MULTIMEDIA INSTRUCTIONS IN MICROPROCESSORS FOR NATIVE SIGNAL PROCESSING

Ruby B. Lee and A. Murat Fiskiran
Department of Electrical Engineering
Princeton University

1. INTRODUCTION

Digital signal processing (DSP) applications on computers have typically used separate DSP chips for each task. For example, one DSP chip is used for processing each audio channel (two chips for stereo); a separate DSP chip is used for modem processing, and another for telephony. In systems already using a general-purpose processor, the DSP chips represent additional hardware resources. Native signal processing is DSP performed in the microprocessor itself, with the addition of general-purpose multimedia instructions. Multimedia instructions extend native signal processing to video, graphics and image processing, as well as the more common audio processing needed in speech, music, modem and telephony applications. In this study, we describe the multimedia instructions that have been added to current microprocessor instruction set architectures (ISAs) for native signal processing, or, more generally for multimedia processing.

Multimedia information processing is becoming increasingly prevalent in the general-purpose processor's workload [1]. Workload characterization studies on multimedia applications have revealed interesting results. More often than not, media applications do not work on very high precision data types. A pixel-oriented application for example, rarely needs to process data that is wider than 16 bits. A low-end digital audio processing program may also use only 16-bit fixed-point numbers. Even high-end audio applications rarely require any precision beyond 32-bit single-precision (SP) floating point (FP). Common usage of low-precision data in such applications translates into low computational efficiency on general-purpose processors, where the register sizes are typically 64 bits. Therefore, efficient processing of low-precision data types on general-purpose processors becomes a basic requirement for improved multimedia performance.

Media applications exhibit another interesting property. The same instructions are often used on many low-precision data elements in rapid succession. Although the large register sizes of the general-purpose processors are more than enough to accommodate a single low-precision data, the large registers can actually be used to process many low-precision data elements in parallel.

Efficient parallel processing of low-precision data elements is therefore a key for high-performance multimedia applications. To that effect, the registers of general-purpose processors can be partitioned into smaller units called *subwords*. A low-precision data element can be accommodated in a single subword. Since the registers of general-purpose processors will have multiple subwords, these can be processed in parallel using a single instruction. A *packed data type* will be defined as data that consists of multiple subwords packed together.

Figure 1.1 shows a 32-bit integer register that is made up of four 8-bit subwords. The subwords in the register can be pixel values from a grayscale image. In this case, the register will be holding four pixels with values 0xFF, 0x0F, 0xF0 and 0x00. Similarly, the same 32-bit register can also be partitioned into two 16-bit subwords, in which case, these subwords would be 0xFF0F and 0xF000. One important point is that, the subword boundaries do not correspond to a physical boundary in the register file. Whether a data is packed or not, does not make any difference regarding its representation in a register.

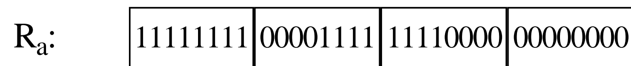

Figure 1.1 Example of a 32-bit integer register holding four 8-bit subwords. The subword values are 0xFF, 0x0F, 0xF0 and 0x00, from the first[1] to the fourth subword respectively.

If we have 64-bit registers, the useful subword sizes will be bytes, 16-bit halfwords or 32-bit words. A single register can then accommodate 8, 4 or 2 of these subwords respectively. The processor can carry out parallel operations on these subwords with a single SIMD-style[2] instruction. SIMD parallelism is said to exist when a single instruction operates on multiple data elements in parallel. In the case of subword parallelism, the *multiple data* elements will correspond to the subwords in the packed register.

Traditionally, however, the term SIMD was used to define a situation where a single instruction operated on multiple registers, rather than on the subwords of a single register. To address this difference, the parallelism exploited by the use of subword parallel instructions, is defined as microSIMD parallelism [2]. Thus, an *add* instruction operating on packed data, can be viewed as a microSIMD instruction, where the *single instruction* is the 'add' and the *multiple data* elements are the 'subwords' in the packed source registers.

---

[1] *Through this chapter, the subwords in a register will be indexed from 1 to n, where n will be the number of subwords in that register. The first subword (index=1) will be in the most significant position in a register; whereas the last subword (index=n) will be in the least significant position. In the figures, the subword on the left end of a register will have index=1, and therefore be the in the most significant position. The subword on the right end of a register will have index=n, and therefore be in the least significant position.*

[2] *The term SIMD stands for 'Single Instruction Multiple Data'.*

For a given processor, the ISA needs to be enhanced to exploit microSIMD parallelism. New instructions are added to allow parallel processing of packed data types. Minor modifications to the underlying functional units will also be necessary. Fortunately, the register file and the pipeline structure need not be changed to support packed data types.

We define *packed instructions* as the instructions that are specifically designed to operate on packed data types. A *packed add,* for example, is an add instruction with the regular definition of addition, but it operates on packed data types. *Packed subtract* and *packed multiply* are other obvious instructions needed to efficiently manipulate packed data types.

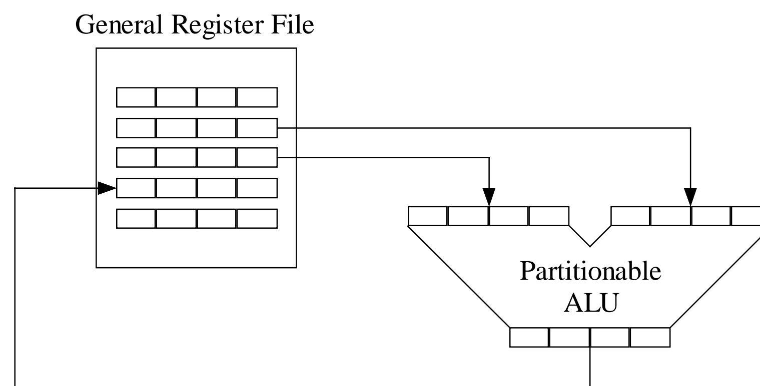

Figure 1.2 microSIMD parallelism uses packed data types and a partitionable ALU structure.

All the architectures in this chapter include varieties of packed instructions. More often than not, they also include other instructions that cannot be classified as packed arithmetic operations. As we shall see shortly, the introduction of subword parallelism to an ISA actually requires that new instructions other than the packed arithmetic instructions are also added. In this study, the multimedia instructions will be classified and discussed in the following order:

- Packed add/subtract operations
- Packed special arithmetic operations
- Packed multiply operations
- Packed compare/minimum/maximum operations
- Packed shift/rotate operations
- Data/subword packing and rearrangement operations
- Approximation operations.

The operations above will be discussed in sections 2 to 8 respectively. To provide examples we will be referring to the following multimedia extensions/architectures:

- IA-64 [3], MMX [4], and SSE-2 [5] from Intel,
- MAX-2 [6,7] from Hewlett-Packard,
- 3DNow![3] [8,9] from AMD and
- AltiVec [10] from Motorola.

Of these architectures, MAX-2 and MMX include only integer microSIMD extensions. SSE-2 and 3DNow! include only FP microSIMD extensions. IA-64 and AltiVec have both integer and FP microSIMD extensions.

Historical Overview

Prior to the ones we discuss in this chapter, there have been other notable multimedia extensions introduced to the general-purpose processors [11]. All of these earlier attempts had the same underlying idea as today's more recent extensions. They were based on subword parallelism: operating in parallel on lower precision data packed into higher precision words.

The first multimedia extensions came from Hewlett-Packard with their introduction of the PA-7100LC processor in January 1994 [12]. This processor featured a small set of multimedia instructions called MAX-1, which was the first version of the 'Multimedia Acceleration Extensions' for the 32-bit PA-RISC instruction-set architecture [13]. MAX-2, although designed simultaneously with MAX-1, was introduced later with the 64-bit PA-RISC 2.0 architecture. The application that best illustrated the performance of MAX-1 was the MPEG-1 video and audio decoding at real-time rates of up to 30 frames per second [14]. For the first time, this performance was made possible on a general-purpose processor in a low-end desktop computer [15]. Until then, such video performance was not achievable without using specialized hardware or high-end workstations.

Next, Sun introduced VIS [16], which was an extension for the UltraSparc processors. Unlike MAX-1, VIS did not have a minimalist approach, thus, it was a much larger set of multimedia instructions. In addition to packed arithmetic operations, VIS provided specialized instructions that were designed for algorithms that manipulated visual data.

---

[3] 3DNow! may be considered as having two versions. In June 2000, 25 new instructions were added to the original 3DNow! specification. In this text, we will actually be considering this extended 3DNow! architecture.

MAX-2 [7], which we discuss in this chapter, was Hewlett-Packard's multimedia extension for its 64-bit PA-RISC 2.0 processors [6]. MAX-2 included a few new instructions; especially subword permutation instructions over MAX-1, to better exploit the increased subword parallelism in 64-bit registers. Like MAX-1, MAX-2 was also a minimalist set of general-purpose media acceleration primitives. Neither included very specialized instructions found in other multimedia extensions.

All multimedia extensions referred in this chapter have the same basic goal: to allow high-performance media processing, or native signal processing on a general-purpose processor. The key idea shared by all of these extensions to achieve this goal is the use of subword parallelism. The instructions included in the extensions are commonly based on operating in parallel on packed data types. As we will address in the following sections, significant differences exist among different ISAs and extensions, in the types and the sizes of the subwords, as well as for the support provided for these subwords.

A Note on Instruction Formatting

Throughout this chapter, we assume that all the instructions (with the possible exclusion of loads and stores) use registers for operand and target fields. The first register in an instruction is the target register and all the remaining registers are the source registers. We index the registers so that the highest index always corresponds to the target register, whereas the source registers appear in increasing indices, starting from *a*. For example, we may represent an add operation as follows:

ADD R_c,R_a,R_b

R_c is the target register, whereas R_a and R_b are the first and the second source registers respectively. For AltiVec and IA-64, where some instructions may have one target and three source fields, R_d is used to represent the target register. VSUMMBM, which will be explained in section 4, is such an AltiVec instruction and it is represented as follows:

VSUMMBM R_d,R_a,R_b,R_c

R_d is the target register, whereas R_a,R_b and R_c are the first, second and the third source registers respectively.

Our initial assumption that all the instructions use registers for source and target fields is not always true. MMX and SSE-2 are two important exclusions. Multimedia instructions in these extensions may use a memory location as a source operand. Thus, using our default representation for such instances will not be conforming to the rules of that particular architecture. However, to keep the notation simple and consistent, this distinction will not be observed, except for being noted here. For the exact instruction formatting and source-target register ordering, the reader is referred to the architecture manuals listed in the references.

2. PACKED ADD / SUBTRACT OPERATIONS

Packed add/subtract operations are nothing but regular add/subtract operations operating in parallel on the subwords of two source registers. Regular (i.e. non-packed) and packed add operations are shown in Figures 2.1 and 2.2 respectively. The packed add operation in Figure 2.2 uses source registers with four subwords each. The corresponding subwords from the two source registers are summed up, and the four sums are written to the target register. Figure 2.3 shows a packed subtract operation that operates on registers holding four subwords each.

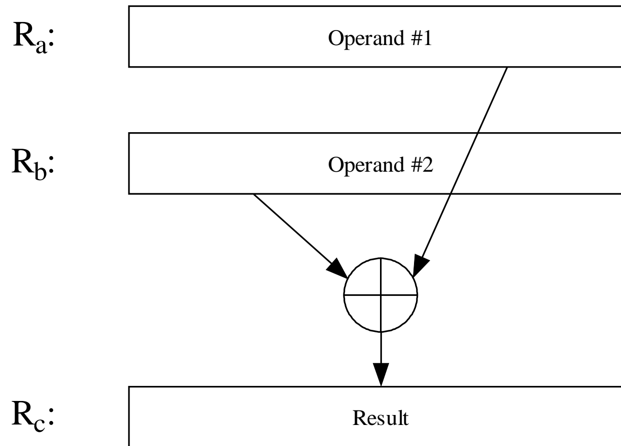

Figure 2.1 ADD[4] R_c,R_a,R_b : In a typical add operation, two source registers are added and the sum is written to the target register.

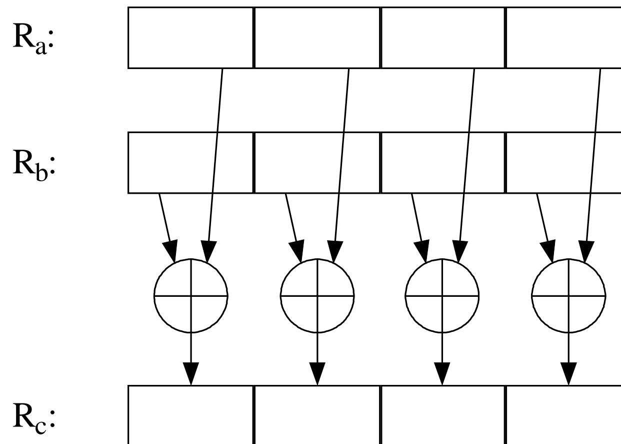

Figure 2.2 PADD R_c,R_a,R_b : Packed add operation. Each register has four subwords.

[4] *For details on instruction formatting used in this discussion, please refer to the endnote that follows section 1.*

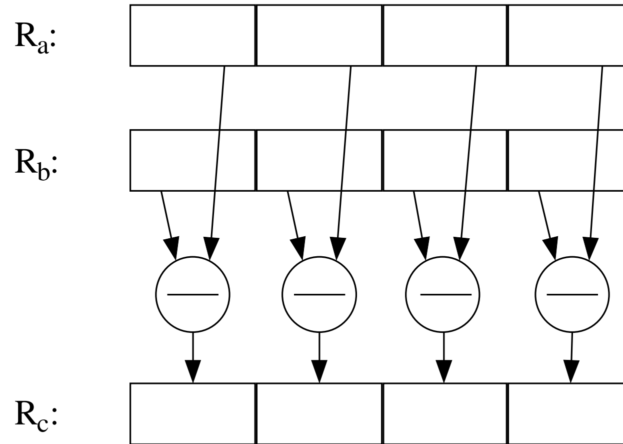

Figure 2.3 PSUB R_c,R_a,R_b : Packed subtract operation. Each register has four subwords.

Implementing Packed Instructions

Very minor modifications to the underlying functional units are needed to implement packed add and subtract operations. Assume that we have an ALU with 32-bit integer registers, and we want to extend this ALU to perform a *packed add* operation that will operate on four 8-bit subwords in parallel. Since subwords are independent, the carry bits generated by the addition of a particular subword pair should not be allowed to affect the sums of other subword pairs. Therefore, to implement this packed add operation, it is necessary and sufficient to block the carry propagation across the subword boundaries.

In Figure 2.4, the packed integer register R_a=[0xFF|0x0F|0xF0|0x00] is being added to another packed register R_b=[0x00|0xFF|0xFF|0x0F]. The result is written to the target register R_c. If the regular addition instruction is used to add these packed registers, the overflows generated by the addition of the second and third subwords will propagate into the first two sums. The correct sums, however, can be achieved easily by blocking the carry bit propagation across the subword boundaries, which are spaced 8 bits apart from one another.

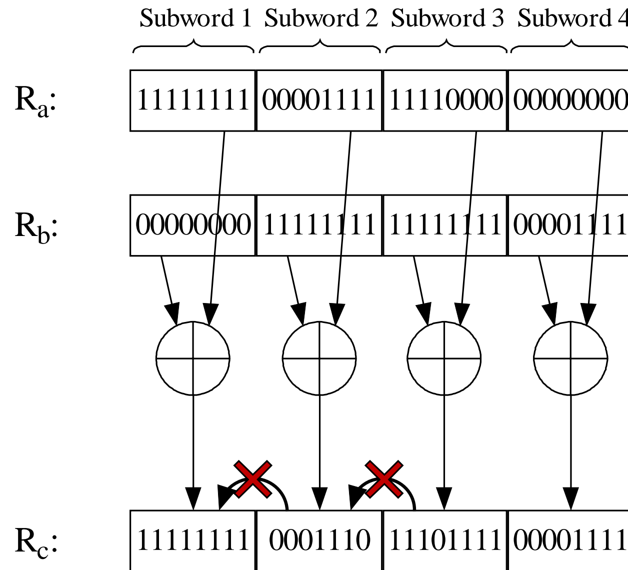

Figure 2.4 To get the correct results in this packed add operation, the carry bits are not propagated into the first and second sums.

As shown in Figure 2.5, a 2-to-1 multiplexer placed at the subword boundaries of the adder can be used to control the propagation or the blocking of the carry bits. If the instruction is a packed add, the multiplexer control is set such that a 0 is propagated into the next subword. If the instruction is a regular add, the multiplexer control is set such that the carry from the previous stage is propagated. By placing such a multiplexer at each subword boundary and adding the control logic, the support for packed add operations will be added to this ALU. If multiple subword sizes must be supported, more multiplexers may be required. In this case, the multiplexer control gets more complicated; nevertheless the area cost is still very insignificant for the performance provided by such microSIMD instructions.

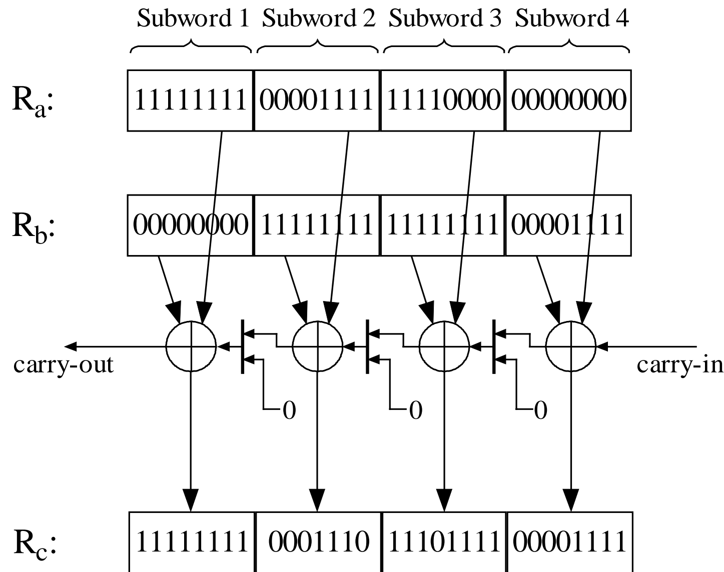

Figure 2.5 In a packed add instruction the multiplexers propagate 0. In a regular add instruction the multiplexers pass on the carry-out of the previous stage into the carry-in input of the next stage.

Packed Subtract Operations

By using 3-to-1 multiplexers instead of 2-to-1 multiplexers, we can also implement packed subtract instructions. In this case, the multiplexer control is set such that:

- For packed add instructions, 0 is propagated into the next stage,
- For packed subtract instructions, 1 is propagated into the next stage,
- For regular add/subtract instructions, the carry bit from the previous stage is propagated into the next stage.

Propagating a 0 through a subword boundary in a packed add operation is equivalent to ignoring any overflow that might have been generated. In Figure 2.4, the two overflows generated in the second and the third subword boundaries were ignored.

Similarly, propagating a 1 through a subword boundary in a packed subtract operation is equivalent to ignoring any borrow that might have been generated.

Ignoring overflows translates into the use of modular arithmetic in add operations. While this can be desirable, there are times when the carry bits should not be ignored and have to be handled differently. The next section addresses these needs and proposes an interesting solution, known as *saturation arithmetic*.

Handling Parallel Overflows

How the overflows are handled in packed add/subtract operations is an important issue. Whenever an overflow is generated, any one of the following actions can be taken:

- The overflow may be ignored (modular arithmetic),
- A flag[5] bit may be set if at least one overflow is generated,
- Multiple flag bits (i.e. one flag bit for each addition operation on the subwords) may be set,
- A software trap can be taken,
- The results may be limited to within a certain range. If the outcome of the operation falls outside this range, the corresponding limiting value will be the result. This is the basis of saturation arithmetic, which will be explained in detail below.

Most non-packed integer add/subtract instructions choose to ignore overflows and perform modular arithmetic. In modular arithmetic, the numbers wrap around from the largest representable number to the smallest representable number. For example, in 8-bit modular arithmetic, the operation 254+2 will give out 0 as result. The expected result, 256, is larger than the largest representable number, which is 255, and therefore is wrapped around to the smallest representable number, which is 0.

Even though modular arithmetic may be an option in packed add/subtract operations as well, there can be specific applications where it cannot be used, and the overflows have to be handled differently. If the numbers in the previous example were pixel values in a grayscale image, by wrapping the values from 255 down to 0, we would have essentially converted white pixels into black ones. This would be an example where modular arithmetic could not be used. One solution to this problem is to use overflow traps, which are implemented in software.

An overflow trap can be used to *saturate* the results so that:

- Any result that is greater than the largest representable value is replaced by that largest value, and
- Any result that is less than the smallest representable value is replaced by that smallest value.

---

[5] *A flag bit is an indicator bit that is set or cleared depending on the outcome of a particular operation. In the context of this discussion, an overflow flag bit is an indicator that is set when an ADD operation generates an overflow. There are occasions where the use of the flag bits are desirable. Consider a loop that iterates many times and in each iteration, performs many ADD operations. In this case, it is not desirable to handle overflows (by taking overflow trap routines) as soon as they occur, since this would negatively impact the performance by interrupting the execution of the loop body. Rather, the overflow flag can be set when the overflow occurs, and the program flow may be resumed as if the overflow did not occur. At the end of each iteration however, this overflow flag can be checked and the overflow trap can be*

The downside of the overflow trap approach is that, since it is handled in software it may take many clock cycles to complete its execution. This can only be acceptable if the overflows are infrequent.

For non-packed add/subtract operations, overflows can be rare enough to justify the use of software traps. Generation of an overflow on a 64-bit register by adding up 8-bit quantities will be rare. In such a case, a software overflow trap will work well. On the other hand, with modular arithmetic implemented in hardware, packed arithmetic operations are much more likely to generate multiple overflows frequently. Generating an overflow in an 8-bit subword is much more likely than in a 64-bit register. Moreover, since a 64-bit register may hold eight 8-bit subwords, there is actually a chance of multiple overflows in a single execution cycle. In such cases, handling the overflows by software traps may severely degrade performance. The time required to process software traps could easily exceed the time saved by executing packed operations. The use of saturation arithmetic comes up as a remedy to this problem.

Saturation Arithmetic

Saturation arithmetic implements in hardware the work done by the overflow trap described above. The results falling outside the allowed numeric ranges are saturated to the upper and lower limits by hardware. This can handle multiple parallel overflows efficiently without any operating system intervention, which can degrade performance.

There can be two types of overflows in arithmetic operations:

- A *positive overflow* occurs when the result is larger than the largest value that is in the defined range for that result, and
- A *negative overflow* occurs when the result is smaller than the smallest value that is in the defined range for that result.

If saturation arithmetic is used in an operation, the result is clipped to the maximum value in its defined range if a positive overflow occurs, and to the minimum value in its defined range if a negative overflow occurs.

For a given instruction, multiple saturation options may exist, depending on whether the operands and the result are treated as signed or unsigned integers. For an instruction that uses three registers (two for source operands and one for the result), there can be eight[6] different saturation options. However, not all of

---

executed if the flag turns out to be set. This way, the program flow would not be interrupted while the loop body executes.

[6] *Each one of the three registers can be treated as containing either a signed or an unsigned integer, which gives* 2^3 *possible combinations.*

these eight options are necessary or even useful; and in practice, multimedia extensions typically prefer to use the following three options:

a) ***sss*** (signed result-signed first operand-signed second operand):

In this saturation option, the result and the two operands are all treated as signed integers (or signed fixed-point numbers). If the operands and the result are 16-bit integer subwords, their most significant bits are considered as the sign bits. The result and the operands are defined in the range $[-2^{15}, 2^{15}-1]$. If a negative or positive overflow occurs, the result is saturated to either -2^{15} or to $2^{15}-1$ respectively. Consider an addition operation that uses the *sss* saturation option. Since the operands are signed numbers, a positive overflow is possible only when both operands are positive. Similarly, a negative overflow is possible only when both operands are negative.

b) ***uuu*** (unsigned result-unsigned first operand-unsigned second operand):

In this saturation option, the result and the two operands are all treated as unsigned integers (or unsigned fixed-point numbers). Considering 16-bit integer subwords, the result and the operands are defined in the range $[0, 2^{16}-1]$. If a negative or positive overflow occurs, the result is saturated to either 0 or to 2^{16} respectively. Consider an addition operation that uses the *uuu* saturation option. Since the operands are unsigned numbers, negative overflow is not a possibility. However, for a subtraction operation using the *uuu* saturation, negative overflow is possible, and any negative result will be clamped to 0.

c) ***uus*** (unsigned result-unsigned first operand-signed second operand):

In this saturation option, the result and the first operand are treated as unsigned integers (or unsigned fixed-point numbers), and the second operand is treated as a signed integer (or as a signed fixed-point number). This option is useful since it allows the addition of a signed increment to an unsigned pixel. As we will see in examples, it also allows negative numbers to be clipped to 0.

Uses of Saturation Arithmetic

In addition to efficient handling of parallel overflows, saturation arithmetic also facilitates several other useful computations:

- Saturation arithmetic can be used to clip results to arbitrary maximum or minimum values. Without saturation arithmetic, these operations could normally take up to five instructions. That would include instructions to check for bounds and then to perform the clipping. Using saturation arithmetic however, this effect can be achieved in as few as two instructions (Table 2.1).

- By using saturation arithmetic, conditional statements can be evaluated in two or three instructions, without the need for conditional branch instructions. Some examples are packed maximum operation shown in Figure 2.6(a) and packed absolute difference operation shown in Figure 2.6(b).

Register					Instruction
R_a:	58	14	12	77	
R_b:	22	192	118	36	

a) $c_i = \max(a_i,b_i)$

Register					Instruction
R_c:	36	0	0	41	PSUB,uuu R_c,R_a,R_b
R_c:	58	192	118	77	PADD R_c,R_c,R_b

b) $c_i = |a_i - b_i|$

Register					Instruction
R_e:	36	0	0	41	PSUB,uuu R_e,R_a,R_b
R_f:	0	178	106	0	PSUB,uuu R_f,R_b,R_a
R_c:	36	178	106	41	PADD R_c,R_e,R_f

Figure 2.6 (a) Packed maximum operation using saturation arithmetic. (b) Packed absolute difference operation using saturation arithmetic. If no saturation option is given, modular arithmetic is assumed.

Table 2.1 contains examples of operations that can be performed using saturation arithmetic. All of the instructions in the table use three registers. The first register is the target register. The second and the third registers hold the first and the second operands respectively. PADD/PSUB operations are packed add/subtract operations. The three-letter field after the instruction mnemonic specifies which saturation option is to be used. If this field is empty, modular arithmetic is assumed. All the examples in the table operate on 16-bit integer subwords.

TABLE 2.1 Examples of operations that are facilitated by saturation arithmetic. This table describes the contents of the registers (e.g. *a* or *b)* as if they contained a single value for simplicity. The same description applies to all the subwords in the registers. Initial contents of R_a and R_b are *a* and *b* unless otherwise noted.

Operation	Instruction Sequence	Notes
Clip *a* to an arbitrary maximum value v_{max}. [$v_{max}<(2^{15}-1)$]	PADD.*sss* R_a, R_a, R_b	R_b contains the value $(2^{15}-1-v_{max})$. If $a>v_{max}$, this operation clips *a* to $2^{15}-1$ on the high end.
	PSUB.*sss* R_a, R_a, R_b	*a* is at most v_{max}.
Clip *a* to an arbitrary minimum value v_{min}. [$(v_{min}>-2^{15})$]	PSUB.*sss* R_a, R_a, R_b	R_b contains the value $(-2^{15}+v_{min})$. If $a<v_{min}$, this operation clips *a* to -2^{15} at the low end.
	PADD.*sss* R_a, R_a, R_b	*a* is at least v_{min}.
Clip *a* to within the arbitrary range [v_{min}, v_{max}]. [$-2^{15}<v_{min}<v_{max}<(2^{15}-1)$]	PADD.*sss* R_a, R_a, R_b	R_b contains the value $(2^{15}-1-v_{max})$. This operation clips *a* to $2^{15}-1$ on the high end.
	PSUB.*sss* R_a, R_a, R_d	R_d contains the value $(2^{15}-1-v_{max}+2^{15}-v_{min})$. This operation clips *a* to -2^{15} at the low end.
	PADD.*sss* R_a, R_a, R_e	R_e contains the value $(-2^{15}-v_{min})$. This operation clips *a* to v_{max} at the high end and to v_{min} at the low end.
Clip the signed integer *a* to an unsigned integer within the range [0, v_{max}]. [$0<v_{max}<(2^{15}-1)$]	PADD.*sss* R_a, R_a, R_b	R_b contains the value $(2^{15}-1-v_{max})$. This operation clips *a* to $2^{15}-1$ at the high end.
	PSUB.*uus* R_a, R_a, R_b	This operation clips *a* to v_{max} at the high end and to 0 at the low end.
Clip the signed integer *a* to an unsigned integer within the range [0, v_{max}]. [$(2^{15}-1)<v_{max}<2^{16}-1$]	PADD.*uus* R_a,R_a,0	If a<0, then a=0 else a=a If *a* was negative, it gets clipped to 0, else remains same.
c=max(a,b) *Maximum* operation: It writes the greater subword to the target register.	PSUB.*uuu* R_c, R_a, R_b	If a>b, then c=(a-b) else c=0
	PADD R_c, R_b R_c	If a>b, then c=a else c=b
c=|a-b| *Absolute difference* operation: It writes the absolute value of the difference of the two subwords to the target register.	PSUB.*uuu* R_e, R_a, R_b	If a>b, then e=(a-b) else e=0
	PSUB.*uuu* R_f, R_b, R_a	If a<=b, then f=(b-a) else f=0
	PADD R_c, R_e, R_f	If a>b, then c=(a-b) else c=(b-a)

Comparison of the Architectures

As far as the packed add/subtract instructions are concerned, differences among architectures are in the register and subword sizes, and supported saturation options.

- IA-64[7] architecture has 64-bit integer registers. Packed add/subtract operations are supported for subword sizes of 1, 2 and 4 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss*, *uuu* and *uus*) exist for only 1 and 2-byte subwords.

- PA-RISC MAX-2 architecture has 64-bit integer registers. Packed add/subtract instructions operate on only 2-byte subwords. MAX-2 instructions support modular arithmetic, and *sss* and *uus* saturation options.

- IA-32 MMX architecture defines eight 64-bit registers for use by the multimedia instructions. Although these registers are given special names, these are indeed aliases to eight registers in the FP data register stack. Supported subword sizes are 1, 2 and 4 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss* and *uuu*) exist for only 1 and 2-byte subwords.

- IA-32 SSE-2 technology introduces a new set of eight 128-bit FP registers to the IA-32 architecture. Each of the 128-bit registers can accommodate four SP or two double-precision (DP) FP numbers. Moreover, these registers can also be used to accommodate packed integer data types. Integer subword sizes can be 1, 2, 4 or 8 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss* and *uuu*) exist for only 1 and 2-byte subwords.

- PowerPC Altivec architecture has 32 128-bit registers. Packed add/subtract operations are defined for 1, 2 and 4-byte subwords. Packed adds can use either modular arithmetic or, *uuu* or *sss* saturation. Packed subtracts can use only modular arithmetic or *uuu* saturation.

Table 2.2 contains a summary of the register and subword sizes and the saturation options for the architectures we discuss.

TABLE 2.2 Summary of the integer register and subword sizes for the different architectures. Not every saturation option indicated is applicable to every subword size. 3DNow! does not have an entry in this table since it does not have integer microSIMD extensions.

Architecture	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**Altivec**
Size of integer registers (bits)	64	64	64	128	128
Number of registers	128	32	8	8	32
Supported subword sizes (bytes)	1, 2, 4	2	1, 2, 4	1,2,4,8	1, 2, 4
Modular arithmetic	Y	Y	Y	Y	Y
Supported saturation options	*sss*, *uuu*, *uus*	*sss*, *uus*	*sss*, *uuu*	*sss*,*uuu*	*sss*, *uuu*

[7] *All the discussions in this chapter consider IA-64 as the base architecture. Evaluations of the other architectures are generally carried out by comparisons to IA-64.*

Other Instructions

AltiVec architecture includes two operations for picking up the carry bits from a packed add or subtract operation. The VADDCUW instruction performs a packed add operation and writes the carry-bits to a right-aligned field in the target register. Figure 2.7 shows this instruction. The VSUBCUW instruction performs a similar operations using packed subtract instead of packed add.

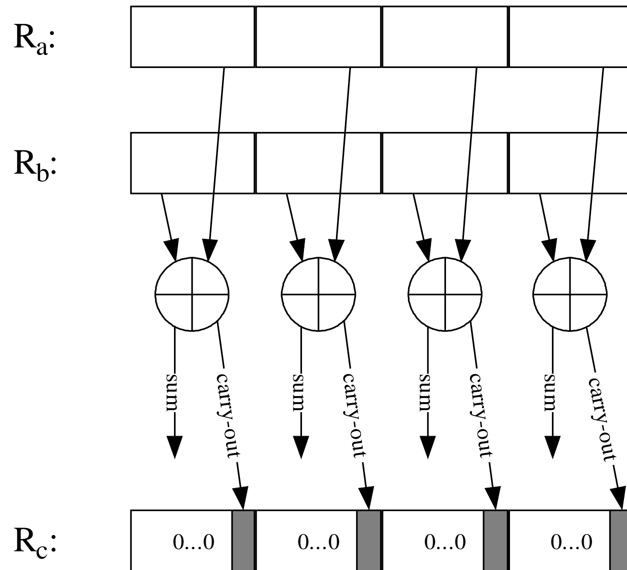

Figure 2.7 VADDCUW R_c,R_a,R_b : VADDCUW instruction of AltiVec writes the carry-out bits of the parallel adds to the least significant bits of the corresponding subwords of the target register. The sums are ignored.

Some instructions of SSE-2 are classified as *scalar* instructions. These instructions operate on packed data types but only the least significant subwords of the two source operands are involved in the operation. The other subwords of the target register are directly copied from the first source operand. Example of a scalar FP add operation is shown in Figure 2.8.

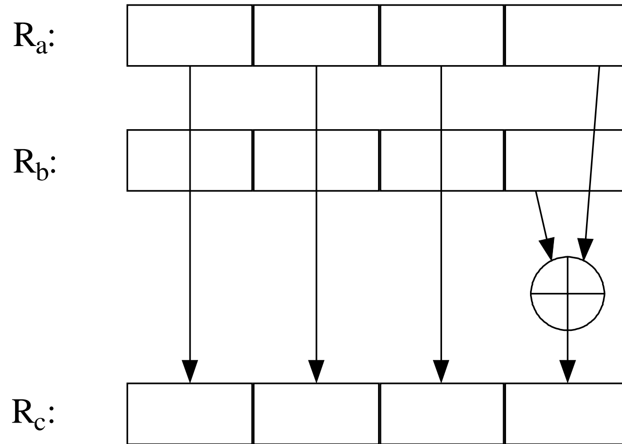

Figure 2.8 Scalar_ADD R_c,R_a,R_b : Scalar add instruction (ADDSS) as defined by SSE and SSE-2 architectures. This instruction uses registers with four subwords each.

Packed Floating Point Add/Subtract Operations

IA-64, SSE-2, 3DNow! and Altivec include packed FP add/subtract instructions. Due to the format of FP numbers, there are no issues like modular arithmetic or saturation options for these instructions. The only difference that exists for packed FP add/subtract operations across various architectures is in the precision levels (Table 2.3).

TABLE 2.3 Supported precision levels for the packed FP add/subtract operations. SP and DP FP numbers are 32 and 64 bits long respectively, as defined by the IEEE-754 FP number standard.

Architecture	IA-64	SSE-2	3DNow!	AltiVec
FP register Size	82 bits	128 bits	128 bits	128 bits
Allowed packed FP data types	2 SP	4 SP or 2 DP	4 SP	4 SP

IA-64 does not have dedicated instructions for packed FP add and subtract. Instead, these operations are realized by using FPMA (Floating Point Parallel Multiply Add) and FPMS (Floating Point Parallel Multiply Subtract) instructions as explained below.

IA-64 architecture specifies 128 FP registers, which are numbered FR0 through FR127. Of these registers, FR0 and FR1 are special. FR0 always returns the value +0.0 when sourced as an operand, and FR1 always reads +1.0. When FR0 or FR1 are used as source operands, the FPMA and FPMS instructions can be used to realize packed FP add/subtract and packed FP multiply operations.

Example:

The format of the FPMA instruction is FPMA R_d,R_a,R_b,R_c and the operation it performs is $R_d = R_a * R_b + R_c$. If FR1 is used as the first or the second source operand (FPMA R_d,FR1,R_b,R_c), a packed FP add operation is realized ($R_d = FR1 * R_b + R_c = 1.0 * R_b + R_c = R_b + R_c$). Similarly, a FPMS instruction can be used to realize a packed FP subtract operation. Using FR0 as the third source operand in FPMA or FPMS (FPMA R_d,R_a,R_b,FR0) results in a packed FP multiply operation ($R_d = R_a * R_b + FR0 = R_a * R_b + 0.0 = R_a * R_b$). Section 3 includes a more detailed discussion of the packed multiply operations.

3DNow! has two packed subtract instructions for FP numbers. The only difference between these two instructions is in the order of the operands. The PFSUB instruction subtracts the second packed source operand from the first, whereas the PFSUBR instruction subtracts the first packed source operand from the second.

Table 2.4 gives a summary of the packed add/subtract operations discussed in this section.

TABLE 2.4 Packed add/subtract operations.

Integer Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3DNow!**	**Altivec**
$c_i = a_i + b_i$	√	√	√	√		√
$c_i = a_i + b_i$ (with saturation)	√	√	√			√
$c_i = a_i - b_i$	√	√	√	√		√
$c_i = a_i - b_i$ (with saturation)	√	√	√			√
$lsbit(c_i) = carryout(a_i + b_i)$						√
$lsbit(c_i) = carryout(a_i - b_i)$						√
FP Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3DNow!**	**Altivec**
$c_i = a_i + b_i$	√[8]			√[9]	√	√
$c_i = a_i - b_i$	√[10]			√[9]	√	√

In the table above, the first column contains the description of the operations. The symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register. A shaded background indicates a packed FP operation.

[8] *This operation is realized by using the FP Multiply and Add instruction.*
[9] *Scalar version of this instruction also exists.*
[10] *This operation is realized by using the FP Multiply and Subtract instruction.*

3. PACKED SPECIAL ARITHMETIC OPERATIONS

This section describes variants of the packed add instructions that are generally designed to further increase performance in multimedia applications.

Packed Averaging

All the architectures we refer to include instructions to support a packed average operation. Packed average operations are very common in media applications such as pixel averaging in MPEG-2 encoding, motion compensation and video scaling.

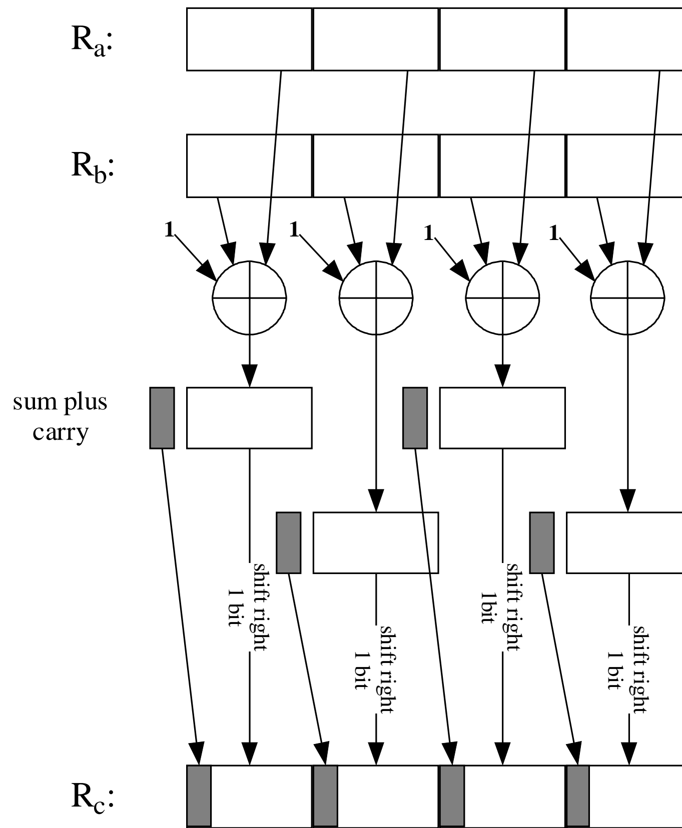

Figure 3.1 PAVG R_c,R_a,R_b : Packed averaging operation using the *round away from zero* option.

In a packed averaging operation, the pairs of corresponding subwords in the two source registers are added to generate intermediate sums. From this point, two different paths may be taken, depending on which rounding option is used. These are addressed below.

1. Round away from zero:

A 1 is added to the intermediate sums, and then the sums are shifted to the right by one bit position. If carry bits were generated during the addition operation, they are inserted into the most significant bit position during the shift right operation. Figure 3.1 shows a packed averaging operation that uses this rounding option.

2. Round to odd:

Instead of adding 1 to the intermediate sums, a much simpler *or* operation is used. The intermediate sums are directly shifted right by one bit position, and the last two bits of each of the subwords of the intermediate sums are ORed to give the least significant bit of the final result. This makes sure that the least significant bit of the final results are set to 1 (odd) if at least one of the two least-significant bits of the intermediate sums are 1.

If the intermediate result is uniformly distributed over the range of possible values, then half of the time, the bit shifted out is 0, and the result remains unchanged with rounding. The other half of the time, the bit shifted out is one: if the next least significant bit is one, then the result loses –0.5, but if the next least significant bit is a 0, then the result gains +0.5. Since these cases are equally likely with uniform distribution of the result, this round to odd option tends to cancel out the cumulative averaging errors that may be generated with repeated use of the averaging instruction. Hence, it is said to provide *unbiased rounding*. Figure 3.2 shows a packed averaging operation that uses this rounding option.

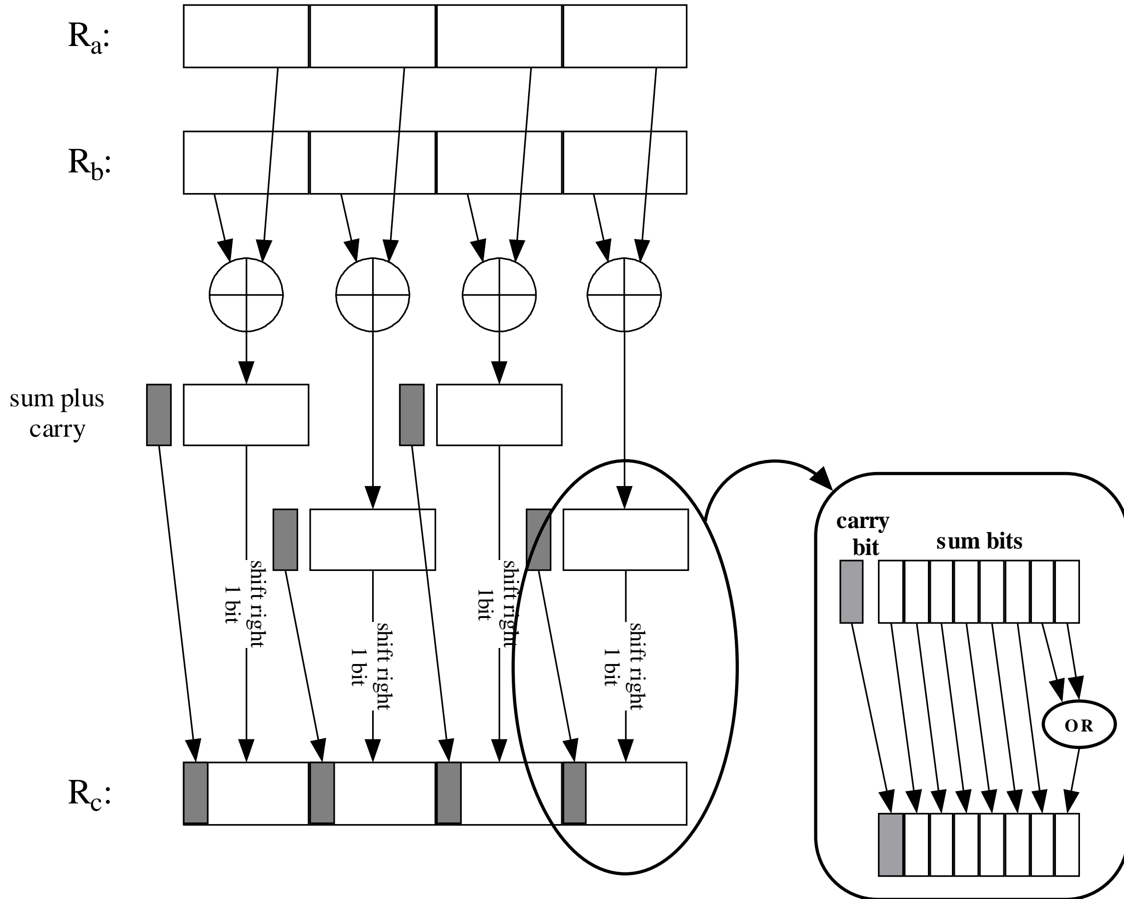

Figure 3.2 PAVG R_c,R_a,R_b : Packed averaging operation using the *round to odd* option.

Accumulate

AltiVec provides an instruction to facilitate the accumulation of streaming data. This instruction performs an addition of the subwords in the same register and places the sum in the upper half of the target register, while repeating the same process for the second source register and using the lower half of the target register (Figure 3.3).

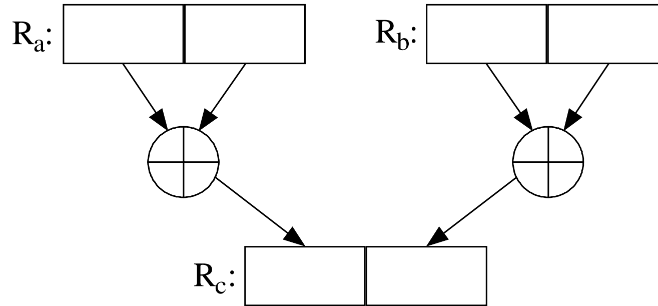

Figure 3.3 ACC R_c,R_a,R_b : Accumulate operation working on registers with two subwords

Sum of Absolute Differences

Sum of Absolute Differences (SAD) is another operation that proves useful in media applications. The motion estimation kernel in the MPEG-2 video encoding application is one example that can benefit from a SAD operation. In a SAD operation, the two packed operands are subtracted from one another. Absolute values of the resulting differences are then summed up. This sum is placed into the target register. Figure 3.5 shows how the SAD operation works. Most architectures feature a special instruction for this operation.

While useful, the SAD instruction is a multi-cycle instruction, with a typical latency of 3 cycles. This can complicate the pipeline control of otherwise single cycle integer pipelines. Hence, minimalist multimedia instruction sets like MAX-2 do not have SAD instructions. Instead, MAX-2 uses generic PADDs and PSUBs with saturation arithmetic to perform the SAD operation (see Figure 2.6(b) and Table 2.1).

Table 3.1 gives a summary of the packed special arithmetic operations discussed in this section.

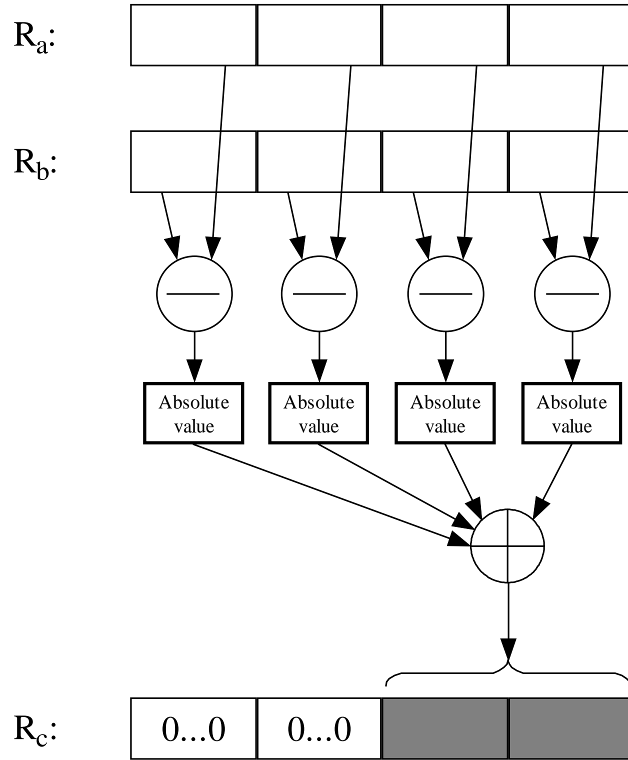

Figure 3.5 SAD R_c,R_a,R_b : Sum of absolute differences operation.

TABLE 3.1 Packed special arithmetic operations.

Integer Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3DNow!**	**Altivec**
$c_i = avg(a_i, b_i)$	√	√		√	√	√
$c_i = neg_avg(a_i, b_i)$	√					
$c = \sum_i \lvert a_i - b_i \rvert$	√			√	√	
Accumulate Integer						√
FP Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3DNow!**	**Altivec**
$c_i = -a_i$	√					
$c_i = \lvert a_i \rvert$	√					
$c_i = -\lvert a_i \rvert$	√					

In the table above, the first column contains the description of the operations. The symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register. A shaded background indicates a packed FP operation.

4. PACKED MULTIPLY OPERATIONS

We begin this section by explaining the multiplication of a packed integer register by a constant number. Next, we consider the more general case of multiplying two packed registers. Finally, we will give some examples of some more specialized instructions that involve packed multiplication operations.

Multiplication of a Packed Integer Register by an Integer Constant

Consider the multiplication of an unpacked integer register by an integer constant. This can be accomplished by a sequence of shift left instructions and add instructions. Shifting a register left by *n* bits is equivalent to multiplying it by 2^n. Since a constant number can be represented as a binary sequence of 1s and 0s, using this number as a multiplier is equivalent to a left shift of the multiplicand of *n* bits for each nth position where there is a 1 in the multiplier and an add of each shifted value to the result register.

As an example, consider multiplying the unpacked integer register R_a with the constant C=11. The following instruction sequence performs this multiplication. Assume R_a=6.

Initial values are: $C=11=1011_2$ $R_a=6=0110_2$		
Instruction	**Operation**	**Result**
Shift_left_by_1 R_b,R_a	$R_b=R_a<<1$	$R_b=1100_2=12$
Add R_b,R_b,R_a	$R_b = R_b + R_a$	$R_b=1100_2+0110_2=010010_2=18$
Shift_left_by_3 R_c,R_a	$R_c=R_a<<3$	$R_c=0110_2*8=110000_2=48$
Add R_b,R_b,R_c	$R_b = R_b + R_c$	$R_b=010010_2+110000_2=1000010_2=66$

This sequence can be shortened by combining the *shift left* instruction and *add* instruction into one new *shift left and add* instruction. The following new sequence performs the same multiplication in half as many instructions and uses one less register.

Initial values are: $C=11=1011_2$ $R_a=6=0110_2$		
Instruction	**Operation**	**Result**
Shift_left_by_1_and_add R_b,R_a,R_a	$R_b=R_a<<1+R_a$	$R_b=18$
Shift_left_by_3_and_add R_b,R_a,R_b	$R_b=R_a<<3+R_b$	$R_b=66$

Multiplication of packed integer registers by integer constants uses the same idea explained above. The *shift left and add* instruction becomes a *packed shift left and add* instruction to support the packed data types. As an example consider multiplying the subwords of the packed integer register R_a=[1|2|3|4] by the constant *C*=11. The instructions to perform this operation are:

Initial values are: *C*=11=1011_2 R_a=[1|2|3|4]=$[0001\|0010\|0011\|0100]_2$		
Instruction	**Operation**	**Result**
Shift_left_by_1_and_add R_b,R_a,R_a	R_b=R_a<<1+R_a	R_b=$[0011\|0110\|1001\|1100]_2$ =[3|6|9|12]
Shift_left_by_3_and_add R_b,R_a,R_b	R_b=R_a<<3+R_b	R_b=$[1011\|10110\|100001\|1010100]_2$ =[11|22|33|44]

The same reasoning used for multiplication by constants applies to multiplication by fractional constants as well. Arithmetic right shift of a register by *n* bits is equivalent to dividing it by 2^n. Using a fractional constant as a multiplier is equivalent to an arithmetic right shift of the multiplicand by *n* bits for each nth position where there is a 1 in the multiplier and an add of each shifted value to the result register. By using a *packed arithmetic shift right and add* instruction, the shift and the add operations can be combined into one to further speed such computations. For instance, multiplication of a packed register by the fractional constant 0.011_2 (=0.375) can be performed by using only two *packed arithmetic shift right and add* instructions.

Initial values are: *C*=0.375=0.011_2 R_a=[1|2|3|4]=$[0001\|0010\|0011\|0100]_2$		
Instruction	**Operation**	**Result**
Arithmetic_Shift_right_by_3_and_add R_b,R_a,0	R_b=R_a>>3+0	R_b=$[0.001\|0.01\|0.011\|0.1]_2$ =[0.125|0.25|0.375|0.5]
Arithmetic_Shift_right_by_2_and_add R_b,R_a,R_b	R_b=R_a>>2+R_b	R_b=[0.011|0.11|1.001|1.1] =[0.375|0.75|1.125|1.5]

These three examples demonstrate the pathlength reduction that can be achieved by the use of multimedia extensions in a general-purpose processor. Only two instructions were necessary to multiply four integer subwords by a constant number. Without subword parallelism, the same operations would take at least four instructions. Considering that a 128-bit register can contain 16 8-bit subwords, the extent of pathlength reduction becomes more pronounced.

In the previous example, it was assumed that the *packed shift right and add* instruction shifted each of the subwords of the source operand by the same amount specified in the immediate operand. It may be that the shift amount is specified in a register or memory operand instead of being specified in an immediate field. This, however, would require the use of three source registers by the *packed shift right and add* instruction. A different *packed shift left* instruction may shift each of the subwords in the source operand by a different amount, which may be given in a register or an immediate operand. *Packed shift left* instructions may cause overflows, which should be detected and handled using one of the ways explained in section 2. *Packed shift right* instructions do not have this problem. For a *shift right* operation, the shift can be arithmetic or logical. All these options create many possibilities for how a particular *packed shift* operation will function. Section 6 will address these details.

The next sub-section discusses different ways of multiplying two packed integer registers. As we shall see shortly, when both of the operands are registers, packed multiplication operations become tricky to handle, since each product is now twice the length of each multiplicand.

Multiplication of two Packed Integer Registers

The most straightforward way to define a packed multiplication is to multiply each subword in the first source register by the corresponding subword in the second source register. This however, is impractical since the target register now has to be twice the size of each of the source registers. Two same size multiplicands will produce a product with a size that is double the size of each of the multiplicands.

Consider the case in Figure 4.1 where the register size is 64-bits and the subwords are 16-bits. The result of the packed multiplication will be four 32-bit products, which cannot be accommodated in a target register, which is only 64 bits wide.

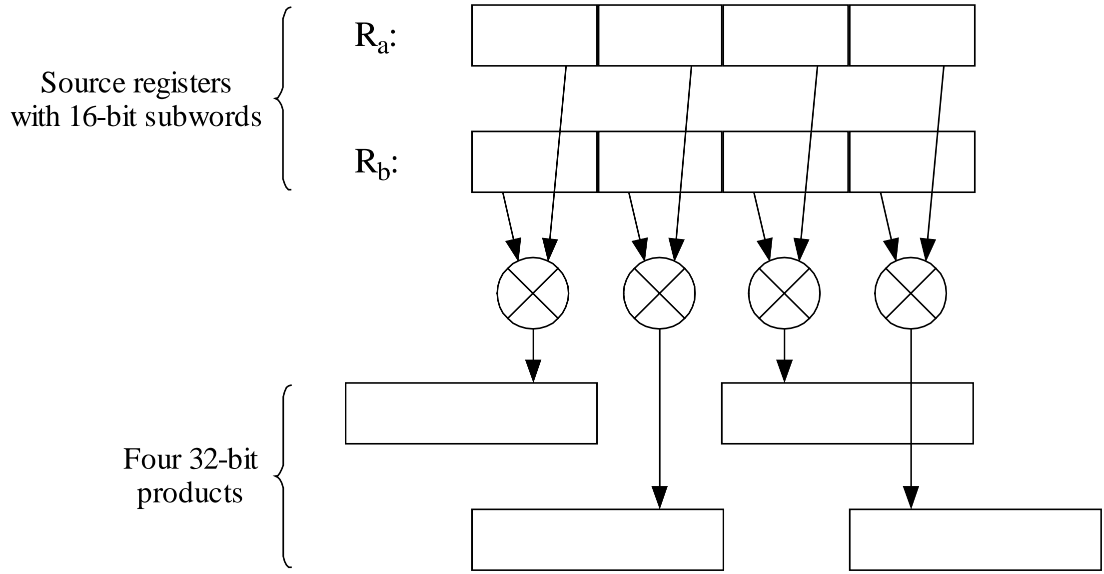

Figure 4.1 Packed multiplication operation using four 16-bit subwords per register. Not all of the full sized products can be accomodated in a single target register.

The architectures approach this problem in different ways. MMX has the PMULHW instruction that only places the most significant upper halves of the products into the target register. Similar to this instruction, SSE-2 has the PMULHUW, AltiVec has the VMUL, and the 3DNow! has the PMULHRW and PMULHUW instructions. All these instructions pick the higher order bits from the products and place them in the target register. These instructions are depicted in Figure 4.2 for the case of four subwords.

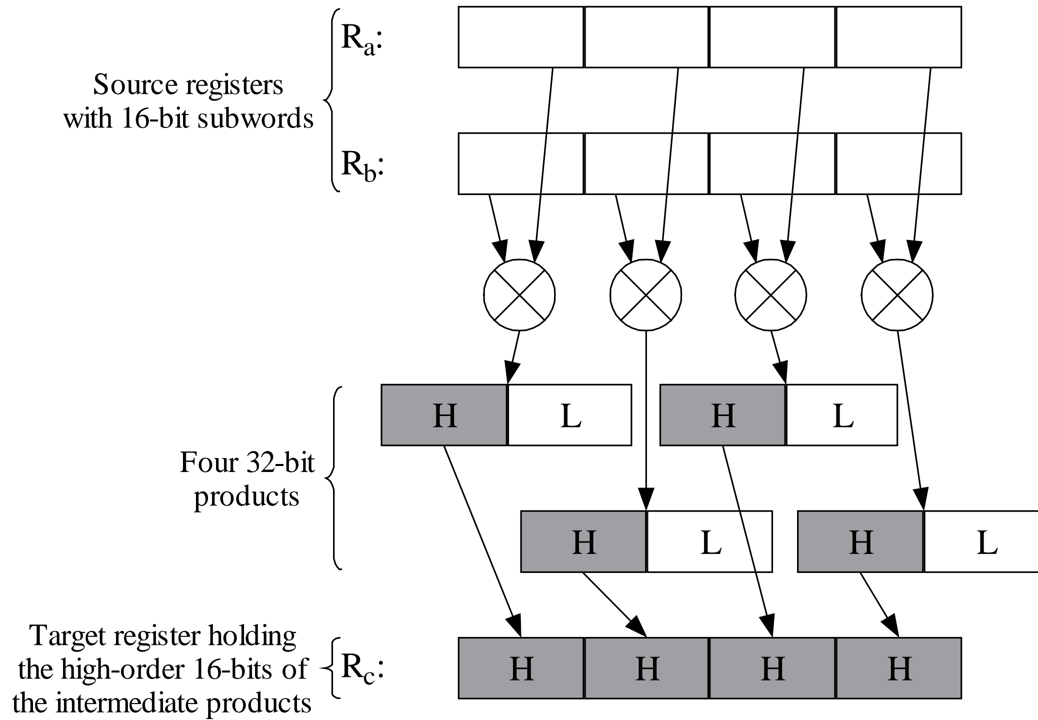

Figure 4.2 PMUL.high R_c,R_a,R_b : Packed multiplication operation using four subwords per register. Only the high order bits of the intermediate products are written to the target register.

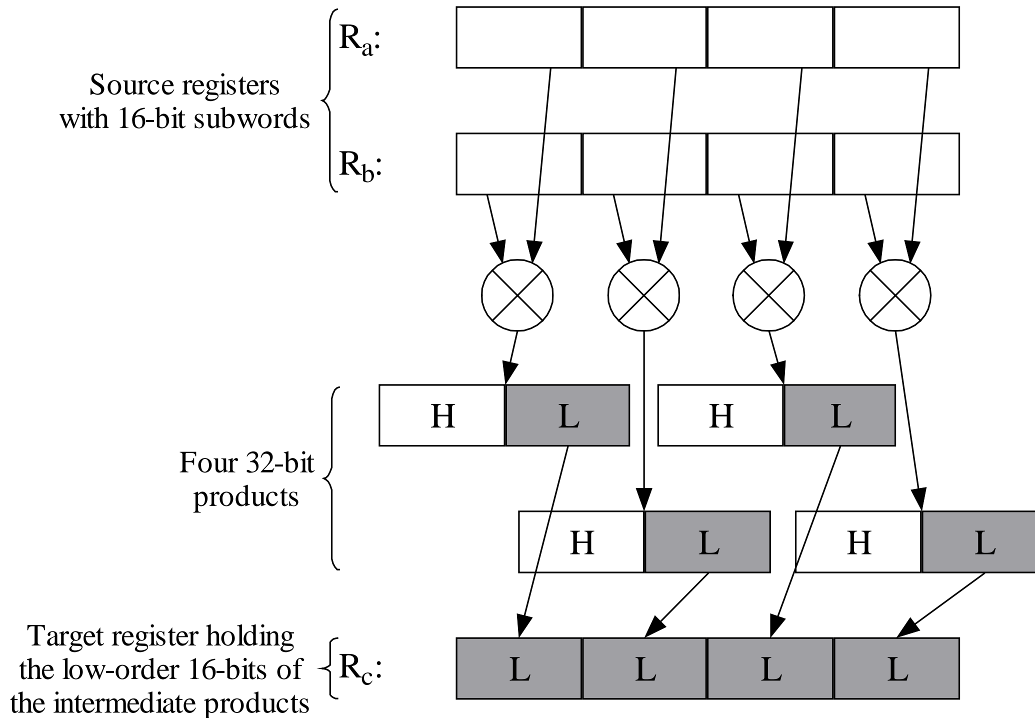

Figure 4.3 PMUL.low R_c,R_a,R_b : Packed multiplication operation using four subwords per register. Only the low order bits of the intermediate products are written to the target register.

The other approach can be to keep the least significant halves of the products in the target register. Some examples to this are the MMX's PMULLW and AltiVec's VMUL. These instructions work as shown in Figure 4.3.

IA-64 architecture comes up with a generalization of this, with its PMPYSHR instruction. This instruction lifts the limit that one has to choose either the upper or the lower half of the products to be put into the target register. PMPYSHR instruction does a parallel multiplication followed by a shift right operation. This allows four[11] possible 16-bit fields (IA-64 has a 64-bit register size) from each of the 32-bit products to be chosen and be placed in the target register. The PMPYSHR instruction is shown in Figure 4.4.

[11] *The right-shift amounts are limited to 0,7,15 or 16 bits. This limitation allows a reduction in the number of bits necessary to encode the instruction.*

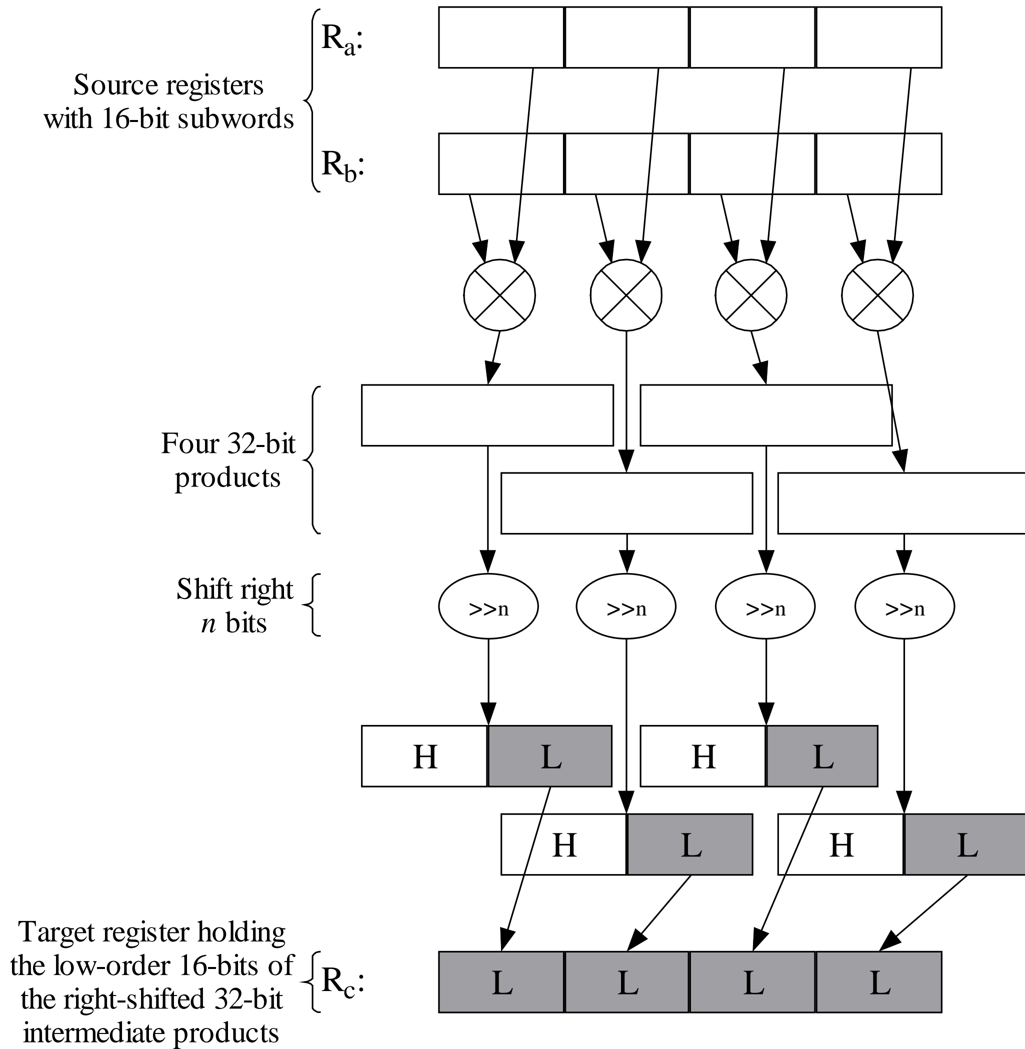

Figure 4.4 PMPYSHR R_c,R_a,R_b : The PMPYSHR instruction of the IA-64 architecture. The shift amount, which is given in the immediate register, is limited to 0,7,15 and 16 bits.

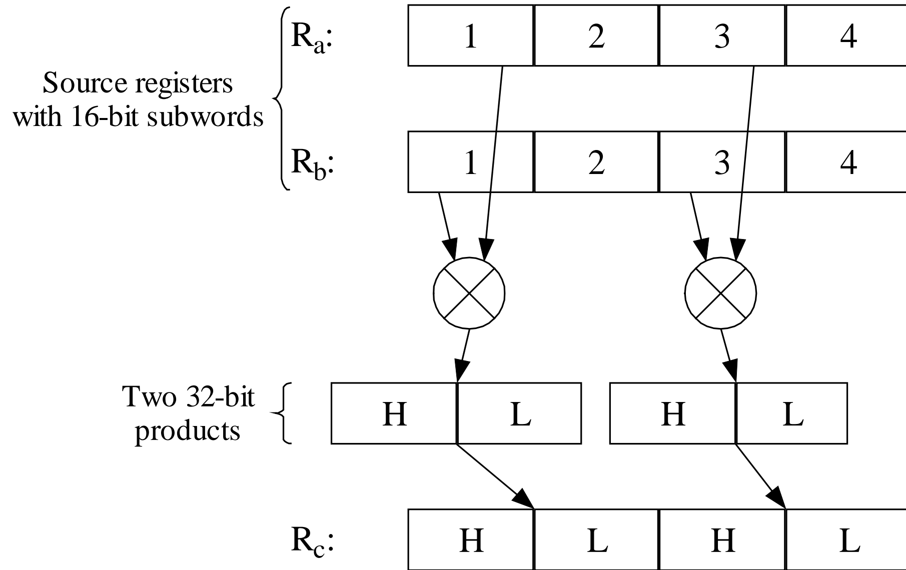

Figure 4.5 PMUL.odd R_c,R_a,R_b : Packed multiplication operation where only the odd indexed subwords of the two source registers are multiplied.

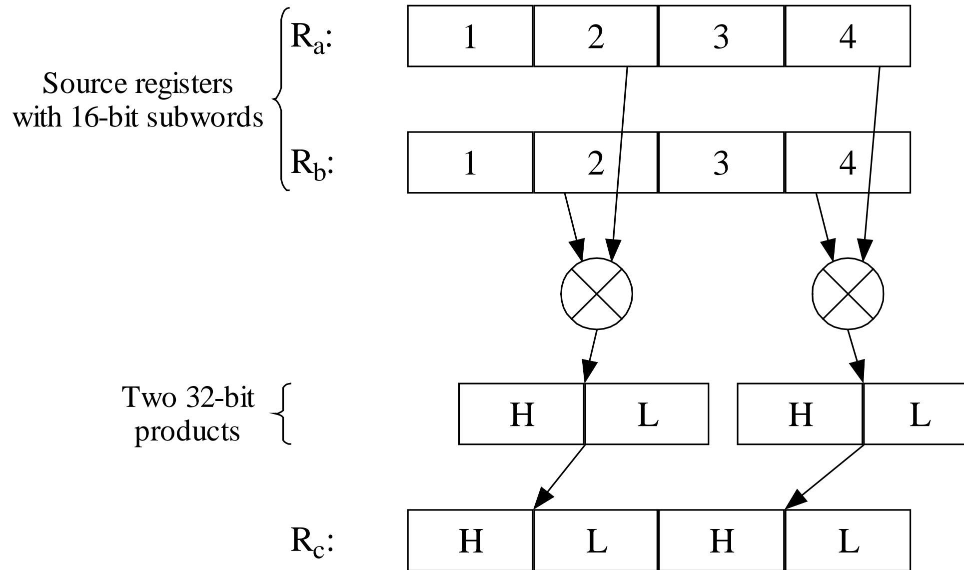

Figure 4.6 PMUL.even R_c,R_a,R_b : Packed multiplication operation where only the even indexed subwords of the two source registers are multiplied.

All of the instructions we have seen thus far performed a full multiplication on all of the subword pairs of the source operands and then decided how to handle the large products. However, instead of truncating the products, the source subwords that will participate in the multiplication may be selected, so that the final product is never larger than can be accommodated in a single target register.

IA-64 has the PMPY instruction, which has two variants. PMPY allows only half of the pairs of the source subwords to go into multiplication. Either the odd or the even indexed subwords are multiplied. This makes sure that only as many full products as can be accommodated in one target register are generated. The two variants of the PMPY instruction are depicted in Figures 4.5 and 4.6.

Other Variants of Packed Integer Multiplication Operations

Intel's MMX technology has an instruction that performs a packed multiplication followed by an addition. This PMADDWD instruction generates four 32-bit intermediate product terms in the packed multiply stage. Later, the first product is added to the second one, and the 32-bit sum is placed into the first half of the target register. The third and the fourth products are also summed and this sum is placed into the second half of the target register (Figure 4.7).

Instructions in the AltiVec architecture may have up to three source registers. This allows AltiVec to realize operations that require three source operands, such as the *packed multiply and add*. The instructions (VMHxADD and VMLxADD) start just like packed multiply instructions, select either the higher or lower order bits of the full-sized products, and then perform a packed addition between the values from a third register and the result of the multiplication operation. These two instructions are shown in Figures 4.8 and 4.9.

The very specialized VSUMMBM instruction of AltiVec performs a vector multiplication and a scalar addition using three packed source operands. First, all the bytes within the corresponding words are multiplied in parallel and 16-bit products are generated. Later, all these products are added to each other to generate the *sum of products*. A third word from the third source operand is added to this *sum of products*. The final sum is placed in the corresponding word field of the target register. This process is repeated for each of the four words (Figure 4.10).

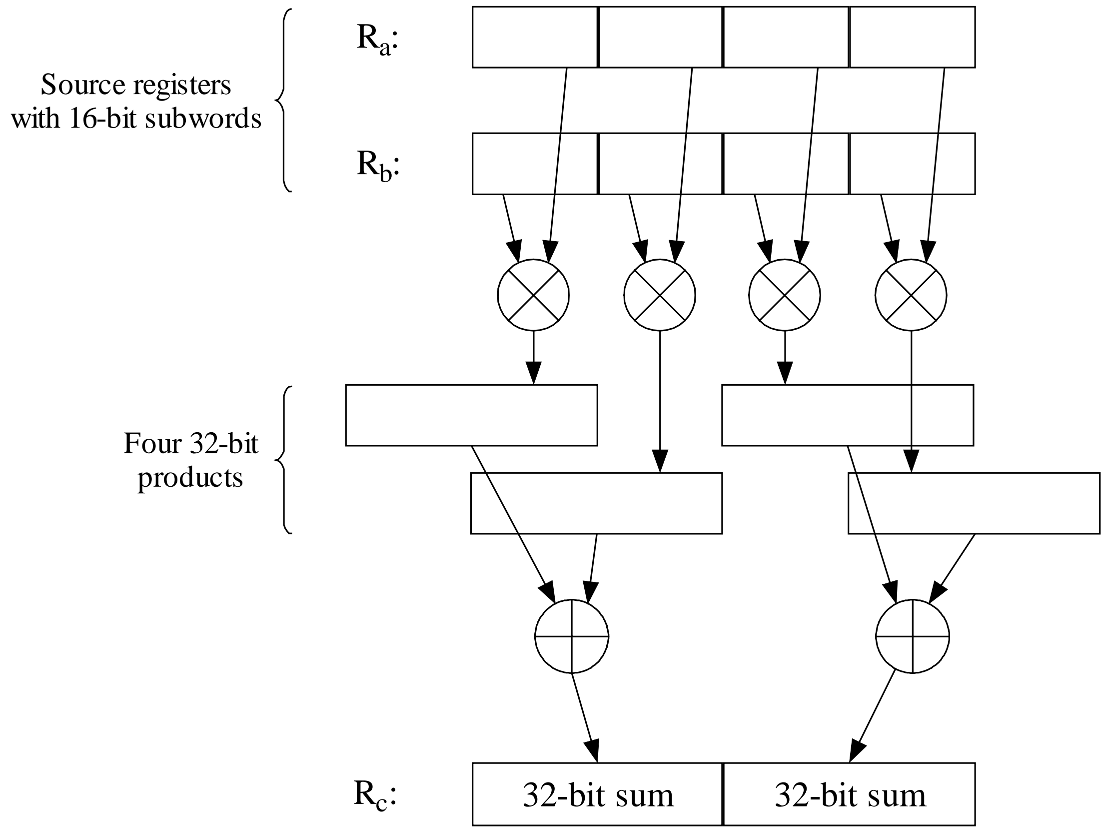

Figure 4.7. Multiply_and_accumulate R_c,R_a,R_b : The *multiply and accumulate* operation of the MMX arhitecture.

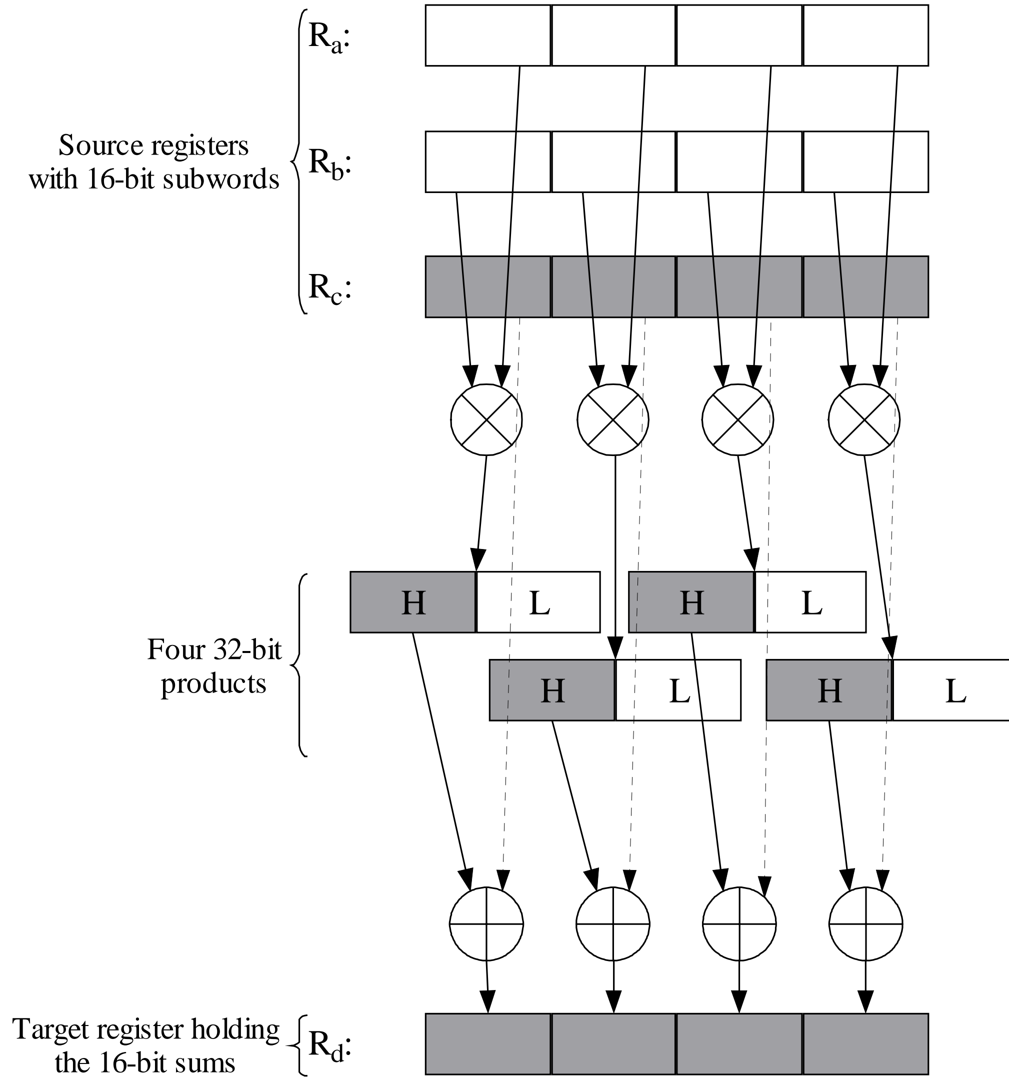

Figure 4.8. Multiply_high_and_add R_d,R_a,R_b,R_c : The *multiply and add* instruction of the AltiVec arhitecture. In this variant of the instruction, only the high order bits of the intermediate products are used in the addition.

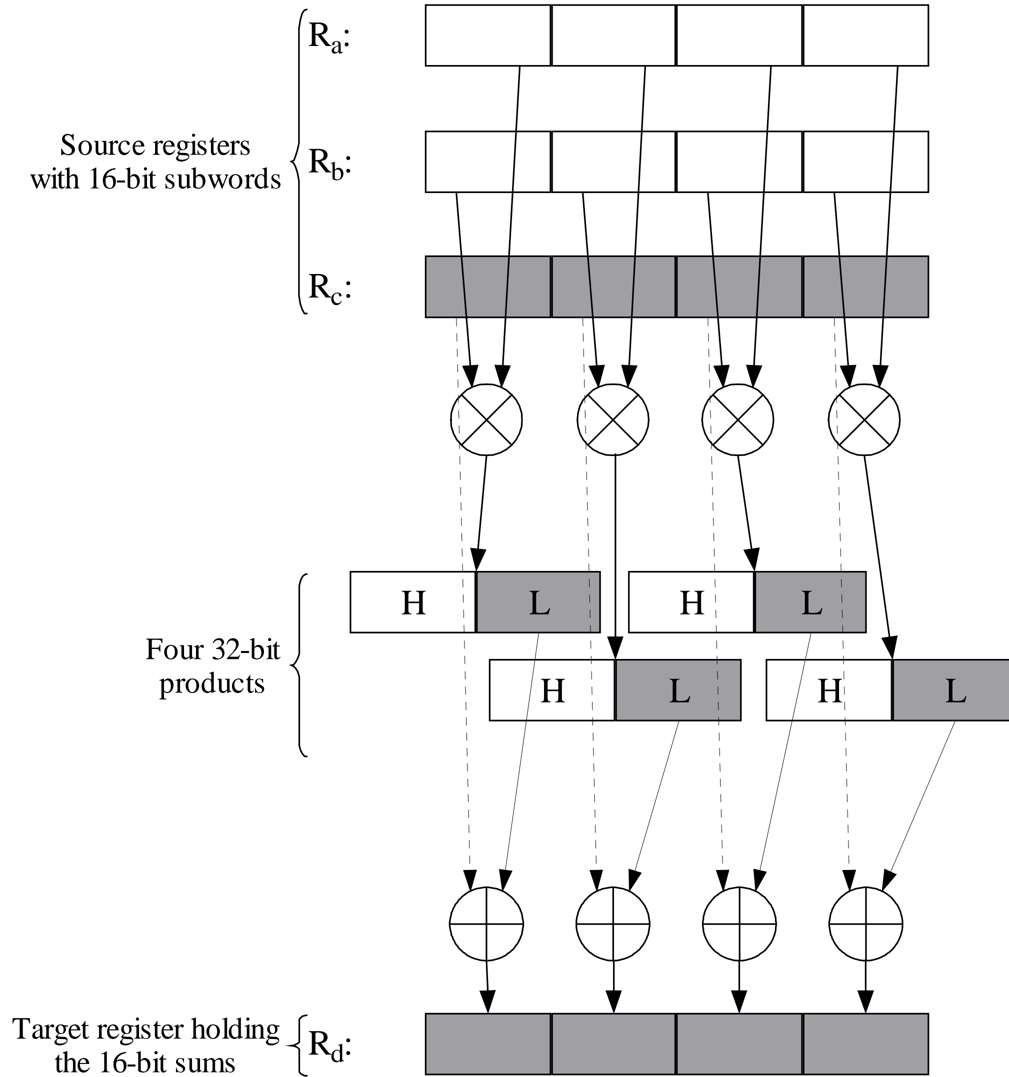

Figure 4.9. Multiply_low_and_add R_d,R_a,R_b,R_c : The *multiply and add* instruction of the AltiVec arhitecture. In this variant of the instruction, only the low order bits of the intermediate products are used in the addition.

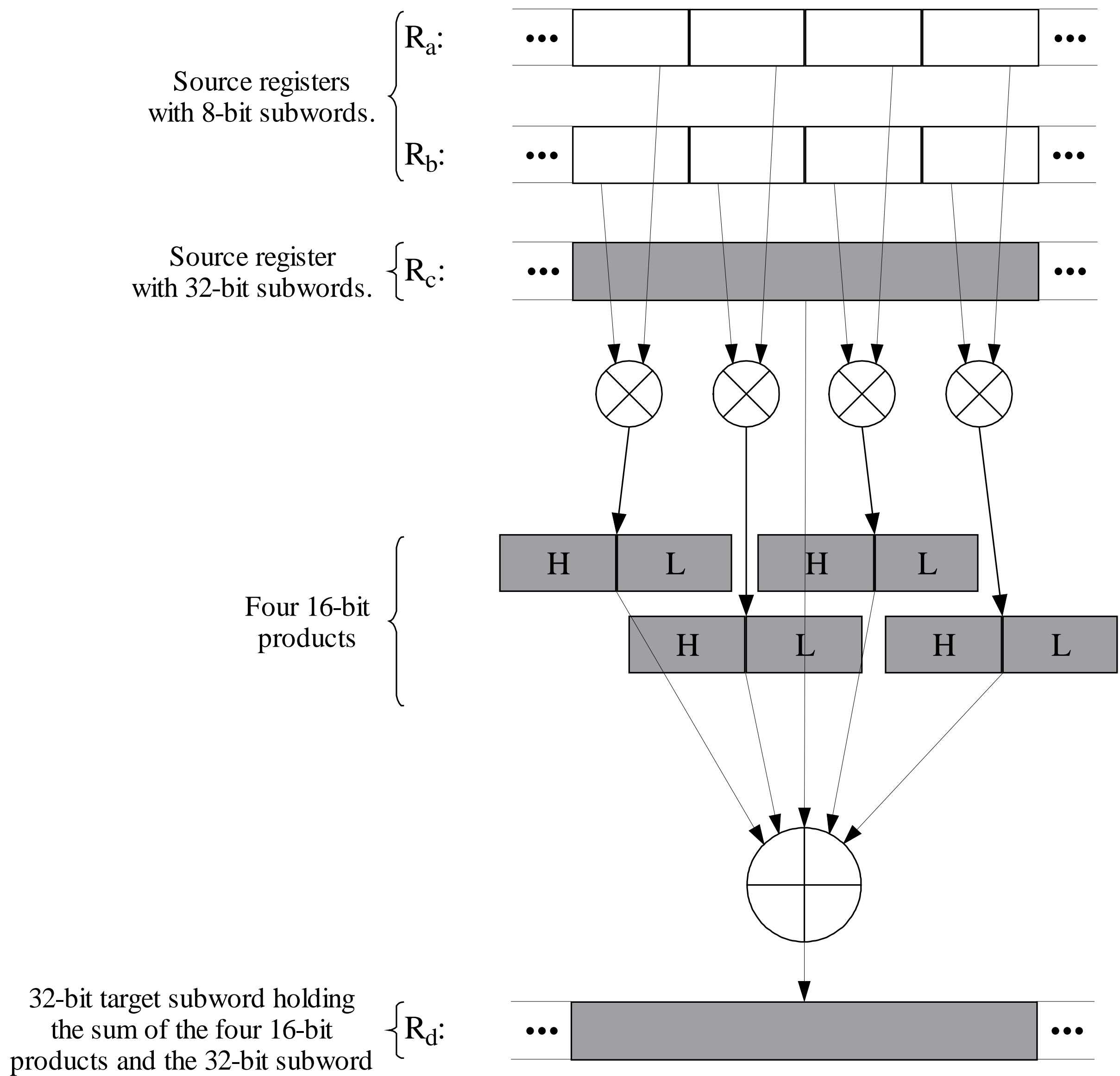

Figure 4.10 VSUMMBM R_d,R_a,R_b,R_c : AltiVec's VSUMMBM instruction proves useful in matrix multiplication operations. In this figure, only one fourth of the instruction is shown. Each box represents a byte. This process is carried out for each 32-bit word in the 128-bit source registers.

A Note about Multiplication of Packed FP registers

Packed FP multiplication does not cause the problems encountered in packed integer multiplication. In integer multiplication, the size of the product term is always greater than either of the multiplicands. This does not allow all of the product terms to be written into the target register. The special format of FP numbers does not cause such a size problem. The same number of bits[12] is used to represent a FP number regardless of how big the number is. In this respect, multiplication of packed FP registers is similar to the addition of packed FP registers. Figure 4.11 shows the multiplication of two packed FP registers, each containing four SP FP numbers.

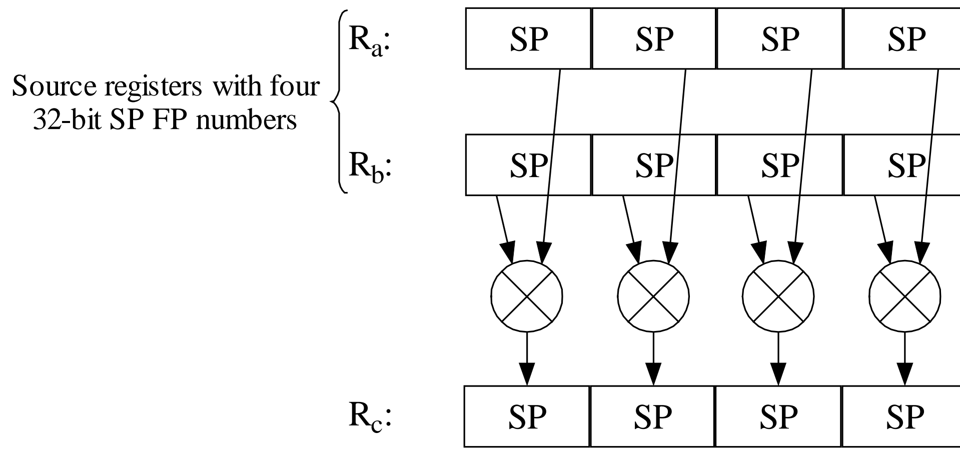

Figure 4.11 FP_PMUL R_c,R_a,R_b : Packed FP multiplication does not introduce the size problems like the packed integer multiplication. This figure shows two packed FP integers being multiplied. The packed registers have two FP numbers each.

Table 4.1 gives a summary of the packed multiply operations discussed in this section.

---

[12] *In general, 32 and 64 bits are used to represent SP and DP FP numbers respectively.*

TABLE 4.1 Packed multiply operations.

Integer Operations	IA-64	MAX-2	MMX	SSE-2	3Dnow!	Altivec
Packed Shift Left and Add[13] $c_i = (a_i << n) + b_i$	√	√				
Packed Shift Right and Add[14] $c_i = (a_i >> n) + b_i$	√	√				
$c_i = lower_half[(a_i * b_i) >> n]$	√[15]					
$c_i = lower_half(a_i * b_i)$	√		√	√	√	√
$c_i = upper_half(a_i * b_i)$	√		√	√	√	√
Multiply Even $[c_{2i}, c_{2i+1}] = a_{2i} * b_{2i}$	√					√
Multiply Odd $[c_{2i}, c_{2i+1}] = a_{2i+1} * b_{2i+1}$	√					√
Multiply and Accumulate $[c_{2i}, c_{2i+1}] = a_{2i} * b_{2i} + a_{2i+1} * b_{2i+1}$			√			
$d_i = upper_half(a_i * b_i) + c_i$						√
$d_i = lower_half(a_i * b_i) + c_i$						√
VMSUMxxx instructions of AltiVec (general form) $[d_{2i}, d_{2i+1}] = a_{2i} * b_{2i} + a_{2i+1} * b_{2i+1} + [c_{2i}, c_{2i+1}]$						√
VSUMMBM instruction of AltiVec $[d_{4i}, d_{4i+1}, d_{4i+2}, d_{4i+3}] = [c_{4i}, c_{4i+1}, c_{4i+2}, c_{4i+3}] + \sum_{j=1}^{4} a_{4i+j} * b_{4i+j}$						√
FP Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3Dnow!**	**Altivec**
$c_i = a_i * b_i$	√			√[16]	√	
$d_i = -a_i * b_i$	√					
$d_i = a_i * b_i + c_i$	√					√

---

[13] *For use in multiplication of a packed register by an integer constant.*
[14] *For use in multiplication of a packed register by a fractional constant.*
[15] *Shift amounts are limited to 0,7,15 or 16 bits.*
[16] *Scalar versions of these instructions also exist.*

$d_i = a_i * b_i - c_i$	√					
$d_i = -a_i * b_i + c_i$	√					√

In the table above, the first column contains the description of the operations. For instructions with three registers, the symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register. For instructions with four registers, the symbols a_i, b_i and c_i represent the subwords from the three source registers. The symbol d_i represents the corresponding subword in the target register. A shaded background indicates a packed FP operation.

5. PACKED COMPARE / MAXIMUM / MINIMUM OPERATIONS

In a packed compare operation, pairs of subwords are compared according to the relation specified by the instruction. If the condition is true for a subword pair, the corresponding field in the target register is written with a 1-mask. If the condition is false, the corresponding field in the target register is written with a 0-mask. Some of the architectures have compare instructions that allows comparison of two numbers for all of the 12 possible relations[17], whereas some architectures allows for a more limited subset of relations. A typical compare instruction is shown in Figure 5.1 for the case of four subwords.

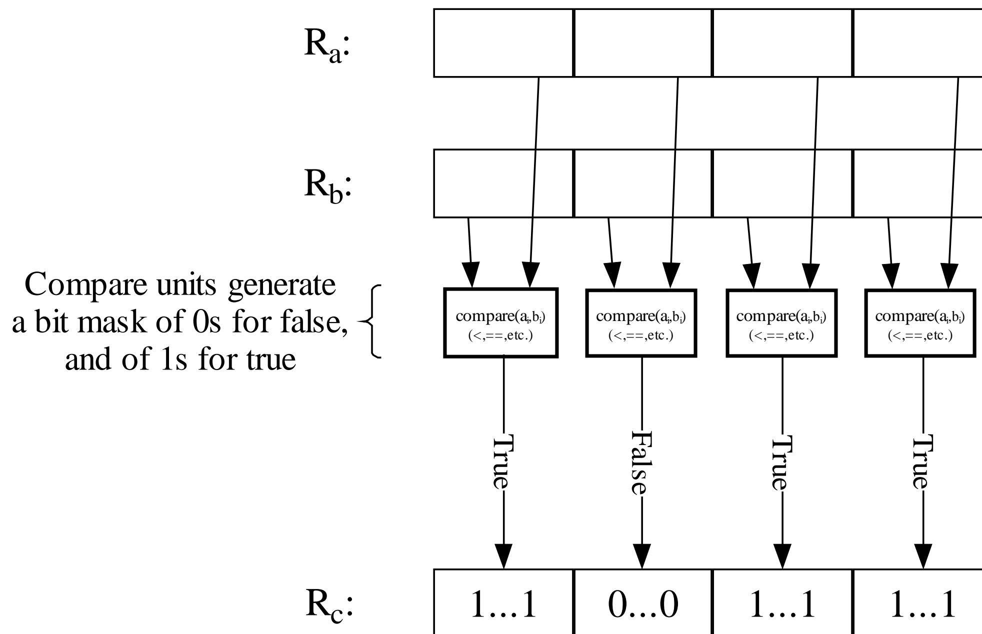

Figure 5.1 PCOMP R_c,R_a,R_b: Packed compare instruction. Bit masks are generated as a result of the comparisons made.

In the packed maximum/minimum operations, the greater/smaller of the subwords in the compared pair gets written to the corresponding field in the target register. Figures 5.2 and 5.3 illustrate the packed maximum/minimum operations. In practice, architectures either include dedicated instructions or make use of other existing instructions to perform a packed maximum/minimum operation. MAX-2, for instance, is in the second category and performs packed max/min operations by using packed add/subtract instructions with saturation arithmetic. How the saturation arithmetic can be used to realize packed max/min operations

[17] *Two numbers a and b can be compared for one of the following 12 possible relations: equal, less-than, less-than-or-equal, greater-than, greater-than-or-equal, undordered, not-equal, not-less-than, not-less-than-or-equal, not-greater-than, not-greater-than-or-equal, ordered. Typical notation for these relations are as follows respectively: ==, <, <=, >, >=, ?, !=, !<, !<=, !>, !>=, !?*

is detailed in section 2. See Figure 2.6 for an example of the packed maximum operation realized by using saturation arithmetic.

An interesting packed compare instruction is the VCMPBFP (‘Vector Compare Bounds Floating-Point’) instruction of AltiVec. This instruction compares corresponding FP number pairs from the two packed source registers, and depending on the relation between the compared numbers, it generates a two-bit result, which is written to the target register. The resulting two-bit field indicates the relation between the two compared FP numbers. For the instruction VCMPBFP R_c,R_a,R_b, the FP number pairs (a_i,b_i) are compared, and a two-bit field is written into c_i, such that:

- Bit 0 of the two-bit field is cleared if $a_i<=b_i$, and is set otherwise,
- Bit 1 of the two-bit field is cleared if $a_i>=(-b_i)$, and is set otherwise.
- Both bits are set if any of the compared FP numbers is a NaN.

The two-bit result field is written to the high-order two bits of c_i; the remaining bits of c_i are cleared to 0. Table 5.1 gives examples of input pairs that result in each of the four different possible outputs for this instruction.

TABLE 5.1 Result of the VCMPBFP instruction for different input pairs.

Input		**Output**	
a_i	b_i	**Bit 0**	**Bit 1**
3.0	5.0	0	0
-8.0	5.0	0	1
8.0	5.0	1	0
3.0	-5.0	1	1

Table 5.2 gives a summary of the packed compare/maximum/minimum operations discussed in this section.

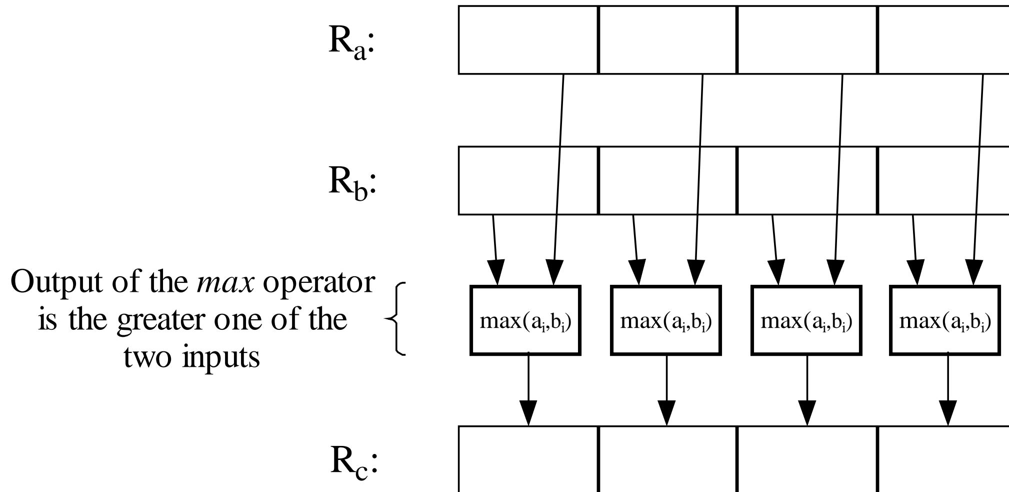

Figure 5.2 PMAX R_c,R_a,R_b: Packed maximum operation. The greater source subword is written to the corresponding location in the target register.

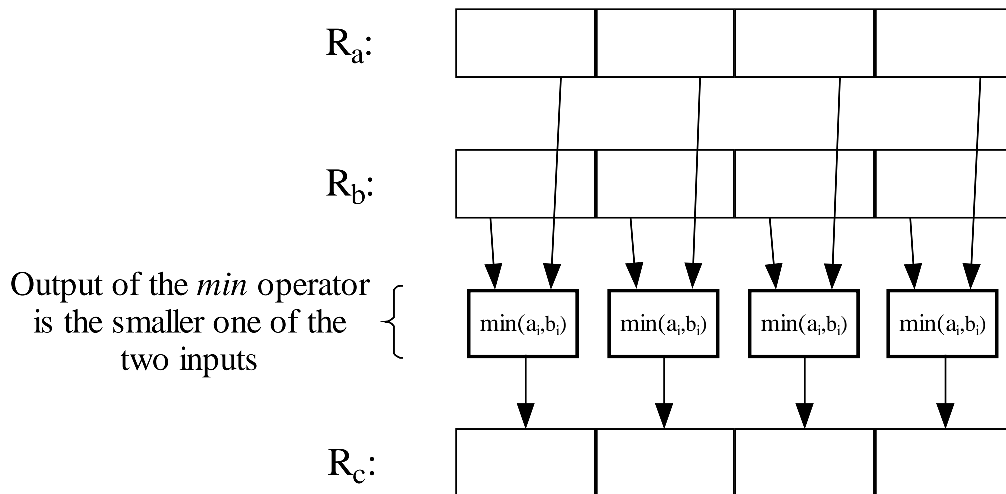

Figure 5.3 PMIN R_c,R_a,R_b: Packed minimum operation. The smaller source subword is written to the corresponding location in the target register.

TABLE 5.2 Packed compare/maximum/minimum operations.

Integer Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3Dnow!**	**Altivec**
$c_i = compare(a_i, b_i)$	√		√			√
$c_i = \max(a_i, b_i)$	√	√[18]		√	√	√
$c_i = \min(a_i, b_i)$	√	√[18]		√	√	√
FP Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3Dnow!**	**Altivec**
$c_i = compare(a_i, b_i)$	√			√[19]	√	√
$c_i = \max(a_i, b_i)$	√			√[19]	√	√
$c_i = \min(a_i, b_i)$	√			√[19]	√	√
$c_i = \max(\|a_i\|, \|b_i\|)$	√					
$c_i = \min(\|a_i\|, \|b_i\|)$	√					
$c_i = VCMPBFP(a_i, b_i)$[20]						√

In the table above, the first column contains the description of the operations. The symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register. A shaded background indicates a packed FP operation.

---

[18] *This operation is realized by using saturation arithmetic.*
[19] *Scalar versions of these instructions also exist.*
[20] *This instruction is explained in the text.*

6. PACKED SHIFT / ROTATE OPERATIONS

Most of the architectures have instructions that support shift/rotate instructions on packed data types. These instructions prove very useful in multimedia, arithmetic and encryption applications. There are usually great differences between the packed shift/rotate instructions of different architectures, since there are several options to be considered.

- A packed shift/rotate instruction shifts/rotates the subwords in a packed register.
- For the packed shift command, one has to decide if the shift will be logical (0s substituted for vacated bits) or algebraic (sign bit is replicated for vacated bits on the left).
- Saturation arithmetic may or may not be used during the packed shift operations.
- The shift/rotate amount can be given by an immediate operand or a register operand.
- Each subword may have to be shifted/rotated by the same amount, or the instruction may be sophisticated so that each subword in a packed register can be shifted/rotated by a different amount.

Given so many options, almost all architectures come up with their own solutions to the problem. The packed shift/rotate instructions are shown in Figures 6.1 to 6.4.

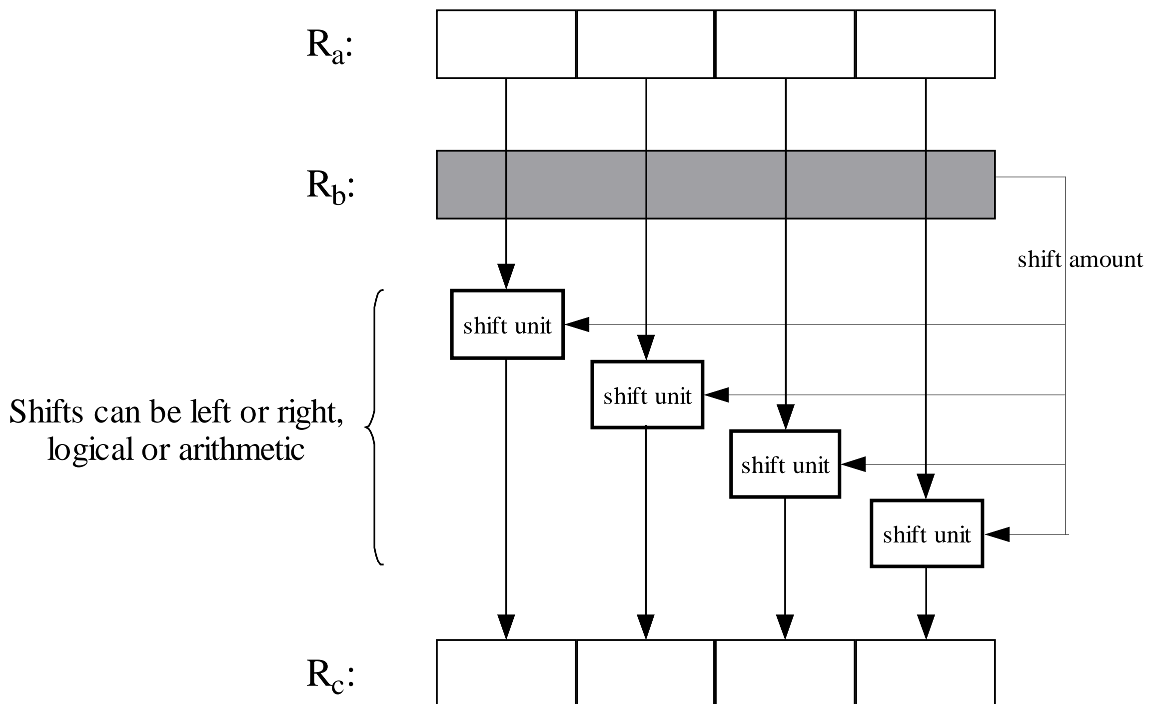

Figure 6.1 PSHIFT R_c,R_a,R_b : Packed shift operation. Shift amount is given in the second operand. Each subword is shifted by the same amount.

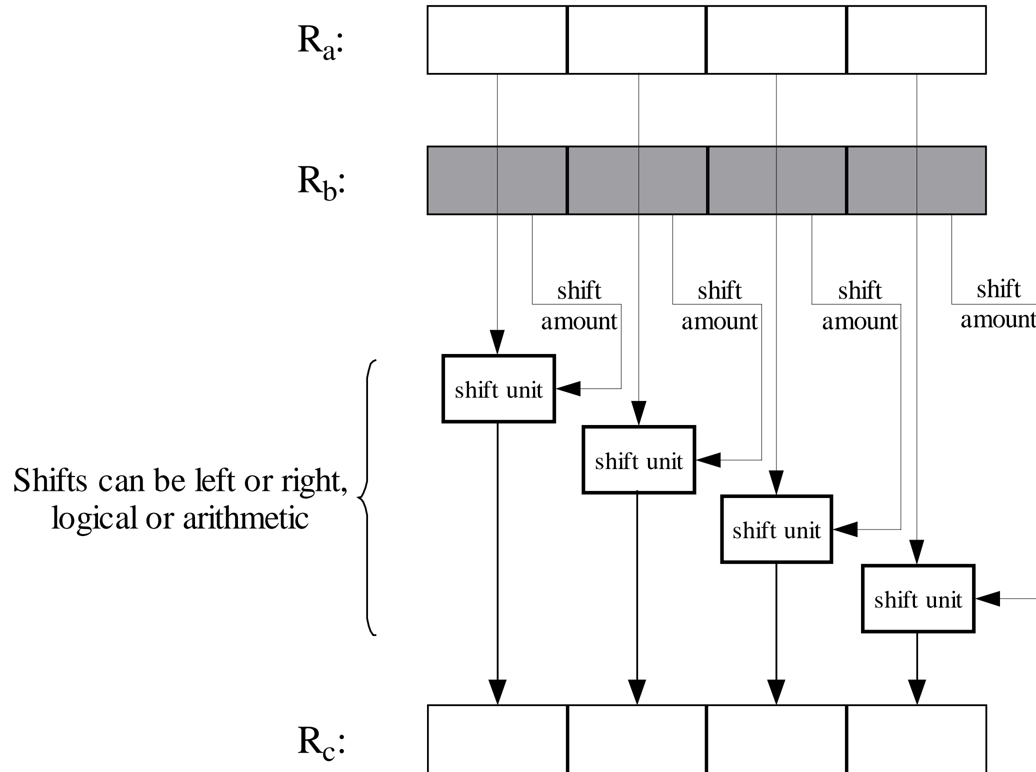

Figure 6.2 PSHIFT R_c,R_a,R_b :Packed shift operation. Shift amount is given in the second operand. Each subword can be shifted by a different amount.

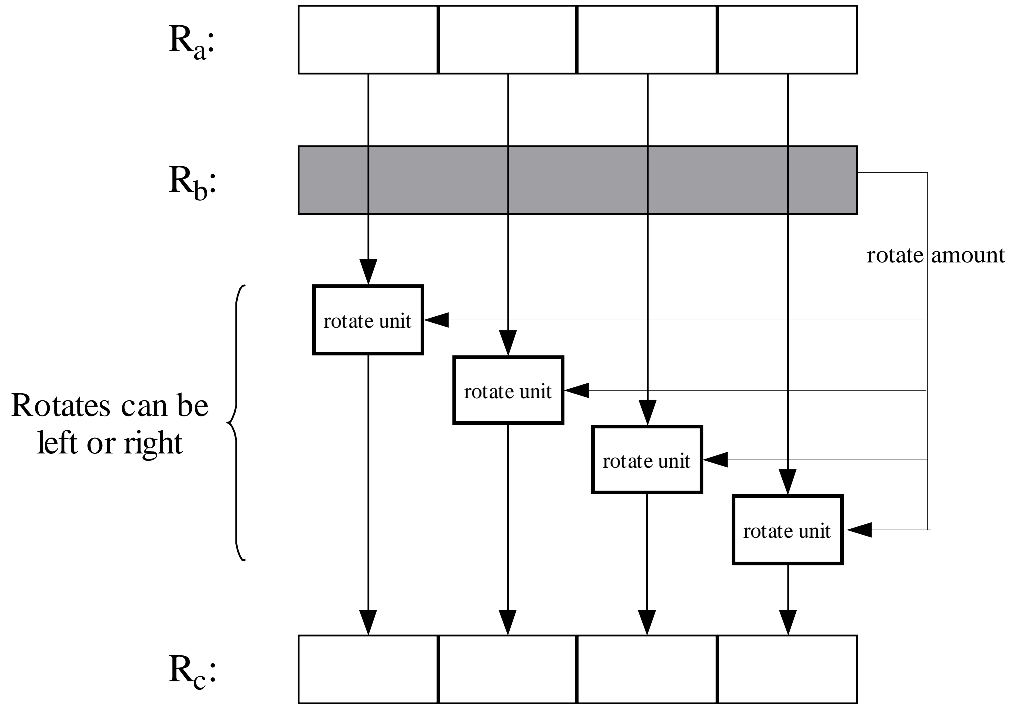

Figure 6.3 PROT R_c,R_a,R_b : Packed rotate operation. Rotate amount is given in the second operand. Each subword is rotated by the same amount.

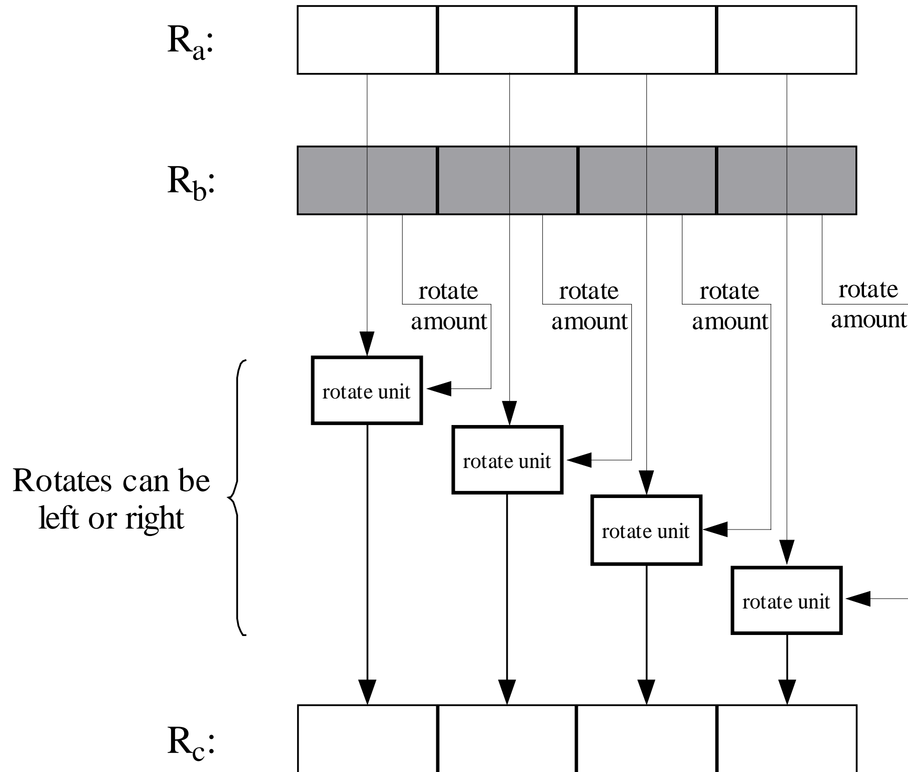

Figure 6.4 PROT R_c,R_a,R_b :Packed rotate operation. Rotate amount is given in the second operand. Each subword can be rotated by a different amount.

Table 6.1 gives a summary of the packed shift/rotate operations discussed in this section.

TABLE 6.1 Packed shift/rotate operations.

Integer Operations	IA-64	MAX-2	MMX	SSE-2	3Dnow!	Altivec
$c_i = a_i << n$	√	√	√			
$c_i = a_i << b$	√		√			
$c_i = a_i << b_i$						√
$c_i = a_i >> n$	√	√	√			
$c_i = a_i >> b$	√		√			
$c_i = a_i >> b_i$						√
$c_i = (a_i << n) + b_i$	√	√				
$c_i = (a_i >> n) + b_i$	√	√				
$c_i = a_i <<< n$						
$c_i = a_i <<< b$						
$c_i = a_i <<< b_i$						√

In the table above, the first column contains the description of the operations. The symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register.

- *n* is used to represent a shift or rotate amount that is specified in the immediate field of an instruction. Hence, in the operation denoted as $c_i = a_i << n$, each subword of *a* is shifted to the left by *n* bits. The results are placed in *c*.

- Similarly, in the operation $c_i = a_i << b$, each subword of *a* is shifted to the left by the amount specified in the source register *b*. The results are placed in *c*.

- In $c_i = a_i << b_i$, each subword of *a* is shifted to the left by the amount specified in the corresponding subword of the source register *b*. The results are placed in *c*.

- $c_i = (a_i << n) + b_i$ represents a *shift left and add* operation. Each subword of *a* is shifted to the left by *n* bits. Corresponding subwords from the source register *b* are added to the shifted values. The sums are placed in their respective locations in *c*.

- In $c_i = a_i <<< n$, each subword of *a* is rotated left by *n* bits. The results are placed in *c*. None of the architectures have this operation.

- In $c_i = a_i <<< b$, each subword of *a* is rotated to the left by the amount specified in the source register *b*. The results are placed in *c*. None of the architectures have this operation.

- In $c_i = a_i <<< b_i$, each subword of *a* is rotated left by the amount specified in the corresponding subword of the source register *b*. The results are placed in *c*.

7. DATA / SUBWORD PACKING AND REARRANGEMENT OPERATIONS

The instructions for data/subword packing and rearrangement are most interesting and have the widest variety among different architectures.

Pack Instructions

All architectures include instructions for conversion between different packed data types. In general, the pack instructions are used to create packed data types from unpacked data types. A pack instruction can also be used to further pack an already-packed data type. Figure 7.1 shows how a packed data-type can be created from two unpacked operands. Figure 7.2 shows how two packed data-types can be packed further using a pack instruction. Differences between any two pack instructions are generally in the size of the supported subwords and in the saturation options that can be used.

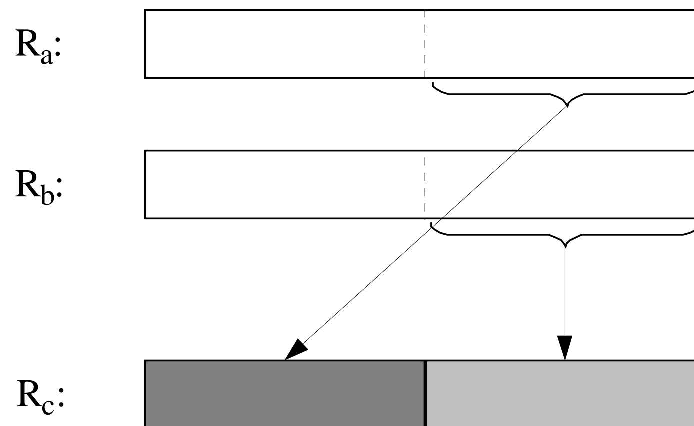

Figure 7.1 PACK R_c,R_a,R_b : Pack operation is used to create packed data types from unpacked data types.

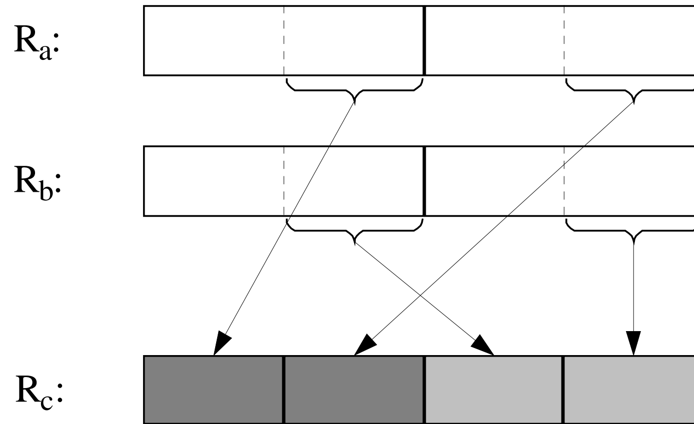

Figure 7.2 PACK R_c,R_a,R_b : Pack operation can be used to further pack already packed data types.

Unpack Instructions

Unpack instructions are used to unpack the packed data types. The subwords in the two source operands are split and written to the target register in alternating order. Since only one half of each of the source registers can be used, the unpack instructions always come with two variants: high or low unpack. These options allow the user to select which subwords in the source operand will be unpacked to the target register. The high/low unpack instructions select and unpack the high/low order subwords of the source operands.

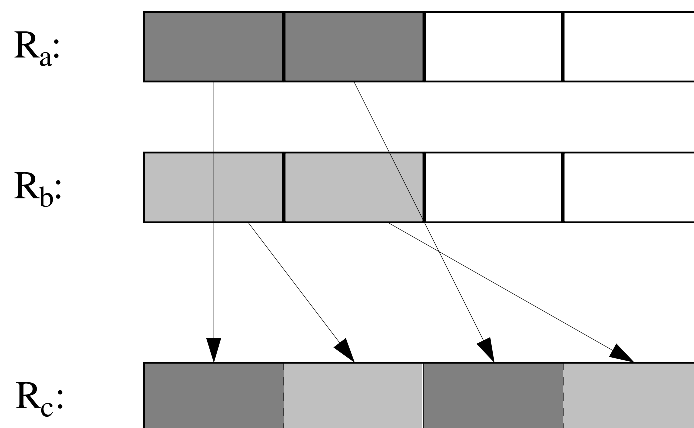

Figure 7.3 UNPACK.high R_c,R_a,R_b : Unpack high operation.

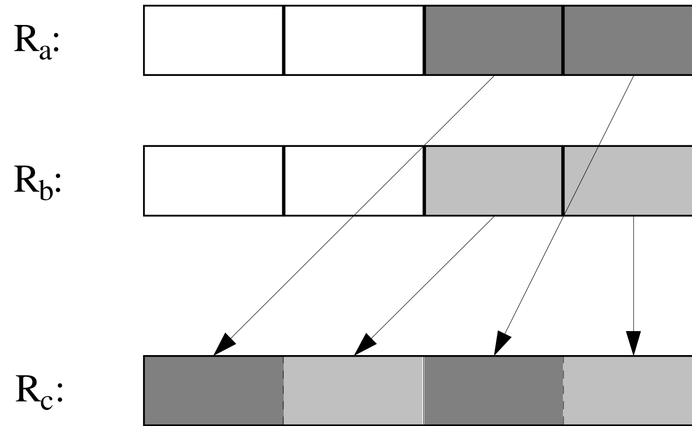

Figure 7.4 UNPACK.low R_c,R_a,R_b : Unpack low operation.

Permutation Instructions

Ideally, it is desirable to be able to perform all the possible permutations on a packed data-type. This is only possible when the subwords in the packed data-type are not very many. When the number of subwords increases beyond a certain value, the number of control bits required to specify arbitrary permutations becomes too many to be encoded in the opcodes. For the case of n subwords, the number of control bits used to specify a particular permutation of these n subwords is calculated as $n*log_2(n)$. Table 7.1 shows how many control bits are required to specify arbitrary permutations for different number of subwords.

TABLE 7.1. The table shows the number of control bits required to specify an arbitrary permutation for a given number of subwords.

Number of subwords in a packed data type	Number of control bits required to specify an arbitrary permutation for a given number of subwords
2	2
4	8
8	24
16	64
32	160
64	384
128	896

As Table 7.1 indicates, when the number of subwords is 16 or more, the number of required control bits exceeds the number of the bits available in the opcodes, which is typically 32. Therefore, it becomes

necessary to use a second register[21] to contain the control bits used to specify the permutation. By using this second register, it is possible to get any arbitrary permutation of up to 16 subwords in one instruction.

AltiVec architecture takes an additional step to use the three source registers it can have in one instruction. The VPERM instruction uses two registers to hold data, and the third register to hold the control bits. Thus, it allows any arbitrary permutation of 16 of the 32 bytes in the two source registers in a single instruction. The number of control bits required to specify this permutation (16 subwords out of 32) is calculated as $16*\log_2(32)=80$.

Due to the problem explained above, only a small subset of all the possible permutations is realizable in practice. There is a great flexibility in the selection of this subset from the set of all the possible permutations. It is sensible to select permutations that can be used as primitives to realize other permutations. One other distinction needs to be made between types of permutations. An instruction can use either one or two source operands for a permutation. In the latter case, only half of the subwords in the two source operands may actually appear in the target register. Examples to these two cases are the MUX and MIX instructions in IA-64 respectively, which correspond to PERMUTE and MIX instructions in MAX-2.

MIX is one useful operation that performs a permutation on two source registers. A MIX instruction picks alternating subwords from two source registers and places them into the target register. Since MIX uses two source registers, it appears in two variants. The first variant (Figure 7.6) is called the *mix left* and uses the odd indexed subwords of the source registers in the permutation, starting from the leftmost subword. The other variant, *mix right* (Figure 7.7), uses the even indexed subwords of the source registers, ending with the rightmost subword

---

[21] *This second register needs to be at least 64-bits wide to fully accommodate the 64 control bits needed for 16 subwords.*

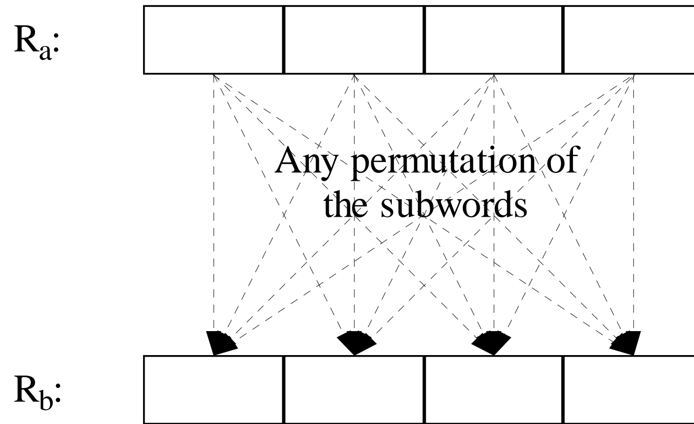

Figure 7.5 PERMUTE R_b,R_{a} : Arbitrary permutation on a register with four subwords.

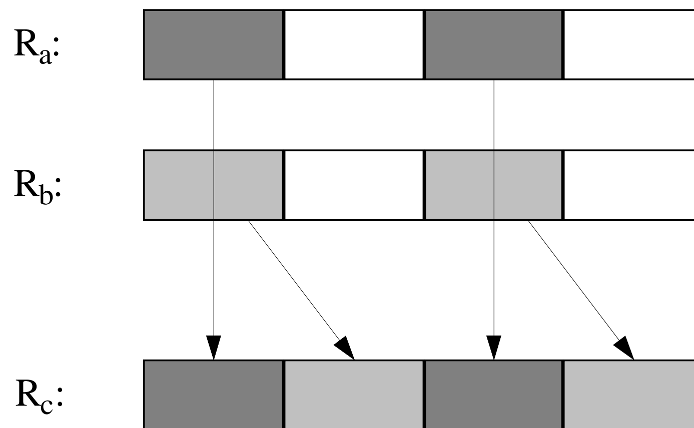

Figure 7.6 MIX_left R_c,R_a,R_b : Mix left operation.

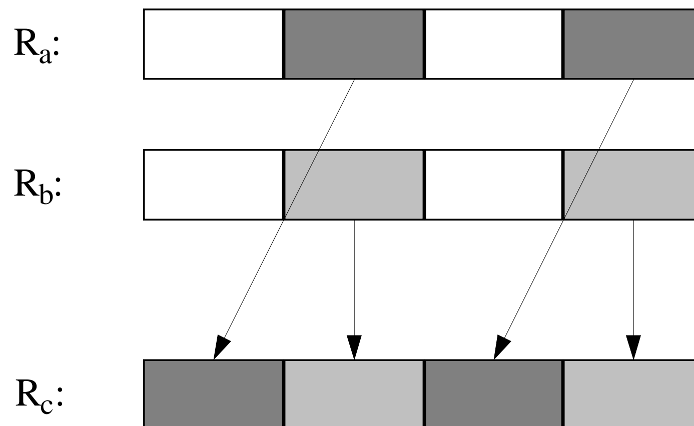

Figure 7.7 MIX_right R_c,R_a,R_b Mix right operation.

IA-64 architecture has the MUX instruction that can be used to perform permutations on 8 or 16-bit subwords. For 16-bit subwords, any arbitrary permutation is allowed. The immediate field is used to select one of the 256 possible permutations of the four 16-bit subwords, with or without repetitions of any subword. For the 8-bit subwords, only the following five permutations are allowed (Figure 7.8):

- MUX.rev: Reverses the order of bytes.
- MUX.mix: Mixes the upper and lower 32-bit fields of the 64-bit register with byte granularity.
- MUX.shuf: Performs a perfect shuffle on the bytes.
- MUX.alt: Selects every other byte placing the even[22] indexed bytes on the left half of the result register followed by the odd indexed bytes.
- MUX.brcst: Replicates the least significant byte into all the byte locations.

[22] *The bytes are indexed from 0 to 7. 0 corresponds to the most significant byte, which is on the left end of the registers.*

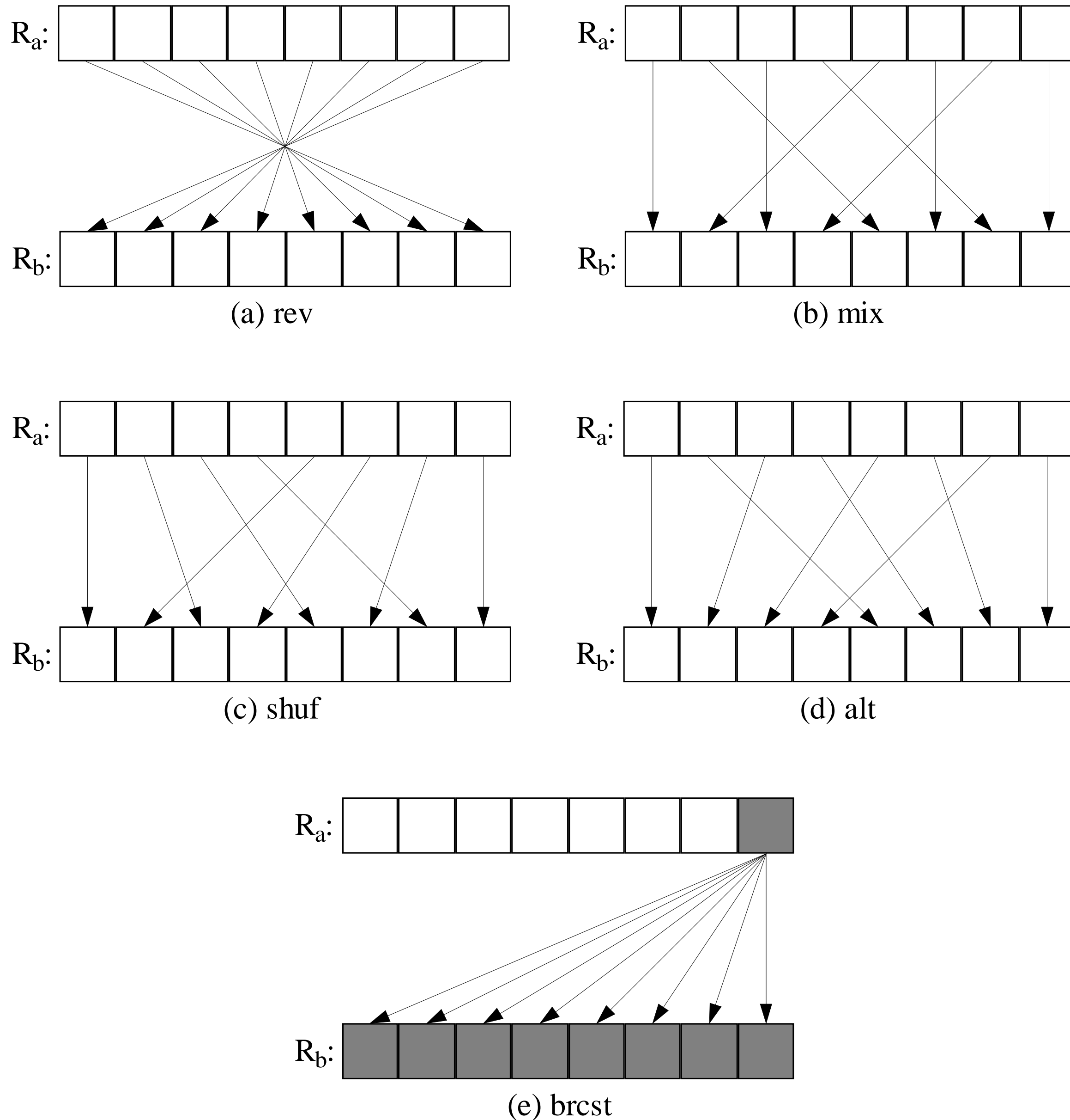

Figure 7.8 MUX.option R_b,R_a : MUX instruction of IA-64 has five permutation options for 8-bit subwords: rev, mix, shuf, alt and brcst. These options are shown above in (a) to (e) respectively.

Extract / Deposit Instructions

An *extract* instruction picks an arbitrary contiguous bit field from the source operand and places it right aligned into the target register. Extract instructions may be limited to work on subwords instead of arbitrarily long bit fields. In general, extract instructions clear the upper bits of the target register. Figures 7.9 and 7.10 show some possible extract instructions.

A *deposit* instruction picks an arbitrarily long right-aligned contiguous bit field from the source register and patches it into an arbitrary location in the target register. The remaining bits of the target register are either zeroed or unchanged. Deposit instructions may be limited to work on subwords instead of arbitrarily long bit fields and arbitrary patch locations. Figures 7.11 and 7.12 show some possible deposit instructions.

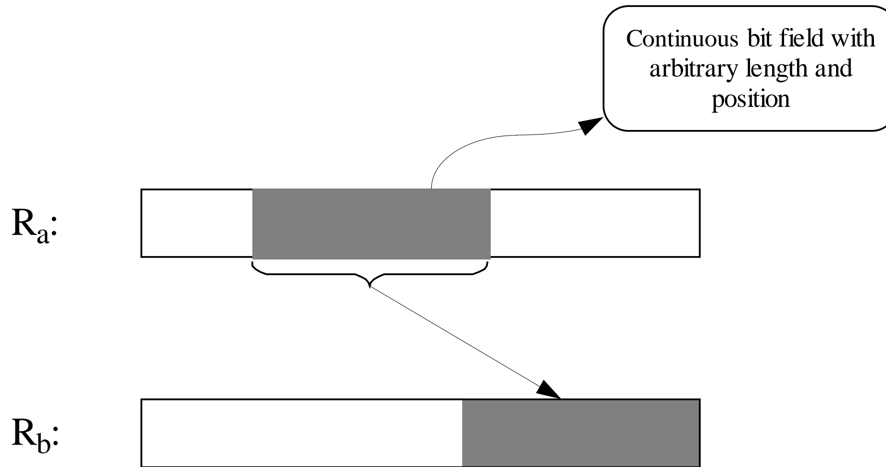

Figure 7.9 EXTRACT R_b,R_a : Extract instructions can be used to extract contiguous bit fields with arbitrary lengths and locations.

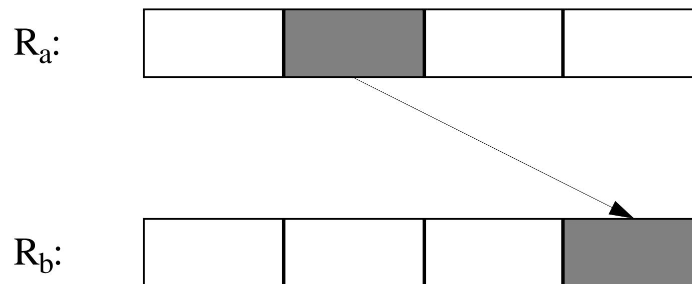

Figure 7.10 EXTRACT R_b,R_a : A more limited extract instruction that can only extract subwords.

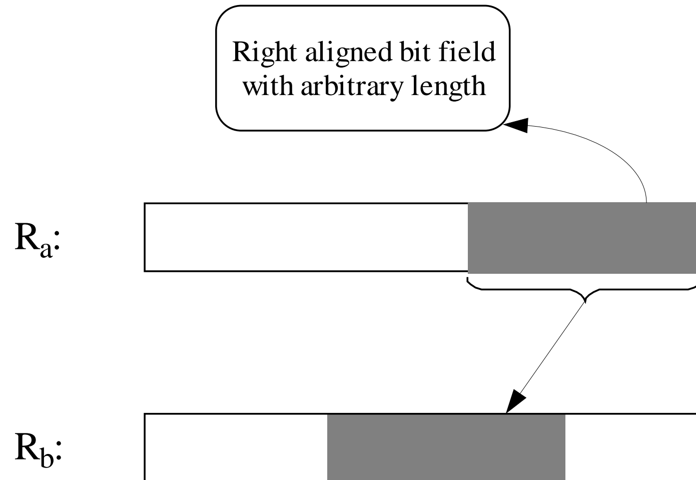

Figure 7.11 DEPOSIT R_b,R_a : Deposit instructions can be used to patch arbitrarily long, right-aligned contiguous bit fields from the source register into any location in the target register.

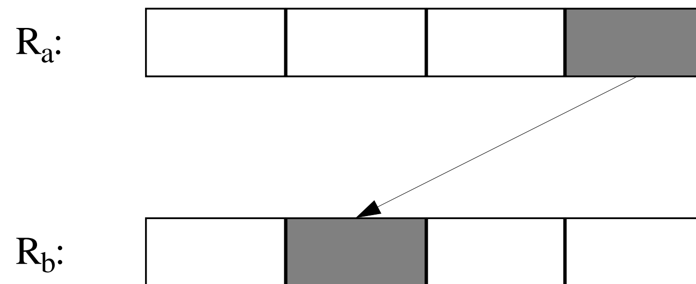

Figure 7.12 DEPOSIT R_b,R_a : A more limited deposit instruction that can only deposit subwords.

Moving a Mask of Most Significant Bits (Move Mask)

3DNow! includes the PMOVMSKB instruction for this operation. During the *Move Mask*, the most significant bit from each subword is picked, and placed in order into the target register, to a right aligned field (Figure 7.13). Remaining bits in the target register are cleared.

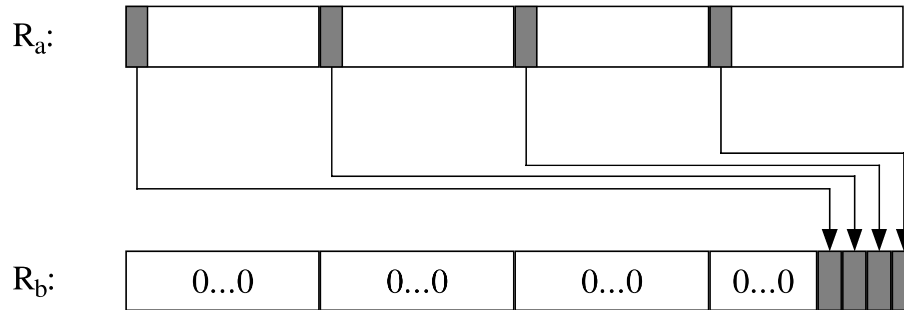

Figure 7.13 Move_Mask R_b,R_a : *Move Mask* operation on a register with four subwords. The most significant bit of each subword is written in order to the least significant byte of the target register.

Table 7.2 gives a summary of the data/subword packing and rearrangement operations discussed in this section.

TABLE 7.2 Data/subword packing and rearrangement operations.

Integer Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3Dnow!**	**Altivec**
Mix Left	√	√				√
Mix Right	√	√				√
MUX.rev	√					
MUX.mix	√					
MUX.shuf	√					
MUX.alt	√					
MUX.brcst	√					
Arbitrary Permutation of *n* subwords	√ (n=4)	√ (n=4)		√ (n=4)	√ (n=4)	√ (n=16,32)[23]
PACK	√		√			√
UNPACK(high/low)	√		√	√		√
Move Mask					√	
FP Operations	**IA-64**	**MAX-2**	**MMX**	**SSE-2**	**3Dnow!**	**Altivec**
Mix Left	√					
Mix Right	√					
PACK	√					
UNPACK(high/low)				√		
Arbitrary Permutation of *n* FP numbers				√ (n=2,4)		

In the table above, the first column contains the description of the operations. PACK, UNPACK, MIX, MUX.option and Arbitrary Permutation operations are explained in the text.

In the FP PACK instruction of the IA-64 architecture, the two source operands are FP numbers in the 82-bit register format (this is a non-standard format IA-64 uses for internal calculations). In the operation, the two numbers are first converted to standard 32-bit SP representation. These two SP FP numbers are then concatenated and the result is stored in the significand field (which is 64 bits) of the 82-bit target FP register. The exponent field of the target register (for its 82-bit register format) is set to the biased exponent for 2.0^{63}, and the sign bit is set to 0, indicating a positive number.

---

[23] *This is the VPERM instruction and it has some limitations for n=32. See text for more details on this instruction.*

FP UNPACK instructions of the SSE-2 architecture are identical to their integer counterparts, except that SP and DP FP numbers are used instead of integer subwords.

8. APPROXIMATION OPERATIONS

Some multimedia applications (e.g. graphics) may be very intensive in computations like $1/x$ or $\sqrt{x}$. Typically, such computations are handled by the FP-ALU and take more execution cycles to complete compared to an operation handled in the Integer-ALU.

Computation of $1/x$ in the FP-ALU involves calculation of the result to infinite precision. This intermediate result is later rounded to either a SP or DP FP number. This process may be too slow for some applications that have stringent time constraints. A real-time graphics application involving intensive $1/x$ computations could be one example.

On the other hand, a different application may not even require SP accuracy. For such applications, waiting for many execution cycles for SP or DP results to complete would degrade performance. To address these problems, multimedia extensions include what is called *approximation instructions*. Approximation instructions return less precise results (than a SP FP number), however execute faster than a full computation, which returns a SP or DP accuracy.

Even in instances that require full SP or DP accuracy, it is undesirable to have a reciprocal instruction that takes many more cycles than a typical FP multiplication or addition. The goal, then, is to break these long operations down into a sequence of simpler operations, each of which takes about the same time (e.g. 3-4 cycles) as a FP multiply, add or *multiply and accumulate* operation. The first operation in such a sequence is typically an approximation, which returns a low-precision estimate of the desired result. If low-precision results are acceptable, no further operations are necessary, and the result can be used at that point. If a higher precision is required, the next operation in the sequence is used. This second operation is typically a *refine* operation. It uses the low-precision estimate generated in the first step, and refines this value to a higher accuracy. Similarly, the next (third) operation in the sequence can be used to refine the output of the second operation, and this process can be repeated until the desired accuracy is reached. Therefore, use of approximation instructions (and further instructions to refine the results) can be used whenever:

a) Using a full computation that gives SP or DP results may be too slow for acceptable performance,
b) A less accurate result (than a SP FP number) may be acceptable, or,
c) A SP or DP result is desired with a sequence of shorter operations rather than a single long operation.

For use in such instances, IA-64, SSE-2, 3Dnow! and Altivec architectures have instructions that approximate the results of the $1/x$ and $1/\sqrt{x}$ operations. AltiVec also includes approximation instructions for $log_2(x)$ and 2^x operations. All these instructions operate on packed FP data types and take SP FP numbers as operands. The results of these instructions are less accurate than a SP FP number. 3DNow! also includes instructions to refine the results of its reciprocal and square-root approximation instructions to SP accuracy. IA-64 architecture does not include any separate instructions to refine the results of its approximation instructions. If standard accuracy is desired (either SP or DP), the approximation instruction is signaled through control bits to continue the computation until the IEEE-754 compliant result is reached.

Table 8.1 gives a summary of the approximation operations discussed in this section.

TABLE 8.1 Approximation operations.

FP Operations	IA-64	MAX-2	MMX	SSE-2	3DNow!	Altivec
$c_i = approx(1/a)$					√	
$c_i = approx(1/\sqrt{a})$					√	
$c_i = approx(1/a_i)$	√			√[24]		√
$c_i = approx(1/\sqrt{a_i})$	√			√[24]		√
$c_i = approx(\log_2 a_i)$						√
$c_i = approx(2^{a_i})$						√
$c_i = refine(est(1/a))$					√	
$c_i = refine(est(1/\sqrt{a}))$					√	

In the table above, the first column contains the description of the operations. The symbols a_i and b_i represent the subwords from the two source registers. The symbol c_i represents the corresponding subword in the target register. A shaded background indicates a packed FP operation.

- $approx(op(x))$ is a function that returns an approximation to the actual result of $op(x)$.
- $est(op(x))$ is an estimate to the actual result of $op(x)$, and is generated by the corresponding $approx(op(x))$ operation.
- $refine(est(op(x)))$ is a function that refines $est(op(x))$.

The example below uses the 3DNow! instructions and it illustrates how to approximate a reciprocal calculation and how to further refine the result.

Example:

Assume that register R_a contains the SP FP number *a*, and we want to calculate $1/a$ by first approximating the result, and then by refining this estimate to achieve the desired SP accuracy. The first step requires the use of the PFRCP instruction:

PFRCP R_b,R_a

This gives a low precision estimate of $1/a$. The result, which gets written into register R_b, is accurate to 14 bits in its significand field, compared to the 24 bit accuracy of a SP value. Increased accuracy requires the use of two additional instructions:

PFRCPIT1 R_c,R_b,R_a
PFRCPIT2 R_d,R_c,R_b

The first stage of refinement uses the PFRCPIT1 instruction with inputs to the instruction as R_a and R_b. PFRCPIT1 performs the first intermediate step of the Newton-Raphson algorithm to refine the approximation produced by PFRCP. The result of PFRCPIT1 instruction is written to R_c. The second and final step of the Newton-Raphson algorithm is performed by the PFRCPIT2 instruction. The inputs to the PFRCPIT2 are R_b and R_c. The result of the PFRCPIT2 instruction has the accuracy of a SP FP number, which is 24-bits in the significand field. Hence, at the end of 3 instructions, the value of $1/a$ is calculated to SP accuracy.

There is another instruction in the SSE-2 architecture that can be included in this section (Table 8.2). This is the *packed square-root* instruction, which operates on packed SP/DP FP operands and computes their square roots to SP/DP accuracy. However, this instruction is not an approximation instruction, and therefore it requires more execution cycles to complete compared to an approximation instruction.

[24] *Scalar versions of these operations also exist.*

TABLE 8.2 SSE-2 features a packed instruction for computing the square root of a FP register. The values in the source register can be SP or DP FP numbers.

FP Operation	IA-64	MAX-2	MMX	SSE-2	3DNow!	Altivec
$c_i = \sqrt{a_i}$				√[25], SP,DP		

[25] *Scalar versions of these operations also exist.*

9. SUMMARY

We have described the latest multimedia instructions that have been added to current microprocessor instruction set architectures (ISAs) for native signal processing, or, more generally, for multimedia processing. We described these instructions by broad classes: packed add/subtract operations, packed special arithmetic operations, packed multiply operations, packed compare operations, packed shift operations, subword rearrangement operations, and approximation operations. For each of these instruction classes, we compared the instructions in IA-64 [3], MMX [4], and SSE-2 [5] from Intel, MAX-2 [6,7] from Hewlett-Packard, 3DNow! [8,9] from AMD, and AltiVec [10] from Motorola.

The common theme in all these multimedia instructions is the implementation of subword parallelism. This is implemented for packed integers or fixed-point numbers in the integer datapaths, and for packed single-precision floating-point numbers in the floating-point datapaths. Visual multimedia data like images, video, graphics rendering and animation involve pixel processing, which can fully exploit subword parallelism on the integer datapath. Higher-fidelity audio processing and graphics geometry processing require single-precision floating-point computations, which exploit subword parallelism on the floating-point datapath. Typical DSP operations like *multiply-accumulate* have also been added to the multimedia repetoire of general-purpose microprocessors. These multimedia instructions have embedded DSP and visual processing capabilities into general-purpose microprocessors, providing native signal processing and media processing [1] capabailities. In fact, most DSPs and media processors have also adopted subword parallelism in their architectures, as well as subword permutation instructions and other features often first introduced in microprocessors for native signal and media processing.

We see two trends in these multimedia Instruction Set Architectures. The first, "less is more" trend, is represented by the minimalist architecture approach of MAX-2, which adheres to the RISC architectural principles of defining as few instructions as necessary for high-performance, with each instruction executable in a single pipeline cycle. The second, "more is better" trend, is represented by Altivec, where very complex sequences of operations are represented by a single multimedia instruction, with such an instruction taking many cycles for execution. An example is the matrix multiply instruction, VSUMMBM, in Altivec (see figure 4.10). More experience and evaluation of these instructions in multimedia processing applications can shed light on the effectiveness of these instructions. In the end, these two trends represent different stylistic preferences, akin to RISC and CISC instruction-set architectural preferences. In fact, sometimes, RISC-like multimedia instructions have been added to CISC processor ISAs, and CISC-like multimedia instructions to RISC processor ISAs. The remarkable fact is that subword parallel multimedia instructions have achieved such rapid and pervasive adoption in both RISC and CISC microprocessors, DSPs and media processors, attesting to their undisputed cost-effectiveness in accelerating multimedia processing in software.

10. REFERENCES

1. R.B. Lee and M. Smith, "Media Processing: A New Design Target", *IEEE Micro*, Vol. 16, No. 4, pp. 6-9, August 1996.
2. R.B. Lee, "Efficiency of microSIMD Architectures and Index-Mapped Data for Media Processors," Proceedings of Media Processors 1999, IS&T/SPIE Symposium on Electric Imaging: Science and Technology, pp. 34-46, January 1999.
3. Intel, "IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference," Revision 1.1, Order Code 245319-002, July 2000.
4. Intel, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference," Order Code 243191, 1999.
5. Intel, "IA-32 Intel Architecture Software Developer's Manual With Preliminary Wilamette Architecture Information, Volume 2: Instruction Set Reference," 2000.
6. G. Kane, "PA-RISC 2.0 Architecture," Prentice Hall, ISBN 0-13-182734-0, 1996.
7. R.B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, Vol. 16, No. 4, pp. 51-59, August 1996.
8. AMD, "3DNow! Technology Manual," Order Code 21928G/0, March 2000.
9. AMD, "AMD Extensions to the 3DNow! and MMX Instruction Sets Manual," Order Code 22466D/0, March 2000.
10. Motorola, "AltiVec Technology Programming Environments Manual," Revision 0.1, Order Code ALTIVECPEM/D. November 1998.
11. R.B. Lee, "Multimedia Extensions for General-Purpose Processors," *Proceedings of IEEE Signal Processing Systems '97*, pp. 9-23, November 1997.
12. R.B. Lee, "Accelerating Multimedia with enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, pp. 22-32, April 1995.
13. R. B. Lee, "Precision Architecture", *IEEE Computer*, Vol. 22 No. 1, pp. 78-91, January 1989.
14. V. Bhaskaran, K. Konstantinides, R.B. Lee and J.P. Beck, "Algorithmic and Architectural Enhancements for Real-Time MPEG-1 Decoding on a General Purpose RISC Workstation," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 5, No. 5, pp. 380-386, October 1995.
15. R.B. Lee, J.P. Beck, J. Lamb and K.E. Severson, "Real-Time Software MPEG Video Decoder on Multimedia-Enhanced PA7100LC Processors", *Hewlett-Packard Journal*, pp.60-68, April 1995.
16. M. Tremblay, J.M. O'Connor, V. Narayanan, H. Liang, "VIS Speeds New Media Processing," *IEEE Micro*, Vol. 16, No. 4, pp. 10-20, August 1996.