). By using mix instructions to perform both expand and contract operations, we preserve the data's original order after contraction.

Replicating subword constants. The permute instruction is useful for replicating subword constants (see Table 3). First, the load offset instruction, LDO, loads a 16-bit signed-immediate

Table 3. Short code examples using MAX-2.

Operation	Code examples*		Cycles**
4×4 matrix transpose of 16-bit subwords in four registers: R1, R2, R3, and R4	MIXH,L MIXH,R MIXH,L MIXH,R MIXW,L MIXW,L MIXW,R MIXW,R	R1,R2,t1; R1,R2,t2; R3,R4,t3; R3,R4,t4 t1,t3,R1; t2,t4,R2; t1,t3,R3; t2,t4,R4	4 (2 SMUs), 2 (4 SMUs)
Expand halfwords to 32-bit words	MIXH,L MIXH,R	R0,Ra,Rx; R0,Ra,Ry	1
Contract words back to halfwords	MIXH,R	Rx,Ry,Rb	1
Replicate subword constant in register	LDO PERMH,3333	const(R0),Rm Rm,Rm	2
Max (a_i, b_i)	HSUB,us HADD,ss	Ra, Rb, Rt Rt, Rb, Rt	2
Min (a_i, b_i)	HSUB,us HSUB,ss HADD,ss	Ra, Rb, Rt R0, Rt, Rt Rt, Ra, Rt	3
Sum of absolute differences SAD (a_i, b_i) accumulated in Rx and Ry	HSUB,us HSUB,us HADD,us HADD,us	Ra, Rb, Rc; Rb, Ra, Rd Rx, Rc, Rx; Ry, Rd, Ry	2
Abs (a_i)	HSUB,us HSUB,us HADD	Ra, R0, Rc; R0, Ra, Rd Rc, Rd, Rx	2
Tree add	EXTRD HADD EXTRD HADD	Ra,31,32,Rt Ra, Rt, Rs Rs,47,16,Rt Rt, Rs, Rt	4
Clip (a_i) signed 16-bit to unsigned n-bit, n<16	HADD,ss HSUB,us [$Rmax_i = (2^{15} - 1) - (2^n - 1)$]	Ra,Rmax,Ra Ra,Rmax,Ra	2
Clip (a_i) to new maximum value, H	HADD,ss HSUB,ss [$Rmax_i = (2^{15} - 1) - H$]	Ra,Rmax,Ra Ra,Rmax,Ra	2
Clip (a_i) to new nonzero minimum value, L	HSUB,ss HADD,ss [$Rmin_i = 2^{15} - L$]	Ra,Rmin,Ra Ra,Rmin,Ra	2
Clip (a_i) to both new maximum value, H, and new nonzero minimum value, L	HADD,ss HSUB,ss HADD,ss [$Rmax_i = (2^{15} - 1) - H$, $Rmin_i = 2^{15} - L$, and $Rboth_i = (2^{15} - 1) - H + 2^{15} - L$]	Ra,Rmax,Ra Ra,Rboth,Ra Ra,Rmin,Ra	3

* A semicolon after an instruction indicates that it may execute in parallel with the following instruction

** Number of cycles is based on data dependencies in the code sequence and on at least two integer ALU or SMU instructions issuing each cycle (as in PA-8000).

value into the target register's low-order bits. (LDO is a PA-RISC instruction used for address computation.) Then permute

Table 4. Register contents in matrix transpose.

Instruction		Register contents			
		R1 = a1	**b1**	**c1**	**d1**
		R2 = a2	**b2**	**c2**	**d2**
		R3 = a3	**b3**	**c3**	**d3**
		R4 = a4	**b4**	**c4**	**d4**
MixH,L	R1,R2,t1	t1 = a1	a2	c1	c2
MixH,R	R1,R2,t2	t2 = b1	b2	d1	d2
MixH,L	R3,R4,t3	t3 = a3	a4	c3	c4
MixH,R	R3,R4,t4	t4 = b3	b4	d3	d4
MixW,L	t1,t3,R1	**R1 = a1**	**a2**	**a3**	**a4**
MixW,L	t2,t4,R2	**R2 = b1**	**b2**	**b3**	**b4**
MixW,R	t1,t3,R3	**R3 = c1**	**c2**	**c3**	**c4**
MixW,R	t2,t4,R4	**R4 = d1**	**d2**	**d3**	**d4**

instruction PERMH replicates this value for the other three subwords in the register.

Conditional execution using saturation arithmetic. Saturation arithmetic efficiently handles the multiple overflows that arise in parallel subword arithmetic. Signed saturation (ss) refers to cases with two signed operands in which the instruction clips subword results to a signed integer. Unsigned saturation (us) in MAX refers to cases with one unsigned and a second signed operand, in which the instruction clips each subword result to an unsigned integer. We found this definition of unsigned saturation more useful than the more common one (where both operands are unsigned) since the algorithms often require adding a signed value to the original unsigned data.

Saturation arithmetic is also very useful for conditional operations in the following class:

If cond(Ra_i, Rb_i) Then Rt_i = Ra_i Else Rt_i=Rb_i,
for i = number of subwords in the word

These are conditional operations that, based on a comparison of corresponding subwords in registers Ra and Rb, select one subword as the result to go into register Rt. For example, condition (cond) could be Max or Min. With Max, result register Rt_i gets the larger of Ra_i or Rb_i. With Min, Rt_i gets the smaller of Ra_i or Rb_i.

Max function. In Table 3, the entry for the Max(a_i, b_i) function shows that we can accomplish this operation with just two in-line instructions using saturation arithmetic. The first instruction, HSUB, subtracts Rb from Ra, and the second instruction adds Rb back. Without saturation arithmetic, these two instructions would cancel each other out. However, with saturation arithmetic, they place the larger of Ra_i or Rb_i in the result Rt_i. If $Ra_i > Rb_i$, then $Rt_i = Ra_i - Rb_i$, for the first instruction, and $Rt_i = Ra_i - Rb_i + Rb_i = Ra_i$ after the second

instruction. If $Ra_i < Rb_i$, then $Rt_i = 0$ for the first instruction using unsigned saturation, and $Rt_i = Rb_i$ after the second instruction. The clipping to zero that unsigned saturation provides in the first instruction is the key to such in-line conditional operations.

Min function. The next entry in Table 3 shows how we can obtain the Min(a_i, b_i) function using saturation arithmetic. This requires three rather than two instructions. The extra instruction subtracts the result of the first instruction from zero to invert the sense of greater than (Max) to less than (Min). If $Ra_i > Rb_i$, then $Rt_i = Ra_i - Rb_i$ for the first instruction; $Rt_i = Rb_i - Ra_i$ (negative) after the second instruction; and $Rt_i = Rb_i - Ra_i + Ra_i = Rb_i$ after the third instruction. If $Ra_i < Rb_i$, then $Rt_i = 0$ for the first instruction; $Rt_i = -0$ after the second instruction; and $Rt_i = Ra_i$ after the third instruction. For example,

	Ra =	260	60	260	60
	Rb =	60	260	−60	−260
HSUB,us	Ra, Rb, Rt	200	0	320	320
HSUB,ss	R0, Rt, Rt	−200	0	−320	−320
HADD,ss	Rt, Ra, Rt	60	60	−60	−260

In practice, Max and Min are usually taken over unsigned numbers, but we also accommodate the cases where the second operand may be negative.

Sum of absolute differences. SAD is an important metric in many compression algorithms, including MPEG-1 and MPEG-2. The code example for SAD in Table 3 does two subtracts with the operands switched. Using subtract with unsigned saturation, negative numbers are clipped to zero. Thus, in each subword track in the two result registers Rc and Rd, only one would have a nonzero positive value; the other would have a zero value. Then, the two add instructions accumulate the absolute differences in two sets of accumulators, Rx and Ry, each of which consists of four 16-bit accumulators. By using two accumulator registers, we can issue the two add instructions simultaneously. At the end of the inner loop, we add up these eight 16-bit accumulated values to get the final sum of absolute differences (see Reduction, later).

In superscalar processors, which issue at least two instructions per cycle, the two subtract and two add instructions would only take two cycles to execute. Since these instructions retire four pixels, their peak execution rate is 0.5 cycles/pixel.

To better illustrate the cycles per pixel metric, we coded the whole block comparison of two 16×16-pixel blocks. We then calculated the sum of absolute differences for these 256 pairs of pixels, using the four instructions described earlier for every four pairs of pixels. Including loads, stores, and loop overhead, this took 0.51 cycles/pixel on a PA-8000 processor. This compares very favorably to special-purpose sum-of-absolute-difference instructions proposed by other processors.[6]

Absolute value. The code example for finding absolute values of the subwords (Table 3) is similar to that for SAD, except that it takes differences from zero, rather than from a second source register. Also, since accumulation is not necessary, we can add intermediate result registers Rc and Rd, saving one instruction.

Reduction. Reduction instructions condense the horizontal subwords in a register into fewer subwords or into a single value. For example, we considered many versions of "tree add," which would add all the subwords packed in a word into a single value. With a programming model that uses parallel-subword accumulators, the final accumulation across the parallel accumulators occurs only once at the end of a loop. Hence, tree add does not require very efficient execution, and we added no new instructions to MAX-2.

Instead, we can implement it as shown in Table 3. Register Ra contains four subword accumulators. The first extract instruction moves the left half of register Ra to the right half of register Rt. HADD adds two pairs of subwords, reducing four accumulated values to two in register Rs. The second extract instruction moves the next-to-rightmost subword in partial-sum register Rs to the rightmost position of Rt. The final HADD instruction reduces these two partial sums to the final accumulated sum in the rightmost subword of register Rt.

Clipping. After computation with data in expanded format for higher precision, it is often necessary to clip the data to fit in a smaller number of bits before storage or transmission. For example, we expand unsigned 8-bit pixel components to signed 16-bit subwords for computation. We must then clip them back to 8 bits before output.

Clip signed to unsigned. We need only two instructions for the common case of clipping a signed 16-bit number to an unsigned 8-bit one (see Table 3). Let register Rmax contain the constant value 32,512 ($2^{15} - 1 - 255$) in each 16-bit subword. This is the largest representable number in signed 16-bit notation minus the largest representable number in unsigned 8-bit notation. The first instruction adds Rmax to Ra, which contains the values to be clipped. The second instruction subtracts Rmax from this result. If the original value in Ra is between 0 and 255, these two instructions cancel each other out. If the original value in Ra is greater than 255, then the first instruction clips it to $2^{15} - 1$, and the second instruction brings this down to 255. If the original value is negative, the first instruction brings it to a value less than $2^{15} - 1$, and the second instruction clips it to zero using unsigned saturation.

One advantage of clipping with saturation arithmetic is its versatility. These two instructions can clip to any n-bit unsigned value. For example, medical-imaging applications use 12-bit instead of 8-bit pixel components. The same two instructions can clip signed 16-bit numbers to unsigned 12-bit numbers. The only change is that the constant register, Rmax, now contains the difference between $2^{15} - 1$ and 4,095, the new maximum. An instruction defined specifically for clipping signed 16-bit subwords to unsigned 8-bit values would be useless in this case.

General clipping. When we use saturation arithmetic for clipping, the new maximum or minimum value of the result does not have to be of the form $2^n - 1$. It can be any value. Table 3 also lists general cases of clipping to any new maximum or minimum value, or both.[3] When clipping to both a new maximum and a new nonzero minimum value, there is a trade-off between using one more instruction or one more register. Three instructions are sufficient if we use three constant registers (as in Table 3). If we use only two constant registers, then we need four instructions.

Performance

The maximum parallelism, measured in operations per cycle, is a function of the number of instructions that can execute in parallel, the number of subwords we can pack into a word, and

the presence of combined-operation[9,12] instructions.

Table 5 shows examples of the operation parallelism achievable by the PA-8000, which is a four-issue superscalar processor with 64-bit integer data paths. In the PA-8000, two load or store instructions may execute with two computation instructions (integer or floating-point) each cycle. FMAC is a combined-operation instruction, representing two operations per cycle. With 16-bit subwords, each subword instruction is equivalent to at least four operations. Some, like the halfword-average and halfword-shift-and-add instructions combine two operations per subword per cycle, so that each instruction represents eight operations per cycle. Since saturation arithmetic replaces five PA-RISC instructions per subword, a halfword add with saturation instruction replaces 20 PA-RISC instructions, bringing the parallelism up to 42 operations per cycle. Such peak parallelism is, of course, not sustainable in real applications.

For a large application, we first achieved real-time software MPEG-1 decode (video, audio, and system layers) on a low-end PA-RISC workstation using MAX-1 instructions in the PA-7100LC processor.[7-8, 13] Without MAX-1, this would not have been attainable on 60- to 80-MHz microprocessors, since MPEG-1 requires about 200 million operations per second. Several large applications, including MPEG-1, H.261, Convolve 512×512, and Zoom 512×512, showed speedups in frames per second of 1.9 to 2.7 on the PA-7100LC with MAX-1.[3]

The PA-8000 doubles the superscalar degree from two to four instructions per cycle, doubles the number of parallel subwords by increasing the word size from 32 to 64 bits, more than doubles the clock frequency, and makes available MAX-2 rather than MAX-1 instructions. With its greater peak performance potential compared to the PA-7100LC, it can, for example, process multiple MPEG-1 streams and MPEG-2 in real time with software alone.

MAX-2 IS A MUCH SMALLER SET of multimedia instructions than those proposed for other processors. However, its simple instructions are often as efficient in media processing as complex, special-purpose instructions, as many of the examples in this article show. As general-purpose primitives, they are usually also more flexible, as illustrated by the clipping examples.

A key advantage of this minimalist approach is that all this subword parallelism and media acceleration essentially come for free with the general-purpose processor. The area impact is truly insignificant: MAX-1 took about 0.2 percent of the PA-7100LC silicon area, and MAX-2 took less than 0.1 percent of the PA-8000 silicon area. Neither caused any cycle time impact on the processors nor lengthened the design schedules.

Subword parallelism provides a cost-effective method of achieving usable operation parallelism for lower precision data. It is a low-overhead technique for speeding up media and other processing in general-purpose microprocessors, media processors, and DSPs. This significant speedup puts difficult media-processing tasks in the realm of software rather than special-purpose hardware. This is especially useful for implementing multiple algorithms for different media data types, adjusting to emerging standards, and exploiting improved algorithms.

As processor performance increases, MAX performance will automatically increase, as will the performance of media-processing programs. Eventually, even complex media-processing tasks like multiway videoconferencing or multiple MPEG-2 streams may take only a small fraction of a processor's computational power.

Table 5.
Operation parallelism in PA-8000 with MAX-2.

Instruction sequence				Operations/cycle
Load	Load	FMAC	FMAC	6
Load	Store	HADD	HSUB	10
Load	Load	HAVG	HSHLADD	18
Load	Load	HADD,ss	HSUB,us	42

Acknowledgments

I thank Larry McMahan and Behzad Razban for help with coding the sum of absolute differences, Max, and Min examples.

References

1. R. Lee and J. Huck, "64-bit and Multimedia Extensions for the PA-RISC 2.0 Architecture," *Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 152-160.
2. M. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, Vol. 54, No. 12, Dec. 1966, pp. 1901-1909.
3. R.B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, Apr. 1995, pp. 22-32.
4. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 3rd ed., Hewlett-Packard, Cupertino, Calif., 1994.
5. G. Kane, *PA-RISC 2.0 Architecture*, Prentice Hall, Old Tappan, N.J., 1996.
6. L. Kohn et al., "The Visual Instruction Set (VIS) in UltraSPARC," *Proc. Compcon*, IEEE CS Press, 1995, pp. 462-469.
7. R. Lee, "Multimedia Acceleration with Subword Parallelism in Microprocessors," *Distinguished Lecture Series X*, recorded March 24, 1995, University Video Communications, Stanford, Calif.
8. R. Lee, "Realtime MPEG Video via Software Decompression on a PA-RISC Processor," *Proc. Compcon*, IEEE CS Press, 1995, pp. 186-192.
9. R. Lee, "Precision Architecture," *Computer*, Vol. 22, No. 1, Jan. 1989, pp. 78-91.
10. D. Hunt, "Advanced Performance Features of the 64-bit PA-8000," *Proc. Compcon*, IEEE CS Press, 1995, pp. 123-128.
11. D. Zucker, M. Flynn, and R. Lee, "Improving Performance for Software MPEG Players," *Proc. Compcon*, IEEE CS Press, 1996, pp. 327-332.
12. R. Lee, M. Mahon, and D. Morris, "Pathlength Reduction Features in the PA-RISC Architecture," *Proc. Compcon*, 1992, IEEE CS Press, pp. 129-135.
13. L. Gwennap, "New PA-RISC Processor Decodes MPEG Video," *Microprocessor Report*, Vol. 8, No. 1, Jan. 24, 1994, pp. 16-17.

Ruby B. Lee's photograph and biography appear on p. 9. Direct questions concerning this article to Ruby B. Lee, Hewlett-Packard, 19410 Homestead Road, MS 43UG, Cupertino, CA 95014; rblee@cup.hp.com.