MIPS Extension for Digital Media with 3D

MIPS Technologies, Inc.

"http://www.mips.com"

March 12, 1997

MIPS Extension for Digital Media with 3D

Table of Contents:

Introduction

On October 21st 1996, *MIPS Technologies, Inc*., the core technology subsidiary of *Silicon Graphics (SGI)*, announced extensions to the MIPS Instruction Set Architecture (ISA) at the Microprocessor Forum. This material is a super-set of the presentation given by Earl Killian. It includes the philosophy and the history behind the extensions, and the goals in defining these extensions for a true digital media architecture.

At *SGI*, digital media has always implied 3D graphics as well as the more traditional forms, video, audio and signal-processing. As higher integration has allowed more functionality and better performance in single-chip designs, the *MIPS RISC architecture* has come to directly support digital media processing. The MIPS Instruction Set Architecture (ISA) is being extended in two distinct parts. The first is the extension to the baseline MIPS architecture, from *MIPS IV* (ref 3)to *MIPS V (ref 1)*. New instructions in MIPS V address 3D graphics algorithms. The second extension, an application specific extension (named *MDMX* for *MIPS Digital Media Extension*, read as "*MaDMaX*" ref 2) addresses audio, video and other signal processing applications such as fax modem. This document describes the key features and advantages of these extensions and are highlighted by some examples. The unique features of *MDMX* and *MIPS V* are useful in accelerating many algorithms, not just a single use. This document shows the mapping of how various algorithms can benefit from each feature. Finally, there is a comparison of the MIPS extensions with those proposed by other architectures.

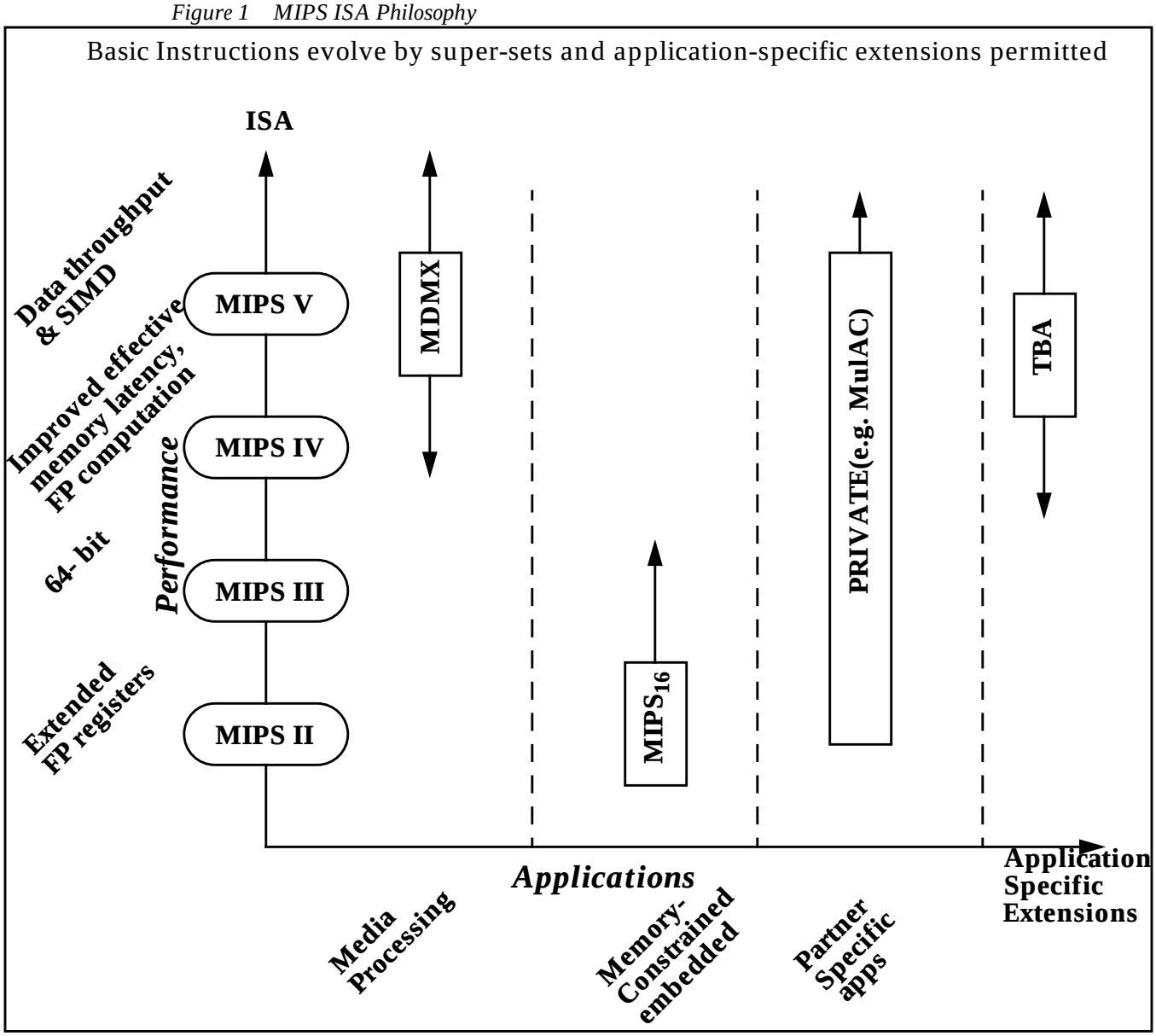
MIPS ISA philosophy and history

MIPS was one of the early pioneers of RISC technology. MIPS proved the technology by designing microprocessors based on a minimalist RISC ISA that has always provided state of the art performance. Whenever the market dictated, MIPS extended the ISA by adding instructions or features that allowed significant advances in performance. New instructions were only added when performance increases on a variety of codes was achievable. This model allowed for performance leadership while maintaining a fast, uncomplicated RISC architecture.

In recent years however, the huge explosion of 32-bit and 64-bit processors in embedded and consumer applications has exerted a new influence on ISA evolution. The requirement of supporting “natural” media types, 3D, video, audio, as well as the stringent cost requirements of consumer volume markets has augured for significantly different styles of instructions and processing of data. The MIPS open technology business model does allow MIPS’ semiconductor partners to add customer-specific extensions (e.g. 32-bit multiply-accumulate instruction) to the baseline ISA to increase performance of certain applications within certain vertical market segments. However, an application specific extension for digital media, implemented in a consistent fashion allows a processor to be tailored for digital media processing while maintaining compatibility, and without burdening the baseline ISA.

In this way, as Figure 1 shows, the ISA can be evolved through new baseline instructions for high-performance requirements, and Application Specific Extensions to support specific vertical market requirements such as digital media.

Figure 1 MIPS ISA Philosophy



The MIPS architecture originated with the MIPS I ISA, the R2000 being the first microprocessor designed, based on MIPS I. MIPS I ISA has been extended in a backward-compatible fashion three times. MIPS I was extended by providing 32 registers for double-precision math and adding instructions to give MIPS II. As applications required 64-bit addressing and data capability, MIPS III added 64-bit instructions. As processors became faster, memory latency became a bottleneck; also, 3-D visualization demanded more FP performance. MIPS IV added new instructions to boost performance in these areas. The latest additions come as a further super-set, MIPS V, for higher FP performance for 3D, and MIPS Digital Media eXtensions (MDMX) as an Application Specific Extension.

Another Application Specific Extension is also defined, for cost-sensitive embedded and

consumer applications. $MIPS_{16}$, announced jointly by LSI Logic Corporation and MIPS Technologies, Inc. are 16-bit instructions geared for having compressed code. These 16-bit instructions certainly use a lot less memory while providing the functionality required for the embedded market.

Table 1 summarizes the history of the different ISAs. It shows the year they were announced and the processors that implemented the ISA

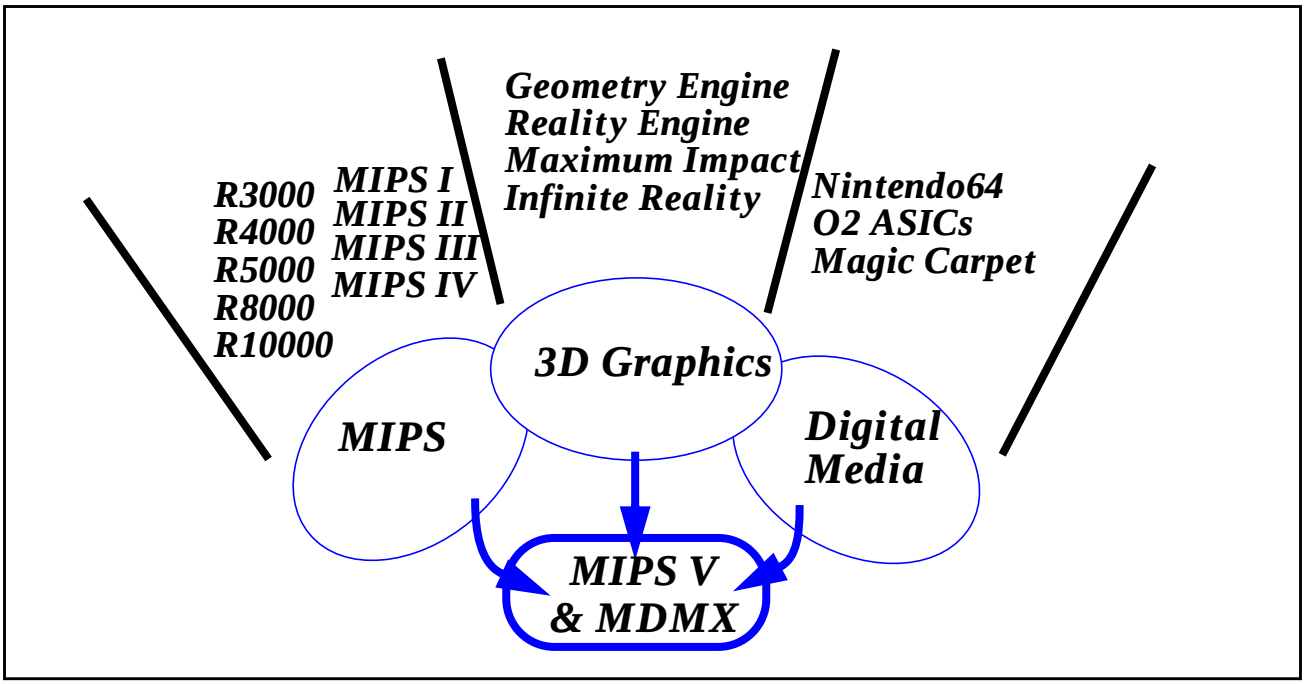
Table 1 Summary of the ISA history.

ISA	Year announced	Processors that implemented ISA
MIPS I	1984	R2000, R3000
MIPS II	1990	R6000
MIPS III	1991	R4000, R4200, R4300i, R4400, R4600, R4700
MIPS IV	1994	R5000, R8000, R10000
MIPS V	1996	TBA
MDMX	1996	TBA
$MIPS_{16}$	1996	LSI Logic Tiny RISC

SGI heritage in digital media

MIPS parent company, Silicon Graphics, Inc. has been an industry leader in the 3D-graphics and digital media technology since its founding. SGI's design experience of 3D-graphics and digital media hardware like "Geometry Engine^{tm}", "Reality Engine^{tm}", "Maximum Impact^{tm}", "Infinite Reality^{tm}", Nintendo64^{tm}, O2^{tm} ASICs and "Magic Carpet^{tm}" has been valuable to MIPS in defining the latest extensions, MIPS V and MDMX.

Figure 2 SGI heritage in digital media



MIPS' Goals for the ISA extensions

- Provide Digital Media support for embedded processors
- Target real algorithms, not trivial examples
- Realize performance improvement over vector/SIMD without promotion of storage format to computational format
- Keep it as simple as possible (but no simpler). Limit to features important in inner loops.
- Keep it fast.

Embedded systems and many low end general purpose systems are very cost sensitive. It is critical to find ways to reduce cost while increasing performance. One way to achieve this is to reduce dedicated hardware and transfer the tasks to be performed by the general purpose processor. To allow the general purpose processor to be operated efficiently, MIPS felt it was imperative to extend the Instruction Set Architecture (ISA) with the necessary digital multi-media specific instructions. The MIPS architecture is established as the leading RISC embedded architecture, and these extensions will cement that position.

MIPS selected and defined the instructions to extend the ISA in such a way that they would address real algorithms and not just over-simplified examples. Consider for example, blue-screening, other presentations might give an impression that conditional selection is all that is important, but in fact, a professional application like Ultimate^{tm}, shows that blending and other more complex operations are the key factors. As another example, the goal was set to improve performance of IEEE-compliant *Discrete Cosine Transform* and i's inverse *(IDCT)* and not just transforms of limited accuracy.

Another goal was to realize the full potential of a SIMD approach. Many architectures ISAs include *Single Instruction Multi Data (SIMD)* instructions. This instructions allow operations to be performed on multiple elements of one vector with corresponding elements of another vector. Operations like multiply or multiply-accumulate which provide wider results, require unpacking the data from the storage format to a larger computational format (promotion). However, there is limited usefulness to such formats because the results of the operations require larger width than the operands. This means that the 16-bit operands with results larger than 32-bits, need to be promoted to 64-bits; and thus, cannot benefit from this format. Computing in the larger format loses a factor of 2 or 4, and that's not counting the cycles unpacking and packing. MIPS' MDMX avoids this pitfall.

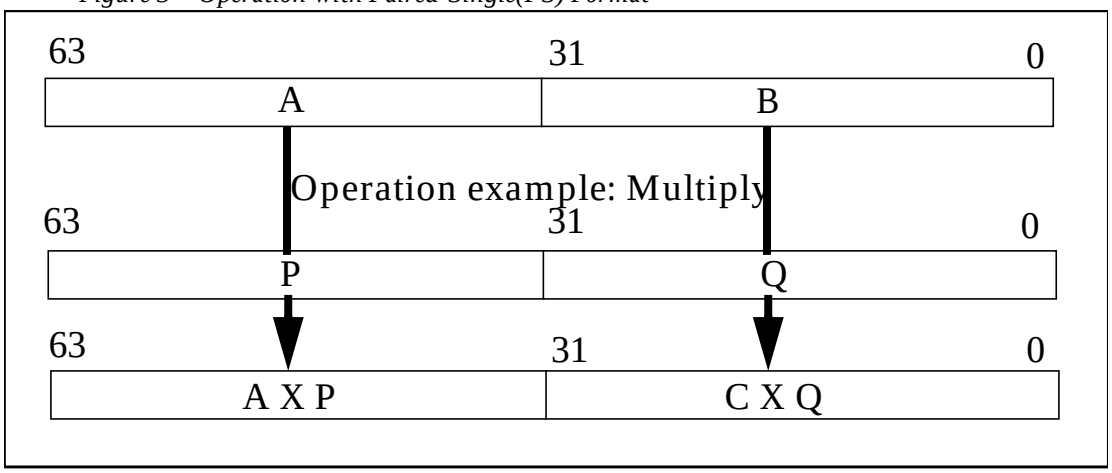
The extensions were selected to accelerate key areas that would most impact performance (for example, inner loops of the algorithms). It is also important to note that the performance not only implies number of cycles/task; but also on the clock frequency, and the selections considered the overall performance including the clock frequency.

- Paired-single (PS) format
- Unique paired-single (PS) opcodes:
  - *PS* to *PS*: *LUXC1, SUXC1, ALNV, PLL, PLU, PUL, PUU*
  - *PS* from two *S: CVT.PS.S*
- MIPS V paired-single provides 2-4 improvement on 3D graphics geometry, and also single precision engineering and scientific applications.
- Other miscellaneous changes include the exception for the PS format; the condition codes for the compare instructions in PS format; and the FP control registers.

Paired Single (PS) format

MIPS V is an extension to the core MIPS architecture. *MIPS IV ISA* includes four basic format for FP instructions. These are IEEE754 single (*S*), IEEE754 double (*D*), 32-bit integer (*W*) and 64-bit integer (*L*). *MIPS V* adds a new format called paired-single (*PS*). The PS format is a single-precision floating point format that allows one to double the performance of the operation of single-precision floating point data. As shown in figure 3, it operates on two single precision values that are stored in two halves of a 64-bit floating point register, and operate element-wise on the data (in effect, a two element vector operation). For this format, a 64-bit *Floating Point Register (FPR)* is interpreted as a vector of two single-precision floating-point numbers.

Figure 3 Operation with Paired-Single(PS) Format



2-4x performance improvement on 3D graphics and other single precision applications

The primary uses of the *paired-singles* (*PS*) are for graphics, where it can be used to implement 3D transforms and clip, and lighting and shading calculations. It can also be useful

in boosting the performance of floating point digital signal processing (e.g. hi-fidelity audio requiring greater precision or dynamic range than 16-bit integers can provide), and engineering calculations such as *matrix multiply* and the *Fast Fourier Transform (FFT)*; and also, in load / store limited algorithms like *SAXPY*. *PS* format can be implemented in a straight forward manner in an existing double precision datapath with only a few changes. As such it doubles the performance of a single precision operation at little cost.

SAXPY is chosen in this example to show the application of the *PS* format because of the wide spread familiarity; even though, modern techniques might make the use of *SAXPY* less relevant. *SAXPY* stands for *Single-precision A*X Plus Y*. This forms an inner loop of the Linpack benchmark (ref 4). The pieces of code for the inner loop using MIPS IV ISA and MIPS V ISA are shown for comparison in figure 4. This code calculates for all n elements, a(i)*x + y(i) and stores it into y(i). Since the vector a(i) is unaligned at the *DWd* boundary, one needs to operate on *words* using MIPS IV; however, one can use *luxc1(*unaligned DWd load) and *alnv.ps* instructions in the MIPS V ISA.

Figure 4 Comparison of SAXPY code with MIPS IV and MIPSV ISA using PS format.

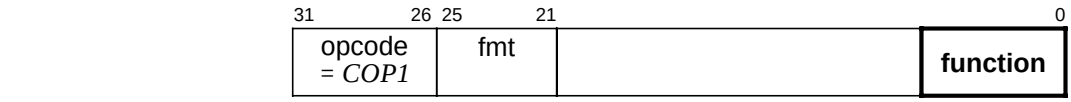
SAXPY using MIPS IV:

```
loop:
    lwxc1   f1, i(a)        # load a(i)
    lwxc1   f2, i(y)        # load y(i)
    madd.s  f2, f2, f1, f0  # calc. a(i)*x+y(i)
    swxc1   f2, i(y)        # store y(i)
    lwxc1   f3, i(ap4)      # load aplus4 (i)
    lwxc1   f4, i(yp4)      # load yplus4(i)
    madd.s  f4, f4, f3, f0
    swxc1   f4, i(yp4)
    lwxc1   f1, i(ap8)
    lwxc1   f2, i(yp8)
    madd.s  f2, f2, f1, f0
    swxc1   f2, i(yp8)
    lwxc1   f3, i(ap12)
    lwxc1   f4, i(yp12)
    madd.s  f4, f4, f3, f0
    daddiu  i, 16
    bne     i, n, loop
    sdxc1   f4, i(am4)      # store DWd at
                            # address aminus4(i)
```

SAXPY using MIPS V:
(Assume that f3 is loaded with un-aligned DWd before entering the loop. Also, assume that a(i) is a vector not aligned on DWd and y(i) is aligned on DWd).

```
loop:
    luxc1   f1, i(a)        # load DWd unaligned
    ldxc1   f2, i(y)        # load DWd aligned
    alnv.ps f3, f3, f1, b   # align the DWd.
    madd.ps f2, f2, f3, f0  # calc. a(i)*x+y(i)
    sdxc1   f2, i(y)        # store y(i)
    luxc1   f3, i(ap8)      # load aplus8(i)
    ldxc1   f5, i(yp8)      # load yplus8(i)
    alnv.ps f1, f1, f3, b
    madd.ps f5, f5, f1, f0
    daddiu  i, 16
    bne     i, n, loop
    sdxc1   f5, i(am8)      # store DWd at aminus8
```

Table 2 Instructions for which the PS format was added.†



function	bits 2..0							
bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1 001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2 010	*	MOV{T,F}	MOVZ	MOVN	*	RECIP	RSQRT	
3 011	*	*	*	*	*	*	*	*
4 100	CVT.S.PU	CVT.D	*	*	CVT.W	CVT.L	*	*
5 101	CVT.S.PL	*	*	*	PLL.PS	PLU.PS	PUL.PS	PUU.PS
6 110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7 111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Table 2 highlights the instructions for which the *PS* format is added. *PS* format was added for basic computational operations; such as multiply / add / subtracts, movf / t and for compare instructions. The *PS* format is left out for complex computational operations due to high cost of implementation; it is left out for conditional move instructions that are based on integer register conditions (*MOVZ, MOVN*) because separate conditions are required for each vector element; and it was left out for the conversions to / from other format because of incompatibility with source / target (e.g. no paired-double).

MIPS V also adds some *unique paired-single opcodes*. These are load / store 8-bytes without alignment trap (*LUXC1/SUXC1*), instructions for handling vectors that are not 8-byte aligned(*ALVN*), instructions for creating a paired single value from various halves of other paired-single halves (*PLL, PLU, PUL, PUU* where L is for lower half and U is for upper half); and creating a paired-single value from two singles (*CVT.PS.S*).

Table 3 Instructions for coprocessor COP1X with MIPS V additions highlighted

	2..0	**function (for opcode = COP1X)**						
bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	LWXC1[4]	LDXC1[4]	=	=	=	LUXC1[5]	=	=
1 001	SWXC1[4]	SDXC1[4]	=	=	=	SUXC1[5]	=	PREFX[4]
2 010	=	=	=	=	=	=	=	=
3 011	=	=	=	=	=	=	ALNV.PS[5]	=
4 100	MADD.S[4]	MADD.D[4]	=	=	=	=	MADD.PS[4]	=
5 101	MSUB.S[4]	MSUB.D[4]	=	=	=	=	MSUB.PS[4]	=
6 110	NMADD.S[4]	NMADD.D[4]	=	=	=	=	NMADD.PS[4]	=
7 111	NMSUB.S[4]	NMSUB.D[4]	=	=	=	=	NMSUB.PS[4]	=

Table3 shows the COP1X instructions that are in *MIPS IV ISA* with the *MIPS V* additions

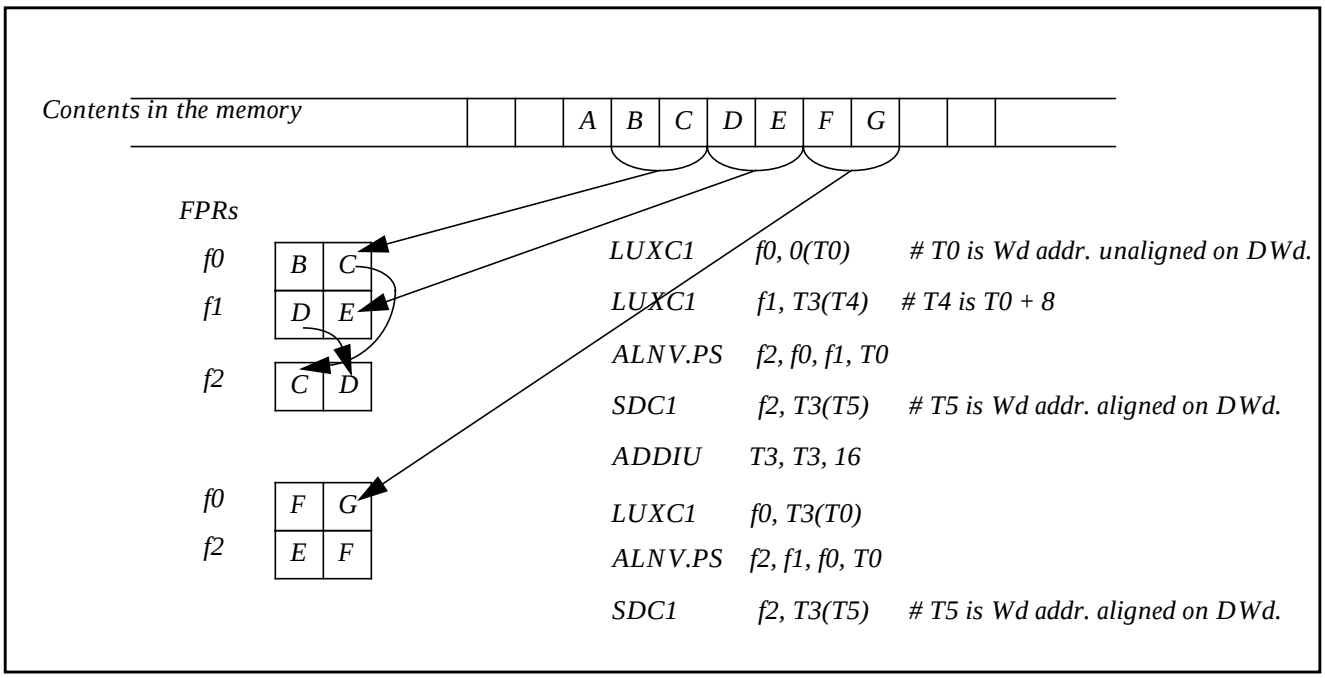
† Not every format supports every opcode. The opcode maps by format are shown in the "MIPS V ISA Manual" (ref 1)

highlighted. The unaligned load / store and the ALNV instructions are part of the COP1X opcodes and the *PLL, PLU, PUL, PUU* and *CVT.PS.S* are part of the COP1 opcodes.

Sample code using ALVN.PS

Figure 5 shows a simple program that loads data that are from an unaligned double-word (DWd) address in the memory, aligns them using *ALVN.PS* and saves it back to the aligned double-word address.

Figure 5 ALNV.PS used with LUXC1 to load a DWd aligned on any Wd boundary.



More detail on the above topics and other miscellaneous topics like condition bits, exception strategy are described in the MIPS V ISA Extension Manual (ref 1).

- Oct Byte (OB) format - 8 unsigned 8-bit Integer with 8 unsigned 24-bit accumulator
- Quad Half (QH) format- 4 unsigned 16-bit Integer with 4- unsigned 48-bit accumulator
- MDMX 192-bit Accumulator provides:
  - 2-4x performance improvement on important algorithms.
  - Sufficient precision for accurate computation.
- *vt select* to choose vector, scalar entry or scalar immediate

The *paired-single* extension added a two-element SIMD capability to the floating point hardware. The *MDMX* adds a four and eight element SIMD capability for integer arithmetic. The MDMX is intended to support video, audio and graphics pixel processing by introducing vectors of small integers. The *MDMX* shares 32 64-bit wide registers and the 8 Condition Code bits with the existing Floating Point Unit (FPU). Data is moved between the shared register file and memory with existing FP load and store *double word* (*LDC1, SDC1, LDXC1, SDCX1, LUXC1* and *SUXC1*) operations; and between FPRs and integer registers (GPRs) using *DMFC1* and *DMTC1* instructions.The use of floating point registers as the home of the MDMX operands, rather than the integer registers, provides more data storage, as the integer registers are used for addresses and the floating point registers are used for MDMX data.

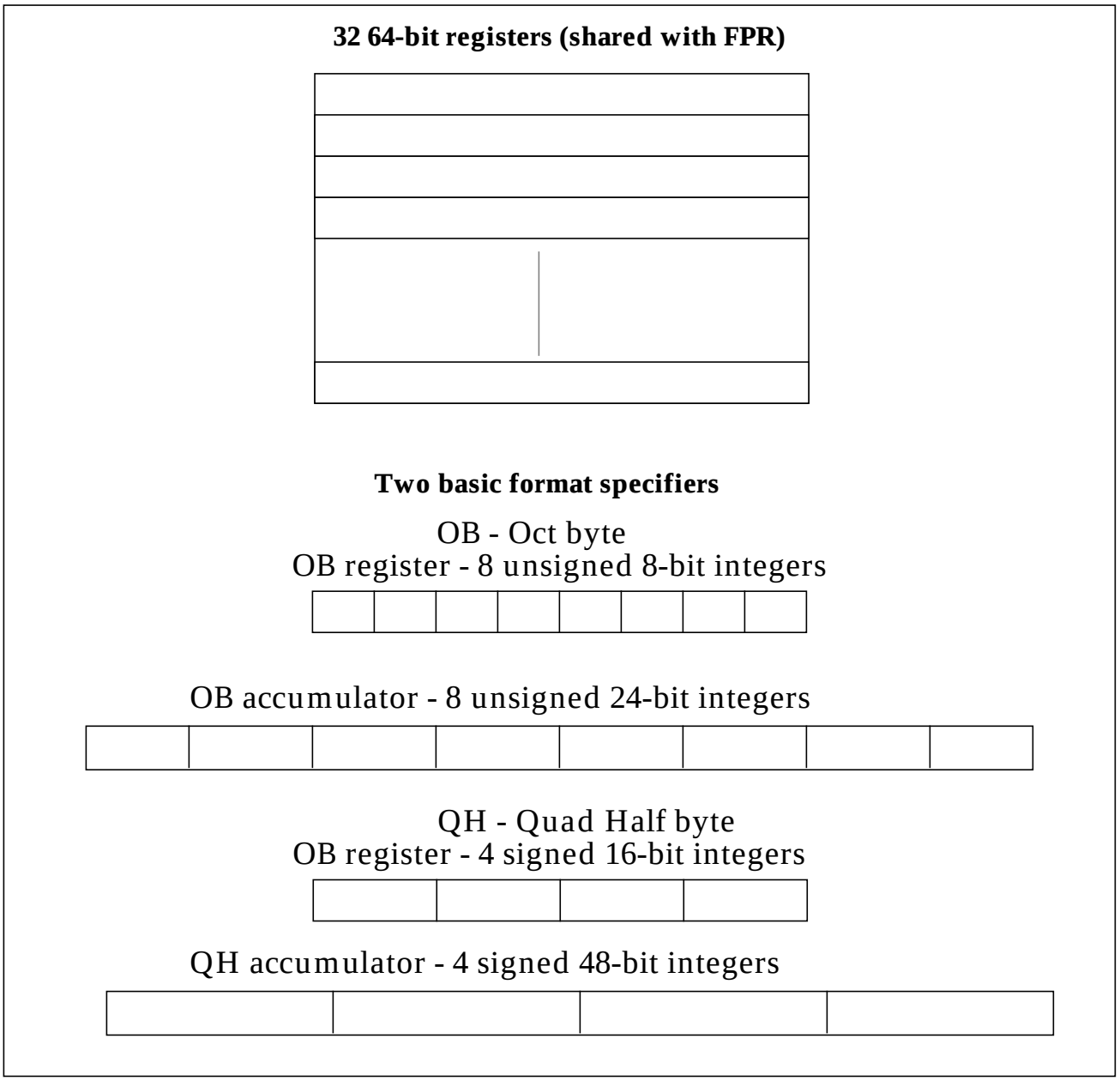
Two formats

In each *MDMX* instruction is a format specifier, of which two are currently defined (figure 6). In these new format specifications, the registers are interpreted as Quad Half (QH) and Oct Byte (OB). In QH the 64-bit FPR is interpreted as a vector of 4 signed 16-bit integers; and in OB format it is interpreted as a vector of 8 unsigned 8-bit integers.

A 192-bit accumulator

The rest of the MDMX architecture is centered around a vector accumulator. The MDMX has a private 192-bit Accumulator register. The format of the accumulator is determined by the format of the elements accumulated. In case of the OB, the accumulator formatted as 8 unsigned 24-bit slices; and in the case of QH, it is formatted as 4 48-bit slices. Figure 6 shows the registers and the accumulator interpreted in the two formats.

Figure 6 Two basic formats in the MDMX

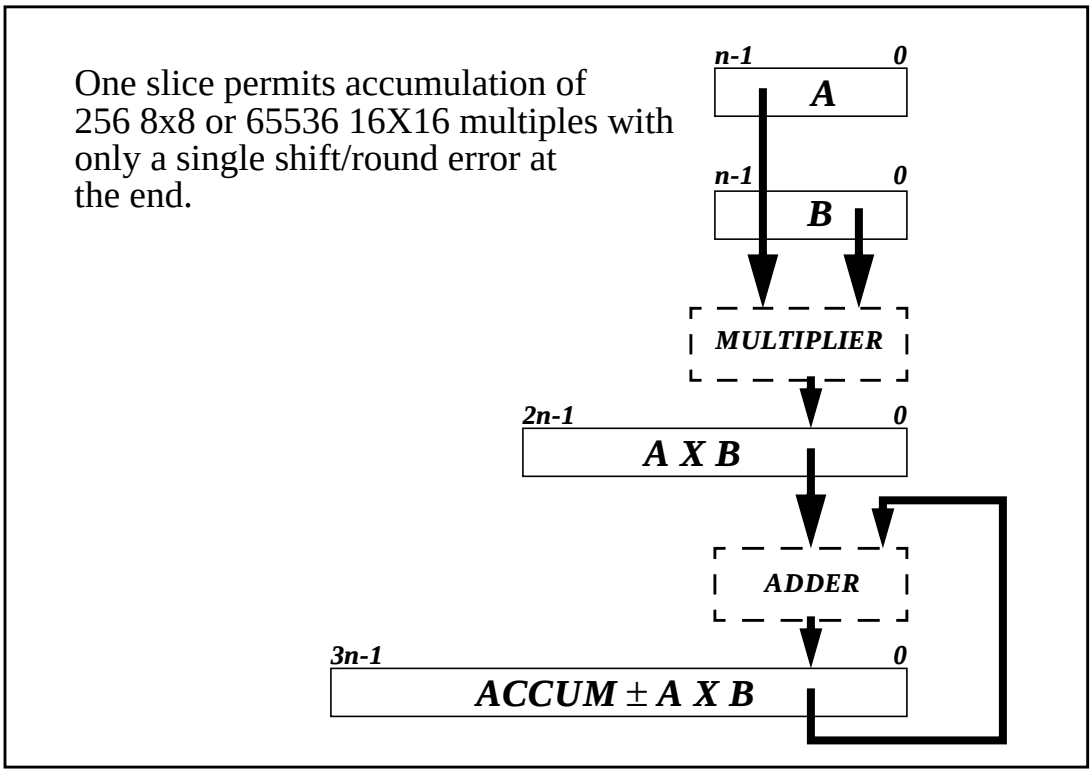


2-4x performance improvement on important algorithms

The accumulator allows MDMX to realize the full potential of the SIMD paradigm by eliminating the need for promotion of data from a storage precision to a larger computation precision. In other words, one can take full advantage of the *QH* (Quad-Half) or *OB* (Octal Byte) format and do multiple accumulations of results with a single rounding at the end. Rounding between each accumulate would certainly reduce accuracy of the final result. This greater accuracy due to additional precision helps implementation of an IEEE-compliant IDCT. As

exemplified in figure 7, a product of n-bit quantities is 2n bits; moreover, 3n-bit wide accumulator permits accumulation of 2^n nxn multiples with only a single shift / round approximation at the end. The wider accumulator also negates the need for *Sum of Absolute Difference (SAD)* instruction for motion search by enabling the higher precision *Sum of Difference Squared (SDS)* computation. The advantages of using *SDS* are discussed later. In using an accumulator, MDMX is similar to many DSP architectures, but unlike extensions such as MMX™‡, VIS™*, and PA-RISC2™. Unlike DSPs, MDMX has a SIMD accumulator for added performance.

Figure 7 MDMX Accumulator.



Sufficient precision for accurate computation

As discussed earlier, an architecture which has the accumulator the same width as the register (or uses destination registers with no accumulator) requires that operands be promoted to the same width as the width of the result. Obviously, this means loss of performance due to extra cycles required for promotion and due to reduction in the vector length. Figure 8 shows the computations with MDMX technology and with those that do not have wide enough accumulator. Depending on whether the promotion is required to 32-bit or 64-bit, the vector length is reduced from 2 to 4 times. Note that in the example without MDMX technology, a 16-

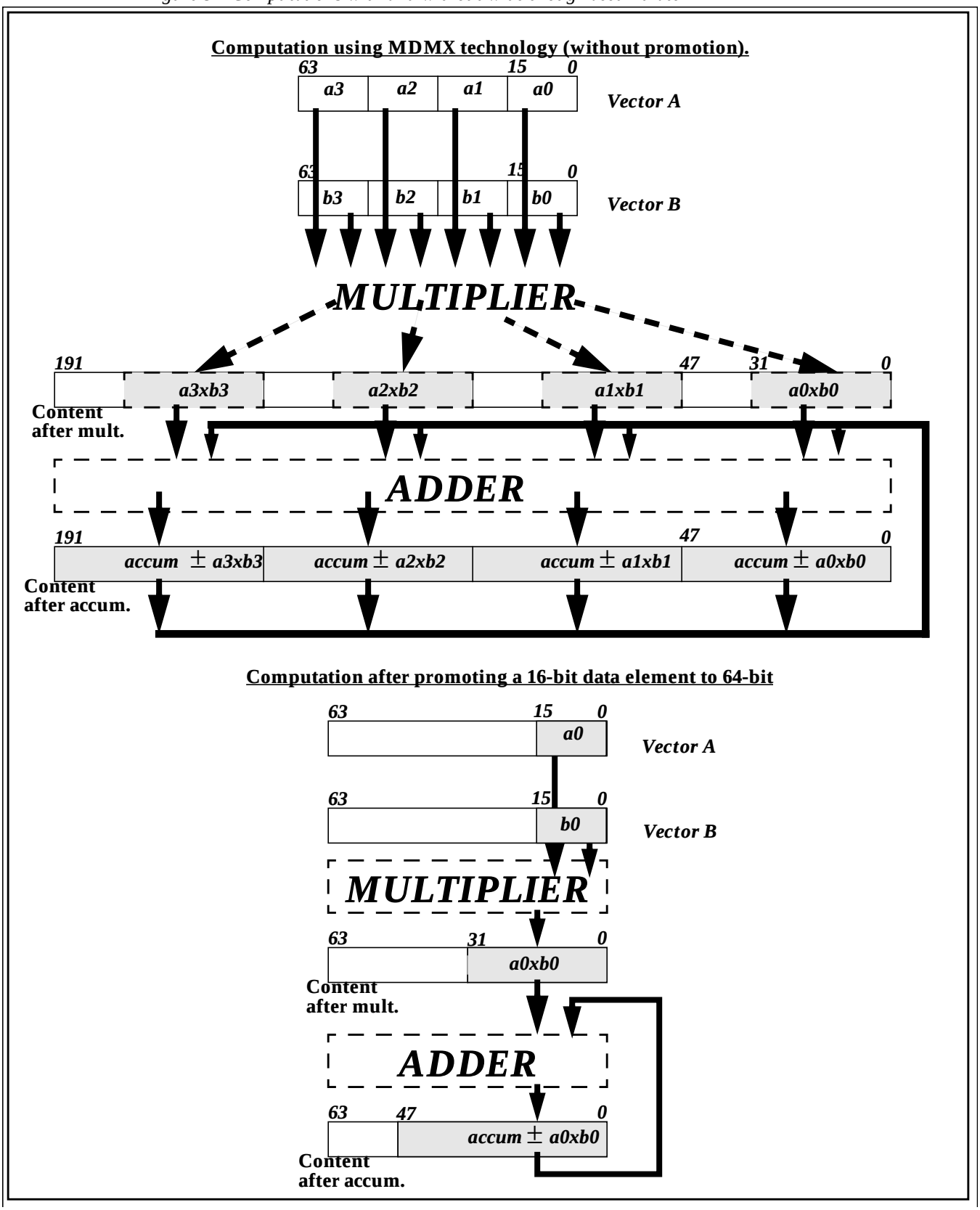
‡ MMX is a trademark of Intel Corporation

* VIS is a trademark of Sun Microsystems Corp.

PA-RISC2 is a trademark of Hewlett Packard Corp.

bit data needs to be promoted to 64-bit (i.e. >32 bits) because the result of a 16 x 16 multiply added to a previous 16 x 16 multiply could be as large as 33 bits wide.

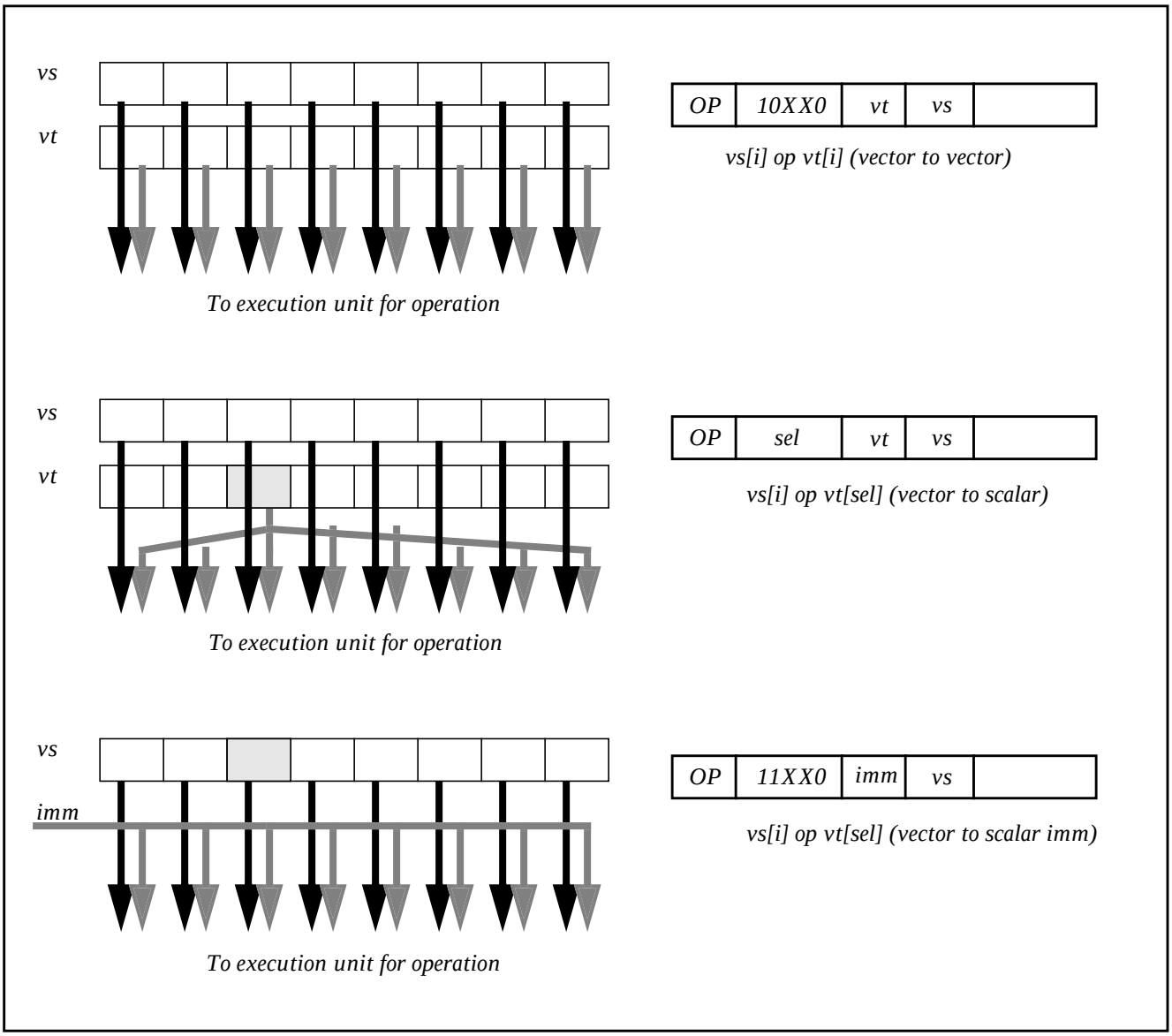
Figure 8 Computations with and without wide enough accumulator



vt select field

MDMX instructions includes a unique vt *select* field that allows selection of either a vector or scalar or an immediate value for the second operand. Operations on a pair of source register can be performed element-wise or one element of the vt vector can be replicated and used with each element of the first vector; or finally, the *select* field can direct the *vt* field to be used as an *immediate* value, as shown in figure 9.

Figure 9 Selection of vector, scalar or immediate operand using vt field in the MDMX instructions.



In many vector algorithms (e.g. motion search), the need for vector / scalar forms of operation arises naturally. Vector instruction sets, such as Cray's, include both vector / vector and vector / scalar forms of all operations. Vector / scalar operations are found to be critical in

implementing extremely efficient motion estimation searches, and a parallel audio filter, to name just a few.

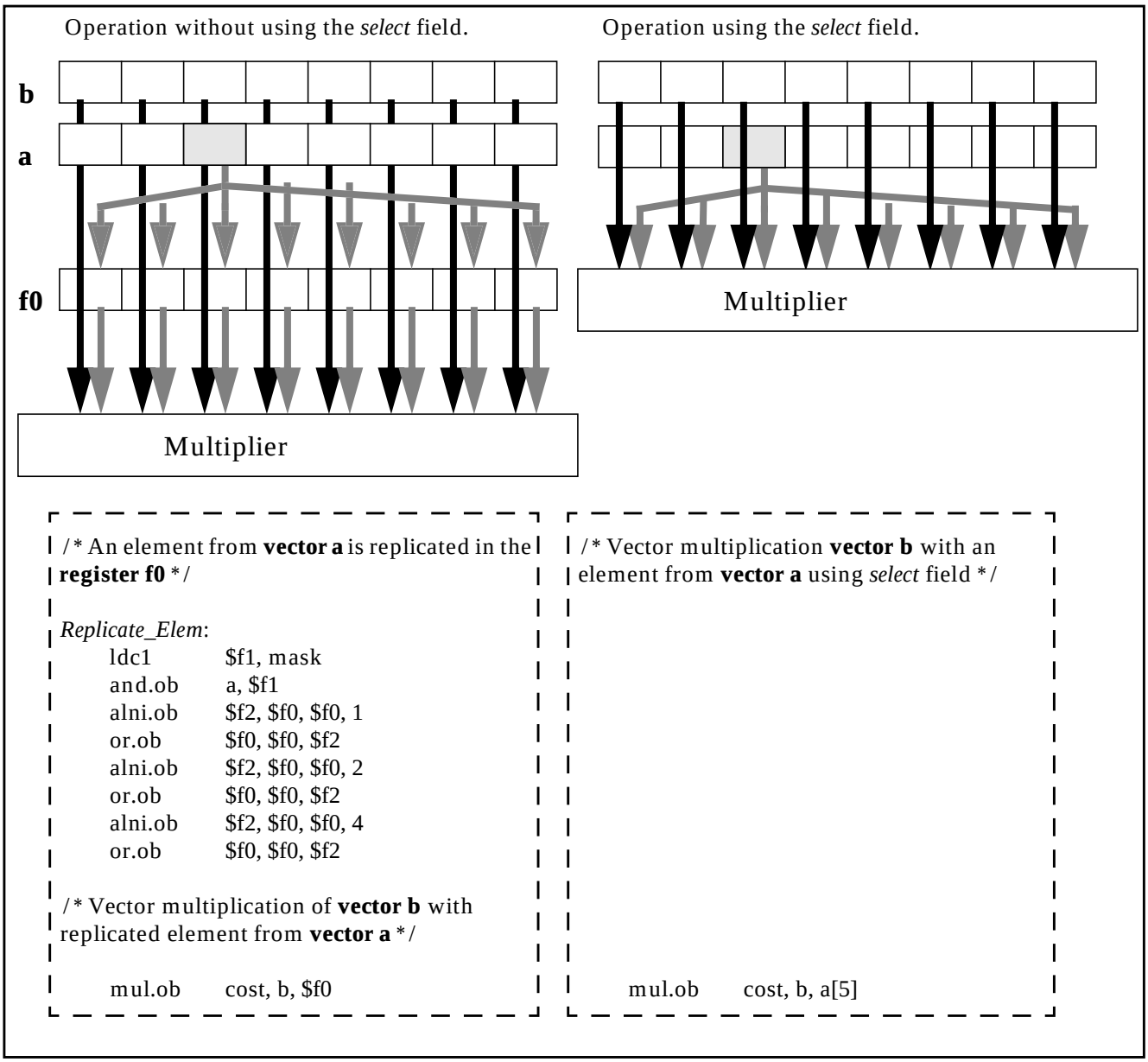
Having the feature *vt select* in the opcodes, saves registers because the scalar element is selected from a vector existing in a register, without requiring the replication of the element into another register. In addition the element select feature allows us to pack several constants into a single register and select one element for *immediate* operand, thereby saving registers. This feature is helpful in implementing *IDCT* for example.

Part of the inner loop of the *motion search* algorithm, has the following equation:

*Cost [dy, dx] = Cost - (a[by, bx] * b[by+dy, bx+dx])*

Eight elements of this equation (dy, dx+0 to dy, dx+7) can be computed in parallel using *OB*-format to reduce the computation cycles. This requires the product of one of the elements of **vector a** (scalar) with eight different elements of **vector b** in one cycle. Figure 10, shows the savings in cycles and registers due to the *select* field in MDMX opcodes. The left-side shows a piece of code (*Replicate_Elem*) that exemplifies a way to take an element of **vector a** and replicate it eight times in a register f0 which is used as one of the operands for vector multiplication. The right side uses *vt select* field and do not need the *Replicate_Elem* code.

Figure 10 Savings due to the use of select field in the MDMX opcodes



About MDMX instruction set

There are two parts of the MDMX instruction set. One part is a conventional register to register set of instructions. For example operations such as add, subtract, logical, shift, min, and max operate on two source registers, the result is clamped to the destination precision, and stored into destination register. Other instructions exist for handling unaligned data, permuting, comparing, or doing conditional selection. The MDMX opcodes were added as part of the coprocessor 2 (COP2), as shown in table 4 and are described in more detail in the ISA manual. Note that instructions to load accumulator with the result of an operation, instructions that add or subtract from the accumulator, a series of instructions that read back the accumulated value

with a shift, round, and clamp, and instructions for reading and writing all the vector accumulator in full precision are also included in the set.

Table 4 MDMX opcode encoding.

	bits 2..0	function (for opcode = COP2)							
	bits 5..3	0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	000	MSGN	C.EQ	PICKF	PICKT	C.LT	C.LE	MIN	MAX
1	001	†	†	SUB	ADD	AND	XOR	OR	NOR
2	010	SLL	†	SRL	SRA	†	†	†	†
3	011	ALNI.OB	ALNV.OB	ALNI.QH	ALNV.QH	†	†	†	SHFL
4	100	RZU	RNAU	RNEU	†	RZS	RNAS	RNES	†
5	101	†	†	†	†	†	†	†	†
6	110	MUL	†	MULS{,L}	MUL{A,L}	†	†	SUB{A,L}	ADD{A,L}
7	111	†	†	†	†	†	†	WAC	RAC

For more detail see the MDMX ISA Extension Manual (ref 2).

Features selectively left out of the extensions for optimization.

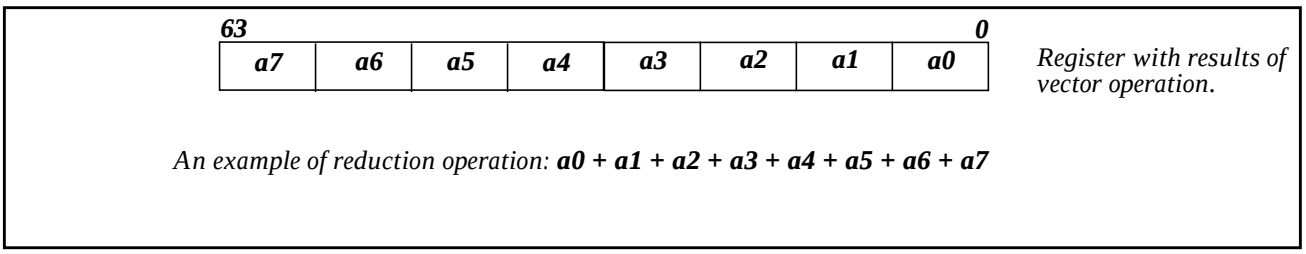
- *Reduction operators*
    - Not needed in inner loops
    - Often not needed at all if parallel computations used and leaving them keeps vector datapaths parallel
    - ALN/op allows fast combination after loop
- *Sum of absolute difference (SAD)* for motion search.
    - use Sum(a-b)^2 instead. This provides higher quality than SAD.
- Accumulator makes the 32-bit and 64-bit integer operations unnecessary.

There are a few things that were purposely left out from the extensions because they didn't fit the initial criteria of simplicity and optimization. *Reduction operators* and *Sum of absolute difference (SAD)* are two such features.

Reduction Operator

A *Reduction operator* is one that doesn't operate pair-wise between two vector operands but instead combines the elements of a single vector. An example is a sum of the elements of a vector shown in figure 11. In general, such operations were not found to be needed in inner loops, and are often not needed if parallel forms of algorithms are used; i.e. if operation is performed on multiple vectors in parallel by packing elements from different vectors in one register. In addition two or four uses of the ALN instructions allows such a computation to be done when it is needed, typically after the end of a loop.

Figure 11 Reduction operation



Sum of absolute difference (SAD)

Compression algorithms for sequence of images (e.g. movies & video conferences) use a technique called *motion estimation* to reduce temporal redundancy. Temporal redundancy is observed in a movie where large portions of an image remain unchanged from frame to adjacent frame. In many situations, such as a camera pan, every pixel in an image will change from frame to frame, but nearly every pixel can be found in a previous image. The process of "finding" copies of pixels in previous (and future) frames is called *motion estimation*. Video compression

standards such as H.261 and MPEG 1 & 2 allow the encoder (compression engine) of an image to remove redundancy by specifying the motion of 16X16 pixel blocks within an image. The image being compressed is broken into blocks of 16X16 pixels and for each block in one image a search is carried out for matching blocks in other images that are in the sequence being compressed. Two measures are typically used to determine the match. One is the *sum of absolute difference (SAD)* which mathematically written as, Sum(Sum $|a_i-b_j|$) and the other is the *sum of differences squared (SDS)* which is mathematically written as Sum(Sum$(a_i-b_j)^2$).¤ The *SAD* measure is easy to implement in the hardware; however, the SDS requires greater precision to generate the results and is generally accepted to be of superior quality.

Considering the innermost loop of the *motion search* algorithm, if Sum$(a-b_i)^2$ less than *Sum*$(a-b_j)^2$, then Sum(b_i^2) - 2∗Sum(b_i) should also be less than Sum(b_j^2) - *Sum*(ab_j). Since Sum(b_i^2) can be computed outside the loop, the operations in the loop are reduced to just accumulation of products which is an existing high performance operation! In addition, the MDMX's vector accumulator handles higher precision requirement of the *SDS*. Implementation of the motion search algorithm using *SDS* is done without the need for a *reduction operator*.

No need for 32-bit and 64-bit integer operations

Finally, the 32-bit and 64-bit integer formats in MDMX are left out. Generally the extra precision of the vector accumulator makes these data formats unnecessary (since no promotion is necessary). Also, for data that requires more than 16 bits of precision (e.g. hi-fidelity audio), the *paired-single* precision floating point extension is usually the best choice, not a 32-bit integer format.

¤ For both SAD and SDS, the inner sum is for all j's and the outer sum is for all i's.

MDMX Features and the applications

Table 5 provides a map of the key MDMX features and the algorithms in which the features can be used. Note that the unique features of MDMX are used in the inner loops of a few important algorithms. In most cases, the unique features are not used by just one or two algorithms, but several. This demonstrates that these features are general purpose and have wide applicability.

Table 5 MDMX Features Map

	Vector 8-bit MUL/ ADD w/ 24-bit accum.	Vector 16-bit Mul/ ADD w/ 48 bit accum	Byte or 16-bit align	Shift/ round	Vector element select	Sat. arithmetic	Shuffle (transpose)
Motion Search (Encode)	√	—	√	—	√	—	—
Motion Vectors (decode)	√	—	√	√	—	√	—
IEEE Compliant IDCT	—	√	—	√	√	√	√
Audio Filter	—	√	√	√	√	√	—
Ultimatte^{tm}	—	√	—	√	√	√	—

Feature comparisons

Table 6 provides comparisons of MIPS V / MDMX with the extensions to several other instruction sets architectures. For each feature, the best of breed among the extensions are highlighted. One thing stands out: the *paired-single* and MDMX have significant functionality and performance advantages compared to the other architectures. Only MIPS has the SIMD single precision floating point for 3D graphics. Only Mad Max has the extra precision vector accumulator and vector element select. Mad Max has greater multiply / accumulate capability than the others. And Mad Max has a high-quality motion search capability without any special purpose instructions.

Table 6 Feature comparisons with extensions to other existing ISAs

Feature	MIPSV/ MDMX	Intel MMX‡	Sun VIS*	HP PA-RISC2†
Paired-single	Yes	No	No	No
Operands	3-4	2	3	3
Vector registers	32 (FP)	8 (FP)	32 (FP)	31 (Int)
Vector/Scalar Element	Yes	No	No	No
Accumulator	Yes	No	No	No
Saturation	Yes	Yes	Pack only	Yes
Compare results	8 CCs	MMX register	Int register	no compare
Integer Storage formats				
8 8-bit	Yes	Yes	Yes	No
4 16-bit	Yes	Yes	Yes	Yes
2 32-bit	No	Yes	No	No
Integer Computation format	8 8/24-bit	8 8-bit		
	4 16/48bit	4 16-bit	4 16-bit	4 16-bit
		2 32-bit	2 32-bit	
Multiply	8 8bit x 8bit 4 16bit x 16bit	4 16bit x 16bit	4 8bit x 16bit	No No
Multiply/Add	Yes	4 mul / 2 add	No	No
Align	Yes	No	Yes	Yes
Shift/Round	Yes	No	No	No
Shifts	Yes	Yes	No	Yes
Expand/Pack	Yes	Yes	Yes	No
Shuffles	8bit + 16bit	No	interleave only	16bit only
SAD	No	No	Yes	No

‡ MMX is a trademark of Intel Corporation.

* VIS is a trademark of Sun Microsystems, Inc.

† PA-RISC is a trademark of Hewlett Packard, Inc.

Summary and Conclusion

This paper presents the extensions to the MIPS ISA for digital media. The extensions are in two parts. First one extends the core MIPS ISA to MIPS V and the second one, MDMX is an independent part. The documentation began with the goals for defining the extensions, followed by the description of the key features of the extensions and their advantages. The MIPS V includes the *paired-single* format for single precision floating point operations. The MDMX not only provides SIMD operations but a vector accumulator which allows one to take full advantage of the SIMD paradigm by eliminating the need for promotion of data from a storage precision to a larger computation precision. Moreover, a *vt select* field is included to allow vector / scalar or vector / immediate operations; there by saving registers and making the code very efficient.

The reasons for leaving out certain features were also discussed. A mapping of features with algorithms showed that unique features addressed several algorithms; thus, raising the applicability of these extensions. A chart was provided to compare the features of MIPS extensions with the extensions of other existing architecture.

The MIPS instruction set extensions provide superior performance and support a full spectrum of applications. The MDMX provides excellent support for integer DSP applications including video and audio, and the MIPS V *paired-single* extension provides excellent floating point DSP capability for things like 3D and audio, as well as single precision engineering applications.

References

1 MIPSV ISA Extension Manual.

2 MDMX ISA Extension Manual.

3 MIPS IV ISA Manual.

4 John Hennessy & David Patterson[1990], “Computer Architecture A Quantitative Approach,” Morgan Kaufmann Publishers, Inc., San Mateo, CA