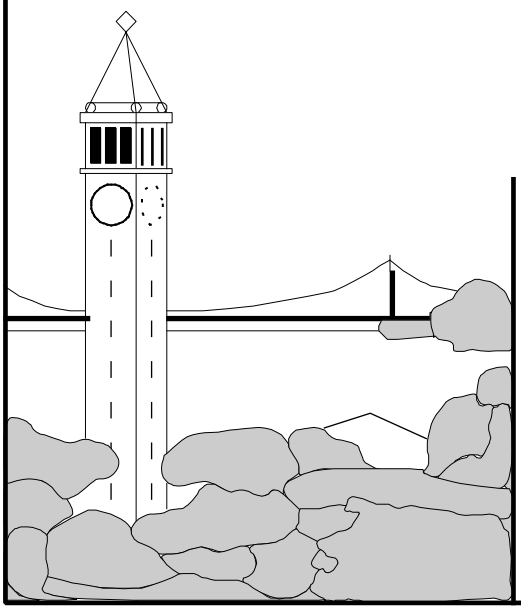
Multimedia Instruction Sets for General Purpose Microprocessors: A Survey

Nathan T. Slingerland and Alan Jay Smith



Report No. UCB/CSD-00-1124

December 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Multimedia Extensions for General Purpose Microprocessors: A Survey

Nathan Slingerland and Alan Jay Smith
{slingn, smith}@cs.berkeley.edu

Computer Science Division
EECS Department
University of California at Berkeley

December 21, 2000

Abstract

The last decade has seen the integration of audio, video, and 3D graphics into traditional workloads as well as the emergence of new workloads dominated by the processing of data representing such information. Initially, these emerging workloads were supported by dedicated, application specific integrated circuits and digital signal processors. In order to avoid the added cost and complexity of these dedicated hardware solutions, microprocessor vendors have extended their architectures with instructions targeting multimedia applications. Despite general agreement on the direction of evolving workloads, there is little agreement on the nature of the architectural changes that should be made to support them. The focus of this work is to survey existing multimedia instruction sets and examine how their functionality maps to a set of computationally important kernels extracted from the previously developed Berkeley multimedia workload.

1 Introduction

The data paths of general purpose microprocessors are 32 or 64-bits wide, while multimedia applications operate on data that is typically 8 (video, imaging) or 16 (audio) bits wide. This mismatch means that only a fraction of the data path and functional units are actually utilized. Multimedia extensions recognize that by partitioning functional units (for example by simply blocking carry bits, in the case of an adder), processing resources can be utilized more efficiently. *Partitioned arithmetic*, *subword parallelism*, and *single instruction multiple data (SIMD) processing* are all used as synonyms for this method of performing parallel operations on lower-precision data packed into word-oriented datapaths (although the term "SIMD" has other meanings as well).

The remainder of this paper is organized as follows: Section 2 gives a brief background on the historical and architectural motivations behind each vendor's multimedia instruction set, while Section 3 compares the instruction set design in detail (first comparing integer, then floating point and finally data type independent functionality). Section 4 surveys the available methods for programming with multimedia instruction sets. Our analysis is summarized in Section 5.

Funding for this research has been provided by the State of California under the MICRO program, and by Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation and Veritas Software Corporation.

2 Background

All major microprocessor vendors have defined multimedia instruction set extensions for their architectures (Table 1). There are large differences in the degree and type of support which the various multimedia extensions offer. In this section we present an overview of the available multimedia instruction sets in the order of their introduction, as well as providing background on design goals and considerations where possible. The date of initial product release is noted in parentheses by the heading for each extension, with an asterisk (*) denoting extensions which have not yet been implemented in released hardware.

2.1 Hewlett Packard MAX-1 (1994)

Hewlett Packard's MAX-1 for speeding partitioned integer operations was the first multimedia extension. It allowed for the implementation of a totally software MPEG-1 (352x240 or SIF resolution) decoder on the low end HP712 workstation (PA-7100LC CPU running at 80MHz). Originally, a scalar high level language implementation of MPEG decoding took 2 seconds (0.5 frames per second) on a 50 MHz HP720 workstation for video only (no audio). The algorithms and software were then optimized (without any multimedia instructions), realizing 4-5 fps. At that time, multimedia enhancements to the forth coming PA-7100LC processor were considered. Since it was already deep in the implementation phase, a requirement was that any chip enhancements that were added should not adversely impact its design schedule, complexity, cycle time or chip area [Lee95b]. Rather than add special purpose MPEG circuitry to the processor, the same design principles were used as when selecting instructions for the original PA-RISC architecture. This involved finding the most frequent

Vendor	Extension	Base ISA	Announced	Product	Instructions	Register File
HP	MAX-1	PA-RISC	unknown	1994 [Lee97b]	9	Integer (31x64b)
Sun	VIS	SPARC v9	unknown	1995 [Kohn95]	121	FP (32x64b)
HP	MAX-2	PA-RISC w/MAX-1	1995 [HP95]	1995 [Lee97b]	8	Integer (32x64b)
MIPS	MIPS-V	MIPS-IV	1996 [SGI96]	-	29	FP (32x64b)
MIPS	MDMX	MIPS-V	1996 [SGI96]	-	74	FP (32x64b), Acc. (1x192b)
Intel	MMX	x86	1996 [Intel96]	1997 [Intel97]	57	FP (8x64b)
DEC	MVI	Alpha	1996 [Bann96]	1997 [Carl97]	13	Integer (31x64b)
Cyrix	Extended MMX	x86 w/MMX	unknown	1997 [Cyrix]	12	FP (8x64b)
AMD	3DNow!	x86 w/MMX	1997 [AMD97]	1998 [AMDWP]	21	FP (8x64b)
Intel	SSE	x86 w/MMX	1998 [Veit98]	1999 [Intel99]	70	8x128b
Motorola	AltiVec	PowerPC	1998 [Phil98]	1999 [Moto99]	162	32x128b
MIPS	MIPS-3D	MIPS64	1999 [Thek99]	-	23	FP (32x64b)
AMD	Enhanced 3DNow!	x86 w/MMX, 3DNow!	1999 [AMD99]	1999 [AMD99]	24	FP (8x64b)
Intel	SSE2	x86 w/MMX, SSE	2000[Intel00a]	-	144*	8x128b

Table 1: **Microprocessor Multimedia Extensions** - the register file used by multimedia instructions may be shared with existing integer (Integer) or floating point (FP) registers, or be separate, with a geometry of N registers each with a width of W bits ($NxWb$). *Note that 68 of the 144 SSE2 instructions are 128-bit wide versions of 64-bit wide MMX/SSE integer instructions.

operations, breaking them down into simple primitives, and accelerating their execution [Lee95]. The changes that were implemented allowed for the decoding of MPEG audio and SIF video simultaneously at up to 30 fps.

2.2 Sun VIS (1995)

Sun's partitioned integer Visual Instruction Set (VIS) was first implemented in hardware on the UltraSPARC I processor, with all successive processors (UltraSPARC II, UltraSPARC IIi, UltraSPARC III) also supporting the instructions included in this extension. The primary motivation behind the creation of the VIS extension was to create a standard platform for multimedia applications such as 3D visualization and MPEG-1/MPEG-2 coding on future SPARC systems. Prior to the VIS extension, multimedia applications required specialized graphics hardware. By implementing VIS on the processor overall system cost is lowered and valuable expansion slots are freed. Unlike a cost conscious multimedia processor or ASIC, VIS is able to take advantage of agressive upgrades in frequency due to cutting edge technology and process improvements in the processor. VIS instructions were defined by examining a variety of graphics and multimedia algorithms. Any potential instruction had to execute in a single cycle or be easily pipelined, be applicable to several algorithms and not affect the cycle time of the processor [Trem96].

2.3 Hewlett Packard MAX-2 (1995)

Hewlett Packard further enhanced its PA-RISC processor multimedia capabilities with PA-RISC 2.0 (of which the partitioned integer MAX-2 extension is a subset), first implemented in the PA-8000 processor. In MAX-2 two approaches were used for multiplication depending on the media stream being processed. For audio and 3D graphics transformations, the full power and versatility of the floating point multiply-accumulate functional units are proffered. In lower precision applications that are typified by multiplications by constants, multiplications are accomplished through a series of compound packed shift and add instructions [Lee97b].

An integer multiplier takes three to four times the area of an integer adder, three times the latency and produces a result that is longer than each operand [Lee97c]. Using the floating-point FMAC (multiply-accumulate) units saved considerable die area. This was determined to outweigh the relatively minor disadvantage of using the same general-purpose integer registers normally used for addresses and loop counter variables for packed data. Architectures which utilize a separate register file or reuse floating point registers for SIMD operations are able to maintain a greater number of pointers and scalar integer variables in the integer register file without spilling to memory. In addition, pipeline complexity was reduced by not having a true integer multiply since it is a multicycle operation, while all other existing integer instructions are single cycle [Lee96b]. For applications such as MPEG decoding, video can be processed in the integer registers by MAX-2 instructions while audio is manipulated in parallel in floating point.

2.4 MIPS MDMX (1996*)

MIPS' MDMX is a partitioned integer instruction set which shares 32 64-bit registers with the floating point register file, but has an architecture centered around a special purpose 192-bit accumulator register. Many of MDMX's arithmetic operations can either explicitly specify a destination register or implicitly accumulate into the accumulator. The parti-

tioning of the accumulator is determined by the format of the elements being accumulated and is set according to the particular instruction executed. When accumulating packed unsigned bytes, the accumulator is partitioned into eight 24-bit unsigned slices. Packed 16-bit operations cause the accumulator to be split into four 48-bit sections. This way multiple accumulations can occur without overflow, although the burden of remembering the current accumulator format is left to the programmer [MIPS97]. MIPS Technologies Inc. no longer manufactures its own processor designs. Instead, it licenses its intellectual property (core designs) to other companies interested in manufacturing MIPS designs for their own products. No processor designs implementing MDMX have yet been announced.

2.5 MIPS MIPS-V/MIPS-64 (1996*)

The MIPS64 instruction set (previously known as MIPS-V) is a superset of all previous MIPS instruction sets (MIPS-I through MIPS-IV), adding a paired single precision floating point data type. Target applications include 3D geometry processing for VRML and other OpenGL applications. MIPS-V will be implemented for the first time on the upcoming MIPS64 20K (code-named "Ruby") processor design, representing 6-8% of the total floating point die area [Thek99]. Both NEC and Toshiba are expected to manufacture the MIPS64 20K processor, which also has low power consumption as a design goal (estimated to be 900 mW at 300 MHz), beginning in late 2000. [Edwa00].

2.6 Intel MMX (1997)

Intel's MMX partitioned integer multimedia extension was first implemented in the P55 Pentium processor (AMD licensed MMX from Intel for their K6 processor around the same time). The design of MMX had as its first priority to substantially improve the performance of multimedia, communications and emerging Internet applications. The applications studied included MPEG-1 and MPEG-2 video, music synthesis, speech compression, speech recognition, image processing, 3D graphics in games, video conferencing, modems and audio. These were collected from a variety of sources and analyzed with profiling tools to determine the characteristics of their critical code sequences. An important design criterion was that processors implementing MMX be able to run existing operating systems without modification. This meant that no new architectural state could be introduced (no new registers or exceptions). MMX technology was mapped onto the existing floating point architecture and registers [Mitt97]. Sharing the x87 floating point register space meant that both multimedia and floating point instructions could not be used concurrently. In addition, in order to return the x87 FP stack to a sane state after MMX operations, an `EMMS` (empty multimedia state) instruction must be used. The `EMMS` instruction had very high latency in early implementations, but current processors have improved upon this significantly.

2.7 DEC MVI (1997)

The DEC Alpha 21164PC processor was the first DEC processor to incorporate the partitioned integer Motion Video Instructions (MVI) multimedia extension. (Note that DEC is now a part of Compaq corporation; we use the now obsolete name "DEC" for historical consistency.) All DEC Alpha processors released after the 21164PC (21264/21264A/21364) include the MVI instructions as well. The target applications during design were H.261 and H.263 teleconferencing at 30 fps, as well as DVD video playback at 30 fps with stereo audio [Carl97]. The goal of Alpha MVI was to enable software video encoders that would produce quality comparable to hardware encoders. DEC architects did not choose to implement a broader repertoire of multimedia instructions because they reasoned that available memory bandwidth limitations minimize the overall benefit of multimedia instructions. They point out that extensions such as MMX essentially try to fix some of the x86 legacy architecture deficiencies which are not present on Alpha, such as a severly limited number of registers and poor floating point organization [Rubi96].

2.8 Cyrix Extended MMX (1997)

Like AMD, Cyrix also licensed MMX from Intel for use in Cyrix's 6x86MX and later processors. They also extended MMX with a set of instructions which attempt to help alleviate the register pressure issues caused by the x86 destructive operand format (one operand register must also act as the destination register). This is accomplished by introducing the concept of implied registers. For a given first source register, an implied destination register is defined, effectively creating a three operand format. Besides augmenting basic MMX register utilization, packed average, sum of absolute differences, conditional moves and a 16-bit rounded multiplication operation were also added [Cyrix].

2.9 AMD 3DNow! (1998)

For the K6-2 processor, AMD took an independent role by defining its own 3DNow! extension, which utilizes the same registers and basic instruction formats as MMX, but adds a partitioned floating point data type. This allows for two parallel single precision floating point operations to be computed in parallel. During their initial analysis of floating point multimedia codes, the architects of 3DNow! found two compelling possibilities in the design space. The first, being a floating point extension to MMX, was the path actually taken. The other was a full set of independent instructions as in Motorola's AltiVec (e.g. a large number of wide registers, four operand instruction format). They found that anything in between these two solutions would require significantly greater hardware area or complexity without a corresponding performance benefit [Ober99].

2.10 Intel SSE (1999)

Intel's follow up to MMX, the Internet Streaming SIMD Extension (SSE), is primarily a partitioned floating point extension which addresses 3D geometry calculations, software rendering, video encoding/decoding, and speech recognition. It also incorporates feedback on MMX from software vendors in the form of new packed integer instructions. Unlike MMX, the floating point side of SSE *does* add new architectural state to the Intel architecture. This reduced implementation complexity and eased programming model issues, as well as allowing for SSE and MMX or x87 (normal floating point capabilities) to be used concurrently. The operating system support issue was avoided by having operating system vendors update their software well in advance of the release of SSE.

The Pentium III processor is the first from Intel to offer an implementation of SSE. Implementing a data path greater than 128-bits was not viewed as a reasonable option when balancing cost against potential benefits when first implementing the extensions during the design of the Pentium III chip. Pentium II floating point busses and registers were already 80-bits wide due to the way x87 floating point is implemented. 128-bits was seen as a marginal increase, where as 256-bits, for example, would have a much larger impact. A 256-bit wide implementation would require doubling the width of existing floating point execution units as well as proportional increase in memory bandwidth in order to feed them. Finally, since the primary focus of the SSE extension was 3D geometry processing, greater than four wide single precision operations were felt to offer diminishing returns as most 3D geometry operations work with 4x4 matrices. In addition, geometry primitives such as the triangular strips that sophisticated 3D objects are composed of tend to be fairly short in current applications (on the order of 20 vertices per strip), limiting the potential benefit of longer vectors.

A 64-bit wide packed floating point implementation was also considered (as in AMD's competing and earlier to market 3DNow!). The AMD approach was not used because the x86 architecture has 3-bit register specifiers (i.e. a maximum of eight registers), thus implementing eight 128-bit wide registers ($8 \times 128\,bits = 1024\,bits$) effectively doubles the register space in comparison to a 64-bit wide register file ($8 \times 64\,bits = 512\,bits$). Despite the 128-bit wide register file width defined by the SSE instruction set architecture, the Pentium-III SSE execution units are actually 64-bits (two single precision floating point elements) wide in hardware. The instruction decoder translates 4-wide (128-bit) SSE instructions into pairs of 2-wide (64-bit) internal micro-ops. Implementing the 128-bit SSE instruction set on a 64-bit datapath in this way limited the necessary changes to the instruction decoder, and allowed for the utilization of existing and new execution units [Thak99].

2.11 Motorola AltiVec (1999)

Motorola's AltiVec is an integral part of the Motorola 7400 (G4) processor, which extended the PowerPC architecture through the addition of a 128-bit wide vector (both partitioned integer and floating point) execution unit. Unlike many other extensions, which have supported media processing by leveraging existing functionality from the integer or floating point data paths, AltiVec devotes a significant portion of the chip area to the new features and emphasizes the growing role of multimedia [Gwen98a]. AltiVec is a 128-bit wide extension with its own dedicated register file and four pipelined execution sub-units. The target applications for AltiVec included IP telephony gateways, multi-channel modems, speech processing systems, echo cancelers, image and video processing systems, scientific array processing systems, as well as network infrastructure such as Internet routers and virtual private network servers. In addition to accelerating next-generation applications, AltiVec can also accelerate many time-consuming traditional computing and embedded processing operations such as memory copies, string compares and page clears. Compared to other vendors, Motorola has targeted a much more general set of applications than just multimedia [Full98].

2.12 MIPS MIPS-3D (1999*)

MIPS-3D consists of application specific extensions (ASEs) which are intended to be separate, optional add-ons to the base MIPS-V instruction set [MIPS99]. MIPS-3D consists of partitioned floating point instructions targeting 3D geometry processing, and is planned for inclusion on the MIPS64 20K processor. It accounts for 3% of the floating point die area; the floating point silicon represents less than 15% of the total processor die area.

2.13 AMD Enhanced 3DNow! (1999)

AMD introduced Enhanced 3DNow! which extended 3DNow! and MMX, in their Athlon (previously known as the K7) processor by adding partitioned floating point and integer operations to make 3DNow! functionally equivalent to Intel's SSE extension. The added integer instructions are opcode equivalent to those included in Intel's SSE.

2.14 Intel SSE2 (2000*)

Intel's SSE2 extension is included in the Pentium IV microprocessor. It extends the existing set of partitioned integer MMX operations to 128-bits wide with the addition of 68 new instructions which utilize the same set of eight 128-bit wide registers (`XMM0-XMM7`) introduced with the original SSE extension. A packed double precision data type is also added, targeted at applications other than multimedia such as scientific and engineering workloads, as well as advanced 3D geometry used in raytracing [Intel00a], [Intel00b]. No new architectural state is added with Intel's SSE2.

3 Comparison

In this section we review the functional architecture of each instruction set by describing the principles of operation of

each. In discussing the functionality present in multimedia instruction sets, we will draw on the computational kernels extracted from the Berkeley Multimedia workload in order to demonstrate how well (or how poorly) a given functionality maps to a real multimedia workload based on our study of it in [Sling00d]. The Berkeley multimedia workload consists of a wide range of open source audio, speech, imaging, document, video and 3D graphics applications along with representative and realistic data sets. [Sling00a] discusses the design and characterization of this workload in detail.

3.1 Integer

All of the multimedia extensions support integer operations, although the types and widths of available operations varies greatly. In our discussion, N-bit unsigned integer values are denoted$U(a.b)$, where a indicates the number of significant bits to the left of the binary point, and b the number of bits to the right. Signed two's complement format is denoted $S(a.b)$, where $a = N - b - 1$ and b is the number bits to the right of the binary point. A full discussion of this notation can be found in Appendix B.

3.1.1 Saturation and Overflow

Traditional integer operations deal with overflow in a modulo fashion as exceeding the maximum or minimum representable range leads to a result in which only the lower N-bits of the intermediate result are retained and placed in an N-bit wide result register. This behavior is undesirable in many multimedia applications, where a better approach is to saturate positive overflow to the largest representable value or negative overflow to the largest negative value. This is because we are operating on media (often visual) data types, so we would like to be able to add two values and have the result not be obviously erroneous to our senses; for audio, for example, clipping is clearly preferably to wrap-around. The difference between the two types of overflow can be seen in the example of adding images with modulo and saturating addition shown in Figure 1, demonstrating the much more aesthetically palatable result achieved with saturating arithmetic.

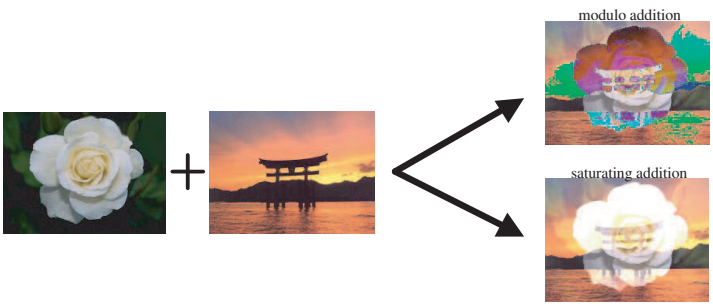


Figure 1: **Modulo and Saturating Operations** - *modulo arithmetic* "wraps around" to the next representable value when overflow occurs, while *saturating arithmetic* clamps the output value to the highest or lowest representable value for positive and negative overflow respectively.

A second reason for employing saturating arithmetic is to have a mechanism to efficiently deal with multiple possible overflow situations in packed values. The main obstacle to exploiting data parallelism is data dependent instruction flow. There are five possible methods for dealing with overflow [Lee95]:

1. ignore it (modulo or wrap-around arithmetic)
2. trap
3. set a shared overflow flag
4. set an individual overflow flag for each element
5. clip the result to the desired range (saturation or clamping arithmetic)

If the overflow in each packed value had to be handled separately, the performance gain of parallel arithmetic would be negated [Lee97b]. This is especially true for the shared flag and trap solutions since the operation would have to be repeated and tested serially to determine which element overflowed. The only added cost of saturation is that unlike the modulo addition of 2's complement numbers, saturation requires separate operations for signed and unsigned values.

A good example of where saturating arithmetic is useful is in the `add_block()`kernel taken from MPEG video decoding. Adding two blocks is an integral part of the block reconstruction step of motion compensation (Algorithm 1).

Algorithm 1 `Add_Block()`**Inner Loop** - `short *bp` and `unsigned char *rfp` are pointers to the two 8x8 input arrays

```
for (i=0; i<8; i++) {
   for (j=0; j<8; j++) {
      *rfp = Clip[*bp++ + *rfp]; rfp++;
   }
   rfp+= iincr;
}
```

As can be seen, the high level language version of the code uses a lookup table to clamp the result of the addition. A SIMD version of the code with saturating addition does not require this additional overhead. An added advantage is that in the case of a 64-bit SIMD implementation, eight values can be loaded simultaneously and operated on simultaneously, further reducing memory overhead.

3.1.2 Addition and Subtraction

Table 2 lists the data types supported for addition and subtraction operations on the architectures we have examined. Motorola's AltiVec includes a saturating 32-bit operation, but it is not clear how necessary this is, since the dynamic range of 32-bit numbers is great enough to avoid overflow for the 8-bit and 16-bit data types used in multimedia. Motorola's AltiVec also has a flag which is set whenever saturation has occurred. This can be useful in code which utilizes block floating point so that the appropriate scaling can be done when needed. (*Block floating point* is a scaling technique in which a single exponent is used for all the integer data in a block of

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Modulo Add/Sub	8,16,32	8,16,32	-	16	8,16,32,64	8,16	8,16,32	16,32
Saturating Add/Sub	U8,U16,S8,S16	U8,U16,S8,S16	-	U16,S16	U8,U16,S8,S16	S16	U8,U16,U32,S8,S16,S32	-
Average	U8,U16	U8	-	S16	U8,U16	-	U8,U16,U32,S8,S16,S32	-
Min/Max	U8, S16	S16	U8,S16	-	U8,S16	U8,S16	U8,U16,U32,S8,S16,S32	-

Table 2: **Basic SIMD Integer Arithmetic Operations** - *Un* indicates n-bit unsigned integer packed elements, *Sn* symbolizes n-bit signed integer packed elements, while values without a prefix, *n*, indicate operations which work for either signed or unsigned values. Average operations arithmetically average the corresponding elements in two partitioned input registers, while min/max output the minimum or maximum values of the corresponding elements in two partitioned input registers, repectively.

data. It has the advantage of being less expensive in terms of hardware than floating point, as well as faster, but is only appropriate where data values are clustered together.) None of the other architectures implement this feature. It is odd that Sun's VIS, which is likely the most imaging focused of all of the extensions does not have saturating operations, except during packing; this operation will be discussed later. Saturating arithmetic is of lesser importance with the MIPS extensions due to the 192-bit wide accumulator used for storing the result of integer operations.

Only DEC's MVI extension does not include any partitioned addition or subtraction operations, instead requiring either that intermediate calculations be done in wide enough precision to guarantee that overflow will not occur, or that max/min operations be used to clamp values appropriately to simulate saturation. An example of synthesizing a partitioned saturating unsigned add with this instruction set is shown in Algorithm 2. A 1's complement operation (`eqv` instruction) is used because the 1's complement of a number is the largest number that can be added to the original number without causing positive overflow to occur. By using the `min` instruction to get the smaller of either the second operand or 1's complement of the first addition operand it is possible to produce a clamped result.

Algorithm 2 MVI Synthesized Saturating Unsigned Add

```
eqv     r6, zero, t0      ;; 1's complement of r6
minuw4  r5, t0, r5        ;; get the smaller values
addq    r5, r6, v0        ;; add r6 to r5 and place in v0
```

3.1.3 Sum of Absolute Differences

One of the few operations that DEC did include in its MVI extension is an instruction to calculate the sum of absolute differences between unsigned packed bytes. Intel's MMX, although a much richer set of instructions, did not include this operation (it *was* included in the AMD, Cyrix and Intel extensions to MMX which came later). Sun's VIS also includes a sum of absolute differences instruction. DEC architects found that this operation provides the most computational benefit of all multimedia extension operations [Rubi96]. We found this to be true as well, since a SAD operation works with 8x parallelism on a 64-bit architecture, and replaces a significant amount of scalar code (Algorithm 3). Most kernels are not able to perform computations with an 8-bit wide data type. It was found that on the Alpha architecture without multimedia extensions, motion estimation (dominated by computing sums of absolute differences) consumes 70% of the time spent encoding MPEG video. This code is found in the `block_match()`kernel, and is centered around the loop listed in Algorithm 3 from MPEG-2 video encoding.

Algorithm 3 `Block_Match()`**Kernel Inner Loop** - `short *blk1` and `short *blk2` are pointers to the 16x16 macro block arrays to be compared

```
for (row_index=0; row_index<height; row_index++) {
   if ((v = blk1[0] - blk2[0])<0) v = -v; sum+= v;
   if ((v = blk1[1] - blk2[1])<0) v = -v; sum+= v;
   if ((v = blk1[2] - blk2[2])<0) v = -v; sum+= v;
   if ((v = blk1[3] - blk2[3])<0) v = -v; sum+= v;
   if ((v = blk1[4] - blk2[4])<0) v = -v; sum+= v;
   if ((v = blk1[5] - blk2[5])<0) v = -v; sum+= v;
   if ((v = blk1[6] - blk2[6])<0) v = -v; sum+= v;
   if ((v = blk1[7] - blk2[7])<0) v = -v; sum+= v;
   if ((v = blk1[8] - blk2[8])<0) v = -v; sum+= v;
   if ((v = blk1[9] - blk2[9])<0) v = -v; sum+= v;
   if ((v = blk1[10] - blk2[10])<0) v = -v; sum+= v;
   if ((v = blk1[11] - blk2[11])<0) v = -v; sum+= v;
   if ((v = blk1[12] - blk2[12])<0) v = -v; sum+= v;
   if ((v = blk1[13] - blk2[13])<0) v = -v; sum+= v;
   if ((v = blk1[14] - blk2[14])<0) v = -v; sum+= v;
   if ((v = blk1[15] - blk2[15])<0) v = -v; sum+= v;
   if (sum >= distlim) break;
   blk1+= row_offset; blk2+= row_offset;
}
```

On 64-bit wide data paths, the central portion of this code can be replaced by two sum of absolute difference (SAD) instructions operating on packed bytes. Although this code is the primary computational aspect of this kernel, MPEG-2 encoding offers three other varieties of block matching involving half-pixel interpolation. *Interpolation* is done by averaging a set of pixel values with pixels offset by one horizontally, vertically or both. The original MPEG-2 code first performs the interpolation, and then computes the sum of absolute differences on the result. An instruction set such as MVI has no way to perform packed averaging, but it can trade some precision for speed. A similar interpolation operation can be done more efficiently (but not identically) on SIMD architectures with a SAD instruction by averaging the result of several SAD operations.

3.1.4 Multiplication

SIMD or partitioned multiplication, involves the multiplication of corresponding packed elements. Partitioned mul-

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Truncating Multiply	S16,U16	S16	-	-	S16,U16	-	S16	S16, S32
Rounded Truncating Multiply	S16	S16	-	-	-	-	S16	-
Full Multiply	S16	S16	-	-	S16,U32	U8,S16	U8, U16, S8,S16	-
Shift Right Logical	16,32,64	16,32,64	-	16	16, 32, 64	8,16	8,16,32	-
Shift Right Arithmetic	16,32	16,32	-	16	16, 32	16	8,16,32	-
Shift Left	16,32,64	16,32,64	-	16	16, 32, 64	8,16	8,16,32	-

Table 3: **Supported SIMD Multiplication and Shift Data Types** - the prefix 'S' indicates a signed operation, 'U' an unsigned operation, and no prefix indicates the same operation works for both signed and unsigned values

tiplication instructions demonstrate the greatest differences among the arithmetic operations available in the multimedia extensions. Compared to implementing packed addition or subtraction, multiplication is significantly more costly in terms of die area (3-4 times that of an integer adder) and latency (3 times that of an integer adder). Semantically, multiplication is difficult to deal with on SIMD architectures because the result of a multiply is longer than either operand [Lee97c]. Latency is of primary concern for those architectures implementing multimedia extensions on the integer data path. DEC's MVI and Hewlett Packard's MAX-1 and MAX-2 are the only extensions in this category, and it is unsurprising that neither have implemented partitioned multiplication.

The length of the result of a multiplication operation is greater than the length of its operands. On a SIMD architecture, a register typically contains the greatest number of packed values that will fit for a given precision. Because of this it necessary to deal with this expansion property of multiplication in some way that maps well to a SIMD architecture. The instruction sets we have examined (Table 3) deal with this in several ways:

1. reduction
2. even/odd
3. truncation
4. higher precision result register

3.1.4.1 Reduction *Reduction* sums result vector subelements to produce fewer result values, and is useful because it is essentially a multiply-add; the core operation in digital signal processing. All that is needed is to ensure efficient processing is that the operands are arranged correctly to make the accumulation work. On Intel's MMX the `pmaddwd` instruction performs the operation depicted in Figure 2.

3.1.4.2 Even/Odd AltiVec takes a different approach to dealing with the multiplication result width problem. It supports selectively multiplying either the *even or odd elements* of the source registers such that the full width results fit in the destination register. This does put a small extra burden on the programmer to take the unconventional result ordering into account (only even or odd elements in a given result register). On AltiVec, this is easily undone with a mix or data shuffle operation as soon as both even and odd results have been computed.

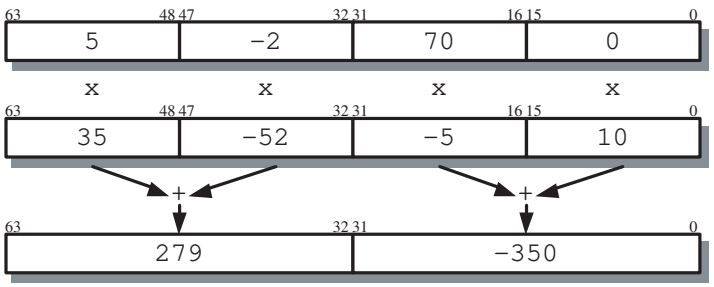
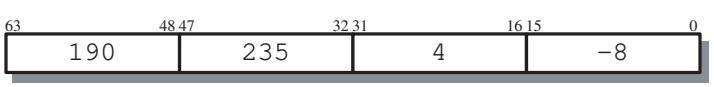


Figure 2: **Partitioned Multiplication with Reduction**

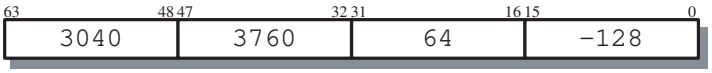
3.1.4.3 Truncation *Truncation* predefines a set number of result bits to be thrown away. This primarily has application when multiplying n-bit fixed point values with fractional components which together take up a total of n-bits of precision. Pre-scaling of one or both of the operands may be needed to meet this criteria. For example, if we want to multiply each element of a vector of $S(15.0)$ fixed-point (integer) values by the coefficient $\sqrt{2} = 1.414$, we can use an instruction such as MMX's `pmulhw`. Recall that our notation $S(a.b)$ refers to a signed 2's complement integer requiring $a + b + 1$ bits of precision. In our example, we will use a $S(3.12)$ representation for the coefficient:

$$\sqrt{2} \simeq \frac{5793}{2^{12}}$$

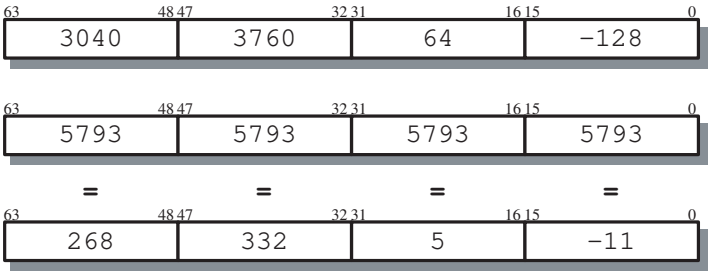
We will assume that the vector of 16-bit integers we wish to multiply by $\sqrt{2} = 1.414$ by is:



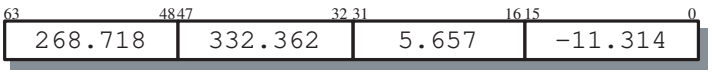
The `pmulhw` instruction truncates the result between the 15th and 16th bit positions of the internal 32-bit result, discarding the lower 16-bits. For this reason, we need to pad the operands with four extra bits so that the sum of the fractional bits in both multiplicands will equal 16 bits (thus the result will be scaled correctly and minimal precision will be lost). The $S(3.12)$ fixed point coefficient 5793 would exceed the 16-bit signed range of $-2^{15} \ldots + 2^{15} - 1$ if it was pre-shifted left by four bits, so instead the second multiplicand must be pre-shifted by left four bits. This can be accomplished through MMX's `psllw` instruction:



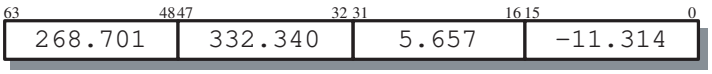
After shifting, the operation with `pmulhw` would be:



where the real numerical result when multiplying by the approximation $5793/2^{12}$would be:



and the actual arithmetic result of multiplying by $\sqrt{2}$ carried out to infinite precision would be:



Note that there was some initial loss of precision in quantizing $\sqrt{2} = 1.4142135$ since $5793/2^{12} = 1.4143066$. The other source of lost precision is due to the fact that the `pmulhw` instruction truncates the result rather than rounding it. In order to reduce this second type of precision loss, many architectures include 16-bit truncating multiplications that internally round the intermediate 32-bit result before the truncation to 16-bits takes place.

3.1.4.4 Higher Precision Result Register MIPS' MDMX extension avoids the multiplication result width problem by including a 192-bit accumulator register to act as the destination register for many operations including multiplication. This way there is the extra space required for the result to be stored in full precision. In addition, full precision results can be accumulated, and the total rounded only once at the end of the loop. While this may seem an elegant solution in principle, it ignores the architectural side of actually making such a design fast. The accumulator is a *shared* resource, and as such has a tendency to limit instruction level parallelism. We found that the existence of only a single accumulator was a severe handicap to avoiding data dependencies. On a non-accumulator but otherwise superscalar architecture it is usually possible to perform some other useful, non data dependent operation in parallel so that the processing can proceed at the greatest degree of instruction level parallelism possible. On MDMX all computations which need to use the accumulator must proceed serially.

In the MPEG decode `pyrogen_add_block()`kernel function listed as Algorithm 4, the bottleneck is quite evident. There is no data dependency which would prevent us from starting to process four new elements every cycle in a pipelined implementation. Unfortunately, there is a resource conflict. Packing and clamping operations in MDMX are performed from the accumulator with the `rx.fmt` instruction.).

Algorithm 4 MDMX `add_block()`

```
loop:
     ldc1      f3, (a1)    # f3:  |r0|r1|r2|r3|r4|r5|r6|r7|
     ldc1      f1, (a0)    # f1:  | bp0 | bp1 | bp2 | bp3 |
     ldc1      f2, 8(a0)   # f2:  | bp4 | bp5 | bp6 | bp7 |
# unpack rfp data 8 -> 16
     shfl.upuh f4, f3      # f4:  | r4 | r5 | r6 | r7 |
     shfl.upul f3, f3      # f3:  | r0 | r1 | r2 | r3 |
# bp + rfp [0..3]
     addl.qh  f1, f3       # acc:  bp + rfp [0..3]
# clamp to 8-bit unsigned
     rzu.ob   f1, f0       # f1:  | 0|r0| 0|r1| 0|r2| 0|r3|
# bp + rfp [4..7]
     addl.qh  f2, f4       # acc:  bp + rfp [4..7]
# clamp to 8-bit unsigned
     rzu.ob   f2, f0       # f2:  | 0|r4| 0|r5| 0|r6| 0|r7|
# pack output
     shfl.pacl f1, f1, f2  # f1:  |r0|r1|r2|r3|r4|r5|r6|r7|
     sdc1     (a1), f1     # store new rfp [0..7] values
     add      a1, a1, a2   # rfp += iincr
     addi     a0, a0, 16   # bp += 8 (pointer to INT16)
     addi     a1, a1, 8    # rfp += 8 (pointer to UINT8)
     subi     a3, a3, 1    # decrement loop counter
     bgtz     a3, loop
```

In this example, it is possible to circumvent the accumulator problem by synthesizing the clamping operation with the `min.fmt` and `max.fmt` register-only instructions, thus avoiding the accumulator bottleneck. This work around costs two extra instructions and two extra registers to hold the maximum and minimum values we wish to clamp to. (See Algorithm 5). Placing exclusive functionality in the accumulator creates unnecessary bottlenecks and thereby inhibits instruction level parallelism. In general, any architectural feature which is not separately and independently available wherever it might be needed has the potential to be a bottleneck.

Algorithm 5 MDMX Alternative `add_block()`

```
     ldc1      f10, max_val # f3:  | 0 | 0 | 0 | 0 |
     ldc1      f11, min_val # f1:  |255|255|255|255|
     ldc1      f2, 8(a0)    # f2:  | bp4 | bp5 | bp6 | bp7 |
# bp + rfp [0..3]
     add.qh   f1, f1, f3    # f1:  bp + rfp [0..3]
# bp + rfp [4..7]
     add.qh   f2, f2, f4    # f2:  bp + rfp [4..7]
# clamp to 8-bit unsigned
     max.qh   f1, f1, f10
     max.qh   f2, f2, f10
     min.qh   f1, f1, f11
     min.qh   f2, f2, f11
# pack output
     shfl.pacl f1, f1, f2  # f1:  |r0|r1|r2|r3|r4|r5|r6|r7|
```

3.1.4.5 Multiplication Primitives A potential problem with the VIS architecture is that it does not actually include full 16-bit multiply instructions. Instead it includes primitives, the results of which must be combined (see Algorithms 6 and 7).

Algorithm 6 Sun VIS 16-bit x 16-bit →16-bit Multiply

```
fmul8sux16   %f0, %f2, %f4
fmul8ulx16   %f0, %f2, %f6
fpadd16      %f4, %f6, %f8
```

Algorithm 7 Sun VIS 16-bit x 16-bit →32-bit Multiply

```
fmuld8sux16  %f0, %f2, %f4
fmuld8ulx16  %f0, %f2, %f6
fpadd32      %f4, %f6, %f8
```

The disadvantage of dividing an operation into several instructions is that it increases register pressure, decreases instruction decoding bandwidth and creates additional data dependencies. Sun's VIS was the only multimedia instruction set with multiplication operations that do not directly support 16-bit operands. The reason for dividing up 16-bit multiplication in this way was to decrease die area. Not providing a full 16x16 multiplier subunit cut the size of the arrays in half [Trem96b].

3.1.5 Shifts

It is a widely known optimization shortcut in binary arithmetic that a multiplication by an integer number N, which is a power of two, can be accomplished by a left shift of the other operand by $log_2 N$ bits. Division by the same class of integers can be performed in the same way, except that a right shift is used. This substitution is often a performance gain because a shift operation typically has a much lower latency than multiplication. Shifts are also needed for proper data alignment at the bit level of granularity, as opposed to the byte or higher level granularity most communication operations work with. The shift operations available on each platform are compared in Table 3.

3.1.5.1 Synthesizing Multiplication Although HP's multimedia instruction sets do not include partitioned multiplication, they do implement packed shift and add instructions, which can be used to synthesize multiplication by fractions with or without small integer components. Because the same shift amount is applied to each packed element, this substitution is only viable on a SIMD architecture when multiplying all of the subwords by the same constant. Although this criterion can often be met in multimedia applications, we found that programming shift-add combinations in place of multiplications made the job of assembly level programming considerably less flexible, more time consuming and more error prone. More importantly, this approach constrains the way in which data parallelism can be exploited.

3.1.5.2 Graphics Status Register Sun's VIS architecture does not include partitioned shift instructions, but does include a graphics status register (GSR). This register has a 3-bit `addr_offset` field which is used implicitly for byte granularity alignment, as well as a 4-bit `scale_factor` field for packing/truncation operations. Like other architectures, Sun included partitioned 8-bit and 32-bit integers as supported data types. What is unusual is that the 16-bit partitioned data type assumes a fixed point number with some fractional component ($S(a.b)$ rather than $S(a.0)$ as is more typical). This has important ramifications for other 16-bit operators. Pack operations work by first left shifting an element by the number of bits specified in the `scale_factor` field of the GSR and clipping at some implicit binary point. For the signed 16-bit to unsigned 8-bit pack operation, the implicit binary point is between bits 6 and 7 of each element, so if 16-bit $S(15.0)$ integers are the actual data type, it is necessary to place `0x7` in the `scale_factor` field of the GSR before executing the `fpack16` instruction.

As is often the case with shared, singular resources, the VIS GSR turns out to be a serializing bottleneck. All of the data scaling functionality and alignment functionality is pushed through the GSR. Because VIS lacks partitioned shift operations, we found ourselves synthesizing such operations with the packing and alignment operations where no other algorithmic path was possible. We found that even with careful planning of packing and alignment operations it was often necessary to write to the GSR several times in each iteration of the loops of our multimedia kernels. The serializing effect of this singular resource prevented VIS operations from proceeding at the full possible degree of parallelism. To see this in action, consider the initial step of the color space conversion kernel which converts from band-interleaved to a more SIMD-friendly band-separated pixel form in Algorithm 8.

Algorithm 8 Color Space Conversion Kernel Inner Loop

```
rgb_to_yuv_loop:
   !  load band interleaved input data
   alignaddr  %i0, %g0, %l0
   ldd        [%l0], %f2
   ldd        [%l0 + 8], %f4
   faligndata %f2, %f4, %f2    !  %f2:  |R0|G0|B0|R1|G1|B1|R2|G2|
   ldd        [%l0 + 16], %f6
   faligndata %f4, %f6, %f4    !  %f4:  |B2|R3|G3|B3|R4|G4|B4|R5|
   ldd        [%l0 + 24], %f8
   faligndata %f6, %f8, %f6    !  %f6:  |G5|B5|R6|G6|B6|R7|G7|B7|
   !  convert from band interleaved to band separated format
                               !  %f2:  |R0|G0|B0|R1|G1|B1|R2|G2|
   wr         %g0, 0x3, %gsr   !  set alignment for <<24
   faligndata %f2, %f4, %f8    !  %f8:  |R1|G1|B1|R2|G2|B2|R3|G3|
   wr         %g0, 0x6, %gsr   !  set alignment for <<48
   faligndata %f2, %f4, %f10   !  %f10:  |R2|G2|B2|R3|G3|B3|R4|G4|
   wr         %g0, 0x1, %gsr   !  set alignment for <<48
   faligndata %f4, %f6, %f12   !  %f12:  |R3|G3|B3|R4|G4|B4|R5|G5|
   wr         %g0, 0x4, %gsr   !  set alignment for <<32
   faligndata %f4, %f6, %f14   !  %f14:  |R4|G4|B4|R5|G5|B5|R6|G6|
   wr         %g0, 0x7, %gsr   !  set alignment for <<56
   faligndata %f4, %f6, %f16   !  %f16:  |R5|G5|B5|R6|G6|B6|R7|G7|
   wr         %g0, 0x2, %gsr   !  set alignment for <<16
   faligndata %f6, %f6, %f18   !  %f18:  |R6|G6|B6|R7|G7|B7|G5|B5|
   wr         %g0, 0x5, %gsr   !  set alignment for <<40
   faligndata %f6, %f6, %f20   !  %f20:  |R7|G7|B7|G5|B5|R6|G6|B6|
   fpmerge    %f2, %f10, %f22  !  %f22:  |R0|R2|G0|G2|B0|B2|XX|XX|
   fpmerge    %f8, %f12, %f24  !  %f24:  |R1|R3|G1|G3|B1|B3|XX|XX|
   fpmerge    %f14, %f18, %f26 !  %f26:  |R4|R6|G4|G6|B4|B6|XX|XX|
   fpmerge    %f16, %f20, %f28 !  %f28:  |R5|R7|G5|G7|B5|B7|XX|XX|
   fpmerge    %f22, %f24, %f2  !  %f2:  |R0|R1|R2|R3|G0|G1|G2|G3|
   fpmerge    %f23, %f25, %f4  !  %f4:  |B0|B1|B2|B3|XX|XX|XX|XX|
   fpmerge    %f26, %f28, %f6  !  %f6:  |R4|R5|R6|R7|G4|G5|G6|G7|
   fpmerge    %f27, %f29, %f8  !  %f8:  |B4|B5|B6|B7|XX|XX|XX|XX|
```

We will first explain what this kernel does and then come back to our discussion of the problematic GSR. Pixel data consists of one or more channels or bands, with each channel representing some independent value associated with a given pixel's (x, y) position. A single channel, for example, represents grayscale. A three channel image is typically color, while a more uncommon four channel image is also color but with the extra channel encoding some extra parameter (e.g. transparency). All of the imaging and video algorithms which

we looked at worked with three-band color image data, such as shown in Figure 3.

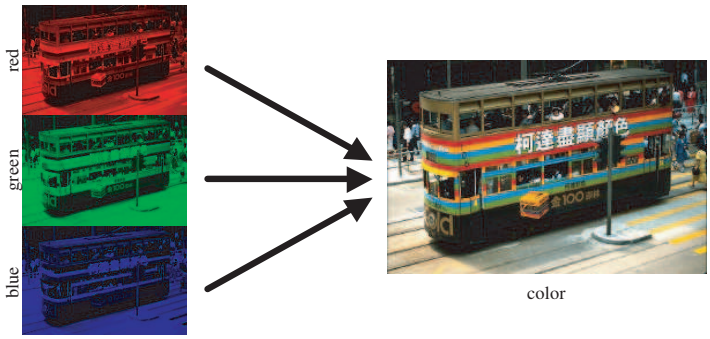


Figure 3: **Three Band Color Images**

The band data may be interleaved (each pixel's red, green, and blue data are adjacent in memory) or separated (e.g. the red data for adjacent pixels are adjacent in memory). In image processing algorithms such as color space conversion we operate on each channel in a different way, so band separated format is the most convenient for SIMD processing. Unfortunately, a lot of image data is stored in band-interleaved format, so it is often necessary to convert between the two. In the color conversion code fragment, each shift operation requires a write to the GSR, with each following instruction depending on the contents of the GSR to operate correctly. This is unfortunate because explicit writes to the GSR stall the processor (6 cycles in the case of UltraSPARC I) [Rice96]. If true shift operations (in this case, simply full register, 64-bit shifts) had been included in VIS, a new shift could begin each cycle, assuming a pipelined implementation.

3.1.6 Data Communication Operations

SIMD instructions perform the same operation on multiple data elements. Because all of the data within a register must be treated identically, instructions with the ability to efficiently rearrange data bytes within and between registers are crucial to efficiently utilizing a SIMD instruction set; we hinted at this earlier when discussing reducing multiplication operations. We refer to these types of instructions as *data communication* operations. The solutions of the various existing multimedia instruction sets are presented in Table 4. These operations are data type independent (they do not depend on knowing how a field of bits is to be interpreted), and so can be shared between partitioned integer and floating point data types assuming the same registers are used for both representations (for example, as in AMD's 3DNow!).

Interleave type communication instructions (also referred to as *mixing*, *unpacking* or *merging*) merge alternate data elements from the upper or lower half of the elements in each of two source registers. *Align* or *rotate* operations allow for arbitrary byte-boundary data realignment of the data in two source registers; essentially a shift operation that is done in multiples of 8-bits at a time. Both interleave and align type operations have hard coded data communication patterns. *Insert* and *extract* operations allow for a specific packed element to be extracted as a scalar or a scalar value to be inserted to a specified location. The desired element location is typically specified as an immediate value encoded in the instruction.

Shuffle (also called *permute*) data communication operations allow greater flexibility than those operations with fixed communication patterns, but this added flexibility requires that the communication pattern be specified either in a third source register or as an immediate value in part of the instruction encoding. Instruction set architects have taken two approaches: either a set of predefined communications patterns can be provided, or full arbitrary mapping capabilities can be implemented. MIPS' MDMX uses a predefined set of eight 8-bit wide shuffles, and eight 16-bit wide shuffles to implement a partial shuffle operation (a canonical set of communication patterns is not provided). Hewlett Packard's `mix` instruction is able to achieve full shuffles for the four 16-bit elements packed within a 64-bit register because there are $4^4 = 256$ possible ways to map the input values. This requires $\lceil log_2(256) \rceil = 8$ bits to encode, which fits within the allocated immediate encoding bits of the HP PA-RISC architecture.

Encoding the desired full shuffle in an immediate field is not always a possibility, especially on architectures with larger register widths or those which shuffle at byte-level granularity. For example, on Motorola's 128-bit wide AltiVec extension, the `vperm` instruction, which concatenates two vectors to use as source elements, a full byte oriented shuffle would involve more than $32^{16} = 1.2089 \times 10^{24}$ mappings, requiring 80 bits to encode as an immediate. Clearly, encoding the desired communication pattern as an immediate is not a reasonable approach. Instead, AltiVec offers a three operand format for shuffling in which the first two specified vectors are concatenated, and the bytes of the resulting 256-bit double register data has its bytes numbered as shown in Figure 4. Each element in `vC` specifies the index of the element in the concatenation of `vA` and `vB` to place in the corresponding element of `vD`.

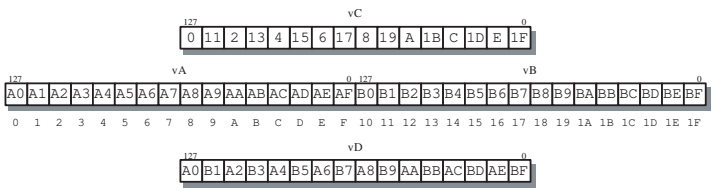


Figure 4: **AltiVec `vperm vd,va,vb,vc` Instruction** - byte elements are selected from the concatenation of registers `va` and `vb` by the byte index specified in `vc`.

Although a full shuffle operation as is found in Motorola's AltiVec is extremely powerful, and is a superset of any other data communication operation, it is costly in terms of register space because every required data communication pattern must be kept in a register. It also increases memory bandwidth requirements to load the communication patterns from memory to registers. In practice, we found that simpler interleave and rotate operations could be used in place of full shuffles in many, although not all cases. The sufficiency of simpler data communication operations is also dependent on

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Interleave/Merge	8,16,32	8,16,32	-	-	8,16,32	32	8,16,32	8
Align/Rotate	-	-	-	-	-	8	8	8
Shuffle/Permute	$16(4^4)$	-	-	$16(4^4)$	$16(4^4)$, $32(4^4)$	8(8), 16(8)	$8(32^{16})$	-
Insert/Extract	16	-	-	-	16	-	8,16,32	-

Table 4: **Data Communication Operations** - Shuffle operations are specified as N(M), for N-bit wide data elements and M possible patterns

the vector length employed. For example, 128-bit AltiVec vectors contain eight elements, while a shorter extension such as Intel's 64-bit MMX contain only four of the same type of element. This means that simple data rearrangement operations (e.g. merge) cover a relatively larger fraction of all possible mappings in the case of the shorter vector length.

Data rearrangement operations which communicate between two registers can provide useful single register functionality when both source operands are set to be the same register. In addition, the functionalities of interleave, shuffle and align operations are certainly not exclusive. For example, the interleave operation on MIPS' MDMX is done through one of the provided communication patterns of the `shfl` (partial shuffle) instruction. Unpack or interleave operations can also function as data width promotion for unsigned values if one of the source operands is zero.

Algorithm 9 AltiVec Matrix Transpose - matrix elements are 16-bits wide

```
;; v8:   |0_0|0_1|0_2|0_3|0_4|0_5|0_6|0_7|
;; v9:   |1_0|1_1|1_2|1_3|1_4|1_5|1_6|1_7|
;; v10:  |2_0|2_1|2_2|2_3|2_4|2_5|2_6|2_7|
;; v11:  |3_0|3_1|3_2|3_3|3_4|3_5|3_6|3_7|
;; v12:  |4_0|4_1|4_2|4_3|4_4|4_5|4_6|4_7|
;; v13:  |5_0|5_1|5_2|5_3|5_4|5_5|5_6|5_7|
;; v14:  |6_0|6_1|6_2|6_3|6_4|6_5|6_6|6_7|
;; v15:  |7_0|7_1|7_2|7_3|7_4|7_5|7_6|7_7|
vmrghh v0, v8, v12   ; v0:  |0_0|4_0|0_1|4_1|0_2|4_2|0_3|4_3|
vmrglh v1, v8, v12   ; v1:  |0_4|4_4|0_5|4_5|0_6|4_6|0_7|4_7|
vmrghh v2, v9, v13   ; v2:  |1_0|5_0|1_1|5_1|1_2|5_2|1_3|5_3|
vmrglh v3, v9, v13   ; v3:  |1_4|5_4|1_5|5_5|1_6|5_6|1_7|5_7|
vmrghh v4, v10, v14  ; v4:  |2_0|6_0|2_1|6_1|2_2|6_2|2_3|6_3|
vmrglh v5, v10, v14  ; v5:  |2_4|6_4|2_5|6_5|2_6|6_6|2_7|6_7|
vmrghh v6, v11, v15  ; v6:  |3_0|7_0|3_1|7_1|3_2|7_2|3_3|7_3|
vmrglh v7, v11, v15  ; v7:  |3_4|7_4|3_5|7_5|3_6|7_6|3_7|7_7|

vmrghh v8, v0, v4    ; v8:  |0_0|2_0|4_0|6_0|0_1|2_1|4_1|6_1|
vmrglh v9, v0, v4    ; v9:  |0_2|2_2|4_2|6_2|0_3|2_3|4_3|6_3|
vmrghh v10, v1, v5   ; v10:  |0_4|2_4|4_4|6_4|0_5|2_5|4_5|6_5|
vmrglh v11, v1, v5   ; v11:  |0_6|2_6|4_6|6_6|0_7|2_7|4_7|6_7|
vmrghh v12, v2, v6   ; v12:  |1_0|3_0|5_0|7_0|1_1|3_1|5_1|7_1|
vmrglh v13, v2, v6   ; v13:  |1_2|3_2|5_2|7_2|1_3|3_3|5_3|7_3|
vmrghh v14, v3, v7   ; v14:  |1_4|3_4|5_4|7_4|1_5|3_5|5_1|7_5|
vmrglh v15, v3, v7   ; v15:  |1_6|3_6|5_6|7_6|1_7|3_7|5_7|7_7|

vmrghh v0, v8, v12   ; v0:  |0_0|1_0|2_0|3_0|4_0|5_0|6_0|7_0|
vmrglh v1, v8, v12   ; v1:  |0_1|1_1|2_1|3_1|4_1|5_1|6_1|7_1|
vmrghh v2, v9, v13   ; v2:  |0_2|1_2|2_2|3_2|4_2|5_2|6_2|7_2|
vmrglh v3, v9, v13   ; v3:  |0_3|1_3|2_3|3_3|4_3|5_3|6_3|7_3|
vmrghh v4, v10, v14  ; v4:  |0_4|1_4|2_4|3_4|4_4|5_4|6_4|7_4|
vmrglh v5, v10, v14  ; v5:  |0_5|1_5|2_5|3_5|4_5|5_5|6_5|7_5|
vmrghh v6, v11, v15  ; v6:  |0_6|1_6|2_6|3_6|4_6|5_6|6_6|7_6|
vmrglh v7, v11, v15  ; v7:  |0_7|1_7|2_7|3_7|4_7|5_7|6_7|7_7|
```

To see the importance of data communication operations, consider a two dimensional discrete cosine transform (DCT). Such a transform is efficiently computed as 1D transforms on each row followed by 1D transforms on each column. A SIMD algorithmic approach requires that multiple data elements from several algorithmic iterations be operated on in parallel for the greatest efficiency. This is straight forward for the 1D column DCT, since the corresponding elements of each column are linearly adjacent in memory, thus several columns can be operated on in parallel without rearranging the data as read from memory. A 1D row DCT is more problematic since the corresponding elements of adjacent rows are not adjacent in memory. It is possible to transpose a matrix making corresponding "row" elements adjacent in memory, perform the desired computation, and then transpose the matrix again (if needed) to put the resulting data back in the correct configuration. An example of a matrix transposition through SIMD data communication instructions for Motorola's AltiVec is shown in Algorithm 9.

3.1.7 Width and Type Conversion

Data width promotion and demotion operations are critical to making a multimedia instruction set applicable to as many different applications as possible. Without a way to convert data between the formats used for storing the data and for operating on the data, SIMD computation cannot proceed efficiently. Although many multimedia data types are small integers (typically 8 or 16 bits), computations on these values often require greater precision. Also, many of the architectures offer operations on partitioned floating point in addition to partitioned integers. Thus, data conversion is a common operation. Supported width and type conversion operations of the examined architectures are listed in Table 5.

Pack or *width demotion* operations convert a larger width data type to a smaller width one either through truncation or by first clamping the values in question to the range of the smaller data type. *Unpack* or *width promotion* operations go the opposite direction, and so require no clamping, although the type of sign extension (zero extend or sign extend) is important. Unsigned unpacking can often be accomplished by utilizing the functionality of interleave or shuffle data communication operations. Saturating packs first clamp the input value to the range of the output data type, while truncating pack operations assume that the input values will fit in the output data type's representable range. Packing with truncation can similarly be done through data communication operations. Few architectures support signed packing and unpacking, although signed unpack can be accomplished somewhat awkwardly through a signed multiplication by 1, since the product will be twice as wide as the operands.

DEC MVI's pack and unpack operations are limited to truncation and zero extension, respectively, so signed values

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Saturating FP Pack	FP32→S32 FP32→S16	-	-	-	FP32→S32	-	FP→S32 FP→U32	-
Truncating FP Pack	-	-	-	-	FP64→FP32 FP64→S32	-	-	-
Saturating 32-bit Pack	S32→S16	S32→S16	-	-	S32→S16	-	S32→S16	S32→U8 S3→S16
Saturating 16-bit Pack	S16→S8 S16→U8	S16→S8 S16→U8	-	-	S16→S8, S16→U8	-	S16→S8 S16→U8	S16→U8
Truncating 32-bit Pack	32→16* 32→8*	32→16* 32→8*	32→8	32→16*	32→16* 32→8*	32→16	32→16	-
Truncating 16-bit Pack	16→8*	16→8*	16→8	-	16→8*	16→8	16→8	-
FP Unpack					FP32→FP64			
32-bit Unpack	S32→FP32	-	-	-	S32→FP32 S32→FP64	-	U32→FP32 S32→FP32	-
16-bit Unpack	U16→U32* S16→FP32	U16→U32*	-	U16→U32*	U16→U32*	U16→U32	U16→U32*	U16→U32
8-bit Unpack	U8→U16*	U8→U16*	U8→U16 U8→U32	-	U8→U16*	U8→U16 S8→S16	U8→U16*	U8→U16*

Table 5: **Packed Data Type Conversion Operations** - The prefix 'S' indicates a signed operation, 'U' an unsigned operation, 'FP' a floating point operation, and no prefix indicates that an operation works for both signed and unsigned integer values. Saturating pack operations first clamp the input value to the range of the output data type, while truncating pack operations assume that the input values will fit in the output data type's representable range, and so simply truncate the input bits to the correct width. (*) designates operations derived from data communication operations.

are not possible except when packing. HP's MAX depends entirely on its `permute` and `mix` instructions for width and type conversion. On HP's MAX-1/MAX-2 architecture, partitioned 8-bit operations were considered, but rejected due to insufficient precision. What this overlooks is that fact that even though many intermediate computations require greater precision than 8-bits, many types of video and imaging data are stored this way in existing multimedia file formats. Thus, packing and unpacking to and from 8-bit precision is a very common operation which is not supported in hardware, making HP's extensions inefficient at processing this type of data. The conclusion to be drawn from this is that *all* data types that occur in multimedia should be supported for packing and unpacking even for those widths not directly supported by arithmetic operations. It should always be possible to convert to a width that is supported for computation.

3.2 Floating Point

Within the Berkeley Multimedia workload, 3D graphics, MPEG audio coding and speech recognition applications all contain floating point intensive kernels. Most traditional scalar floating point architectures support single precision (32-bit) and double precision (64-bit) data types. Single precision is generally sufficient for all but numerical scientific applications requiring great precision. Intel's SSE2 is thus far the only extension to offer packed double precision floating point operations, and is targeted at applications beyond the domain of multimedia. A SIMD approach to floating point is useful, although it does not offer as much potential speedup as integer SIMD.

Cyrix, DEC, Hewlett Packard and Sun currently do not offer packed floating point capabilities. Basic arithmetic functionality (add, subtract, multiply) is included by in all architectures supporting packed floating point extensions (AMD's 3DNow, Intel's SSE, and Motorola's AltiVec). These operations are generally useful on both audio and 3D applications. We found multiply-accumulate operations to be useful when implementing the fast Fourier transform, which is found in the LAME MPEG-1 layer III audio encoding and Rasta speech recognition applications. It was also useful in the synthesis filter bank kernel from the mpg123 MPEG-1 layer III audio decoding application.

3.2.1 Square Root / Reciprocal Approximation

Beside basic arithmetic functionality, all of the floating point extensions include some form of reciprocal approximation and square-root approximation. These operations are targeted at the 3D geometry pipeline. A kernel derived from the Mesa source code (Mesa is an open source implementation of SGI's OpenGL API) is listed in Algorithm 10. Both the square root and reciprocal function can be seen to be central to this kernel.

Function approximation instructions are typically implemented as hardware lookup tables, returning k-bits of precision. In Intel's SSE, for example, approximate reciprocal (`rcp`) and reciprocal square root (`rsqrt`) return 12-bits of mantissa. Full IEEE compliant operations return 24-bits of mantissa. Intel's SSE supplies the full precision but slower

Algorithm 10 Transform and Normalize Kernel Code Fragment

```
for (i=0;i<n;i++) {
   FLOAT64 tx, ty, tz;
   {
      FLOAT32 ux = u[i][0], uy = u[i][1], uz = u[i][2];
      tx = ux * m0 + uy * m1 + uz * m2;
      ty = ux * m4 + uy * m5 + uz * m6;
      tz = ux * m8 + uy * m9 + uz * m10;
   }
   {
      FLOAT64 len, scale;
      len = sqrt( tx*tx + ty*ty + tz*tz );
      scale = (len>1E-30) ? (1.0 / len):  1.0;
      v[i][0] = tx * scale;
      v[i][1] = ty * scale;
      v[i][2] = tz * scale;
   }
}
```

divide (`div`) and square root (`sqrt`) instructions, as well as full double precision (but no fast double precision approximations) in SSE2. None of the other architectures include full precision instructions, as the Newton-Raphson method can usually be employed where greater precision than the approximated value is required. One iteration of the Newton-Raphson method on a 12-bit precise approximation returns a 22-bit precise result [Thak99]. Motorola's AltiVec also returns 12-bits of precision for both the reciprocal and reciprocal square root approximation instructions. AltiVec also includes approximate log_2 and exp_2 instructions, which find application in the lighting stage of a 3D rendering pipeline.

AMD's 3Dnow! and MIPS' MIPS-3D extensions include instructions to automatically utilize the Newton-Raphson method to make initial approximations more precise. Intel's SSE architecture includes no such instructions nor does Motorola's AltiVec, but both point the programmer to using the Newton-Raphson method for greater precision. AMD's 3DNow! returns 14-bits of precision for the reciprocal approximation (`pfrcp`) instruction and 15-bits for the reciprocal square root approximation (`pfrsqrt`) instruction. The Newton-Raphson iteration instructions of 3Dnow! return full (24-bit) precision, so full precision versions of these operations are unnecessary. It should be noted that the AMD reciprocal and square root estimation instructions are actually scalar operations - only the lower element of a packed single precision register is used, with the scalar result being placed in both the top and bottom packed elements of the destination register. Intel's SSE, on the other hand, is a true vector operation, as it operates on each of the four specified packed single precision elements, computing four results in parallel.

3.2.2 Exceptions

Exception handling has the same problem dealing with packed values as overflow and other instruction stream dependencies on data within SIMD architectures. Checking result flags or generating an exception from a packed operation requires considerable time to determine which packed element caused the problem. For this reason, AMD's 3DNow! instructions take the view that packed instructions should never raise exceptions. In multimedia applications there is little desire for hardware status and exceptions at the expense of lower real time performance. Like saturating integer arithmetic, AMD's implementation of 3DNow! generates properly signed maximum representable numbers in the case of numeric overflow, and flushes results to zero in underflow situations [Ober99].

Motorola's AltiVec does not report IEEE floating point exceptions, although of course the regular scalar floating point instructions are IEEE compliant and report all appropriate exceptions. In most cases where an exception might be raised it is possible to fill in a value which will give reasonable results in most applications. This is similar to saturating integer arithmetic where maximum or minimum result values are substituted rather than checking for and reporting positive or negative overflow. This speeds execution because no error condition checking need be done. In the case of AltiVec, default values for all floating point exceptions are as specified by the Java Language Specification.

Intel's SSE and SSE2 include a control/status register (`MXCSR`) to mask or unmask packed floating point numerical exceptions, set rounding modes and check status flags (if any numerical exceptions have occurred). By default, all numerical exceptions are masked. The status flags are "sticky", and can only be cleared by writing zeros to their locations in the `MXCSR`. The numerical exceptions include the pre-compute exceptions *Invalid Operation*, *Divide by Zero* and *Denormal Operation*, as well as post-compute exceptions *Numeric Overflow*, *Numeric Underflow*, and *Inexact Result*.

When an exception occurs in an SSE instruction the actual element that caused the exception is not reported and neither of the 64-bit micro-ops retires (writes a result to the destination register) in a 128-bit instruction. MIPS-V takes a similar approach in dealing with exceptions which occur during the processing of its paired floating point format. Exception flags from both packed elements are combined together and the hardware makes no effort to determine which element caused the exception.

3.2.3 Rounding

Intel's SSE and SSE2 offer two modes of rounding: IEEE compliant and another, faster, flush to zero (FTZ) mode. Flush to zero (FTZ) clamps to a minimum representable result in the event of underflow (a number too small to be represented in single precision floating point). Most real time 3D applications use the FTZ rounding mode since they are not particularly sensitive to a slight loss in precision [Thak99]. 3DNow! supports only truncated rounding (round to zero). Fully compliant IEEE floating point supports four rounding modes.

Motorola's AltiVec offers two modes for floating point. An IEEE compliant mode and a potentially faster non-IEEE compliant mode. All AltiVec floating point arithmetic instructions use the IEEE default rounding mode of round to nearest. The IEEE directed rounding modes are not provided. Instructions for explicitly rounding towards minus infinity, nearest integer, positive infinity or zero are included for converting from floating point to integer formats.

3.2.4 Scalar Operations

Intel was the only vendor to explicitly add scalar floating point operations along with their packed floating point extension, duplicating the existing x87 floating point capabilities. This was done in order to aid clean up at the end of a strip mined loops without reverting to the cumbersome stack-based x87 floating point available on the Pentium III. In addition, results generated via x87 floating point operations can potentially differ from SSE floating point results because x87 FP computations are carried out in 80-bit precision, while SSE utilizes 32-bit operations or 64-bits in the case of SSE2. Masked (selective) operations allowing for any element of an SSE floating point register to be disabled or enabled during computation were also considered, but rejected for design complexity and lack of compelling applications [Thak99]. The AMD Athlon processor also of course supports x87 floating point, but does not implement a separate set of 32-bit scalar floating point instructions. None of the other vendors found it necessary to add scalar floating point operations, as their existing floating point designs do not suffer from the aforementioned x87 deficiencies.

3.3 Polymorphic Operations

Polymorphic operations are those for which the same instruction can be used independently of the partition width (bitwise operations). Table 6 briefly summarizes these types of operations present on the architectures we have examined.

DEC's MVI and Hewlett Packard's MAX-1 and MAX-2 include no additional logical or other bit-wise operations because they are implemented on the integer data paths of their respective architectures. The integer data paths already offer considerable bit-wise and logical functionality so additional instructions are unnecessary. MIPS-V (as well as MIPS-I through MIPS-IV) does not include logical or other bit-wise operations on the floating point data path. Only MDMX defines these operations on the MIPS floating point data path.

ANDN and ORN refer to bitwise AND and OR operations respectively in which the complement of one of the source registers is used as one of the operands. The combination of AND and ANDN operations is useful for working with bit masks since we often want to use the bit mask to select some data elements from one register with AND and then select elements with the reverse bit mask from another register using ANDN. Both results can then be merged with an OR or addition operation. The operation of zeroing a register is easily synthesized through an exclusive or operation between source operands taken from the same register.

3.4 Comparisons and Control Flow

Operating on multiple data elements in parallel can be problematic if a computation should only be performed if an operand value passes some conditional check. Most interesting computational loops exhibit some sort of control flow variation. Conventional single condition code flags are meaningless for the results of packed operations, and control flow through software (e.g. check each operation separately) defeats the attempted extraction of data parallelism [Luml97].

Traditional SIMD architectures with multiple parallel processors performing the same operation on separate data elements simply disable the necessary processing elements for portions of computation. Contemporary SIMD within a register architectures do not support disabling arbitrary fields. Hardware designers have so far taken two approaches to this problem: bit vector flags and bit masks. An overview of the types of partitioned comparison and control flow operations supported by each architecture is given in Table 7.

3.4.1 Element Masks

An *element mask* is a vector in which each packed element contains either all 1's or all 0's. Compare operations result in an element mask corresponding to the length of the packed operands. For example, the result of a comparison between packed 16-bit values using the Intel MMX `pcmpeqw` instruction results in a 64-bit wide element mask containing four 16-bit sub-elements, each consisting either of all 1's (0xFFFF) where the comparison condition is true or all 0's (0x0000) when it is false. These masks are then used in conjunction with logical operations such as AND, ANDN and OR to achieve the desired conditional assignment. Element masks are by far the most popular means of packed comparisons because they allow for computation to continue without any disruption of actual control flow - the instruction stream is not data dependent, only the result.

3.4.2 Bit Vectors and Partial Stores

Another approach to control flow is the generation of bit vectors, in which a single bit represents the result of a comparison for each element. Bit vectors are typically used in conjunction with partial store operations to generate the desired result in memory. An inverted version of the bit vector is then used to partially store the remaining (non-computed) elements if necessary; the uncomputed version may already exist in memory. Depending on timing, successive stores to portions of a word may be combined in a store buffer and therefore may not generate separate bus transactions [Rice96].

MIPS' MDMX implements eight comparison bits (only four bits are used when comparing 16-bit values) for comparing unsigned bytes much like traditional integer or floating point comparison flags. MMX supports both element masks and bit-vectors with partial stores. This is done through instructions that convert a mask to a bit vector and copy the result to an integer register. Sun's VIS takes a similar approach, but writes a bit vector (each bit indicates the true/false result of the comparison between the corresponding elements in two partitioned source registers) resulting from packed comparisons to an integer register. This bit-vector is subsequently used for partial store instructions to indicate which packed elements should be stored to a given memory address. Sun's VIS depends entirely on bit masks with partial stores for comparisons. The disadvantage of only having partial store operations to utilize the results of comparisons is that extra memory

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
Operation	AND,ANDN, OR,XOR	AND,ANDN, OR,XOR	-	-	AND,ANDN, OR,XOR	AND,NOR, OR,XOR	AND,ANDN,NOR, OR,XOR	NOT,OR,NOR,ORN,XNOR AND,ANDN,NAND,XOR

Table 6: **Polymorphic (Bitwise) Operations** - AND $= A \cdot B$, NAND $= \overline{A \cdot B}$, ANDN $= A \cdot \overline{B}$, OR $= A + B$, NOR $= \overline{A + B}$, ORN $= A + \overline{B}$, XOR $= A \oplus B$, XNOR $= \overline{A \oplus B}$

	AMD	Cyrix	DEC	HP	Intel	MIPS	Motorola	Sun
$==$	8,16,32,FP32	8,16,32	-	-	8,16,32,FP32,FP64	8,16,FP32	8,16,32,FP32	16,32
$!=$	-	-	-	-	FP32,FP64	-	-	16,32
$>$	S8,S16,S32,FP32	S8,S16,S32	-	-	S8,S16,S32,FP32,FP64	FP32	U8,U16,U32,S8,S16,S32,FP32	S16,S32
$\geq$	FP32	-	-	-	FP32,FP64	-	FP32	-
$<$	-	-	-	-	FP32,FP64	U8,S16,FP32	-	-
$\leq$	-	-	-	-	FP32,FP64	U8,S16	-	S16,S32
$<>$	-	-	-	-	-	FP32	FP32	-
?	-	-	-	-	FP32,FP64	FP32	-	-
!?	-	-	-	-	FP32,FP64	-	-	-

Table 7: **Packed Data Control Flow** - integer comparison operations are either sign independent or denoted as signed (S), or unsigned (U), with the width of the packed data type in bits indicated. Floating point (FP) comparisons are also listed for both single (FP32) and double (FP64) precision. The "?" comparison checks if operands are ordered (?) or unorderd (!?) - this is further explained in the Appendix.

traffic is generated, especially in cases where the comparison result must be reloaded for further computation.

Although not all types of comparison tests are supported by every architecture, it is usually possible to synthesize any required type of comparison operation from another by reversing the operands:

$$\begin{aligned} A \leq B &\Longrightarrow B > A \\ A < B &\Longrightarrow B \geq A \end{aligned} \tag{1}$$

Because of this property, only one set of comparison operations ({>, >,=} or {<, <, =}) are needed to provide full functionality. Depending on how comparison operations are encoded this may or may not conserve instruction encoding space. For example, the MIPS architecture allows arbitrary combinations of any of the available tests and their inverses because each available test {<,>, =, ?} is encoded as a single immediate bit within a comparison instruction.

Motorola's AltiVec and MIPS' MIPS-3D both include a specialized comparison instruction for dealing with boundary testing as utilized in the clip test kernel from Mesa's 3D rendering pipeline (Algorithm 11). The MIPS-3D comparison tests two clip values in parallel to see if any clipping is needed, utilizing a specialized branch instruction to act as a fast out if no clipping is necessary. This works by noticing the following equivalency:

$$\begin{aligned} (x \geq -w, \quad x \leq w) \quad &= |x| \leq w \\ (y \geq -w, \quad y \leq w) \quad &= |y| \leq w \\ (z \geq -w, \quad z \leq w) \quad &= |z| \leq w \end{aligned}$$

A compare absolute value instruction (`CABS.cc`) is part of the MIPS-3D instruction set, and can compares the absolute values of two packed registers. So, two `CABS` comparisons must be issued to cover a triplet of x, y, and z values. Comparing absolute values has the advantage of setting only three condition code bits (as opposed to six). This allows for a specialized branch instruction like `bc1any4f` (branch if any one of four consecutive comparison code bits are false) to branch and begin computing the clipping of the next vertex if no clipping occurred for the current vertex. Motorola's AltiVec also introduces a specialized comparison for clip checking called a bounds comparison (`vcmpbfpx`) which is similar to the MIPS-3D method, except that a two-bit vector is generated for each element. Because the AltiVec extension is 128-bits wide, all three comparisons (x, y and z compared to w) can be done with a single instruction.

Algorithm 11 Project and Clip Test Kernel C Code Fragment

```
for (i=0;i<n;i++) {
   FLOAT32 ex = vEye[i][0], ey = vEye[i][1];
   FLOAT32 ez = vEye[i][2], ew = vEye[i][3];
   volatile FLOAT32 cx = m0 * ex + m8 * ez ;
   volatile FLOAT32 cy = m5 * ey + m9 * ez ;
   volatile FLOAT32 cz = m10 * ez + m14 * ew;
   volatile FLOAT32 cw = -ez ;
   UINT8 mask = 0;
   vClip[i][0] = cx;
   vClip[i][1] = cy;
   vClip[i][2] = cz;
   vClip[i][3] = cw;
   if (cx > cw) mask |= CLIP_RIGHT_BIT;
   else if (cx < -cw) mask |= CLIP_LEFT_BIT;
   if (cy > cw) mask |= CLIP_TOP_BIT;
   else if (cy < -cw) mask |= CLIP_BOTTOM_BIT;
   if (cz > cw) mask |= CLIP_FAR_BIT;
   else if (cz < -cw) mask |= CLIP_NEAR_BIT;
   if (mask) {
      clipMask[i] |= mask;
      tmpOrMask |= mask;
   }
   tmpAndMask &= mask;
}
*orMask = tmpOrMask;
*andMask = tmpAndMask;
```

3.5 Memory

3.5.1 Load/Store Operations

In many multimedia algorithms, there are several levels at which data parallelism can potentially be exploited. For example, within the DCT for MPEG decoding, we can operate at any level from elements within a block to macroblocks to pictures. The only problem is that on SIMD architectures, load and store operations work only with adjacent elements in memory. Parallel computers have in the past offered architectural features such as strided loads and stores. AltiVec comes closest to this by implementing strided prefetching, which we will discuss shortly. Even so, there is considerable overhead involved when using data communication (rearrangement) operations to assemble otherwise non-adjacent data elements together.

The AMD, Cyrix and Intel architectures are unlike the others in our study in that they are CISC, and therefore allow complex addressing modes such as having one instruction operand in memory. When programming for these architectures, it is rare to use an explicit load or store instruction, as this would reduce instruction decoding bandwidth (requiring at least one additional instruction) and increase register pressure because of the need to explicitly retain an operand in a register before computation (only eight registers are available) [AMDOpt]. Explicit load and store instructions can be useful when scheduling memory operations far ahead of computation in order to hide memory latency (in the absence of prefetch instructions) for these architectures.

3.5.1.1 Alignment An aligned memory access is one that accesses a 2^N byte size data element at an address in which the lower N bits are all zeroes. Alignment is generally important on all architectures for performance reasons, as unaligned accesses typically suffer a significant performance penalty or cause a run time exception, depending on the behavior specified by the architecture. Intel's MMX does not allow the programmer to specify alignment; instead the hardware deals with alignment issues transparently such that access to unaligned addresses simply have a higher latency than for aligned addresses. During the design of SSE, Intel found that software vendors preferred being alerted to misalignment of load/store data via an explicit fault. For this reason, all computation instructions in SSE that use a memory operand must access data that is 16-byte (128-bit) aligned. Unaligned load/store instructions are also provided for cases where alignment cannot be guaranteed [Thak99]. Such a situation occurs when in the motion compensation step of MPEG video coding, unaligned memory access is needed depending on the motion vector [Kuro98], as the addresses of the reference macroblock can be random depending on the type of motion search being performed.

The AltiVec instruction set architecture does not provide for alignment exceptions when loading and storing data. Alignment is maintained by forcing the lower four bits of any address to be zero. This is transparent to the programmer, so the programmer is responsible for guaranteeing alignment so that the correct data is loaded or stored. We agree with the software vendors that Intel interviewed - it is better that performance and correctness issues due to alignment be made explicit. The loading of incorrect data due to a mistaken assumption about alignment would be an extremely difficult and possibly intermittent bug to track down.

3.5.1.2 Array Addressing Sun's VIS includes two sets of instructions for accelerating multimedia operations with sophisticated memory addressing needs. The first, `edge8`, `edge16`, and `edge32`, produce bit vectors to be used in conjunction with partial store instructions to deal with the boundaries in 2D images. The second group of addressing instructions include `array8`, `array16` and `array32` which find use in *volumetric imaging* (the process of displaying a two dimensional slice of a three dimensional data). An array instruction converts (x, y, z) coordinates into a memory address. The Berkeley multimedia workload does not include any volumetric imaging applications, so it is unsurprising that these instructions found no utility in our workload.

3.5.2 Prefetching

Prefetching is a hardware or software technique which tries to predict data access needs in advance, so that a specific piece of data is loaded into the cache from the main memory before it is actually needed by an application. Programmers and compilers have, and do, use dummy load instructions to achieve prefetching functionality, but at the expense of possibly blocking other (necessary) loads as well as added register pressure since any load must have a target register. Also, load instructions can cause TLB misses and page faults, which may not be acceptable. This form of prefetching is inferior to true software prefetch instructions, which use special purpose instructions to fetch data into the cache from main memory without blocking true load/store instruction accesses. They also do not waste registers on dummy targets, and do not cause exceptions. Unlike hardware prefetching which is performed automatically by the memory subsystem, software prefetching has the overhead of additional instructions which prefetch only when they are executed.

Memory access types can be divided into three classes: *temporal* (data that will be used again soon), *spatial* (data from adjacent locations will be used soon) and *non-temporal* (data which are referenced once and not reused). Data that will only be read once need not be copied into the L2 cache. Data that will not be needed immediately, but should be loaded to reduce latency, can be loaded only into the L2 cache. Data that will be referenced several times may need to be loaded into both the L1 and L2 cache levels, depending on the intervening data references. Indicating the expected type of temporal and spatial locality to the hardware from software is known as a *cache hint*. Cache hints are often implicit functions of load and store instructions which indicate to the memory subsystem the type of memory access. The memory system can then determine at what cache level the data should be loaded, at what initial MESI state to set a cache block, or if a data

element will not be reused, the data cache hierarchy can be bypassed entirely.

On the UltraSPARC architecture, prefetch instructions were introduced to the SPARC v9 instruction set architecture before actually being implemented on a processor. The Sun UltraSPARC I processor treats prefetch instructions as `nop`s. Because prefetch instructions do not update any architectural state it is possible for an UltraSPARC processor which does not support prefetching to correctly execute a program containing prefetch instructions. Sun did this to keep from stratifying the UltraSPARC architecture according to which processor model code is targeted for. UltraSPARC II does support prefetch operations, and allows for up to three simultaneous outstanding cache miss requests and two cache write back requests.

Hewlett Packard's PA-RISC architecture supports prefetching for reading and writing. Prefetch instructions are encoded as normal load and store operations with a target register r0 (always zero, discards writes). All load and store instruction support a cache hint for spatial locality (no data reuse), so this can be used in combination with prefetch.

DEC's prefetch implementation is not part of their MVI multimedia extension, but rather part of the Alpha architecture in general. A prefetch instruction (`fetch`) gives a hint to the hardware to prefetch a 512-byte aligned block of data. Prefetching for writing is accomplished through the `fetch_m` instruction. The Alpha architecture also provides a cache hint instruction (`wh64`) to indicate that a 64-byte block of data will not be read again but will be written to soon. This allows the cache to allocate resources for the block of data without reading it from memory (prevent write allocation).

Determining the ideal location for prefetch instructions in a piece of code depends on many architectural parameters, including the amount of memory to be prefetched, cache latency, system memory latency, and intervening computation time. If the time between when a prefetch is issued and the time it is used is too short, the prefetch will not effectively hide the latency of the fetch behind computation. If the prefetch is too far ahead other data may dislodge it from the cache hierarchy before it can be used, wasting the prefetch operation. Excessive prefetching squanders memory bandwidth and instruction decoding throughput, and results in lowering performance rather than improving it [Smit78], [Tse98].

Intel provides an equation for estimating prefetch distance on the Pentium II processor, but cautions that the parameters given are for "illustration only." [IAOpt] (It is not clear how some of these parameters can be determined accurately by the end programmer.) The prefetch scheduling distance, psd, in loop iterations, is defined:

$$psd = \left| \frac{N_{lookup} + N_{xfer} \cdot (N_{pref} + N_{st})}{CPI \cdot N_{inst}} \right| \qquad (2)$$

where, (Intel's sample parameters are placed in parentheses):

N_{lookup} number of clocks for memory latency (60)

N_{xfer} number of clocks to transfer a cache line (25)

N_{pref} number of cache lines to be prefetched

N_{st} number of cache lines to be stored

N_{inst} number of instructions in one loop iteration

CPI clock cycles per instruction (1.5)

In AMD's 3DNow!, the prefetch for write instruction (`prefetchw`) sets the cache line MESI state of the prefetched data to modified. [AMDOpt] states that this can save an additional 15-25 cycles for write only data in comparison to the prefetch for read instruction (`prefetch`), which sets the cache line MESI state to exclusive. [AMDOpt] also provides an equation for estimating the appropriate prefetch distance:

$$psd = 200^{(DS/C)}\ bytes \qquad (3)$$

where,

DS data stride bytes per loop iteration

C number of cycles for one loop iteration to execute entirely from the L1 cache

Of all the architectures examined, Motorola's AltiVec provided the most innovative and most powerful prefetch mechanism. Rather than issuing an explicit prefetch instruction for each desired data prefetch, a single *data stream touch instruction* (`dst`) is issued indicating the memory sequence or pattern that is likely to be accessed soon before the access is to take place. We will refer to this hybrid of hardware and software prefetching as *software directed prefetching* to indicate that a separate prefetch instruction need not be issued for each data element. A data stream is defined by:

effective address address of the first unit in a sequence

unit size number of quad words (128-bits) in each unit

count total number of units in a sequence

stride number of bytes between the effective address of one unit in the sequence and the next

A `dst` instruction specifies the starting address, a block size (1-32 vectors, where a vector is 128-bits long), a number of blocks to prefetch (1-256 blocks) and a signed stride (-32768...+32768). A 2-bit tag indicates the stream ID, so up to four concurrent data streams are possible, although the available data bandwidth would likely be saturated trying to prefetch four streams at once with current implementations. Hardware optimizes the number of cache blocks to prefetch so it is not necessary for the programmer to know the parameters of the cache system. A stream is fetched either until all of the requested blocks have been brought into the cache or another `dst` instruction is issued with the same tag ID.

A data stream touch instruction (`dst`) implies to the hardware that the requested data will exhibit a high degree of locality. Other flavors of `dst` indicate transient data (`dstt`) or a read once situation, and touch for store (`dstst`), which rather than placing a cache block in a state most efficient

for reading, sets it for writing. A fourth instruction allows for transient stores (`dststt`). Because unnecessary prefetching is wasteful of bandwidth, software can stop any tag ID's associated prefetch operation with the data stream stop instruction (`dss`) or all streams with `dssall`.

Unlike other variations of software prefetching, the stream construct eliminates the instruction issue overhead as well as the problem of determining the optimal prefetch distance. And, unlike software prefetching, all of the required parameters are easily accessible to the programmer because they are only those characteristics which describe the memory access pattern (unlike parameters such as the memory system's latency or processor's issue width or instruction latency).

Unfortunately, using AltiVec's data stream mechanism is not quite as clean as it initially might seem. Motorola suggests "short frequent `dst` instructions" because a prefetch stream is restarted automatically after an exception. Process task switching can possibly cause a new process to prefetch according to prefetch streams set up for the prior process, so it is not advisable to issue a single `dst` instruction to completely cover a large data stream. There are a finite number of streams (four in the case of the MPC7400 processor). It is typically the responsibility of the operating system to manage access to limited or shared resources, but AltiVec does not specify a mechanism for reading the state of an outstanding prefetch operation, so it is not clear how this could be implemented for data streams. In addition, if several multimedia (or other) applications are using data streams concurrently performance may be degraded because of their contention for system resources.

4 Programming with SIMD

Simply including a multimedia extension on a general purpose processor is not in itself a solution to handling multimedia workloads. A powerful SIMD multimedia instruction set is worthless without the means to utilize it.

4.1 Shared Libraries

One of the simplest ways to improve application performance through SIMD instructions is to rewrite shared system libraries with them. Existing applications can immediately take advantage of the new instructions without recompilation. However, performance will not improve unless an application already calls or is modified to call the appropriate system functions. Even if the appropriate functions are used, data must be formatted as specified by the API [Lee96]. Often there is a mismatch between the functions available in a library and what the target application requires to be efficient.

4.2 Macros

Macros are high level language "wrappers" which programmers utilize in order to use multimedia instructions like function calls within their C or C++ code. The advantages to this approach are that the compiler rather than the developer performs machine specific optimizations such as instruction scheduling and register allocation, and the added level of abstraction means that code can be transparently recompiled for use on platforms without multimedia extensions by replacing macros with their high level language equivalents.

[Chen96] found that programming C applications with multimedia instruction macros was difficult, likening it to typical DSP (assembly) coding. Additionally, [Allen99] found that upon examining output of the SPARCompiler (v5.0), that the instruction scheduling of expanded macro code is poor and that macros can inhibit a compiler's most aggressive optimizations.

4.3 Compilers

Ideally, high level language compilers would be able to systematically and automatically identify parallelizable sections of code and generate the appropriate SIMD instructions. SIMD optimizations would then not just be limited to multimedia applications, but could be more generally applied to any application exhibiting the appropriate type of data parallelism. Currently, none of the multimedia instruction sets are supported by commercially available compilers in this way on any of the platforms. The lack of languages which allow programmers to specify data types and overflow semantics at variable declaration time has hindered the development of automated compiler support for multimedia instruction sets [Cont97].

4.4 Assembly Language

The most effective method of programming with multimedia extensions is through hand-coding by expert programmers, just as in DSP approaches [Kuro98]. Although this method is more tedious and error prone than the other methods that we have looked at, it is available on every platform, and allows for the greatest flexibility and precision when coding.

5 Summary

This work has surveyed the field of multimedia instruction set extensions for general purpose processors. We will now summarize the general conclusions we have drawn about multimedia instruction set designs.

5.1 Singular Resources

Resources that are highly utilized should be duplicated in order to allow for parallel instruction execution, otherwise a bottleneck is created which prevents the extraction of instruction level parallelism. Although the approach taken with MIPS MDMX and its single 192-bit accumulator register was found to be flawed from this perspective, a multiple accumulator architecture could possibly achieve the latent potential of this idea. We also saw a similar serializing bottleneck in the graphics status register (GSR) which is part of Sun's VIS extension. Sun's inclusion of a GSR was not in itself necessarily a mistake, but the lack of partitioned shift operations,

which we have found to be quite important in multimedia algorithms, forced us to use the GSR in a manner that it was not really intended to be used. In general we found that architectural features such as this or the accumulator in MIPS' MDMX are poor from a design perspective because their singular nature tends to serialize instruction execution.

Poor design choices are compounded by the use of split instructions (for example, Sun's multi-part partitioned integer multiplication primitives). Dividing an operation into several instructions (which are not otherwise useful in and of themselves) increases register pressure, decreases instruction decoding bandwidth and creates additional data dependencies. Splitting SIMD instructions (which have been introduced for their ability to extract data parallelism) can actually cripple a superscalar processor's ability to extract instruction level parallelism. A multi-cycle operation can be a better solution than a multi-instruction operation because instruction latencies can be transparently upgraded in future processors, while poor instruction semantics can not be repaired without adding new instructions.

5.2 Data Types

We have seen that the data types supported by each architecture vary, especially for very short (e.g. 8-bit signed) and very long (e.g. 64-bit integer or floating point) operations. We have found that arithmetic operations on signed eight-bit data types are not used and should not be included in a multimedia instruction set. Thirty-two bit wide data types are almost exclusively used for accumulation - other operations at this width (or wider 64-bit data types) are of lesser importance. Packed, signed 16-bit multiplication is very useful, and is central to multimedia algorithms such as the discrete cosine transform. Full precision floating point and square root operations are not useful for multimedia. Less precise approximations are often sufficient, with the option of utilizing the Newton-Raphson method where more accuracy is desired or required. These observations are based on our experience programming with many different multimedia instruction sets in order to measure their performance on the Berkeley multimedia workload [Sling00d].

Data communication and data type conversion operations do not perform directly useful computation, but they are necessary to allow for SIMD computations to proceed efficiently. In the case of data type conversion operations, this is because although some data types are not common to the actual arithmetic computation of multimedia algorithms, they may be used for storage and compression. Thus, it is crucial to include a wide set of type conversion instructions. We found in many cases that the lack of the appropriate conversion operation was the limiting factor in making a SIMD implementation effective.

Along with new arithmetic functionality, many multimedia instruction set extensions have also included cache hint and prefetch instructions to exploit the largely predictable memory accesses found in many multimedia applications. The software directed prefetching with hardware assistance as found in Motorola's AltiVec appears to be a much more compelling design than the software prefetching instructions found in the other architectures, as there is much lower overhead involved; fewer prefetch instructions need to be issued.

5.3 Error Handling

Traditional error handling for issues such as positive/negative overflow, as well as many floating point exceptions must be dealt with in new ways for SIMD multimedia extensions. This is exemplified in the saturating arithmetic operations that are central to many of the instruction sets. The soft boundaries on precision in most multimedia applications, mean that many error conditions (e.g.overflow) and techniques (e.g. fully IEEE compliant floating point rounding modes and exceptions) which slow down computation when dealt with in a strict sense, can be instead corrected automatically in ways that are adequate for the domain of multimedia processing. However, the recent trend towards supporting workloads other than multimedia applications (e.g. Intel's SSE2 [Intel00b]) may limit the utility of these multimedia specific techniques.

References

[Allen99] Gregory E. Allen, Brian L. Evans, Lizy K. John, "Real-Time High-Throughput Sonar Beamforming Kernels Using Native Signal Processing and Memory Latency Hiding Techniques," *Proc. of the 33rd IEEE Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, California, October 24-27, 1999, pp. 137-141

[AMD97] Advanced Micro Devices, "AMD-3D Technology will Lead Visual Computing Revolution, Says AMD Chairman and CEO Jerry Sanders," October 14, 1997 Press Release, *http://www.amd.com/newsroom/display/ 1,1528,327,00.html*, retrieved April 24, 2000

[AMD99] Advanced Micro Devices, "AMD Introduces the 650MHz AMD Athlon Processor, the Fastest, Most Powerful Engine for x86 Computing," August 9, 1999 Press Release, *http://www.amd.com/newsroom/display/ 1,1528,400,00.html*, retrieved April 24, 2000

[AMDOpt] Advanced Micro Devices, *AMD Athlon Processor x86 Code Optimization Guide*, Publication 22007G/0, April 2000, *http://www.amd.com/products /cpg/athlon/techdocs/index.html*, retrieved April 24, 2000

[AMDWP] Advanced Micro Devices, "3DNow! Technology vs. KNI," White Paper, *http://www.amd.com/products/cpg/3dnow/vskni.html*, retrieved April 24, 2000

[Bann96] P. Bannon, "Enhancing Motion Video Performance in the Alpha Architecture," *Presented at Microprocessor Forum '96*, San Jose, California, October 22-23, 1996, *http://www.digital.com/semiconductor/mvi/index.html*, retrieved April 24, 2000

[Carl97] David A. Carlson, Ruben W.Castelino, Robert O.Mueller, "Multimedia Extensions for a 550-MHz RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol.32, No.11, November 1997, pp. 1618-1624

[Chen96] Wilam Chen, H. John Reekie, Sunil Bhave, Edward A. Lee, "Native Signal Processing on the UltraSparc in the Ptolemy Environment," *Proc. of the 30th Annual Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, California, November 3-6, 1996, Vol. 2, pp. 1368-1372

[Cont97] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, Andrew Wolfe, "Challenges to Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Vol. 30, No. 12, December 1997, pp. 33-37

[Cyrix] VIA-Cyrix, "Application Note 108: Cyrix Extended MMX Instruction Set," Application Note, April 21, 1998, *http://www.via.com.tw/pdf/cyrix/108ap.pdf*, retrieved April 24, 2000

[Edwa00] Chris Edwards, "MIPS shows 20K 64-bit core," *EE Times*, June 12, 2000, *http://www.dotelectronics.com/story/OEG20000612S0003*, retrieved June 27, 2000

[Full98] Sam Fuller, "Motorola's AltiVec Technology," White Paper, May 6, 1998, *http://www.mot.com/SPS/PowerPC /teksupport/teklibrary/papers/altivec_wp.pdf*, retrieved April 24, 2000

[Gwen98a] L. Gwennap, "AltiVec Vectorizes PowerPC," *Microprocessor Report*, Vol.12, No.6, May 11, 1998

[HP95] Hewlett Packard Inc., "HP Announces Release of PA-8000 to PRO Partners," November 2, 1995 Press Release, *http://www.hp.com/ahp/framed/technology/micropro/pa-8000/docs/111995pr.html*, retrieved April 24, 2000

[IAOpt] Intel Corporation, *Intel Architecture Optimization Reference Manual*, *http://developer.intel.com/design/pentiumii/manuals*, retrieved April 24, 2000

[Intel96] Intel Corporation, "Intel Releases MMX Technology Details to Software Community to Drive New Multimedia, Game, and Internet Applications," March 5, 1996 Press Release, *http://www.intel.com/pressroom/ archive/1996.htm*

[Intel97] Intel Corporation, "Intel Introduces The Pentium Processor With MMX Technology," January 8, 1997 Press Release, *http://www.intel.com/pressroom /archive/releases/dp010897.htm*

[Intel99] Intel Corporation, "Intel Launches The Pentium III Processor," February 26, 1999 Press Release, *http://www.intel.com/pressroom /archive/releases/dp022699.htm*, retrieved April 24, 2000

[Intel00a] Intel Corporation, "IA32 Intel Architecture Software Developer's Manual with Preliminary Intel Pentium 4 Processor Information Volume 1: Basic Architecture," *http://developer.intel.com/design/processor/future/ manuals/24547001.pdf*, retrieved September 26, 2000

[Intel00b] Intel Corporation, "Intel Announces New NetBurst Micro-Architecture for Pentium IV Processor," *http://www.intel.com/pressroom/archive/releases/ dp082200.htm*, retrieved September 27, 2000

[Kohn95] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC," *Proc.of Compcon '95*, San Francisco, California, March 5-9, 1995, pp.462-469

[Kuro98] Ichiro Kuroda, Takao Nishitani, "Multimedia Processors," *Proc. of the IEEE*, Vol. 86, No. 6, June 1998, pp. 1203-1221

[Lee95] Ruby B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, April 1995, pp. 22-32

[Lee95b] Ruby B. Lee, "Realtime MPEG Video via Software Decompression on a PA-RISC Processor, "*Proc. of the IEEE Compcon*, San Francisco, California, March 5-9, 1995, pp. 186-192

[Lee96] Ruby B. Lee, Michael D. Smith, "Media Processing: A New Design Target," *IEEE Micro*, Vol. 16, No. 4, August 1996, pp. 6-9

[Lee96b] Ruby B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, Vol. 16, No .4, August 1996, pp. 51-59

[Lee97b] Ruby Lee, Larry McMahan, "Mapping of Application Software to the Multimedia Instructions of General Purpose Microprocessors," *IS&T/SPIE Symp. on Electric Imaging: Science and Technology*, San Jose, California, February 10-14, 1997, pp. 122-133

[Lee97c] Ruby B. Lee, "Multimedia Extensions for General Purpose Processors," *Proc. of the IEEE Workshop on VLSI Signal Processing*, Leicester, UK, November 3-5, 1997, pp. 1-15

[Luml97] John Lumley, "Packed Arithmetic - Architectural Influence on Compilation," *IEE Colloquium on Multimedia Instruction Sets: Design and Application*, Birmingham, United Kingdom, February 25, 1997

[MIPS99] MIPS Technologies, Inc., "MIPS Technologies Information Backgrounder," April 30, 1999, *http://www.mips.com/pressReleases/ MIPSISAback0430.pdf*, retrieved April 24, 2000

[Mitt97] Millind Mittal, Alex Peleg, Uri Weiser "MMX Technology Architecture Overview," *Intel Technology Journal*, Q3 1997, *http://developer.intel.com/technology/itj/q31997.htm*, retrieved April 24, 2000

[MIPS97] MIPS Technologies, Inc., "MIPS Extension for Digital Media with 3D," WhitePaper, March 12, 1997 *http://www.mips.com/Documentation/ isa5_tech_brf.pdf*, retrieved April 24, 2000

[Moto99] Motorola Inc., "Motorola's New MPC7400 Microprocessor Delivers Quantum Performance Leap for Embedded and Desktop Systems," August 31, 199 Press Release, *http://www.mot.com/SPS/PowerPC/library /press_releases/7400_pr.html*, retrieved April 24, 2000

[Ober99] Stuart Oberman, Greg Favor, Fred Weber, "AMD 3DNow! Technology: Architecture and Implementations," *IEEE Micro*, Vol. 19, No. 2, March-April, 1999, pp. 37-48

[Phil98] Mike Philip, "A Second Generation SIMD Microprocessor Architecture," *Proc.of Hot Chips 10*, Palo Alto, California, August 16-18, 1998, pp. 111-122

[Rice96] Daniel S. Rice, "High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set," *University of California at Berkeley, Master's Report*, March 19, 1996

[Rubi96] Paul Rubinfeld, Bob Rose, Michael McCallig, "Motion Video Instruction Extensions for Alpha," White Paper, October 18, 1996 *http://www.digital.com/alphaoem/papers/pmvi-abstract.htm*, retrieved April 24, 2000

[SGI96] Silicon Graphics Inc., "Silicon Graphics Introduces Enhanced MIPS Architecture to Lead the InteractiveDigital Revolution,"October 21, 1996 Press Release, *http://www.mips.com/pressReleases/100196B.html*, retrieved April 24, 2000

[Sling00a] Nathan T. Slingerland, Alan Jay Smith, "Design and Characterization of the Berkeley Multimedia Workload," *University of California at Berkeley Technical Report CSD-00-1122*, December 2000

[Sling00b] Nathan T. Slingerland, Alan Jay Smith, "Cache Performance for Multimedia Applications," *University of California at Berkeley Technical Report CSD-00-1123*, December 2000

[Sling00d] Nathan T. Slingerland, Alan Jay Smith, "Measuring the Performance of Multimedia Instruction Sets," *University of California at Berkeley Technical Report CSD-00-1125*, December 2000

[Smit78] Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies", *IEEE Computer*, Vol. 11, No. 12, December, 1978, pp. 7-21

[Thak99] Shreekant (Ticky) Thakkar, Tom Huff, "The Internet Streaming SIMD Extensions," *Intel Technology Journal*, Q2 1999, *http://developer.intel.com/technology/itj/q21999.htm*, retrieved April 24, 2000

[Thek99] Radhika Thekkath, Mike Uhler, Chandlee Harrell, Ying-wai Ho, "An Architecture Extension for Efficient Geometry Processing," *Proc. of Hot Chips 11*, Palo Alto, California, August 15-17, 1999, pp. 263-274

[Trem96] Marc Tremblay, J. Michael O'Connor, "UltraSPARC I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, Vol. 16, No. 2, April 1996, pp. 42-49

[Trem96b] Marc Tremblay, J.Michael O'Connor, Venkatesh Narayanan, Liang He, "VIS Speeds New Media Processing," *IEEE Micro*, Vol. 16, No. 4, August 1996, pp. 10-20

[Tse98] J. Tse, A. J. Smith, "CPU cache prefetching: Timing evaluation of hardware implementations," *IEEE Trans. on Computers*, Vol. 47, No.5, May 1998. pp. 509-26

[Veit98] Martin Veitch, "Intel Details Katmai Chip Plan," ZDNet Article, January 15, 1998, *http://www.zdnet.co.uk/news/news1/ns-3539.html*, retrieved April 24, 2000

1 Appendix

1.1 History of Multimedia Processing

1.1.1 Digital Signal Processors (DSPs)

First introduced in 1980, digital signal processors (DSPs) are special purpose microprocessors that have architectures and instruction sets specifically designed for real time signal processing algorithms. A typical DSP has a Harvard memory architecture, supports modulo addressing modes for speeding algorithms like the Fast Fourier transform (FFT), and has instructions such as a pipelined multiply-accumulate - an operation which is at the heart of digital filters. DSPs have been developed mainly for speech processing and communications processing in modems, pagers and cellular phones [Kuro98].

1.1.2 Application Specific Integrated Circuits (ASICs)

A function or application specific integrated circuit (ASIC) is strictly tied to a single well defined algorithm. This is why an ASIC is able to meet demanding processing requirements at minimal cost. Typically, a multimedia algorithm is broken down into subtasks until each subtask is of a complexity that can be assigned to a hardware block. An example of such a device would be an MPEG video decoder chip for set top boxes. ASICs usually offer very little if any programmability, but some ASICs embed a RISC processor for control and to be able to have a small degree of flexibility [Pirs97].

1.1.3 Multimedia Co-Processors

Multimedia processors are built from the ground up to be programmable devices, and therefore have an adaptability advantage over ASIC solutions. They can replace several ASICs, in addition to being able to adapt to changing or emerging standards. Architecturally, multimedia processors are an extension of DSPs, but utilize VLIW (very long instruction word) techniques [Pirs97]. Like superscalar microprocessors, VLIW processors attempt to exploit instruction parallelism through multiple execution units (e.g. adders, multipliers). However, unlike superscalar architectures which extract instruction level parallelism at run time with dedicated hardware (e.g. reservation stations, reorder buffers) to resolve conflicts and pipeline hazards, VLIW architectures rely entirely on the compiler to generate efficient, sane code at compile time. This approach conserves die area and the hardware is easier to implement than superscalar microprocessors which are notoriously difficult to design and debug [Pirs97].

In VLIW multimedia processors, one instruction controls several functional units, with special purpose functional blocks used to achieve high multimedia performance at lower clock frequencies. Like many DSPs, a high speed memory interface to memory is used instead of the cache based hierarchy found in microprocessors. Virtual memory support is also typically foregone, enabling programmers to make hard guarantees as to the latency and throughput of their code.

An important drawback of the VLIW approach is that there are many situations where the compiler cannot take advantage of parallel functional units because of limited information available at compile time [Hunt95]. VLIW coding also has the discouraging side effect of low code density because if the compiler can not supply a full quota of instructions for execution in a given clock cycle, the remaining positions in the word must be filled in with no-ops [Gepp98]. This means that an excessive amount of memory is often needed to store the code for a VLIW application.

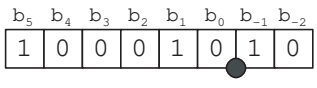
Multimedia processors have been introduced as accelerator boards for desktop personal computers, but are vanishing from personal computers due to the increased multimedia performance of microprocessors. The accelerator board approach inherently suffers from a cost disadvantage and tends to be less compatible because many such competing boards are typically available.

1.2 Numerical Methods

Before delving into the details of implementing multimedia algorithms on processors with supporting instructions, it will be useful to briefly review numerical representation on digital computers. These concepts are easily overlooked or forgotten when programming in a high level language like C or Java, but are of central importance to a correct implementation in assembly language.

1.2.1 Integer Values

The meaning assigned to any of the 2^N states of an N-bit binary word depends entirely on its interpretation. Common binary representations include unsigned integers, unsigned fixed-point rationals, signed two's complement integers and signed two's complement fixed-point rationals. Rational numbers are those numbers expressible as a/b where a and b are both members of the set of integers, Z. Like decimal (base 10) representations of rational numbers, binary (base 2) representations assign a weighted power of the respective base to each position of a number. Although we do not usually think of a decimal number as having a limited number of positions, binary numbers on computers necessarily have a limited width. Consider an unsigned 8-bit binary fixed point number format, which we will designate $U(6.2)$:



As is conventional with decimal numbers, the above binary number has been written with its most significant bits to the left, and its least significant bits to right. The value of an N-bit $U(a.b)$ format fixed point number, x, is given by:

$$x = \left(\frac{1}{2}\right)^b \sum_{n=0}^{N-1} 2^n x_n \qquad (1)$$

The variable, a, is the number of significant bits to the left of the binary point in an N-bit value, and b is the number of

bits to the right. Like decimal (base 10) arithmetic, binary (base 2) arithmetic contains an implicit binary point indicating the boundary between positions representing powers of the base greater than zero and those signifying powers of the base less than zero. Unsigned integer (or "natural binary") representations are a special case of $U(a.b)$ format fixed point numbers where $b = 0$. Each bit, b_k, has a weight of 2^k, so the value of the above example binary number is 34.5 (in decimal).

Two's complement is a method for representing signed numbers which simplifies the underlying hardware implementation on digital computers. The two's complement of a binary number, x, is given by taking the *one's complement* (negating all of the bits) and adding one. We will denote signed two's complement format $S(a.b)$, where $a = N - b - 1$ and b is the number bits to the right of the binary point. The value of an N-bit $S(a.b)$ format number, x, is given by:

$$x = \left(\frac{1}{2}\right)^b \left[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n\right] \qquad (2)$$

We will use the notation $X(a.b)$ to note when a rule is applicable to either $U(a.b)$ or $S(a.b)$ format numbers. Fixed point arithmetic has the following fundamental rules [Yates]:

1. Unsigned Wordlength - the number of bits required to represent $U(a.b)$ is $a + b$
2. Signed Wordlength - the number of bits required to represent $S(a.b)$ is $a + b + 1$
3. Unsigned Range - the range of a $U(a.b)$ fixed-point number is $0 \leq x \leq 2^a - 2^{-b}$
4. Signed Range - the range of an $S(a.b)$ fixed-point number is $-2^a \leq x \leq 2^a - 2^{-b}$
5. Addition Operands - the binary points of two numbers must be aligned in order for addition or subtraction to be performed. $X(c.d) + X(e.f)$ is only valid if $c = e$, and $d = f$.
6. Addition Result - the sum of two N-bit binary numbers requires $N + 1$ bits
7. Unsigned Multiplication - $U(a_1.b_1) \times U(a_2.b_2) = U(a_1 + a_2.b_1 + b_2)$
8. Signed Multiplication - $S(a_1.b_1) \times S(a_2.b_2) = S(a_1 + a_2 + 1.b_1 + b_2)$
9. Shifting- a shift can either be considered a scaling operation, moving an entire binary value along with its binary point:

$$X(a.b) \gg n = X(a + n.b - n)$$
$$X(a.b) \ll n = X(a - n.b + n)$$

or a multiplication/division by a power of two:

$$X(a.b) \gg n = X(a - n.b + n)$$
$$X(a.b) \ll n = X(a + n, b - n)$$

1.2.2 Floating Point

Floating point representations offer greater dynamic range in the same number of bits as natural binary or fixed point representations. This is accomplished by using a format similar to scientific notation in decimal. Bits of precision are traded to extended the range of representable values. The IEEE 754 floating point standard is used by almost all modern floating point hardware (the notable exception being x86 based machines which internally use an 80-bit format, but are able to convert to and from standard IEEE 754 representations). Two sets of formats are defined by the IEEE standard; the first set consisting of the basic formats (32-bit single precision and 64-bit double precision), are defined in Figure 1. IEEE 754 also defines an extended set of formats which assume that a machine will devote an entire word (32 or 64 bits) to the mantissa, and anther 10 or 14 bits to the exponent for the extended single precision and double precision formats, respectively.

The IEEE standard assigns the largest and smallest numbers supported by the standard to be $\pm 3.4 \times 10^{38}$, and $\pm 1.2 \times 10^{-38}$ respectively for single precision, leaving some bit patterns free for special values:

1. ± 0 - all of the mantissa bits and exponent bits being 0s
2. $\pm\infty$ - all of the mantissa bits 0s and all of the exponent bits 1s
3. NaN - not a number
4. group of small unnormalized numbers $\pm 1.2 \times 10^{-38}$ to $\pm 1.4 \times 10^{-45}$

Double precision IEEE floating point extends single precision by adding extra bits to the exponent and mantissa, extending the range to: $\pm 1.8 \times 10^{+308}$ to $\pm 2.2 \times 10^{-308}$.

1.2.3 Ordered and Unorderd

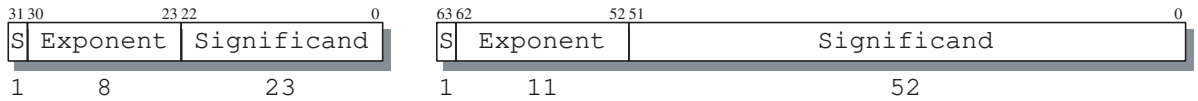
A comparison is said to be ordered if one can define the relations $<$ (less-than), $>$ (greater-than), and $=$ (equal-to) so that they have the following properties.

Trichotomy For all x and y, exactly one of $x < y$, $x > y$, or $x = y$, is true.

Reflexive For all x and y, if $x < y$, then $y > x$, and if $x = y$, then $y = x$.

Transitive. For all x, y, and z, if $x < y$ and $y < z$ then $x < z$. Also, if $x = y$ and $y = z$ then $x = z$.

Relations in the IEEE-754 floating point standard need a way to express comparisons with NaNs (for Not-a-Number). Since NaNs have no value, the result of comparing them with normal numbers is unordered, which is symbolically specified as "?". That is, a NaN is not greater-than, nor less-than, nor equal-to any number. So, the best one can say about any two numbers in the IEEE-754 system is that either $x < y$, $x > y$, $x = y$, or $x!?y$ (read as x is unordered with respect to y). [Zuras]



(a) Single Precision

(b) Double Precision

Figure 1: **IEEE 754 Floating Point Formats**

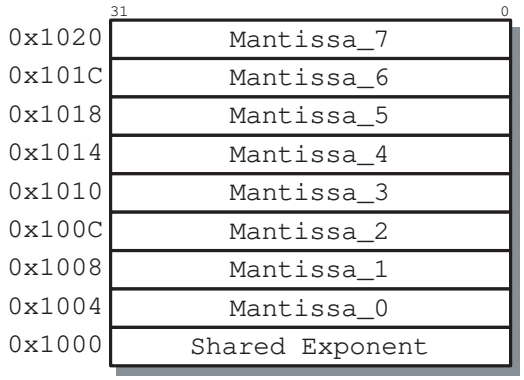


Figure 2: **Example Block Floating Point Memory Map**

1.2.4 Block Floating Point

Block floating point is a scaling technique in which a single exponent is used for all of the elements in a block of data. This originated as a software technique (as did floating point), but some digital signal processors (DSPs) support this mode directly in hardware. It has the advantage of being less expensive in terms of hardware than floating point, as well as being faster.

To visualize how this works, consider the memory map shown in Figure 2, which diagrams how eight block floating point numbers and their shared exponent might be laid out in memory. Floating point numbers have a greater dynamic range because the distance between numbers gets larger as the magnitude of the numbers get larger. The disadvantage of block floating point when compared to full IEEE 754 floating point is that all numbers must share the greatest exponent of all the actual values. The mantissas must be scaled to match this shared exponent, but because there is a finite number of bits in the mantissa, precision can be lost. Block floating point only works well if a block of data can have a wide possible range of values, but values are clustered for a particular computation.

1.3 Programming Methods

In programming with the fixed and floating point formats we have discussed, there are many heuristics for performing operations quickly in assembly language. Additionally, if a needed operation is required by an algorithm but unsupported directly by the instruction set in question it is often possible to synthesize the needed operation. The only costs are a possibly larger number of instructions and potentially greater register pressure and execution dependencies. Here we present those equivalencies which were of utility in our programming.

1.3.1 Synthesizing Width Promotion

Width promotion is the expansion of an n-bit value to some larger width. For unsigned fixed point numbers in the $U(a.b)$ format discussed previously, this involves zero extension or filling any additional bits with zeros. Zero extension is usually not specified as such in a multimedia architecture because it overlaps in functionality with data communication instructions such as unpack or merge. These operations interleave source elements from two registers. If a SIMD register is merged (interleaved) with another register which has been zeroed, the result will be a zero extended version of the upper or lower elements of the first register, depending on if a merge-hi (most significant half of the register) or merge-lo (least significant half of the register) is used. Signed element unpacking is not as simple, but is rarely supported directly by hardware; only the MIPS' MDMX and AltiVec instruction sets include it. It can, however, be synthesized with multiplication by '1' since width of the result of multiplication is the sum of the width of both its operands.

1.3.2 Synthesizing Max/Min

Signed minimum and maximum operations are often used with a constant second operand to saturate results to arbitrary ranges. This operation can be simulated with packed signed saturating addition. As was discussed earlier, the representable range of an $S(a.b)$ number is $-2^a \leq x \leq 2^a - 2^{-b}$. For example, assume we want to limit an $S(a.b)$ result, X, to $-j..+k$. The maximum representable value is $M = 2^a - 2^{-b}$.

1. $T_{pos} = (2^a - 2^b) - k$
2. $X + T_{pos} \Longrightarrow X$
3. $X - T_{pos} \longrightarrow X$

These three steps limit X to $+K$. ($\mathrm{A} \Longrightarrow$ represents saturating overflow, where as a $\longrightarrow$ symbolizes modulo overflow.) Three more operations are required to limit X to the desired floor value:

1. $T_{neg} = -2^a + j$
2. $X + T_{neg} \Longrightarrow X$
3. $X - T_{neg} \longrightarrow X$

1.3.3 Synthesizing Multiply/Shift

Multiplication of fixed-point values by integer and fractional constants can be simulated with a right/left shift and add operation. This is in fact the only type of partitioned (and integer) multiplication operation supported in hardware by Hewlett-Packard's PA-RISC architecture. Consider packed 16-bit values multiplied by the constant:

$$\sqrt{2} = 1.41421356 \cong 1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} = 1.4140625$$

With the MAX-2 instruction set, such a multiplication is accomplished as listed in Algorithm 1.

Algorithm 1 Partitioned Multiplication by a Constant with MAX-2

```
hshradd    r10, 2, r10, r8      ; r8:  x[0..3]*(1+1/4)
hshradd    r8, 3, r10, r8       ; r8:  x[0..3]*(1+1/8+1/32)
hshradd    r10, 1, r8, r8       ; r8:  x[0..3]*(1+1/2+1/8+1/32)
hshradd    r8, 2, r10, r2       ; r2:  x[0..3]*(1+1/4+1/8+1/32+1/128)
```

Likewise, shifts can be simulated with multiplication and division. None of the multimedia extensions offer partitioned integer divide instructions, so only left shifts can be simulated in this way. It is also sometimes useful to keep in mind that a multiplication by a power of 2 on a platform without left shift operations can be decomposed into a series of additions. If the desired power of two is small this can be a performance win if other operations can be done at the same time to combat the added data dependencies. We found this useful on Sun's VIS when bit-wise left shifts were needed.

1.3.4 Synthesizing Absolute Value

The absolute value of an $S(a.b)$integer, X, can be synthesized as follows:

1. $X \longrightarrow X_{pos}$, $0 - X \longrightarrow X_{neg}$
2. $max(X_{neg}, X_{pos}) \longrightarrow X_{pos}$

Floating point absolute value can be performed by AND'ing a single precision value with `0x7FFFFFFF`. This clears the sign bit.

1.3.5 Synthesizing Floating Point Sign Negation

Floating point negation can be performed by XOR'ing a single precision value with `0x80000000`. This clears the sign bit if it was set or sets it if it was cleared.

1.3.6 Newton-Raphson Method

The Newton-Raphson formula for finding the root of an equation is defined:

$$x_{i+1} = x_i - f(x_i)/f'(x_i) \tag{3}$$

Utilizing the above method, it is possible to generate equations for increasing the precision of approximations to $1/a$:

$$x_1 = x_0 - (a \cdot x_0^2 - x_0) \tag{4}$$

as well as 1/sqrt(a):

$$x_1 = x_0 - (0.5 \cdot a \cdot x_0^3 - 0.5 \cdot x_0) = 0.5 \cdot x_0 \cdot (3.0 - a \cdot x_0^2) \tag{5}$$

Both the reciprocal and reciprocal operations are common in floating point 3D geometry routines. In the above equations, x_0 represents the first approximation, and x_1 is the iteratively higher precision result. This method approximately doubles the number of significant digits for each iteration if the initial guess is close.[IAP803]

1.3.7 Synthesizing Square Root

Because many floating point instruction sets include approximations for $\frac{1}{\sqrt{a}}$, it is helpful to recognize that if a true square-root value is what is actually needed, the following equivalency can be used:

$$\sqrt{a} = a \cdot \frac{1}{\sqrt{a}} \tag{6}$$

Although all of the SIMD floating point instruction sets included reciprocal operations, they are typically approximations. The above prevents the loss of any precision, other than any original loss due to approximating the square root.

1.4 Instruction Set Descriptions

AMD 3DNow!	Functionality
FEMMS	clear floating point state
PAVGUSB	$F_{U8} \Lambda F_{U8} \rightarrow F_{U8}$ $F_{U8} \Lambda mem_{U8} \rightarrow F_{U8}$
PF2ID	$\varnothing(F_{FP32}) \rightarrow F_{S32}$ $\varnothing(mem_{FP32}) \rightarrow F_{S32}$
PFACC	$\Re_{+}(F_{FP32}) \| \Re_{+}(F_{FP32}) \rightarrow F_{FP32}$ $\Re_{+}(F_{FP32}) \| \Re_{+}(mem_{FP32}) \rightarrow F_{FP32}$
PFADD	$F_{FP32}+F_{FP32} \rightarrow F_{FP32}$ $F_{FP32}+mem_{FP32} \rightarrow F_{FP32}$
PFCMPEQ	$m(F_{FP32}==F_{FP32}) \rightarrow F_{32}$ $m(F_{FP32}==mem_{FP32}) \rightarrow F_{32}$
PFCMPGE	$m(F_{FP32} \geq F_{FP32}) \rightarrow F_{32}$ $m(F_{FP32} \geq mem_{FP32}) \rightarrow F_{32}$
PFCMPGT	$m(F_{FP32} > F_{FP32}) \rightarrow F_{32}$ $m(F_{FP32} > mem_{FP32}) \rightarrow F_{32}$
PFMAX	$\lceil F_{FP32},F_{FP32} \rceil \rightarrow F_{FP32}$ $\lceil F_{FP32},mem_{FP32} \rceil \rightarrow F_{FP32}$
PFMIN	$\lfloor F_{FP32},F_{FP32} \rfloor \rightarrow F_{FP32}$ $\lfloor F_{FP32},mem_{FP32} \rfloor \rightarrow F_{FP32}$
PFMUL	$F_{FP32} \times F_{FP32} \rightarrow F_{FP32}$ $F_{FP32} \times mem_{FP32} \rightarrow F_{FP32}$
PFRCP	$\approx_{14} 1/F_{\bullet FP32} \rightarrow F_{\bullet FP32}$ $\approx_{14} 1/mem_{\bullet FP32} \rightarrow F_{\bullet FP32}$
PFRCPIT1	$\approx_{XX}^{1} 1/F_{\bullet FP32} \rightarrow F_{\bullet FP32}$ $\approx_{XX}^{1} 1/mem_{\bullet FP32} \rightarrow F_{\bullet FP32}$
PFRCPIT2	$\approx_{24}^{2} 1/\sqrt{F_{\bullet FP32}} \rightarrow F_{\bullet FP32}$ $\approx_{24}^{2} /F_{\bullet FP32} \rightarrow F_{\bullet FP32}$ $\approx_{24}^{2} 1/\sqrt{mem_{\bullet FP32}} \rightarrow F_{\bullet FP32}$ $\approx_{24}^{2} /mem_{\bullet FP32} \rightarrow F_{\bullet FP32}$
PFRSQIT1	$\approx_{XX}^{1} 1/\sqrt{F_{\bullet FP32}} \rightarrow F_{\bullet FP32}$ $\approx_{XX}^{1} 1/\sqrt{mem_{\bullet FP32}} \rightarrow F_{\bullet FP32}$
PFRSQRT	$\approx_{15} 1/\sqrt{F_{\bullet FP32}} \rightarrow F_{\bullet FP32}$ $\approx_{15} 1/\sqrt{mem_{\bullet FP32}} \rightarrow F_{\bullet FP32}$
PFSUB	$F_{FP32}-F_{FP32} \rightarrow F_{FP32}$ $F_{FP32}-mem_{FP32} \rightarrow F_{FP32}$
PFSUBR	$F_{FP32}-F_{FP32} \rightarrow F_{FP32}$ $F_{FP32}-mem_{FP32} \rightarrow F_{FP32}$
PI2FD	$F_{S32} \rightarrow F_{FP32}$ $F_{S32} \rightarrow mem_{FP32}$
PMULHRW	$\{U_{16}(F_{S16} \times F_{S16}) \rightarrow F_{S16}$ $\{U_{16}(F_{S16} \times mem_{S16}) \rightarrow F_{S16}$
PREFETCH	prefetch into L1
PREFETCHW	prefetch for write into L1

Source: 3DNow! Technology Manual 21928E/0 – November 1998

AMD Extension to 3DNow! and MMX	Functionality
PF2IW	$\varnothing(F_{FP32}) \Rightarrow F_{S16} \rightarrow F_{S32}$ $\varnothing(mem_{FP32}) \Rightarrow F_{S16} \rightarrow F_{S32}$
PFNACC	$\Re_{+}(F_{FP32}) \| \Re_{+}(F_{FP32}) \rightarrow F_{FP32}$ $\Re_{+}(F_{FP32}) \| \Re_{+}(mem_{FP32}) \rightarrow F_{FP32}$
PFPNACC	$\Re_{+}(F_{FP32}) \| \Re_{-}(F_{FP32}) \rightarrow F_{FP32}$ $\Re_{+}(F_{FP32}) \| \Re_{-}(mem_{FP32}) \rightarrow F_{FP32}$
PI2FW	$E(F_{S16}) \rightarrow F_{FP32}$ $E(mem_{S16}) \rightarrow F_{FP32}$
PSWAPD	$\aleph F_{FP32} \rightarrow F_{FP32}$ $\aleph mem_{FP32} \rightarrow F_{FP32}$
MASKMOVQ	$if(F_{[(i*8)-1]})\ F_8[i] \rightarrow mem_8[i]$
MOVNTQ	F→mem (no write allocate)
PAVGB	$F_{U8} \Lambda F_{U8} \rightarrow F_{U8}$ $F_{U8} \Lambda mem_{U8} \rightarrow F_{U8}$
PAVGW	$F_{U16} \Lambda F_{U16} \rightarrow F_{U16}$ $F_{U16} \Lambda mem_{U16} \rightarrow F_{U16}$
PEXTRW	$F_{16}[imm_8] \rightarrow R_{[15..0]}$
PINSRW	$R_{[0..15]} \rightarrow F_{16}[I_8]$ $mem \rightarrow F_{16}[I_8]$
PMAXSW	$\lceil F_{S16},F_{S16} \rceil \rightarrow F_{S16}$ $\lceil F_{S16},mem_{S16} \rceil \rightarrow F_{S16}$
PMAXUB	$\lceil F_{U8}, F_{U8} \rceil \rightarrow F_{U8}$ $\lceil F_{U8}, mem_{U8} \rceil \rightarrow F_{U8}$
PMINSW	$\lfloor F_{S16},F_{S16} \rfloor \rightarrow F_{S16}$ $\lfloor F_{S16},mem_{S16} \rfloor \rightarrow F_{S16}$
PMINUB	$\lfloor F_{U8}, F_{U8} \rfloor \rightarrow F_{U8}$ $\lfloor F_{U8}, mem_{U8} \rfloor \rightarrow F_{U8}$
PMOVMSKB	$F_{[(i*8)-1]} \rightarrow R_{[i]}$
PMULHUW	$U_{16}(F_{U16} \times F_{U16}) \rightarrow F_{U16}$ $U_{16}(F_{U16} \times mem_{U16}) \rightarrow F_{U16}$
PREFETCHNTA	prefetch with minimal L1/L2 pollution
PREFETCHT0	prefetch to all cache levels
PREFETCHT1	prefetch to all but L1 cache level
PFEFETCHT2	prefetch to all but L1 and L2 cache levels
PSADBW	$\Re_{+}(F_{U8} \nabla F_{U8}) \rightarrow F_{\bullet U16}$ $\Re_{+}(F_{U8} \nabla mem_{U8}) \rightarrow F_{\bullet U16}$
PSHUFW	$F_{16}[imm_{8[2*i+1..2*i]}] \rightarrow F_{16}[i]$ $mem_{16}[imm_{8[2*i+1..2*i]}] \rightarrow F_{16}[i]$
SFENCE	force completion of weakly ordered stores

Source: AMD Extensions to the 3DNow! and MMX Instruction Sets Manual – 22466B/0 – August 1999

Cyrix Extended MMX	Functionality
PADDISW	$F_{S16}+F_{S16} \Rightarrow \partial F_{S16}$ $F_{S16}+mem_{S16} \Rightarrow \partial F_{S16}$
PAVEB	$F_{U8} \Lambda F_{U8} \rightarrow F_{U8}$ $F_{U8} \Lambda mem_{U8} \rightarrow F_{U8}$
PDISTIB	$F_{U8} \nabla mem_{U8} \rightarrow \Sigma \partial F_{U8}$
PMACHRIW	$U_{16}(\{(F_{S16} \times mem_{S16})) \rightarrow \Sigma \partial F_{S16}$
PMAGW	$\lceil \|F_{S16}\|,\|F_{S16}\| \rceil \rightarrow F_{S16}$ $\lceil \|F_{S16}\|,\|mem_{S16}\| \rceil \rightarrow F_{S16}$
PMULHRW	$U_{16}(\{(F_{S16} \times F_{S16})) \rightarrow F_{S16}$ $U_{16}(\{(F_{S16} \times mem_{S16})) \rightarrow F_{S16}$
PMULHRIW	$U_{16}(\{(F_{S16} \times F_{S16})) \rightarrow \partial F_{S16}$ $U_{16}(\{(F_{S16} \times mem_{S16})) \rightarrow \partial F_{S16}$
PMVZB	$if(\partial F_8[i]==0),\ mem_8 \rightarrow F_8$
PMVNZB	$if(\partial F_8[i]!=0),\ mem_8 \rightarrow F_8$
PMVLZB	$if(\partial F_{S8}[i]<0),\ mem_8 \rightarrow F_8$
PMVGEZB	$if(\partial F_{S8}[i] \geq 0),\ mem_8 \rightarrow F_8$
PSUBSIW	$F_{S16}-F_{S16} \Rightarrow \partial F_{S16}$ $F_{S16}-mem_{S16} \Rightarrow \partial F_{S16}$

Source: Application Note 108 – Cyrix Extended MMX Instruction Set

DEC MVI	Functionality
MAXSB8	$\lceil R_{S8},R_{S8} \rceil \rightarrow R_{S8}$
MAXSW4	$\lceil R_{S16},R_{S16} \rceil \rightarrow R_{S16}$
MAXUB8	$\lceil R_{U8},R_{U8} \rceil \rightarrow R_{U8}$
MAXUW4	$\lceil R_{U16},R_{U16} \rceil \rightarrow R_{U16}$
MINSB8	$\lfloor R_{S8},R_{S8} \rfloor \rightarrow R_{S8}$
MINSW4	$\lfloor R_{S16},R_{S16} \rfloor \rightarrow R_{S16}$
MINUB8	$\lfloor R_{U8},R_{U8} \rfloor \rightarrow R_{U8}$
MINUW4	$\lfloor R_{U16},R_{U16} \rfloor \rightarrow R_{U16}$
PERR	$\Re_{+}(R_{U8} \nabla R_{U8}) \rightarrow R_{\bullet U64}$
PKLB	$0 \| R_{32} \rightarrow R_8$
PKWB	$0 \| R_{16} \rightarrow R_8$
UNPKBW	$R_{U8}[3..0] \rightarrow R_{U16}$
UNPKBL	$R_{U8}[1..0] \rightarrow R_{U32}$

Source: James Hicks, Richard Weiss, "Motion Video Instructions", February 1999

HP MAX-1	Functionality
HADD	$R_{16}+R_{16}\rightarrow R_{16}$
HADD,ss	$R_{S16}+R_{S16}\Rightarrow R_{S16}$
HADD,us	$R_{U16}+R_{U16}\Rightarrow R_{U16}$
HAVG	$R_{S16}\Lambda R_{S16}\Rightarrow R_{S16}$
HSUB	$R_{16}-R_{16}\rightarrow R_{16}$
HSUB,ss	$R_{S16}-R_{S16}\Rightarrow R_{S16}$
HSUB,us	$R_{U16}-R_{U16}\Rightarrow R_{U16}$
HSLADD	$(R_{S16}<<imm_2)+R_{S16}\Rightarrow R_{S16}$
HSHRADD	$(R_{S16}\underline{>>}imm_2)+R_{S16}\Rightarrow R_{S16}$

Source: Ruby B. Lee, "Subword Parallelism with MAX-2", IEEE Micro, August 1996, pg. 51-59

HP MAX-2	Functionality
HSHR	$R_{S16}\underline{>>}imm_2\rightarrow R_{S16}$
HSHR,u	$R_{U16}>>imm_2\rightarrow R_{U16}$
HSHL	$R_{16}<<imm_2\rightarrow R_{16}$
MIXH,L	$U(R_{16})\wedge\vee U(R_{16})\rightarrow R$
MIXH,R	$L(R_{16})\wedge\vee L(R_{16})\rightarrow R$
MIXW,L	$U(R_{32})\wedge\vee U(R_{32})\rightarrow R$
MIXW,R	$L(R_{32})\wedge\vee L(R_{32})\rightarrow R$
PERMH	$R_{16}[imm_{8[2*i+1..2*i]}]\rightarrow R_{16}[i]$

Source: Ruby B. Lee, "Subword Parallelism with MAX-2", IEEE Micro, August 1996, pg. 51-59

Intel MMX	Functionality
EMMS	clear floating point state
MOVD	$F_{[0..31]}\rightarrow R_{[0..31]}$ $F_{[0..31]}\rightarrow$mem $R_{[0..31]}\rightarrow F_{[0..31]}$ mem$\rightarrow F_{[0..31]}$
MOVQ	$F_{[0..63]}\rightarrow F_{[0..63]}$ $F_{[0..63]}\rightarrow$mem mem$\rightarrow F_{[0..63]}$
PACKSSDW	$F_{S32}\|F_{S32}\Rightarrow F_{S16}$ $F_{S32}\|mem_{S32}\Rightarrow F_{S16}$
PACKSSWB	$F_{S16}\|F_{S16}\Rightarrow F_{S8}$ $F_{S16}\|mem_{S16}\Rightarrow F_{S8}$
PACKUSWB	$F_{S16}\|F_{S16}\Rightarrow F_{U8}$ $F_{S16}\|mem_{S16}\Rightarrow F_{U8}$
PADDB	$F_8+F_8\rightarrow F_8$ $F_8+mem_8\rightarrow F_8$
PADDD	$F_{32}+F_{32}\rightarrow F_{32}$ $F_{32}+mem_{32}\rightarrow F_{32}$
PADDSB	$F_{S8}+F_{S8}\Rightarrow F_{S8}$ $F_{S8}+mem_{S8}\Rightarrow F_{S8}$
PADDSW	$F_{S16}+F_{S16}\Rightarrow F_{S16}$ $F_{S16}+mem_{S16}\Rightarrow F_{S16}$
PADDUSB	$F_{U8}+F_{U8}\Rightarrow F_{U8}$ $F_{U8}+mem_{U8}\Rightarrow F_{U8}$
PADDUSW	$F_{U16}+F_{U16}\Rightarrow F_{U16}$ $F_{U16}+mem_{U16}\Rightarrow F_{U16}$
PADDW	$F_{16}+F_{16}\rightarrow F_{16}$ $F_{16}+mem_{16}\rightarrow F_{16}$
PAND	F&F$\rightarrow$F F&mem$\rightarrow$F
PANDN	~(F)&F$\rightarrow$F ~(F)&mem$\rightarrow$F
PCMPEQB	$m(F_8==F_8)\rightarrow F$ $m(F_8==mem_8)\rightarrow F$
PCMPEQD	$m(F_{16}==F_{16})\rightarrow F_{16}$ $m(F_{16}==mem_{16})\rightarrow F_{16}$
PCMPEQW	$m(F_{32}==F_{32})\rightarrow F_{32}$ $m(F_{32}==mem_{32})\rightarrow F_{32}$
PCMPGTB	$m(F_{S8}>F_{S8})\rightarrow F_8$ $m(F_{S8}>mem_{S8})\rightarrow F_8$
PCMPGTD	$m(F_{S16}>F_{S16})\rightarrow F_{16}$ $m(F_{S16}>mem_{S16})\rightarrow F_{16}$
PCMPGTW	$m(F_{S32}>F_{S32})\rightarrow F_{32}$ $m(F_{S32}>mem_{S32})\rightarrow F_{32}$
PMADDWD	$\Re_+(F_{S16}\times F_{S16})\rightarrow F_{S32}$ $\Re_+(F_{S16}\times mem_{S16})\rightarrow F_{S32}$
PMULHW	$U_{16}(F_{S16}\times F_{S16})\rightarrow F_{S16}$ $U_{16}(F_{S16}\times mem_{S16})\rightarrow F_{S16}$
PMULLW	$L_{16}(F_{S16}\times F_{S16})\rightarrow F_{16}$ $L_{16}(F_{S16}\times mem_{S16})\rightarrow F_{16}$
POR	F|F$\rightarrow$F F|mem$\rightarrow$F
PSLLD	$F_{32}<<F_{\bullet 64}\rightarrow F_{32}$ $F_{32}<<mem_{\bullet 64}\rightarrow F_{32}$ $F_{32}<<imm_8\rightarrow F_{32}$
PSLLQ	$F_{\bullet 64}<<F_{\bullet 64}\rightarrow F_{\bullet 64}$ $F_{\bullet 64}<<mem_{\bullet 64}\rightarrow F_{\bullet 64}$ $F_{\bullet 64}<<imm_8\rightarrow F_{\bullet 64}$
PSLLW	$F_{16}<<F_{\bullet 64}\rightarrow F_{16}$ $F_{16}<<mem_{\bullet 64}\rightarrow F_{16}$ $F_{16}<<imm_8\rightarrow F_{16}$
PSRAD	$F_{S32}\underline{>>}F_{\bullet 64}\rightarrow F_{S32}$ $F_{S32}\underline{>>}mem_{\bullet 64}\rightarrow F_{S32}$ $F_{S32}\underline{>>}imm_8\rightarrow F_{S32}$
PSRAW	$F_{S16}\underline{>>}F_{\bullet 64}\rightarrow F_{S16}$ $F_{S16}\underline{>>}mem_{\bullet 64}\rightarrow F_{S16}$ $F_{S16}\underline{>>}imm_8\rightarrow F_{S16}$
PSRLD	$F_{32}>>F_{\bullet 64}\rightarrow F_{32}$ $F_{32}>>mem_{\bullet 64}\rightarrow F_{32}$ $F_{32}>>imm_8\rightarrow F_{32}$
PSRLQ	$F_{\bullet 64}>>F_{\bullet 64}\rightarrow F_{\bullet 64}$ $F_{\bullet 64}>>mem_{\bullet 64}\rightarrow F_{\bullet 64}$ $F_{\bullet 64}>>imm_8\rightarrow F_{\bullet 64}$
PSRLW	$F_{16}>>F_{\bullet 64}\rightarrow F_{16}$ $F_{16}>>mem_{\bullet 64}\rightarrow F_{16}$ $F_{16}>>imm_8\rightarrow F_{16}$
PSUBB	$F_8-F_8\rightarrow F_8$ $F_8-mem_8\rightarrow F_8$
PSUBD	$F_{32}-F_{32}\rightarrow F_{32}$ $F_{32}-mem_{32}\rightarrow F_{32}$
PSUBSB	$F_{S8}-F_{S8}\Rightarrow F_{S8}$ $F_{S8}-mem_{S8}\Rightarrow F_{S8}$
PSUBSW	$F_{S16}-F_{S16}\Rightarrow F_{S16}$ $F_{S16}-mem_{S16}\Rightarrow F_{S16}$
PSUBUSB	$F_{U8}-F_{U8}\Rightarrow F_{U8}$ $F_{U8}-mem_{U8}\Rightarrow F_{U8}$
PSUBUSW	$F_{U16}-F_{U16}\Rightarrow F_{U16}$ $F_{U16}-mem_{U16}\Rightarrow F_{U16}$
PSUBW	$F_{16}-F_{16}\rightarrow F_{16}$ $F_{16}-mem_{16}\rightarrow F_{16}$
PUNPCKHBW	$U(F_8)\wedge\vee U(F_8)\rightarrow F$ $U(F_8)\wedge\vee U(mem_8)\rightarrow F$
PUNPCKHDQ	$U(F_{32})\wedge\vee U(F_{32})\rightarrow F$ $U(F_{32})\wedge\vee U(mem_{32})\rightarrow F$
PUNPCKHWD	$U(F_{16})\wedge\vee U(F_{16})\rightarrow F$ $U(F_{16})\wedge\vee U(mem_{16})\rightarrow F$
PUNPCKLBW	$L(F_8)\wedge\vee L(F_8)\rightarrow F$ $L(F_8)\wedge\vee L(mem_8)\rightarrow F$
PUNPCKLDQ	$L(F_{32})\wedge\vee L(F_{32})\rightarrow F$ $L(F_{32})\wedge\vee L(mem_{32})\rightarrow F$
PUNPCKLWD	$L(F_{16})\wedge\vee L(F_{16})\rightarrow F$ $L(F_{16})\wedge\vee L(mem_{16})\rightarrow F$
PXOR	F$\oplus$F$\rightarrow$F F$\oplus$mem$\rightarrow$F

Source: AMD-K6 MMX Enhanced Processor Multimedia Technology Manual 1997 – 20726C/0 – June 1997

Intel SSE	Functionality
ADDPS	$V_{FP32}+V_{FP32}\rightarrow V_{FP32}$ $V_{FP32}+mem_{FP32}\rightarrow V_{FP32}$
ADDSS	$V_{\bullet FP32}+V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $V_{\bullet FP32}+mem_{\bullet FP32}\rightarrow V_{\bullet FP32}$
ANDNPS	~(V)&V→V ~(V)&mem→V
ANDPS	V&V→V V&mem→V
CMPPS	$m(V_{FP32}\{==, <, \leq, !?, !=, !<, !\leq, ?\}V_{FP32})\rightarrow V_{32}$ $m(V_{FP32}\{==, <, \leq, !?, !=, !<, !\leq, ?\}mem_{FP32})\rightarrow V_{32}$
CMPSS	$m(V_{\bullet FP32}\{==, <, \leq, !?, !=, !<, !\leq, ?\}V_{\bullet FP32})\rightarrow V_{\bullet 32}$ $m(V_{\bullet FP32}\{==, <, \leq, !?, !=, !<, !\leq, ?\}mem_{\bullet FP32})\rightarrow V_{\bullet 32}$
COMISS	$bv(V_{\bullet FP32}\{!?,>,<\}V_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags $bv(V_{\bullet FP32}\{!?,>,<\}mem_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags
CVTPI2PS	$U_{64}(V_{FP32})\|F_{S32}\rightarrow V_{FP32}$ $U_{64}(V_{FP32})\|mem_{S32}\rightarrow V_{FP32}$
CVTPS2PI	$L_{64}(V_{FP32})\rightarrow F_{S32}$ $L_{64}(mem_{FP32})\rightarrow F_{S32}$
CVTSI2SS	$U_{96}(V_{FP32})\|R_{[32..0]}\rightarrow V_{FP32}$ $U_{96}(V_{FP32})\|mem_{[32..0]}\rightarrow V_{FP32}$
CVTSS2SI	$V_{\bullet FP32}\rightarrow R_{[32..0]}$ $mem_{\bullet FP32}\rightarrow R_{[32..0]}$
CVTTPS2PI	$L_{64}(V_{FP32})\rightarrow F_{S32}$ $L_{64}(mem_{FP32})\rightarrow F_{S32}$
CVTTSS2SI	$V_{\bullet FP32}\rightarrow R_{[32..0]}$ $mem_{\bullet FP32}\rightarrow R_{[32..0]}$
DIVPS	$V_{FP32}\div V_{FP32}\rightarrow V_{FP32}$ $V_{FP32}\div mem_{FP32}\rightarrow V_{FP32}$
DIVSS	$V_{\bullet FP32}\div V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $V_{\bullet FP32}\div mem_{\bullet FP32}\rightarrow V_{\bullet FP32}$
FXRSTOR	restore FP/MMX and SSE state
FXSAVE	save FP/MMX and SSE state
LDMXCSR	load SSE control/status word
MASKMOVQ	$if(F_{8[(i*8)-1]})\ F_8[i]\rightarrow mem_8[i]$
MAXPS	$\lceil V_{FP32}, V_{FP32}\rceil\rightarrow V_{FP32}$ $\lceil V_{FP32}, mem_{FP32}\rceil\rightarrow V_{FP32}$
MAXSS	$\lceil V_{\bullet FP32}, V_{\bullet FP32}\rceil\rightarrow V_{\bullet FP32}$ $\lceil V_{\bullet FP32}, mem_{\bullet FP32}\rceil\rightarrow V_{\bullet FP32}$
MINPS	$\lfloor V_{FP32}, V_{FP32}\rfloor\rightarrow V_{FP32}$ $\lfloor V_{FP32}, mem_{FP32}\rfloor\rightarrow V_{FP32}$
MINSS	$\lfloor V_{\bullet FP32}, V_{\bullet FP32}\rfloor\rightarrow V_{\bullet FP32}$ $\lfloor V_{\bullet FP32}, mem_{\bullet FP32}\rfloor\rightarrow V_{\bullet FP32}$
MOVAPS	V→V V→mem (exception if unaligned mem) mem→V (exception if unaligned mem)
MOVHLPS	$U_{64}(V)\|L_{64}(V)\rightarrow V$
MOVHPS	$U_{64}(V)\rightarrow mem$ $mem\rightarrow U_{64}(V)$
MOVLHPS	$L_{64}(V)\|L_{64}(V)\rightarrow V$
MOVLPS	$L_{64}(V)\rightarrow mem$ $mem\rightarrow L_{64}(V)$
MOVMSKPS	$V_{[(i*32)-1]}\rightarrow R_{[i]}$
MOVNTPS	V→mem (no write allocate)
MOVNTQ	F→mem (no write allocate)
MOVSS	$V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $V_{\bullet FP32}\rightarrow mem$ $mem\rightarrow V_{\bullet FP32}$
MOVUPS	V→V V→mem mem→V
MULPS	$V_{FP32}\times V_{FP32}\rightarrow V_{FP32}$ $V_{FP32}\times mem_{FP32}\rightarrow V_{FP32}$
MULSS	$V_{\bullet FP32}\times V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $V_{\bullet FP32}\times mem_{\bullet FP32}\rightarrow V_{\bullet FP32}$
ORPS	$V\|V\rightarrow V$ $V\|mem\rightarrow V$
PAVGB	$F_{U8}\Lambda F_{U8}\rightarrow F_{U8}$ $F_{U8}\Lambda mem_{U8}\rightarrow F_{U8}$
PAVGW	$F_{U16}\Lambda F_{U16}\rightarrow F_{U16}$ $F_{U16}\Lambda mem_{U16}\rightarrow F_{U16}$
PEXTRW	$F_{16}[I_8]\rightarrow R_{[15..0]}$
PINSRW	$R_{[0..15]}\rightarrow F_{16}[I_8]$ $mem\rightarrow F_{16}[I_8]$
PMAXSW	$\lceil F_{S16}, F_{S16}\rceil\rightarrow F_{S16}$ $\lceil F_{S16}, mem_{S16}\rceil\rightarrow F_{S16}$
PMAXUB	$\lceil F_{U8}, F_{U8}\rceil\rightarrow F_{U8}$ $\lceil F_{U8}, mem_{U8}\rceil\rightarrow F_{U8}$
PMINSW	$\lfloor F_{S16}, F_{S16}\rfloor\rightarrow F_{S16}$ $\lfloor F_{S16}, mem_{S16}\rfloor\rightarrow F_{S16}$
PMINUB	$\lfloor F_{U8}, F_{U8}\rfloor\rightarrow F_{U8}$ $\lfloor F_{U8}, mem_{U8}\rfloor\rightarrow F_{U8}$
PMOVMSKB	$F_{[(i*8)-1]}\rightarrow R_{[i]}$
PMULHUW	$U_{16}(F_{U16}\times F_{U16})\rightarrow F_{U16}$ $U_{16}(F_{U16}\times mem_{U16})\rightarrow F_{U16}$
PREFETCHNTA	prefetch with minimal L1/L2 pollution
PREFETCHT0	prefetch to all cache levels
PREFETCHT1	prefetch to all but L1 cache level
PREFETCHT2	prefetch to all but L1 and L2 cache levels
PSADBW	$\Re_+(F_{U8}\nabla F_{U8})\rightarrow F_{\bullet U16}$ $\Re_+(F_{U8}\nabla mem_{U8})\rightarrow F_{\bullet U16}$
PSHUFW	$F_{16}[imm_{8[2*i+1..2*i]}]\rightarrow F_{16}[i]$ $mem_{16}[imm_{8[2*i+1..2*i]}]\rightarrow F_{16}[i]$
RCPPS	$\approx_{12} 1/V_{FP32}\rightarrow V_{FP32}$ $\approx_{12} 1/mem_{FP32}\rightarrow V_{FP32}$
RCPSS	$\approx_{12} 1/V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $\approx_{12} 1/mem_{\bullet FP32}\rightarrow V_{\bullet FP32}$
RSQRTPS	$\approx_{12} 1/\sqrt{V_{FP32}}\rightarrow V_{FP32}$ $\approx_{12} 1/\sqrt{mem_{FP32}}\rightarrow V_{FP32}$
RSQRTSS	$\approx_{12} 1/\sqrt{V_{\bullet FP32}}\rightarrow V_{\bullet FP32}$ $\approx_{12} 1/\sqrt{mem_{\bullet FP32}}\rightarrow V_{\bullet FP32}$
SFENCE	force completion of weakly ordered stores
SHUFPS	$V_{FP32}[I_{8[2*i+3.2*i+2]}]\|\| V_{FP32}[I_{8[2*i+1..2*i]}]\|\|\rightarrow V$ $mem_{FP32}[I_{8[2*i+3..2*i+2]}]\|\| V_{FP32}[I_{8[2*i+1..2*i]}]\|\|\rightarrow V$
SQRTPS	$\sqrt{V_{FP32}}\rightarrow V_{FP32}$ $\sqrt{mem_{FP32}}\rightarrow V_{FP32}$
SQRTSS	$\sqrt{V_{\bullet FP32}}\rightarrow V_{\bullet FP32}$ $\sqrt{mem_{\bullet FP32}}\rightarrow V_{\bullet FP32}$
STMXCSR	store SSE control/status word
SUBPS	$V_{FP32}-V_{FP32}\rightarrow V_{FP32}$ $V_{FP32}-mem_{FP32}\rightarrow V_{FP32}$
SUBSS	$V_{\bullet FP32}-V_{\bullet FP32}\rightarrow V_{\bullet FP32}$ $V_{\bullet FP32}-mem_{\bullet FP32}\rightarrow V_{\bullet FP32}$
UCOMISS	$bv(V_{\bullet FP32}\{!?,>,<\}V_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags $bv(V_{\bullet FP32}\{!?,>,<\}mem_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags
UNPCKHPS	$U(V_{32})\wedge\vee U(V_{32})\rightarrow V$ $U(V_{32})\wedge\vee U(mem_{32})\rightarrow V$
UNPCKLPS	$L(V_{32})\wedge\vee L(V_{32})\rightarrow V$ $L(V_{32})\wedge\vee L(mem_{32})\rightarrow V$
XORPS	$V\oplus V\rightarrow V$

Source: Intel Architecture Software Developer's Manual, Volume II: Instruction Set Reference, 1999

Intel SSE2	Functionality
ADDPD	$V_{FP64}+V_{FP64}\rightarrow V_{FP64}$ $V_{FP64}+mem_{FP64}\rightarrow V_{FP64}$
ADDSD	$V_{\bullet FP64}+V_{\bullet FP64}\rightarrow V_{\bullet FP64}$ $V_{\bullet FP64}+mem_{\bullet FP64}\rightarrow V_{\bullet FP64}$
ANDNPD	~(V)&V→V ~(V)&mem→V
ANDPD	~(V)&V→V ~(V)&mem→V
CFLUSH	Flush and invalidate memory operand in all cache levels
CMPPD	$m(V_{FP64}\{==,<,\leq,!?,!=,!<,!\leq,?\}V_{FP64})\rightarrow V_{64}$ $m(V_{FP64}\{==,<,\leq,!?,!=,!<,!\leq,?\}mem_{FP64})\rightarrow V_{64}$
CMPSD	$m(V_{\bullet FP64}\{==,<,\leq,!?,!=,!<,!\leq,?\}V_{\bullet FP64})\rightarrow V_{\bullet 64}$ $m(V_{\bullet FP64}\{==,<,\leq,!?,!=,!<,!\leq,?\}mem_{\bullet FP64})\rightarrow V_{\bullet 64}$
COMISD	$bv(V_{\bullet FP64}\{!?,>,<\}V_{\bullet FP64})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags $bv(V_{\bullet FP64}\{!?,>,<\}mem_{\bullet FP64})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags
CVTDQ2PD	$V_{S32}\rightarrow V_{FP64}$ $mem_{S32}\rightarrow V_{FP64}$
CVTDQ2PS	$V_{S32}\rightarrow V_{FP32}$ $mem_{S32}\rightarrow V_{FP32}$
CVTPD2DQ	$V_{FP64}\rightarrow 0\|\|V_{S32}$ $mem_{FP64}\rightarrow 0\|\|V_{S32}$
CVTPD2PI	$V_{FP64}\rightarrow F_{S32}$ $mem_{FP64}\rightarrow F_{S32}$
CVTPD2PS	$V_{FP64}\rightarrow 0\|\|V_{FP32}$ $mem_{FP64}\rightarrow 0\|\|V_{FP32}$
CVTPI2PD	$F_{S32}\rightarrow V_{FP64}$ $mem_{S32}\rightarrow V_{FP64}$
CVTPS2DQ	$V_{FP32}\rightarrow V_{S32}$ $mem_{FP32}\rightarrow V_{S32}$
CVTPS2PD	$L_{64}(V_{FP32})\rightarrow V_{FP64}$ $L_{64}(mem_{FP32})\rightarrow V_{FP64}$
CVTSD2SI	$V_{\bullet FP64}\rightarrow R_{[32..0]}$ $mem_{\bullet FP64}\rightarrow R_{[32..0]}$
CVTSD2SS	$V_{\bullet FP64}\rightarrow V_{\bullet FP32}$ $mem_{\bullet FP64}\rightarrow V_{\bullet FP32}$
CVTSI2SD	$U_{64}(V_{FP64})\|\|R_{[32..0]}\rightarrow V_{FP64}$ $U_{64}(V_{FP64})\|\|mem_{[32..0]}\rightarrow V_{FP64}$
CVTSS2SD	$V_{\bullet FP32}\rightarrow V_{\bullet FP64}$ $mem_{\bullet FP32}\rightarrow V_{\bullet FP64}$
CVTTPD2DQ	$V_{FP64}\rightarrow 0\|\|V_{S32}$ $mem_{FP64}\rightarrow 0\|\|V_{S32}$
CVTTPD2PI	$V_{FP64}\rightarrow F_{S32}$ $mem_{FP64}\rightarrow F_{S32}$
CVTTPS2DQ	$V_{FP32}\rightarrow V_{S32}$ $mem_{FP32}\rightarrow V_{S32}$
CVTTSD2SI	$V_{\bullet FP64}\rightarrow R_{[32..0]}$ $mem_{\bullet FP64}\rightarrow R_{[32..0]}$
DIVPD	$V_{FP64}\div V_{FP64}\rightarrow V_{FP64}$ $V_{FP64}\div mem_{FP64}\rightarrow V_{FP64}$
DIVSD	$V_{\bullet FP64}\div V_{\bullet FP64}\rightarrow V_{\bullet FP64}$ $V_{\bullet FP64}\div mem_{\bullet FP64}\rightarrow V_{\bullet FP64}$
LFENCE	Serialize load operations
MASKMOVDQU	$if(V_{8[(i*8)-1]})\ V_8[i]\rightarrow mem_8[i]$
MAXPD	$\lceil V_{FP64},V_{FP64}\rceil\rightarrow V_{FP64}$ $\lceil V_{FP64},mem_{FP64}\rceil\rightarrow V_{FP64}$
MAXSD	$\lceil V_{\bullet FP64},V_{\bullet FP64}\rceil\rightarrow V_{\bullet FP64}$ $\lceil V_{\bullet FP64},mem_{\bullet FP64}\rceil\rightarrow V_{\bullet FP64}$
MFENCE	Serialize load and store operations
MINPD	$\lfloor V_{FP64},V_{FP64}\rfloor\rightarrow V_{FP64}$ $\lfloor V_{FP64},mem_{FP64}\rfloor\rightarrow V_{FP64}$
MINSD	$\lfloor V_{\bullet FP64},V_{\bullet FP64}\rfloor\rightarrow V_{\bullet FP64}$ $\lfloor V_{\bullet FP64},mem_{\bullet FP64}\rfloor\rightarrow V_{\bullet FP64}$
MOVAPD	V→V V→mem (exception if unaligned mem)
	mem→V (exception if unaligned mem)
MOVDQA	V→V V→mem (exception if unaligned mem) mem→V (exception if unaligned mem)
MOVDQU	V→V V→mem
MOVHPD	$U_{64}(V)\rightarrow$mem mem→$U_{64}(V)$
MOVLPD	$L_{64}(V)\rightarrow$mem mem→$L_{64}(V)$
MOVMSKPD	$V_{[(i*32)-1]}\rightarrow R_{[i]}$
MOVNTDQ	V→mem (no write allocate)
MOVNTI	R→mem (no write allocate)
MOVNTPD	V→mem (no write allocate)
MOVQ2DQ	$L_{64}(F)\rightarrow 0\|\|V_{64}$
MOVSD	$V_{\bullet FP64}\rightarrow V_{\bullet FP64}$ $V_{\bullet FP64}\rightarrow$mem mem→$V_{\bullet FP64}$
MOVUPD	V→V V→mem mem→V
MOVDQ2Q	$L_{64}(V)\rightarrow F$
MULPD	$V_{FP64}\times V_{FP64}\rightarrow V_{FP64}$ $V_{FP64}\times mem_{FP64}\rightarrow V_{FP64}$
MULSD	$V_{\bullet FP64}\times V_{\bullet FP64}\rightarrow V_{\bullet FP64}$ $V_{\bullet FP64}\times mem_{\bullet FP64}\rightarrow V_{\bullet FP64}$
ORPD	V|V→V V|mem→V
PADDQ	$V_{64}+V_{64}\rightarrow V_{64}$
PAUSE	Delay execution of next instruction
PMULUDQ	$V_{U32}\times V_{U32}\rightarrow V_{U64}$ $V_{U32}\times mem_{U32}\rightarrow V_{U64}$ $F_{\bullet U32}\times F_{\bullet U32}\rightarrow F_{\bullet U64}$ $F_{\bullet U32}\times mem_{\bullet U32}\rightarrow F_{\bullet U64}$
PSHUFD	$V_{32}[imm_{8[2*i+1..2*i]}]\rightarrow V_{32}[i]$ $mem_{32}[imm_{8[2*i+1..2*i]}]\rightarrow V_{32}[i]$
PSHUFHW	$U_{64}(V_{16})[imm_{8[2*i+1..2*i]}]\|\|L_{64}(V_{16})\rightarrow U_{64}(V_{16})[i]$ $U_{64}(mem_{16})[imm_{8[2*i+1..2*i]}]\|\|L_{64}(mem_{16})\rightarrow U_{64}(V_{16})[i]$
PSHUFLW	$L_{64}(V_{16})[imm_{8[2*i+1..2*i]}]\|\|U_{64}(V_{16})\rightarrow L_{64}(V_{16})$ [i] $L_{64}(mem_{16})[imm_{8[2*i+1..2*i]}]\|\|U_{64}(mem_{16})\rightarrow L_{64}(V_{16})[i]$
PSLLDQ	$V<<8*imm_8\rightarrow V$
PSRLDQ	$V>>8*imm_8\rightarrow V$
PSUBQ	$V_{64}-V_{64}\rightarrow V_{64}$
PUNPCKHQDQ	$U(V_{32})\wedge\vee U(V_{32})\rightarrow V$ $U(V_{32})\wedge\vee U(mem_{32})\rightarrow V$
PUNPCKLQDQ	$L(V_{32})\wedge\vee L(V_{32})\rightarrow V$ $L(V_{32})\wedge\vee L(mem_{32})\rightarrow V$
SHUFPD	$V_{FP64}[I_{8[2*i+3.2*i+2]}]\|\|\ V_{FP64}[I_{8[2*i+1..2*i]}]\|\|\rightarrow V$ $mem_{FP64}[I_{8[2*i+3..2*i+2]}]\|\|\ V_{FP64}[I_{8[2*i+1..2*i]}]\|\|\rightarrow V$
SQRTPD	$\sqrt{V_{FP64}}\rightarrow V_{FP64}$ $\sqrt{mem_{FP64}}\rightarrow V_{FP64}$
SQRTSD	$\sqrt{V_{\bullet FP64}}\rightarrow V_{\bullet FP64}$ $\sqrt{mem_{\bullet FP64}}\rightarrow V_{\bullet FP64}$
SUBPD	$V_{FP64}-V_{FP64}\rightarrow V_{FP64}$ $V_{FP64}-mem_{FP64}\rightarrow V_{FP64}$
SUBSD	$V_{\bullet FP64}-V_{\bullet FP64}\rightarrow V_{\bullet FP64}$ $V_{\bullet FP64}-mem_{\bullet FP64}\rightarrow V_{\bullet FP64}$
UCOMISD	$bv(V_{\bullet FP64}\{!?,>,<\}V_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags $bv(V_{\bullet FP64}\{!?,>,<\}mem_{\bullet FP32})\rightarrow$ ZF, PF, CF flags, 0→OF, SF, AF flags
UNPCKHPD	$U(V_{64})\wedge\vee U(V_{64})\rightarrow V$ $U(V_{64})\wedge\vee U(mem_{64})\rightarrow V$
UNPCKLPD	$L(V_{64})\wedge\vee L(V_{64})\rightarrow V$ $L(V_{64})\wedge\vee L(mem_{64})\rightarrow V$
XORPD	$V\oplus V\rightarrow V$

Source: IA-32 Intel Architecture Software Developer's Manual with Preliminary Willamette Architecture Information Vol. 2: Instruction Set Reference, 2000

MIPS MDMX	Functionality
ADD.QH	$F_{S16}+F_{S16} \Rightarrow F_{S16}$
ADD.OB	$F_{U8}+F_{U8} \Rightarrow F_{U8}$
ADDA.QH	$F_{S16}+F_{S16} \rightarrow \Sigma A_{S48}$
ADDA.OB	$F_{U8}+F_{U8} \rightarrow \Sigma A_{U24}$
ADDL.QH	$F_{S16}+F_{S16} \rightarrow A_{S48}$
ADDL.OB	$F_{U8}+F_{U8} \rightarrow \Sigma A_{U24}$
ALNI.QH	$(F_8 \| F_8)[imm_3+7..imm_3] \rightarrow F$
ALNI.OB	$(F_8 \| F_8)[imm_3+7..imm_3] \rightarrow F$
ALNV.QH	$(F_8 \| F_8)[R_{[2..0]}+7..R_{[2..0]}] \rightarrow F$
ALNV.OB	$(F_8 \| F_8)[R_{[2..0]}+7..R_{[2..0]}] \rightarrow F$
AND.QH	F&F→F
AND.OB	F&F→F
C.cond.QH	if(F_{S16}[i]{<,≤,==}F_{S16}[i]), CC[i]=1, else CC[i]=0
C.cond.OB	if(F_{U8}[i]{<,≤,==}F_{U8}[i]), CC[i]=1, else CC[i]=0
MAX.QH	$\lceil F_{S16}, F_{S16} \rceil \rightarrow F_{S16}$
MAX.OB	$\lceil F_{U8}, F_{U8} \rceil \rightarrow F_{U8}$
MIN.QH	$\lfloor F_{S16}, F_{S16} \rfloor \rightarrow F_{S16}$
MIN.OB	$\lfloor F_{U8}, F_{U8} \rfloor \rightarrow F_{U8}$
MSGN.QH	$F_{S16} \otimes F_{S16} \rightarrow F_{S16}$
MUL.QH	$F_{S16} \times F_{S16} \Rightarrow F_{S16}$
MUL.OB	$F_{U8} \times F_{U8} \Rightarrow F_{U8}$
MULA.QH	$F_{S16} \times F_{S16} \rightarrow \Sigma A_{S48}$
MULA.OB	$F_{U8} \times F_{U8} \rightarrow \Sigma A_{U24}$
MULL.QH	$F_{S16} \times F_{S16} \rightarrow A_{S48}$
MULL.OB	$F_{U8} \times F_{U8} \rightarrow A_{U24}$
MULS.QH	$F_{S16} \times F_{S16} \rightarrow \Sigma A_{S48}$
MULS.OB	$F_{U8} \times F_{U8} \rightarrow \Sigma A_{U24}$
MULSL.QH	$-(F_{S16} \times F_{S16}) \rightarrow A_{S48}$
MULSL.OB	$-(F_{U8} \times F_{U8}) \rightarrow A_{U24}$
NOR.QH	!(F|F)→F
NOR.OB	!(F|F)→F
OR.QH	F|F→F
OR.OB	F|F→F
PICKF.QH	if(!cc[i]) F_{16}[i]→F_{16}[i], else F_{16}[i]→F_{16}[i]
PICKF.OB	if(!cc[i]) F_8[i]→F_8[i], else F_8[i]→F_8[i]
PICKT.QH	if(cc[i]) F_{16}[i]→F_{16}[i], else F_{16}[i]→F_{16}[i]
PICKT.OB	if(cc[i]) F_8[i]→F_8[i], else F_8[i]→F_8[i]
RZS.QH	$\varnothing(A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RZU.QH	$\varnothing(A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RZU.OB	$\varnothing(A_{U8} >> F_{U8}) \Rightarrow F_{U8}$
RNAS.QH	$\lfloor (A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RNAU.QH	$\lfloor (A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RNAU.OB	$\lfloor (A_{U8} >> F_{U8}) \Rightarrow F_{U8}$
RNES.QH	$\{ (A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RNEU.QH	$\{ (A_{S48} >> F_{S16}) \Rightarrow F_{S16}$
RNEU.OB	$\{ (A_{U8} >> F_{U8}) \Rightarrow F_{U8}$
RACL.QH	$A_{64}[0] \rightarrow F$
RACM.QH	$A_{64}[1] \rightarrow F$
RACH.QH	$A_{64}[2] \rightarrow F$
RACL.OB	$A_{64}[0] \rightarrow F$
RACM.OB	$A_{64}[1] \rightarrow F$
RACH.OB	$A_{64}[2] \rightarrow F$
SHFL.func.QH	$f(F_{16}, F_{16}) \rightarrow F_{16}$
SHFL.func.OB	$f(F_8, F_8) \rightarrow F_8$
SLL.QH	$F_{S16} << F_{S16} \rightarrow F_{S16}$
SLL.OB	$F_{U8} << F_{U8} \rightarrow F_{U8}$
SRA.QH	$F_{S16} >> F_{S16} \rightarrow F_{S16}$
SRL.QH	$F_{S16} >> F_{S16} \rightarrow F_{S16}$
SRL.OB	$F_{U8} >> F_{U8} \rightarrow F_{U8}$
SUB.QH	$F_{S16} - F_{S16} \Rightarrow F_{S16}$
SUB.OB	$F_{U8} - F_{U8} \Rightarrow F_{U8}$
SUBA.QH	$F_{S16} - F_{S16} \rightarrow \Sigma A_{S48}$
SUBA.OB	$F_{U8} - F_{U8} \rightarrow \Sigma A_{U24}$
SUBL.QH	$F_{S16} - F_{S16} \rightarrow A_{S48}$
SUBL.OB	$F_{U8} - F_{U8} \rightarrow \Sigma A_{U24}$
WACH.QH	$F \rightarrow A_{64}[2]$
WACH.OB	$F \rightarrow A_{64}[2]$
WACL.QH	$F \rightarrow A_{64}[0]$, $F \rightarrow A_{64}[1]$
WACL.OB	$F \rightarrow A_{64}[0]$, $F \rightarrow A_{64}[1]$
XOR.QH	$F_{S16} \oplus F_{S16} \rightarrow F_{S16}$
XOR.OB	$F_{U8} \oplus F_{U8} \rightarrow F_{U8}$

Source: MIPS Digital Media Extension Rev 1.0,
MIPS Extension for Digital Media with 3D, March 12, 1997

MIPS MIPS-3D

Instruction	Functionality
ADDR	$\Re_+(F_{FP32}) \| \Re_+(F_{FP32}) \rightarrow F_{FP32}$
MULR	$\Re_\times(F_{FP32}) \| \Re_\times(F_{FP32}) \rightarrow F_{FP32}$
RECIP1*	$\approx_{14} 1/F_{FP32} \rightarrow F_{FP32}$
RECIP2*	$\approx_{24} 1/F_{FP32} \rightarrow F_{FP32}$
RSQRT1*	$\approx_{15} 1/\sqrt{} F_{FP32} \rightarrow F_{FP32}$
RSQRT2*	$\approx_{24} 1/\sqrt{} F_{FP32} \rightarrow F_{FP32}$
CVT.PS.PW	$F_{S32} \rightarrow F_{FP32}$
CVT.PW.PS	$F_{FP32} \rightarrow F_{S32}$
CABS	bv($\|F_{FP32}\|${0,?,<,>,=}$\|F_{FP32}\|$)→$cc_{[i..i+1]}$
BC1ANY2F	branch if (cc_i==0|cc_{i+1}==0)
BC1ANY2T	branch if (cc_i==1|cc_{i+1}==1)
BC1ANY4F	branch if (cc_i==0|cc_{i+1}==0|cc_{i+2}==0|cc_{i+3}==0)
BC1ANY4T	branch if (cc_i==1|cc_{i+1}==1|cc_{i+2}==1|cc_{i+3}==1)

Source: Radhika Thekkath, Mike Uhler, Chandlee Harrell, Ying-wai Ho,
"An Architecture Extension for Efficient Geometery Processing,"
Proceedings of Hot Chips 11, August 15-17, 1999, pg. 263-274
"MIPS-3D ASE Product Brief", MIPS
*RECIP1/2 and RSQRT1/2 precisions are estimates based on other architectures and the claim of full precision (24 mantissa bits) after the second step in each case

MIPS MIPS-V	Functionality
ABS.PS	$\|F_{FP32}\| \rightarrow F_{FP32}$
ADD.PS	$F_{FP32}+F_{FP32} \rightarrow F_{FP32}$
ALNV.PS	$(F_8 \| F_8)[R_{[0..2]}..R_{[0..2]}+7] \rightarrow F$
C.cond.PS	if(F_{FP32}[i]{{0,?,<,>,=}}F_{FP32}[i])→cc[i]
CVT.PS.S	$F_{\bullet FP32} \| F_{\bullet FP32} \rightarrow F_{FP32}$
CVT.S.PL	$L(F_{FP32}) \rightarrow F_{\bullet FP32}$
CVT.S.PU	$U(F_{FP32}) \rightarrow F_{\bullet FP32}$
LUXC1	load doubleword indexed unaligned
MADD.PS	$(F_{FP32} \times F_{FP32})+F_{FP32} \rightarrow F_{FP32}$
MOV.PS	F→F
MOVF.PS	if(!cc_i) F_{FP32}[0]→F_{FP32}, if(!cc_{i+1}) F_{FP32}[1]→F_{FP32}
MOVT.PS	if(cc_i) F_{FP32}[0]→F_{FP32}, if(cc_{i+1}) F_{FP32}[1]→F_{FP32}
MSUB.PS	$(F_{FP32} \times F_{FP32}) - F_{FP32} \rightarrow F_{FP32}$
MUL.PS	$(F_{FP32} \times F_{FP32}) \rightarrow F_{FP32}$
NEG.PS	$-F_{FP32} \rightarrow F_{FP32}$
NMADD.PS	$-((F_{FP32} \times F_{FP32})+F_{FP32}) \rightarrow F_{FP32}$
NMSUB.PS	$-((F_{FP32} \times F_{FP32}) - F_{FP32}) \rightarrow F_{FP32}$
PLL.PS	$L(F_{32}) \| L(F_{32}) \rightarrow F_{32}$
PLU.PS	$L(F_{32}) \| U(F_{32}) \rightarrow F_{32}$
PUL.PS	$U(F_{32}) \| L(F_{32}) \rightarrow F_{32}$
PUU.PS	$U(F_{32}) \| U(F_{32}) \rightarrow F_{32}$
SUB.PS	$F_{FP32} - F_{FP32} \rightarrow F_{FP32}$
SUXC1	store doubleword indexed unaligned

Source: MIPS V Instruction Set Rev 1.0,
MIPS Extension for Digital Media with 3D, March 12, 1997

Motorola AltiVec	Functionality
DSS	data stream stop
DSSALL	data stream stop all
DST	data stream touch
DSTT	data stream touch transient
DSTST	data stream touch for store
DSTSTT	data stream touch for store transient
LVEBX	$mem_8 \rightarrow V_8[addr_{3..0}]$, addr = R + R
LVEHX	$mem_{16} \rightarrow V_{16}[addr_{2..0}]$, addr = R + R
LVEWX	$mem_{32} \rightarrow V_{32}[addr_{1..0}]$, addr = R + R
LVSL	$f(R+R) \rightarrow V$
LVSR	$f(R+R) \rightarrow V$
LVX	$mem_{128} \rightarrow V$ (forces alignment)
LVXL	$mem_{128} \rightarrow V$ (forces alignment), transient
MFVSCR	move from vector status/control register
MTVSCR	move to vector status/control register
STVEBX	$V_8[addr_{3..0}] \rightarrow mem_8$, addr = R + R
STVEHX	$V_{16}[addr_{2..0}] \rightarrow mem_{16}$, addr = R + R
STVEWX	$V_{32}[addr_{1..0}] \rightarrow mem_{32}$, addr = R + R
STVX	$V \rightarrow mem_{128}$ (forced alignment)
STVXL	$V \rightarrow mem_{128}$ (forced alignment), transient
VADDCUW	$C(V_{32}+V_{32}) \rightarrow V_{32}$
VADDFP	$V_{FP32}+V_{FP32} \rightarrow V_{FP32}$
VADDSBS	$V_{S8}+V_{S8} \Rightarrow V_{S8}$
VADDSHS	$V_{S16}+V_{S16} \Rightarrow V_{S16}$
VADDSWS	$V_{S32}+V_{S32} \Rightarrow V_{S32}$
VADDUBM	$V_{U8}+V_{U8} \rightarrow V_{U8}$
VADDUBS	$V_{U8}+V_{U8} \Rightarrow V_{U8}$
VADDUHM	$V_{U16}+V_{U16} \rightarrow V_{U16}$
VADDUHS	$V_{U16}+V_{U16} \Rightarrow V_{U16}$
VADDUWM	$V_{U32}+V_{U32} \rightarrow V_{U32}$
VADDUWS	$V_{U32}+V_{U32} \Rightarrow V_{U32}$
VAND	$V \& V \rightarrow V$
VANDC	$!V \& V \rightarrow V$
VAVGSB	$V_{S8} \Lambda V_{S8} \rightarrow V_{S8}$
VAVGSH	$V_{S16} \Lambda V_{S16} \rightarrow V_{S16}$
VAVGSW	$V_{S32} \Lambda V_{S32} \rightarrow V_{S32}$
VAVGUB	$V_{U8} \Lambda V_{U8} \rightarrow V_{U8}$
VAVGUH	$V_{U16} \Lambda V_{U16} \rightarrow V_{U16}$
VAVGUW	$V_{U32} \Lambda V_{U32} \rightarrow V_{U32}$
VCFSX	$V_{S32} >> imm_5 \rightarrow V_{FP32}$
VCFUX	$V_{U32} >> imm_5 \rightarrow V_{FP32}$
VCMPBFP	$bv(V_{FP32} <> V_{FP32}) \rightarrow V_{32}$
VCMPBFP.	$bv(V_{FP32} <> V_{FP32}) \rightarrow V_{32}$, set CR6 bits
VCMPEQFP	$m(V_{FP32} == V_{FP32}) \rightarrow V_{32}$
VCMPEQFP.	$m(V_{FP32} == V_{FP32}) \rightarrow V_{32}$, set CR6 bits
VCMPEQUB	$m(V_{U8} == V_{U8}) \rightarrow V_8$
VCMPEQUB.	$m(V_{U8} == V_{U8}) \rightarrow V_8$, set CR6 bits
VCMPEQUH	$m(V_{U16} == V_{U16}) \rightarrow V_{16}$
VCMPEQUH.	$m(V_{U16} == V_{U16}) \rightarrow V_{16}$, set CR6 bits
VCMPEQUW	$m(V_{U32} == V_{U32}) \rightarrow V_{32}$
VCMPEQUW.	$m(V_{U32} == V_{U32}) \rightarrow V_{32}$, set CR6 bits
VCMPGEFP	$m(V_{FP32} \geq V_{FP32}) \rightarrow V_{32}$
VCMPGEFP.	$m(V_{FP32} \geq V_{FP32}) \rightarrow V_{32}$, set CR6 bits
VCMPGTFP	$m(V_{FP32} > V_{FP32}) \rightarrow V_{32}$
VCMPGTFP.	$m(V_{FP32} > V_{FP32}) \rightarrow V_{32}$, set CR6 bits
VCMPGTSB	$m(V_{S8} > V_{S8}) \rightarrow V_8$
VCMPGTSB.	$m(V_{S8} > V_{S8}) \rightarrow V_8$, set CR6 bits
VCMPGTSH	$m(V_{S16} > V_{S16}) \rightarrow V_{16}$
VCMPGTSH.	$m(V_{S16} > V_{S16}) \rightarrow V_{16}$, set CR6 bits
VCMPGTSW	$m(V_{S32} > V_{S32}) \rightarrow V_{32}$
VCMPGTSW.	$m(V_{S32} > V_{S32}) \rightarrow V_{32}$, set CR6 bits
VCMPGTUB	$m(V_{U8} > V_{U8}) \rightarrow V_8$
VCMPGTUB.	$m(V_{U8} > V_{U8}) \rightarrow V_8$, set CR6 bits
VCMPGTUH	$m(V_{U16} > V_{U16}) \rightarrow V_{16}$
VCMPGTUH.	$m(V_{U16} > V_{U16}) \rightarrow V_{16}$, set CR6 bits
VCMPGTUW	$m(V_{U32} > V_{U32}) \rightarrow V_{32}$
VCMPGTUW.	$m(V_{U32} > V_{U32}) \rightarrow V_{32}$, set CR6 bits
VCTSXS	$V_{FP32} \times (1 << imm_5) \Rightarrow V_{S32}$
VCTUXS	$V_{FP32} \times (1 << imm_5) \Rightarrow V_{U32}$
VEXPTEFP	$\approx_{12} exp_2(V_{FP32}) \rightarrow V_{FP32}$
VLOGEFP	$\approx_{12} log_2(V_{FP32}) \rightarrow V_{FP32}$
VMADDFP	$(V_{FP32} \times V_{FP32}) + V_{FP32} \rightarrow V_{FP32}$
VMAXFP	$\lceil V_{FP32}, V_{FP32} \rceil \rightarrow V_{FP32}$
VMAXSB	$\lceil V_{S8}, V_{S8} \rceil \rightarrow V_{S8}$
VMAXSH	$\lceil V_{S16}, V_{S16} \rceil \rightarrow V_{S16}$
VMAXSW	$\lceil V_{S32}, V_{S32} \rceil \rightarrow V_{S32}$
VMAXUB	$\lceil V_{U8}, V_{U8} \rceil \rightarrow V_{U8}$
VMAXUH	$\lceil V_{U16}, V_{U16} \rceil \rightarrow V_{U16}$
VMAXUW	$\lceil V_{U32}, V_{U32} \rceil \rightarrow V_{U32}$
VMHADDSHS	$U_{16}(V_{S16} \times V_{S16}) + V_{S16} \Rightarrow V_{S16}$
VMHRADDSHS	$U_{16}(\lfloor (V_{S16} \times V_{S16}) \rfloor) + V_{S16} \Rightarrow V_{S16}$
VMINFP	$\lfloor V_{FP32}, V_{FP32} \rfloor \rightarrow V_{FP32}$
VMINSB	$\lfloor V_{S8}, V_{S8} \rfloor \rightarrow V_{S8}$
VMINSH	$\lfloor V_{S16}, V_{S16} \rfloor \rightarrow V_{S16}$
VMINSW	$\lfloor V_{S32}, V_{S32} \rfloor \rightarrow V_{S32}$
VMINUB	$\lfloor V_{U8}, V_{U8} \rfloor \rightarrow V_{U8}$
VMINUH	$\lfloor V_{U16}, V_{U16} \rfloor \rightarrow V_{U16}$
VMINUW	$\lfloor V_{U32}, V_{U32} \rfloor \rightarrow V_{U32}$
VMLADDUHM	$L_{16}(V_{S16} \times V_{S16}) + V_{S16} \rightarrow V_{S16}$
VMRGHB	$U(V_8) \wedge \vee U(V_8) \rightarrow V$
VMRGHH	$U(V_{16}) \wedge \vee U(V_{16}) \rightarrow V$
VMRGHW	$U(V_{32}) \wedge \vee U(V_{32}) \rightarrow V$
VMRGLB	$L(V_8) \wedge \vee L(V_8) \rightarrow V$
VMRGLH	$L(V_{16}) \wedge \vee L(V_{16}) \rightarrow V$
VMRGLW	$L(V_{32}) \wedge \vee L(V_{32}) \rightarrow V$
VMSUMMBM	$\Re_+(V_{U8} \times V_{S8}) + V_{S32} \rightarrow V_{S32}$
VMSUMSHM	$\Re_+(V_{S16} \times V_{S16}) + V_{S32} \rightarrow V_{S32}$
VMSUMSHS	$\Re_+(V_{S16} \times V_{S16}) + V_{S32} \Rightarrow V_{S32}$
VMSUMUBM	$\Re_+(V_{U8} \times V_{U8}) + V_{U32} \rightarrow V_{U32}$
VMSUMUHM	$\Re_+(V_{U16} \times V_{U16}) + V_{U32} \rightarrow V_{U32}$
VMSUMUHS	$\Re_+(V_{U16} \times V_{U16}) + V_{U32} \Rightarrow V_{U32}$
VMULESB	$E(V_{S8}) \times E(V_{S8}) \rightarrow V_{S16}$
VMULESH	$E(V_{S16}) \times E(V_{S16}) \rightarrow V_{S32}$
VMULEUB	$E(V_{U8}) \times E(V_{U8}) \rightarrow V_{U16}$
VMULEUH	$E(V_{U16}) \times E(V_{U16}) \rightarrow V_{U32}$
VMULOSB	$O(V_{S8}) \times O(V_{S8}) \rightarrow V_{S16}$
VMULOSH	$O(V_{S16}) \times O(V_{S16}) \rightarrow V_{S32}$
VMULOUB	$O(V_{U8}) \times O(V_{U8}) \rightarrow V_{U16}$
VMULOUH	$O(V_{U16}) \times O(V_{U16}) \rightarrow V_{U32}$
VNMSUBFP	$-((V_{FP32} \times V_{FP32}) - V_{FP32}) \rightarrow V_{FP32}$
VNOR	$!(V \vert V) \rightarrow V$
VOR	$V \vert V \rightarrow V$
VPERM	$(V_8 \| V_8)[V_{U8}] \rightarrow V_8[i]$
VPKPX	$(V \| V) \rightarrow V_{pixel}$
VPKSHSS	$(V_{S16} \| V_{S16}) \Rightarrow V_{S8}$
VPKSHUS	$(V_{S16} \| V_{S16}) \Rightarrow V_{U8}$
VPKSWSS	$(V_{S32} \| V_{S32}) \Rightarrow V_{S16}$
VPKSWUS	$(V_{S32} \| V_{S32}) \Rightarrow V_{U16}$
VPKUHUM	$(V_{U16} \| V_{U16}) \rightarrow V_{U8}$
VPKUHUS	$(V_{U16} \| V_{U16}) \Rightarrow V_{U8}$
VPKUWUM	$(V_{U32} \| V_{U32}) \rightarrow V_{U16}$
VPKUWUS	$(V_{U32} \| V_{U32}) \Rightarrow V_{U16}$
VREFP	$\approx_{12} 1/V_{FP32} \rightarrow V_{FP32}$
VRFIM	$\lfloor (V_{FP32}) \rightarrow V_{FP32}$
VRFIN	$\{ (V_{FP32}) \rightarrow V_{FP32}$
VRFIP	$\lceil (V_{FP32}) \rightarrow V_{FP32}$
VRFIZ	$\varnothing (V_{FP32}) \rightarrow V_{FP32}$
VRLB	$V_8 \langle\langle \rightarrow V_8$
VRLH	$V_{16} \langle\langle \rightarrow V_{16}$

VRLW	$V_{32}\langle\langle\rightarrow V_{32}$
VRSQRTEFP	$\approx_{12} 1/\sqrt{V_{FP32}} \rightarrow V_{FP32}$
VSEL	$(V_i==1)?V_i:V_i \rightarrow V_i$
VSL	$V<<V_{[0..2]} \rightarrow V$
VSLB	$V_8<<V_{[0..2]} \rightarrow V_8$
VSLDOI	$(V_8 \| V_8)[imm_4+15..imm_4] \rightarrow V$
VSLH	$V_{16}<<V_{[0..3]} \rightarrow V_{16}$
VSLO	$V<<V_{[6..3]\|000} \rightarrow V$
VSLW	$V_{16}<<V_{[0..4]} \rightarrow V_{16}$
VSPLTB	$V_8[imm_{U5}] \rightarrow V_8[i]$
VSPLTH	$V_{16}[imm_{U5}] \rightarrow V_{16}[i]$
VSPLTISB	$imm_{S8} \rightarrow V_{S8}[i]$
VSPLTISH	$imm_{S16} \rightarrow V_{S16}[i]$
VSPLTISW	$imm_{S32} \rightarrow V_{S32}[i]$
VSPLTW	$V_{32}[imm_{U5}] \rightarrow V_{32}[i]$
VSR	$V>>V_{[0..2]} \rightarrow V$
VSRAB	$V_{S8} \underline{>>} V_{[0..2]} \rightarrow V_{S8}$
VSRAH	$V_{S16} \underline{>>} V_{[0..3]} \rightarrow V_{S16}$
VSRAW	$V_{S32} \underline{>>} V_{[0..4]} \rightarrow V_{S32}$
VSRB	$V_8>>V_{[0..2]} \rightarrow V_8$
VSRH	$V_{16}>>V_{[0..2]} \rightarrow V_{16}$
VSRO	$V_8>>V_{[6..3]000} \rightarrow V_8$
VSRW	$V_{32}>>V_{[0..4]} \rightarrow V_{32}$
VSUBCUW	$!C(V_{U32}-V_{U32}) \rightarrow V_{32}$
VSUBFP	$V_{FP32}-V_{FP32} \rightarrow V_{FP32}$
VSUBSBS	$V_{S8}-V_{S8} \Rightarrow V_{S8}$
VSUBSHS	$V_{S16}-V_{S16} \Rightarrow V_{S16}$
VSUBSWS	$V_{S32}-V_{S32} \Rightarrow V_{S32}$
VSUBUBM	$V_{U8}-V_{U8} \rightarrow V_{U8}$
VSUBUBS	$V_{U8}-V_{U8} \Rightarrow V_{U8}$
VSUBUHM	$V_{U16}-V_{U16} \rightarrow V_{U16}$
VSUBUHS	$V_{U16}-V_{U16} \Rightarrow V_{U16}$
VSUBUWM	$V_{U32}-V_{U32} \rightarrow V_{U32}$
VSUBUWS	$V_{U32}-V_{U32} \Rightarrow V_{U32}$
VSUMSWS	$\Re_+(V_{S32})+V_{\bullet S32} \Rightarrow V_{S32}$
VSUM2SWS	$\Re_+(V_{S32})+E(V_{S32}) \Rightarrow E(V_{S32})$
VSUM4SBS	$\Re_+(V_{S8})+V_{S32} \Rightarrow V_{S32}$
VSUM4SHS	$\Re_+(V_{S16})+V_{S32} \Rightarrow V_{S32}$
VUPKHPX	$U(V_{pixel}) \rightarrow V_{32}$
VUPKHSB	$U(V_{S8}) \rightarrow V_{S16}$
VUPKHSH	$U(V_{S16}) \rightarrow V_{S32}$
VUPKLPX	$L(V_{pixel}) \rightarrow V_{32}$
VUPKLSB	$L(V_{S8}) \rightarrow V_{S16}$
VUPKLSH	$L(V_{S16}) \rightarrow V_{S16}$
VXOR	$V \oplus V \rightarrow V$

Source: "AltiVec Technology Programming Environments Manual," Rev. 0.1, 11/1998

Sun VIS	Functionality
RDASR	read graphics status register (GSR)
WRASR	write graphics status register (GSR)
FPADD16	$F_{16}+F_{16} \rightarrow F_{16}$
FPADD16S	$L(F_{16}+F_{16}) \rightarrow F_{16}$
FPADD32	$F_{32}+F_{32} \rightarrow F_{32}$
FPADD32S	$L(F_{32}+F_{32}) \rightarrow F_{32}$
FPSUB16	$F_{16}-F_{16} \rightarrow F_{16}$
FPSUB16S	$L(F_{16}-F_{16}) \rightarrow F_{16}$
FPSUB32	$F_{32}-F_{32} \rightarrow F_{32}$
FPSUB32S	$L(F_{32}-F_{32}) \rightarrow F_{32}$
FPACK16	$(F_{16}<<GSR_SCALE)_{[15+GSR_SCALE..7]} \Rightarrow F_{U8}$
FPACK32	$(F_{32}<<GSR_SCALE)_{[31+GSR_SCALE..22]} \Rightarrow (F_{U8}<<8)$
FPACKFIX	$(F_{32}<<GSR_SCALE)_{[31+GSR_SCALE..15]} \Rightarrow F_{S16}$
FEXPAND	$F_{U16} \rightarrow F_{U32}$
FPMERGE	$F_8 \wedge \vee F_8 \rightarrow F_8$
FMUL8X16	$(\lfloor (L(F_{U8})\times F_{S16}) \rfloor)_{[7..23]} \rightarrow F_{S16}$
FMUL8X16AU	$(\lfloor (L(F_{U8})\times F_{S16}[1]) \rfloor)_{[7..23]} \rightarrow F_{S16}$
FMUL8X16AL	$(\lfloor (L(F_{U8})\times F_{S16}[0]) \rfloor)_{[7..23]} \rightarrow F_{S16}$
FMUL8SUX16	$(\lfloor (U_8(F_{16})\times F_{S16})_{[7..23]} \rfloor \rightarrow F_{S16}$
FMUL8ULX16	$(\sim_{32}(L_8(F_{16})\times F_{S16}))_{[31..16]} \rightarrow F_{S16}$
FMULD8SUX16	$(\lfloor (U_8(L(F_{16}))\times F_{S16})_{[7..23]} \rfloor \rightarrow F_{S16}$
FMULD8ULX16	$(\sim_{32}(L_8(L(F_{16}))\times F_{S16})) \rightarrow F_{S16}$
ALIGNADDR	$(R+R)_{[63..3]} \| 000 \rightarrow R$, $(R+R)_{[2..0]} \rightarrow GSR_ALIGN$
ALIGNADDRL	$(R+R)_{[63..3]} \| 000 \rightarrow R$, $-(R+R)_{[2..0]} \rightarrow GSR_ALIGN$
FALIGNDATA	$(F_8 \| F_8)\ \langle\langle 8*GSR_ALIGN \rightarrow F$
FZERO	$0 \rightarrow F_{63..0}$
FZEROS	$0 \rightarrow F_{31..0}$
FONE	$1 \rightarrow F_{63..0}$
FONES	$1 \rightarrow F_{31..0}$
FSRC1	$F \rightarrow F$
FSRC1S	$F_{31..0} \rightarrow F_{31..0}$
FSRC2	$F \rightarrow F$
FSRC2S	$F_{31..0} \rightarrow F_{31..0}$
FNOT1	$!F \rightarrow F$
FNOT1S	$!F_{31..0} \rightarrow F_{31..0}$
FNOT2	$!F \rightarrow F$
FNOT2S	$!F_{31..0} \rightarrow F_{31..0}$
FOR	$F \| F \rightarrow F$
FORS	$(F \| F)_{31..0} \rightarrow F_{31..0}$
FNOR	$!(F \| F) \rightarrow F$
FNORS	$!(F \| F)_{31..0} \rightarrow F_{31..0}$
FAND	$F \& F \rightarrow F$
FANDS	$(F \& F)_{31..0} \rightarrow F_{31..0}$
FNAND	$!(F \& F) \rightarrow F$
FNANDS	$!(F \& F)_{31..0} \rightarrow F_{31..0}$
FXOR	$F \oplus F \rightarrow F$
FXORS	$(F \oplus F)_{31..0} \rightarrow F_{31..0}$
FXNOR	$!(F \oplus F) \rightarrow F$
FXNORS	$!(F \oplus F)_{31..0} \rightarrow F_{31..0}$
FORNOT1	$!F \| F \rightarrow F$
FORNOT1S	$(!F \| F)_{31..0} \rightarrow F_{31..0}$
FORNOT2	$(F \| !F) \rightarrow F$
FORNOT2S	$(F \| !F)_{31..0} \rightarrow F_{31..0}$
FANDNOT1	$(!F \& F) \rightarrow F$
FANDNOT1S	$(!F \& F)_{31..0} \rightarrow F_{31..0}$
FANDNOT2	$(F \& !F) \rightarrow F$
FANDNOT2S	$(F \& !F)_{31..0} \rightarrow F_{31..0}$
FCMPGT16	$bv(F_{S16}>F_{S16}) \rightarrow R_{[3..0]}$
FCMPGT32	$bv(F_{S32}>F_{S32}) \rightarrow R_{[1..0]}$
FCMPLE16	$bv(F_{S16} \underline{<} F_{S16}) \rightarrow R_{[3..0]}$
FCMPLE32	$bv(F_{S32} \underline{<} F_{S32}) \rightarrow R_{[1..0]}$
FCMPNE16	$bv(F_{16}!=F_{16}) \rightarrow R_{[3..0]}$
FCMPNE32	$bv(F_{32}!=F_{32}) \rightarrow R_{[1..0]}$
FCMPEQ16	$bv(F_{16}==F_{16}) \rightarrow R_{[3..0]}$
FCMPEQ32	$bv(F_{32}==F_{32}) \rightarrow R_{[1..0]}$
EDGE8	m(R,R)→R partial store mask for F_8
EDGE8L	m(R,R)→R partial store mask for F_8 (little endian)
EDGE16	m(R,R)→R partial store mask for F_{16}
EDGE16L	m(R,R)→R partial store mask for F_{16} (little endian)
EDGE32	m(R,R)→R partial store mask for F_{32}
EDGE32L	m(R,R)→R partial store mask for F_{32} (little endian)
PDIST	$\Re_+(F_{U8} \nabla F_{U8}) \rightarrow \Sigma R$
ARRAY8	convert 8-bit 3D address to blocked byte address
ARRAY16	convert 16-bit 3D address to blocked byte address
ARRAY32	convert 32-bit 3D address to blocked byte address
STDA	partial store scalar store (8/16 bit) for big/little endian 64-byte block store for big/little endian
LDDA	scalar load (8/16 bit) for big/little endian 64-byte block load for big/little endian atomic quad load

Source: UltraSPARC IIi User's Manual, October 1997

Register Types

[Register Type]$_{[DataType]}$

R	integer register
F	floating point register
V	vector register
A	accumulator register
cc_i	condition code register bit i
mem	memory
imm_k	k-bit immediate value
∂	implicit register
U	unsigned integer
S	signed integer
FP	floating point
pixel	32-bit pixel format (AltiVec)
k	partition size in bits
•	scalar value (assume vector)

Data Communication

$_{[x..y]}$	bits x through y
I	immediate value
→	modulo result
⇒	saturated result
[i]	for each byte
‖	concatenate
∧∨	interleave
⟨⟨k	rotate k bytes left
⟩⟩k	rotate k bytes right
⇵	exchange elements
f()	partial permute function

Control Flow

| |	magnitude
==	equal
bv	bit vector
m	mask
{ }	set of allowed comparisons
{{ }}	arbitrary combinations of comparisons
<	less than
>	greater than
?	ordered
<>	bounds
≤	less than or equal
≥	greater than or equal
0	vector of zeros
br	branch

Width/Type Conversion

‖	concatenate
⌈	round up (to +∞)
⌇	round to nearest (even)
⌇	round to nearest (away from zero)
⌊	round down (to –∞)
∅	round to zero (truncate)
L	low half of bytes
U	upper half of bytes

Arithmetic

+	add
–	subtract
×	multiply
⊗	multiply by sign
√	square root
÷	division
/	reciprocal
$≈_k$	approximate to k-bits
$≈_{k1}$	first iter. Newton-Raphson
$≈_{k2}$	second iter. Newton-Raphson
log_2	base-2 logarithm
exp_2	2 raised to the power
Σ	positive accumulate
Σ̲	negative accumulate
⌈ ⌉	maximum
⌊ ⌋	minimum
$ℜ_+$	additive reduction
$ℜ_-$	subtractive reduction
$ℜ_×$	multiplicative reduction
| |	absolute value
!	negate
>>	right arithmetic shift
>>	right logical shift
<<	left (logical) shift
|	or
&	and
⊕	exclusive or
!	not
0…	zero fill
1…	one fill
L_k	lower k bits
U_k	upper k bits
⌇	round to nearest integer
C	carry out
E	even values
O	odd values
Λ	average
∇	sum of abs differences
→	assignment/modulo result
⇒	saturated result
⟩⟩k	rotate k bits right
⟨⟨k	rotate k bits left
$~_k$	sign extend to k bits

References

[Gepp98] Linda Geppert, "High Flying DSP Architectures," *IEEE Spectrum*, Vol. 35, No. 11, November 1998, pp. 53-56

[Hunt95] Doug Hunt, "Advanced Performance Features of the 64-bit PA-8000," *Proc. of IEEE Compcon*, San Francisco, California, March 5-9, 1995, pp. 123-128

[IAP803] Intel Corporation, "Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method," *Intel Application Note AP-803*, February 4, 1999, *http://developer.intel.com /vtune/cbts/strmsimd/appnotes.htm*, retrieved April 24, 2000

[Kuro98] Ichiro Kuroda, Takao Nishitani, "Multimedia Processors," *Proc. of the IEEE*, Vol. 86, No. 6, June 1998, pp. 1203-1221

[Pirs97] P. Pirsch, A. Freimann, M. Berekovic, "Architectural Approaches for Multimedia Processors," *Proc. of Multimedia Hardware Architectures*, *SPIE Vol. 3021*, San Jose, California, February 2-14, 1997, pp. 2-13

[Yates] Charles R.Yates, "Fixed-Point Arithmetic: An Introduction," *http://www.shadow.net/~yates/papers.htm*, retrieved April 24, 2000

[Zuras] Dan Zuras, "Floating-Point Fallacies," *http://www.labs.agilent.com/personal/Dan_Zuras/FPFallacies.html*, retrieved December 9, 2000