MULTIMEDIA EXTENSIONS FOR GENERAL-PURPOSE PROCESSORS

Ruby B. Lee
Hewlett-Packard Company
19410 Homestead Road
Cupertino, CA 95014.
rblee@cup.hp.com

Abstract - This paper gives an overview of the multimedia instructions that have been added to the instruction set architectures of general-purpose microprocessors to accelerate media processing. Examples are MAX, MMX and VIS, the multimedia extensions for PA-RISC, ix86, and SPARC processor architectures. We describe *subword parallelism*, a low overhead form of SIMD parallelism, and the classes of instructions needed to support subword parallel computations efficiently. Features described include arithmetic operations with saturation, averaging, multiply alternatives, data rearrangement primitives like Permute and Mix, formatting instructions, conditional execution, and complex instructions.

1. INTRODUCTION

The general-purpose information processing workload is changing to include an increasing amount of media processing. *Media processing* is the processing of digital multimedia information, such as images, video, 2-dimensional and 3-dimensional graphics, animations, audio and text. The definition of a general-purpose processor implies that it can process any program, including media processing programs. However, some media processing programs have demanding real-time performance requirements, such as the encoding and decoding of high-resolution, high-fidelity video at 30 frames per second. Hence, *multimedia extensions* are new instructions and resources added to general-purpose processors to improve the performance of media processing programs. The more versatile the base architecture, the fewer the new features needed, since the goal is to exploit the existing structure and resources of the general-purpose processor as much as possible. Since this paper assumes that the processor is general-purpose, we are not really interested in special-purpose instructions or resources that are only useful for a specific algorithm. Rather, we are searching for the “native alphabet” of a very broad class of media processing programs - the general-purpose primitives from which more complicated operations, loops and programs can be built efficiently.

Most of the major microprocessor architectures have already added or proposed multimedia extensions to their Instruction Set Architectures (ISAs). PA-RISC was the first ISA to introduce multimedia extensions, MAX-1 (Multimedia Acceleration eXtensions), in products introduced in January 1994 [1]. To demonstrate the effectiveness of MAX-1, a software MPEG-1 decoder was introduced at the same time, that could decode high-fidelity MPEG-1 video, audio and system layers at 30 frames per second, on a low-end, 80 Mhz PA-RISC workstation [2]. Next, Sun added VIS (Visual Instruction Set) to the Sparc ISA [3]. HP then introduced MAX-2, its second generation multimedia extensions for its 64-bit PA-RISC 2.0 processor ISA [4]. In January 1997, Intel introduced chips with MMX (Multi-Media Extensions) added to the ix86 ISA [5]. SGI has announced the MDMX (Mips Digital Media Extensions) for MIPS processors [6], and Alpha has announced a small set of MVI (Motion Video Instructions) for Alpha processors [6] specifically to accelerate MPEG-2 encoding.

Table 1 summarizes the instruction set features available in the first three multimedia extensions: MAX-2, VIS and MMX. Since published papers on MDMX and MVI are not readily available, these are not included. The instruction mnemonics used are illustrative, rather than identical to those in the specific ISAs. Many features in Table 1 will be described in the course of the paper. However, it is beyond the scope of this paper to describe all the memory instructions, and other features present in the base ISAs, which are particularly useful to media processing, although some of these are included in Table 1 for illustrative purposes. We first discuss subword parallelsim (section 2), the key feature common to these multimedia extensions, that provides the main performance acceleration. Then, we describe different classes of instructions that either implement or support subword parallelism, drawing from example instructions in MAX, MMX and VIS. These classes include subword parallel arithmetic instructions (section 3), data rearrangement instructions (section 4), formatting instructions (section 5), and conditional instructions (section 6). We also briefly mention complex instructions (section 7) and memory instructions (section 8), before concluding (section 9).

2. SUBWORD PARALLELISM

Pixel-oriented media processing programs exhibit a high degree of data-parallelism on lower-precision (less than 16 bits) data. These include image processing, video processing and graphics rendering computations. Pixels are input to, or output from, the computation as 8 bit or 12 bit (medical imaging) components, but intermediate processing usually requires slightly greater precision, such as 16 bit precision. Based on this observation, the concept of subword parallelism was introduced into microprocessor ISAs [4],[1].

In *subword parallelism*, a standard unit of computation or storage, a word, is partitioned into smaller units called subwords. The same operation can be performed on the subwords in parallel, providing a form of SIMD (Single Instruction Multiple Data) parallel processing. Instructions can be performed on the whole word, or on subwords in parallel. Subwords can be of different sizes, the most useful size being 16 bits for media processing. 8 or 32 bit subwords are also useful in some situations, but lack in precision or parallelism, respectively, compared to 16 bits. Theoretically, subwords can be either overlapping or non-overlapping, either partially or completely fill the word, either implemented by software or hardware, and deal with either integer or floating-point data. In practice, subwords are non-overlapping, completely fill the word, and are implemented by hardware. This subset of subword parallelism is also referred to as *packed parallelism*. In MAX, VIS and MMX, the subwords are integer subwords, but the concept of subword parallelsim can also be applied to floating-point subwords.

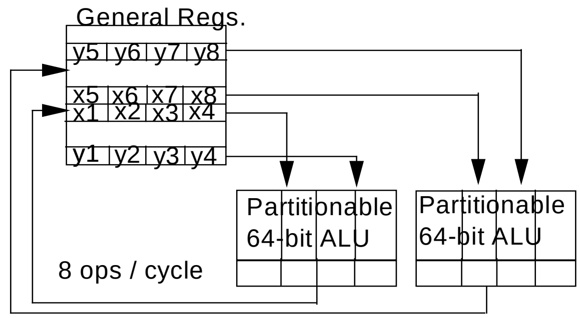

Figure 1: Subword Parallelism saves on Register Ports and Instructions

Subword parallelism provides a very low-cost form of small-scale SIMD parallelism in a word-oriented processor. A word-wide integer functional unit can be partitioned into parallel subword units, with very little hardware overhead. For example, a 64-bit integer two's-complement adder may be partitioned into four 16-bit subword integer adders, by merely blocking the carry at the three 16-bit boundaries. Such a partitionable adder allows four 16-bit adds, or a single 64-bit add, to be performed in a single processor cycle. The overhead cost is very small since the same datapaths are used in either case: two 64-bit register reads and one register write. A supercalar microprocessor with two ALUs could support 8 parallel operations with just a 6-ported register file (see figure 1), rather than a 24-ported register file, which would be required if 8 independent 16-bit functional units were used. This suggests that the subword parallel organization is desirable for special-purpose media processors as well as for general-purpose processors, which already have word-oriented datapaths.

3. SUBWORD PARALLEL ARITHMETIC INSTRUCTIONS

3.1 Add and Subtract

The most frequent operations in many important image and video computations are still integer add and subtract, as is the case for general-purpose computations. The difference is that the precision required is often lower: 16 bits or even just 8 bits. Hence, Parallel Subword Add (Padd) and Parallel Subword Subtract (Psub) are often the first two instructions added to a 32-bit or 64-bit general-purpose processor for multimedia acceleration.

	PA-RISC/ MAX-2	SPARC/ VIS	ix86 / MMX
Padd (mod.arith) (signed sat.) (unsigned sat.)	16 16 16	16,32 - -	8,16,32 8,16 8,16
Psub (mod.arith) (signed sat.) (unsigned sat.)	16 16 16	16,32 - -	8,16,32 8,16 8,16
Pavg	16	-	-
PshRadd	16xn=16	-	-
PshLadd	16xn=16	-	-
Pmul	-	16x8=16	16x16=16
Pmadd	[Fmac: two FP 32x32=32]	-	two (16x16)+ (16x16)=32
PshR, PshRa	16	-	16,32,64
PshL	16	-	16,32,64
PcmpEq PcmpGt	n [XNOR] -	16,32 (Eq, NEq) 16,32 (Gt, NGt)	8,16,32 8,16,32
MixL, MixR	16,32	8 (fpmerge)	(UnpackH/L)
Permute	16	-	-
[Shift Pair]	128 [shift 2 regs.]	-	-
Unpack	8to16u(PshR,PshL) 16to32u(Mix) 32to64u (Mix) 8to16s (PshRA)	8to16u(fexpand)	8to16u 16to32u 32to64u 8to16s (PshRA)
Pack	32u to 16u (Mix), 16s to 8u (Paddss, Psubus)	32u to 16u 16u to 8u 32u to 8u	32s to 16s 16s to 8s 16s to 8u
duplicate 64-bit logicals on FRs.	[share logicals on GRs]	16 logical instructions	AND, ANDN, OR, XOR
Pdist (Sum of Absolute Diffs.)	- (sat. arith.)	$c = c + \sum \|ai-bi\|$ for i=1 to 8	- (sat. arith.)
Edge (boundary processing)	[Ftest, multiple cond bits]	8,16,32(generate bit-mask)	-
Alignaddr, Faligndata	[Set ShiftAmtReg, Shift Pair]	set up, and align data	-
Array (3D addr. to blocked addr.)	-	8,16,32	-
block load/store	-	ld/st 8 regs.	-
Partial store	[Store 0-8 bytes]	store w/ mask	-
Prefetch to cache	[Prefetch R/W]	-	-
EMMS	-	-	Empty FR tags

Table 1: Multimedia Extensions for General-Purpose Processors

Note: [ ] indicates feature present in base ISA
( ) indicates feature in multimedia ISA extension
8,16,32,64,128 are subword widths, in bits, supported for that instruction

3.2 Parallel Overflows and Saturation Arithmetic

The key difference in the Padd and Psub instructions between ISAs in Table 1 is in how overflows are handled. Because of the smaller number of bits available to a subword, overflows can be more frequent. Since multiple overflows can occur in a single Padd or Psub instruction, inefficient handling of even infrequent overflows can reduce any performance gain from subword parallel execution.

In modulo arithmetic, the overflow is ignored, and the result is taken modulo 2^n, for n-bit subwords. For example, for 8-bit subwords, (255+1) mod 256 is 0. This is the desired result when implementing pointer arithmetic for a circular buffer of 256 elements, for example. Modulo arithmetic is implicitly assumed by integer add and subtract operations in C.

In pixel-oriented arithmetic, where values represent a color spectrum, such modulo arithmetic is undesirable, since a small change in color intensity may result in a huge change in the resulting color, e.g., from white to black. Here, saturation arithmetic is more desirable. In *saturation arithmetic*, a *positive overflow*, viz., an overflow beyond the largest representable number in n bits, causes the result to be clamped to that largest representable number. Similarly, a *negative overflow*, viz., an overflow below the smallest representable number in n bits, causes the result to be clamped to that smallest representable number. The result can be defined as a signed or unsigned number, resulting in signed or unsigned saturation.

All three architectures, MAX, MMX and VIS implement modulo arithmetic. MAX and MMX also implement signed and unsigned saturation arithmetic. For architectures like VIS which do not have saturation arithmetic, unexpected overflows can potentially cause worse artifacts. Also, explicit software checking and handling of overflows may result in performance degradations, compared to architectures with automatic overflow handling via saturation arithmetic.

3.3 Derivatives of Add

There are many useful derivatives of a parallel add instruction, reusing the partitionable 2's complement adder. For example, the *Average* of two numbers is obtained by adding them, then dividing the sum by two. This is equivalent to a right shift of one bit, after the add. This 1-bit right shift shifts in the overflow bit of the add operation as the most significant bit of the result, while shifting out the least significant bit. By combining a common sequence of two operations into a single instruction, not only is performance improved, but no overflow is possible for the combined pair of operations.

Another derivative of Parallel Add are the *Parallel Shift and Add* instructions, which shift one of the operands by a few bits before adding the other operand. These are very useful as integer multiply and accumulate primitives (see below).

MAX is currently the only multimedia ISA including Parallel Subword Average (Pavg) and Parallel Subword Shift Right/Left and Add (PshRadd, PshLadd) instructions.

3.4 Multiply

Multiplication is one area where the greatest differences exist for the different multimedia instruction-set extensions. The problems with multiplication are: an integer multiplier takes about three to four times the area of an integer adder, it takes about three times the latency, and it produces a result that is longer than each operand.

VIS and MMX both include new hardware for 16-bit integer multiply. VIS tries to reduce the area by restricting parallel subword multiplication to a 16-bit operand multiplied by an 8-bit operand, with the result being the high-order 16 bits of the 24-bit product. To complete a 16-bit by 16-bit multiply, at least three such instructions are needed.

Current implementations of MMX try to reduce the area by implementing only one 16-bit multiplier which is reused iteratively to perform the four subword multiplications in a *Parallel Subword Multiply* (pmul) instruction. The full 16-bit by 16-bit multiplication is performed, with either the upper or lower sixteen bits of the 32-bit product selected as the result. MMX also has a *Parallel Subword Multiply-Accumulate* instruction which does four 16-bit by 16-bit multiplies, then adds the first two products and the last two products, giving two 32-bit results (figure 2).

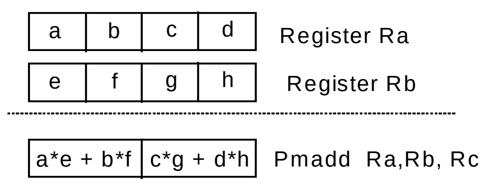

Figure 2: Subword Multiply-Accumulate Instruction in MMX

MAX has a two-pronged strategy for supporting multiplication, without adding any 16-bit multiply hardware. This is based on observing that there appears to be two main classes of media computations based on their multiplication needs: those requiring extensive, full-function multiplication with precision greater than 16 bits, and those where multiplication is frequently by constants with the precision required being less than 16 bits. By full-function multiplication, we mean both multiplication by variables (i.e., neither operand’s value is known at compile time), and multiply-accumulate functionality. Audio and graphics transformations fall into the first class, while image, video and graphics rendering tend to fall into the second class.

For the first class of full-function, greater than 16-bit precision media computations, MAX assumes that they are implemented using the existing PA-RISC floating-point functional units and registers for single-precision (32-bit) floating-point data. For example, the PA-8000 [7], PA-8200 and PA-8500 all have two full-function, floating-point multiply-accumulate (FMAC) units, and two floating-point divide and square-root units. Since two FMAC instructions can be issued every cycle, this is already equivalent to four operations per cycle.

For the second class of computations, 16-bit precision is sufficient and multiplication is often by constants. No new integer multiply hardware is added. Instead, the existing integer adders with preshifters are used. The *Parallel Shift Left and Add (PshLadd)* instruction shifts one operand left by 1, 2 or 3 bits, before adding the other operand. This is equivalent to multiplying one operand by 2, 4, or 8, before adding the second operand. The *Parallel Shift Right and Add (PshRadd)* instruction shifts one operand right by 1, 2 or 3 bits, before adding the other operand. This is equivalent to multiplying one operand by 1/2, 1/4 or 1/8, before adding the second operand. Together, they are low-cost multiply and accumulate primitives for integer or fractional multiplication. Figure 3 illustrates how the parallel multiplication of four subwords by the constant, SQRT(2), is built up using only 3 PshRadd instructions.

$$SQRT(2) = 1.4142_{10} = 1.01101010_{2}$$

t = 1.01*x	t = x + x>>2	PshRadd x,2,x, t
s = 1.0101*x	s = x + t>>2	PshRadd t,2,x, s
t = 1.0110101*x	t = t + s>>3	PshRadd s,3,t, t

Fig. 3: Parallel Subword Multiply using Parallel ShiftRight&Add instructions

The *Parallel Shift Left (PshL)* and *Parallel Shift Right logical/arithmetic (PshR, PshRa)* instructions perform parallel subword shifts: bits shifted out of one subword are not shifted into the adjacent subword. They perform n-bit shifts, rather than just 1, 2 or 3 bit shifts, and may also be used to multiply subwords by larger positive or negative powers of two. Since these instructions are implemented on a shifter, no checking for overflow is done on PshL, unlike in the PshLadd instructions, where saturation arithmetic is performed.

Table 2 shows the equivalent number of 16-bit hardware multipliers that would be needed to obtain the same performance as Parallel Shift and Add instructions, using one or more 64-bit integer adders. For example, if two adders are available, and the multiplication requires a sequence of 3 PshRadd instructions as in figure 3, then 8/3=2.67 hardware multipliers are needed to equal the performance of the low-cost Pshadd instructions. Eight 16-bit subwords are being multiplied simultaneously by two adders, and each PshRadd instruction takes one cycle.

	1 instruction	2 instructions	3 instructions	4 instructions
1 adder	4	2	1.33	1
2 adders	8	4	2.67	2
3 adders	12	6	4	3

Table 2: Equivalent number of 16-bit multiplies performed per cycle

4. DATA REARRANGEMENT INSTRUCTIONS

With four or eight subwords packed into a word-wide register, it is often necessary to rearrange the order of the subwords.

4.1 Permute

The Permute instruction allows any arbitrary rearrangement of the subwords from one source register, with or without repetition of any of the subwords. Only one source register, rather than two, can be handled if all possible permutations are allowed, since only one result register is generated.

Figure 4 shows examples of some permutations that can be generated. In figure 4a, one subword b is replicated across the result register . This is useful when b is a scalar to be used in subsequent vector-scalar operations. A single load instruction can load four 16-bit scalars into a register. Next, a Permute instruction can replicate any one of these scalars in another register, in a single cycle. This allows any subword parallel instruction to be used for either vector-vector operation, or vector-scalar operation. An alternative is to duplicate instructions for vector-scalar operations (e.g., in MDMX), which often results in a more restricted set of vector-scalar instructions.

Figure 4b shows a permutation where the order of the subwords is reversed. This is useful when dealing with bi-endian data, i.e., both little-endian and big-endian data. Figure 4c shows a permutation, where one subword is repeated twice.

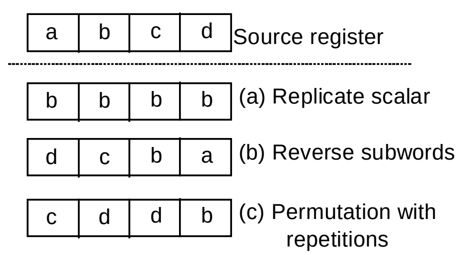

Figure 4: Permutation Instruction Examples

4.2 Mix

Often it is necessary to rearrange subwords spread across multiple registers, in order to proceed with SIMD processing at maximum performance. The architectural finesse is in selecting only those combinations and orderings of subwords, from the myriad possibilities, that provide the greatest usefulness in building up common data rearrangement cases.

```
for i=1 step 4 to 8 do <1-D 8-point IDCT >;
8x8 matrix transpose;
for i=1 step 4 to 8 do  <1-D 8-point IDCT >;
8x8 matrix transpose;
```

Figure 5: 8x8 IDCT using Subword Parallel Instructions

For example, a 2-dimensional 8x8 Inverse Discrete Cosine Transform (IDCT) may be processed as eight 1-dimensional 8-point IDCTs on the columns, followed by eight 1-dimensional 8-point IDCTs on the rows. Assuming that data for four columns of the 8x8 block are initially packed in the four subword tracks of the registers, subword parallel instructions can execute four 1-D IDCTs simultaneously on four columns ("step 4" in figure 5, means perform 4 iterations simultaneously). If an 8x8 matrix transpose is then performed, the subwords in the registers are rearranged so that data for four rows are now packed in the subword tracks. Each of the "for-loops" is now executed in 2 iterations rather than 8, using subword parallel arithmetic instructions for the 1-D 8-point IDCT computation. To perform the 8x8 matrix transpose, new data rearrangement instructions are desirable.

Mix is a data-rearrangement primitive in MAX-2, based on the even-odd paradigm. It interleaves either even or odd subwords selected alternately from each of two source registers. MAX-2 actually uses the terms MixLeft and MixRight rather than MixEven and MixOdd, because being even or odd depends on whether the subwords are numbered from 0 or 1, and from the left or from the right. MixLeft interleaves alternate subwords starting with the leftmost subwords in the source registers, while MixRight interleaves alternate subwords ending with the rightmost subwords. Figure 6 shows MixLeft (MixL) and MixRight (MixR) for 16-bit and 32-bit subwords. Mix can also be defined for other sizes of subwords, e.g., 8, 4, 2 or 1 bit subwords.

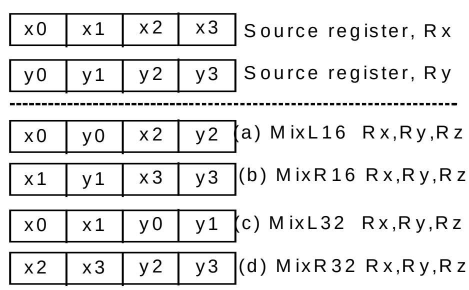

Figure 6: Subword Rearrangement Instructions, Mix, on 2 Source Regsters

Figure 7 shows how Mix instructions may be used to perform a 4x4 matrix transpose of 16-bit subwords, in two steps. First, four Mix16 instructions combine the even or odd 16-bit subwords from a pair of registers. Then, four Mix32 instructions combine the even or odd 32-bit subwords from pairs of registers generated in the first step. In general, for a matrix transpose of an nxn matrix, where n subwords are packed into a register, we need $\log_2(n)$ steps using $n*\log_2(n)$ Mix instructions. If n Mix instructions can be issued simultaneously, then only $\log_2(n)$ cycles are needed. However, if only m Mix instructions may be executed in one cycle, m<n, then $[n/m]* \log_2(n)$ cycles are needed. For example, in figure 7, where n=4 subwords and m=2 shifters implementing Mix instructions, then a 4x4 matrix transpose can be completed in 4 cycles.

An 8x8 matrix transpose is just four 4x4 matrix transposes, where the upper right 4x4 sub-matrix is swapped with the lower-left 4x4 sub-matrix. No extra operations are needed for this swapping: just register renaming suffices.

An alternative solution to the 8x8 matrix transpose problem is to build a special-purpose, matrix transpose functional unit, invoked with a multi-cycle matrix transpose instruction. This would be of no use for other types of data rearrangement. Primitives such as Mix, not only require much less hardware, but also have other general-purpose, data-rearrangement uses.

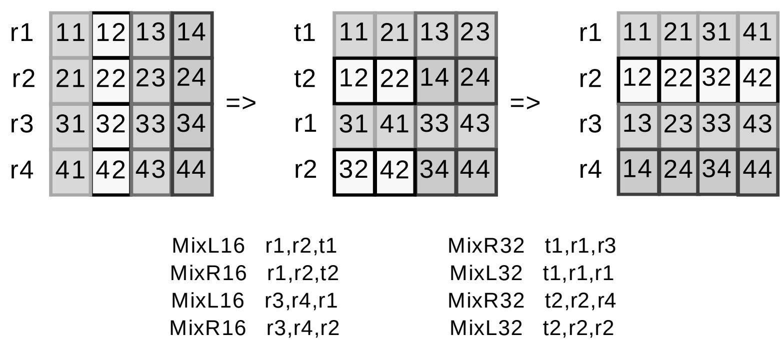

Figure 7: 4x4 Matrix Transpose Using Mix instructions

5. FORMATTING INSTRUCTIONS

Formatting instructions, which convert one subword size to another are necessary to support subword paralleism, since the level of precision required for computations may need to be greater than the input and output precision of the data. However, they are usually not performance critical operations, since the conversion is usually done before and after an inner loop computation.

Unpack instructions are designed to expand smaller subwords into larger ones, and *pack* instructions reduce larger subwords back into smaller ones. MMX has Unpack instructions which expand packed unsigned 8-bit to 16-bit subwords, unsigned 16-bit to 32-bit subwords, or unsigned 32-bit to 64-bit subwords. Its Pack instructions first saturate the result to the lower-precision desired, then contract it to the smaller subwords.

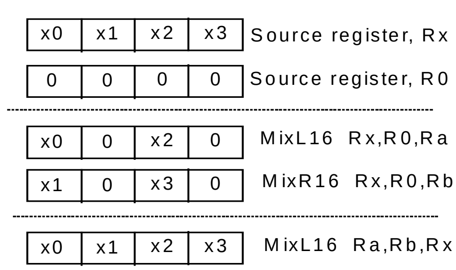

Figure 8: Use of Mix for unpacking then packing in MAX-2

The Mix instructions, in MAX-2, used with an all-zero operand (register R0), can also be used for unpacking data. For fractional data (binary point to the left), zeros are added to the right, and for integer data (binary point to the right), zeros are added to the left, in unpacking. Figure 8 shows Mix instructions used to unpack fractional data in Rx by adding zeros to the right (i.e., R0 is the second operand in the Mix instructions). By switching the order of Rx and R0 in figure 8, integer data may be unpacked. After computation is done with the higher precision subwords, the Mix instruction is used again to pack the smaller subwords in the original order.

The Parallel Shift Left or Right instructions (PshL, PshR, PshRa), described earlier for multiplication by a power of 2, can also be used for unpacking and packing signed or unsigned, integer or fractional numbers.

6. CONDITIONAL INSTRUCTIONS

6.1 Compare and Select

A *general conditional select* operation compares pairs of operands in two registers, and selects subwords from two other registers, based on the result of this comparison:

$$\text{if } \mathrm{cond}(a_i, b_i) \text{ then } c_i = s_i \text{ else } c_i = t_i\ , \quad \text{for } i = 1 \text{ to } n \qquad (1)$$

The *Parallel Subword Compare (Pcmp)* instruction performs the “cond(a_i, b_i)” testing by comparing pairs of subwords in the two source registers, and generating either a 1-bit true or false indicator, or a mask of all ones or all zeros, for each subword comparison. We call the former a *bit-mask compare* (e.g., in VIS) and the later a *mask-mask compare* (e.g., in MMX).

The mask-mask Pcmp instruction may be used to select subwords from two registers by using 64-bit logical operations. Equation (1) is implemented with four instructions:

Pcmp	Ra,Rb, Rmask;	Compare and generate mask-mask in Rmask
And	Rs,Rmask, Ru;	If cond(a_i, b_i) true, select subword s_i from Rs
AndN	Rt,Rmask, Rv;	If cond(a_i, b_i) false, select subword t_i from Rt
Or	Ru,Rv, Rc;	Combine selected subwords in Rc

The bit-mask Pcmp instruction is used in VIS to control the *Partial Store* instruction. This instruction stores a 64-bit register at an aligned 64-bit boundary in memory. Only those subwords corresponding to a “1” bit in the bit-mask are actually written to memory; the other subwords remain unchanged. The Edge instruction in VIS also generates a bit-mask for the Partial Store instruction. MAX does not need a Partial Store instruction, since its base PA-RISC architecture already has a Store Bytes instructions, which performs a similar function.

6.2 In-line Conditionals with Saturation Arithmetic

A general conditional select operation is not often needed in media computations. Often, the two registers being compared, and the two sources of subword data to be selected, are in fact the same, reulting in a *self-conditioned select* operation:

$$\text{if } \mathrm{cond}(a_i,\ b_i) \text{ then } c_i = a_i \text{ else } c_i = b_i\ , \quad \text{for } i = 1 \text{ to } n \qquad (2)$$

Examples are max(a_i, b_i) where the condition, "cond(a_i, b_i)" is "a_i greater than b_i", or min(a_i, b_i), where "cond(a_i, b_i)" is "a_i less than b_i". These conditional operations can be implemented using saturation arithmetic. A subword parallel add or subtract operation, with saturation arithmetic, has a built-in conditional operation on each pair of subwords: the subword result is either the true arithmetic result, or one of the two possible saturated values. Every pair of operands and the operation uniquely determine the direction of saturation.

With saturation arithmetic, two operations which normally cancel each other out, may no longer do so. For example, a subtract of a constant, followed by an add of the same constant, may not give the original value, due to possible saturation of the first operation. This is used in figure 9a to implement the conditional operation, max(a_i, b_i), using unsigned saturation. Figure 9b shows how absolute differences of pairs of subwords may also be obtained using saturation arithmetic.

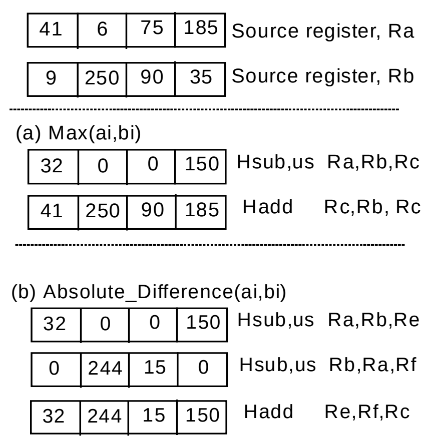

Figure 9: Conditional Operations using Saturation Arithmetic

7. COMPLEX INSTRUCTIONS

A compound operation may not be used in many media computations, but may be performance critical in a few important ones. One example is the Sum of Absolute Differences, a metric used extensively to find the error between a reference block and the current block, in MPEG-like video encoding. The *pdist* instruction, in VIS, subtracts 8-bit subwords in the second source register from the corresponding 8-bit subwords in the first source register, takes the absolute values of these 8 differences, adds them together, and accumulates them to the result register (see figure 10). This instruction has a latency of three cycles, requires three source registers, and special-purpose hardware.

MMX and MAX use generic padd and psub instructions, on 8-bit or 16-bit subwords, with unsigned saturation to achieve the same result (see figure 9b). In actual SAD code, MAX uses two accumulator registers (Rc and Rd), so that the subword absolute values in Re and Rf can be accumulated in parallel. Within the inner loop, four instructions are needed, but a latency of only two cycles. Although the pdist instruction is a single instruction, its latency is 3 cycles. Pipelining the pdist instruction can help to lower the effective latency down towards one cycle, but the cost is still significantly greater.

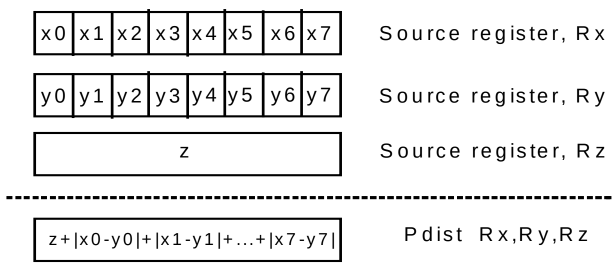

Figure 10: Pdist instruction: Sum of Absolute Differences

A different type of complex instruction is the EMMS instruction in MMX. This instruction is needed to switch the use of the 8 floating-point registers between multimedia instructions, where they are used in random access mode, and floating-point instructions, where they are used in stack mode. Not only is MMX constrained by the paucity of registers in its base ix86 architecture, but its performance gains can be significantly reduced by the large number of cycles needed for this EMMS instruction.

8. MEMORY INSTRUCTIONS

An advantage of subword parallelism is that a single Load64 instruction can load multiple packed subwords into a register. This is similarly true for store instructions. Hence, the number of load and store instructions needed is also reduced, in subword parallel computations. These Load and Store instructions are already present in all 64-bit microprocessors.

Most media computations have very predictable memory access patterns which can benefit from the judicious use of prefetch instructions to reduce cache miss penalties. PA-RISC processors have a Prefetch instruction [8] which will fetch the cache-line containing the addressed word to the cache, if it is not already there. VIS has special-purpose instructions for loading a block of 8 registers (Block Load), and changing three-dimensional array addresses into a pre-defined blocked linear address (Array) [3]. It is beyond the scope of this paper to describe, in detail, the memory access patterns of media computations, and the many possible approaches for improving the performance of memory accesses.

9. CONCLUSIONS

We have described the multimedia extensions, MAX, VIS and MMX, that are available in current general-purpose microprocessors. They all implement the concept of subword parallel instructions, which accelerate not only media processing, but also any data-parallel computation with lower-precision data. Although all implement integer subwords, only MAX-2 instructions use existing integer functional units, while MMX and VIS have new subword-parallel integer functional units operating out of the floating-point registers. The multi-cycle multimedia instructions in VIS and MMX fit more easily with the multi-cycle floating-point instructions in microprocessors. Becauses MAX-2 has only single-cycle instructions, it can fit easily with the single-cycle integer instructions, using the existing integer aithmetic-logical and shifter functional units.

Ideally, each multimedia instruction, or feature, has multiple uses. For example, saturation arithmetic is used for overflow handling as well as for in-line conditional execution; Mix instructions are used for data rearrangement as well as formatting; and parallel shift instructions are used for shifting, multiplication by a power of two, and formatting.

Significant performance improvements are achievable at insignificant incremental cost, with these multimedia extensions [2],[3],[4],[5],[9]. For example, MAX-2 occupies less than 0.1% of the area of the PA-8000 microprocessor, but achieves better than linear speedup with respect to its degree of subword parallelism [9]. At the application level, challenging media processing like real-time MPEG-2 decoding is already achievable by some of these general-purpose microprocessors.

In conclusion, high-performance, programmable media processing is here. The ubiquity of subword parallel instructions in microprocessors is assured. These general-purpose microprocessors with multimedia extensions can process increasingly difficult media processing tasks, raising the bar for special-purpose solutions. Since multimedia instructions are part of the main processor, their performance will improve at the relentless rate of microprocessor performance improvements. In addition, future Instruction Set Architecture innovations are likely, as we continue the symbiotic research into more efficient algorithms and architectures.

10. REFERENCES

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

[2] Lee R., "Real-time MPEG Video via Software Decompression on a PA-RISC

Processor", *Proceedings of IEEE Compcon*, March 5-9, 1995, San Francisco, California, pp. 186-192.

[3] Trembley M., O'Connor M., Narayanan V., He L., "VIS Speeds New Media Processing", *IEEE Micro*, vol. 16 no. 4, August 1996, pp. 10-20.

[4] Lee R., "Subword Parallelism with MAX2", *IEEE Micro*, vol. 16 no. 4, August 1996, pp. 51-59.

[5] Peleg A. and Weiser U., "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, vol. 16 no. 4, August 1996, pp. 42-50.

[6] Ninth Annual Microprocessor Forum, October 21 - 24, 1996, San Jose, California.

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

[8] Lee R., and Huck J., "64-bit and Multimedia Extensions for the PA-RISC 2.0 Architecture", *Proceedings of IEEE Compcon*, February 25-28, 1996, Santa Clara, California.

[9] Lee R., and McMahan L., "Mapping of Application Software to the Multimedia Instructions of General-Purpose Microprocessors", *Proceedings of IS&T/SPIE Symposium on Electric Imaging: Multimedia Hardware Architectures 1997*, February 10-14 1997, San Jose, California, pp. 122-133.