AltiVec™: Bringing Vector Technology to the PowerPC™ Processor Family

Jon Tyler, Jeff Lent and Anh Mather, Huy Nguyen
Motorola Incorporated
6200 Bridgepoint Pkwy,
Bldg.4, Austin, TX 78730, USA

Abstract

Motorola's AltiVec™ Technology provides a new, SIMD vector extension to the PowerPC™ architecture. AltiVec adds 162 new instructions and a powerful new 128-bit datapath, capable of simultaneously executing up to 16 operations per clock. AltiVec instructions allow parallel operation on either 8, 16 or 32-bit integers, as well as 4 IEEE single-precision floating-point numbers. AltiVec technology includes highly flexible "Permute" instructions, which give the data re-organization power needed to maintain a high level of data parallelism. Fine grained data prefetch instructions are also included, which help hide the memory latency of data hungry multimedia applications.

All of these features add up to a dramatic performance improvement with the first implementation of AltiVec technology: routines written with AltiVec instructions can execute significantly faster, sometimes by a factor of 10 or more, than traditional scalar PowerPC code. Yet AltiVec technology is flexible enough to be useful in a wide variety of applications.

1: Introduction

Microprocessor performance has always been a difficult metric to measure. Clock frequency is often correlated to, and confused as, microprocessor performance. But, as clock frequency increases, this correlation becomes less clear. If a microprocessor can run at twice the frequency, but has twice the pipeline stages, the performance gain is hazy at best. Often a microprocessor architect has to make a trade-off between the clock frequency, and the number of instructions per cycle(ipc).

In addition to clock frequency, design teams have come up with numerous ways of increasing microprocessor performance. One popular way is to add multiple execution units. Motorola has increased the performance of PowerPC Microprocessors with the addition of a vector execution unit. The vector execution unit enables the processor to execute Single Instruction Multiple Data (SIMD) extensions of the PowerPC architecture and is called AltiVec technology [1].

The first microprocessor to include the AltiVec technology is Motorola's fourth generation PowerPC microprocessor[2]. The addition of AltiVec technology enables the PowerPC microprocessors to concurrently address high bandwidth data-processing and algorithmically intensive applications.

2: Organization

AltiVec Technology adds 162 new instructions and a new 32 entry, 128 bit vector register file to the PowerPC microarchitecture. The operands of these new instructions are 128-bit wide vectors, and can be subdivided into either 16, 8-bit integers, eight, 16-bit integers, four, 32-bit integers, or four, 32-bit single-precision floating-point numbers as seen in Figure 1. The instructions operate on either one, two, or three operand input vectors from the Vector Register File (VRF). Vector instructions are dispatched to the Vector Execution Unit in parallel with instructions dispatched to the scalar fixed-point, scalar floating-point, or load/store units. Data for dispatched vector instructions is stored in the VRF, separate from the scalar register files, and results are placed on separate Vector Rename Buses.

Motorola's fourth generation PowerPC microprocessor has a short pipeline and dual dispatch design similar to the current PowerPC 750™ microprocessors [3]. It adds to that a powerful new memory subsystem, a faster floating point unit, and the new vector execution unit and register file. By including a separate register file, AltiVec technology supports a 128-bit datapath without sacrificing the total number of registers available to the vector execution units. This simplifies compiler instruction scheduling, minimizes stalls in the pipeline caused by resource limitations, and eliminates context switching performance penalties when mixing vector and scalar instructions.

AltiVec operations include integer addition and subtraction (with and without carry-out); multiply odd and even (eight and 16 bit only); multiply and add (high or low, with or without rounding, 16 bit only); multiply sum (eight or 16 bit only); average; sum across (32 bit only); sum across partial; logical AND, OR, XOR, AND with complement, and NOR; element rotate left; ele-

0-7803-5258-0/99 $10.00 © 1999 IEEE

ment shift left or right; 128-bit shift left or right; compare equal-to; compare greater-than; conditional select; shift left or right by octet, shift left double (two 128-bit concatenated vectors) by octet; maximum; and minimum. In addition, many of the operations can act on signed or unsigned operands and can produce modulo or saturated results. The multiply and add high can be used for fractional fixed-point numbers instead of integers.

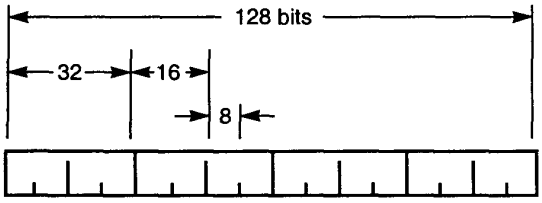


Figure 1: Vector Operand Format

Operations on single-precision, floating-point numbers include addition and subtraction; multiply-add; compare equal-to; compare greater-than; compare greater-than or equal-to; compare bounds; maximum; minimum; reciprocal estimate; reciprocal square root estimate; log2 estimate; 2 raised to the exponent estimate; negative multiply-subtract; round to floating-point integral (with selective rounding modes); convert from fixed-point word; and convert to fixed-point word with saturation. Because there is no separate multiply instruction, in order to perform multiply-only operations, a register with zeros must be used for the add operand of a multiply-add instruction.

The vector execution unit, as implemented, can be thought of as two additional execution units of the PowerPC microprocessor: the Permute Unit and the Vector Arithmetic Logic Unit (Vector ALU) as shown in Figure 2. The Vector ALU can be further divided into three separate execution units: the Vector Simple-fixed Unit (VSFX), the Vector Complex-fixed Unit (VCFX), or the Vector Floating-point Unit (VFPU). Vector load and stores are handled by the existing scalar load/store unit.

Vector instructions are assigned one of six Vector Rename Buses at dispatch time, and sent to either the Permute Unit, or Vector ALU. The Permute and Vector ALU may both be dispatched to simultaneously. If a required vector register is not available for the dispatched instruction, the instruction is held in one of two, single entry reservation stations. One reservation station is for the Permute unit, while the other handles the Vector ALU. When the needed resources become available, the instruction is then sent to the corresponding execution unit.

Both the Permute and Simple Fixed unit will then execute the instruction in one cycle, while the Complex unit will execute in three cycles, and the VFPU will execute in four. Each execution unit places its results on the assigned Rename Bus. These rename buses have "keeper" circuits that allow them to act as rename registers. Data on a rename bus has two possible destinations: it can either be transferred back to the register file for later computations or memory accesses, or it may be forwarded directly to the inputs of one of the four vector execution units for subsequent instruction execution, thereby decreasing the latency time for data dependencies between instructions.

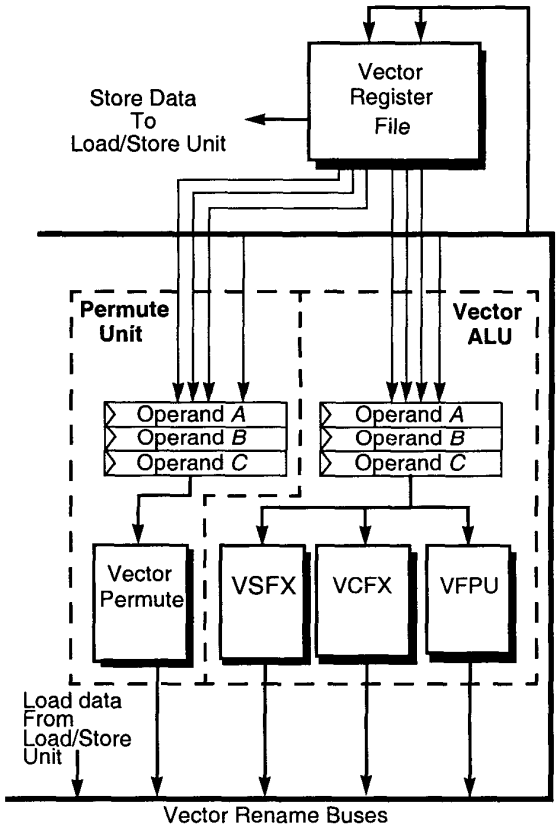


Figure 2: Vector Execution Unit Block Diagram

3: The Vector Permute Unit (VPERM)

As vector units get wider, more work must be done to line up data before the vector unit can operate on it. Here the permute unit provides all the power necessary. Permute instructions shuffle data from any of 32 byte

locations in 2 operands to any of 16 byte locations in the destination register. Permute instructions all have a one cycle latency, and can be dispatched and executed in parallel with other execution units.

It can be seen in Figure 3 that a major portion of the Permute unit is a 32 byte to 16 byte crossbar switch network. Each byte of the result vector can receive a data byte from any of the bytes of Operand *A* or *B* depending on the executing instruction. The result mux is used to determine whether the instruction being executed gets its result from the crossbar network, a saturated value, or from a pixel operation. Saturated values are derived from pack instructions having source data-size that is not representable in the target. Pixel pack and unpack instructions are bit-level operations that can not be generated by the crossbar network; dedicated pixel logic is present that bypasses the crossbar network to the result mux.

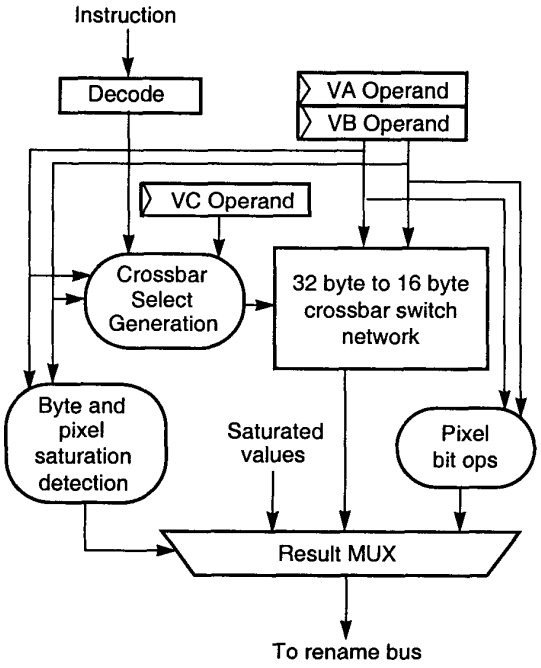


Figure 3: Permute Unit Block Diagram

The vector permute unit has many powerful applications. For example, the vperm instruction can be used by itself as a small parallel table lookup, as shown in Figure 4. The instruction vperm, for each destination byte, selects a byte in VA or VB, depending on the value in the control vector VC. (See the *AltiVec Programming Environments Manual* for a complete description of all AltiVec technology instructions [4]).

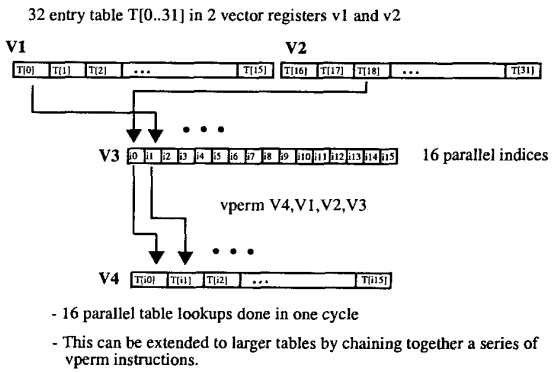


Figure 4: Parallel Table Lookup

Although there is no explicit Matrix Transpose instruction, Figure 5 shows how the permute unit can do a 4x4 word transpose with just 8 single-cycle instructions.

The permute unit is frequently used in almost all AltiVec technology code, setting up data for other powerful vector operations.

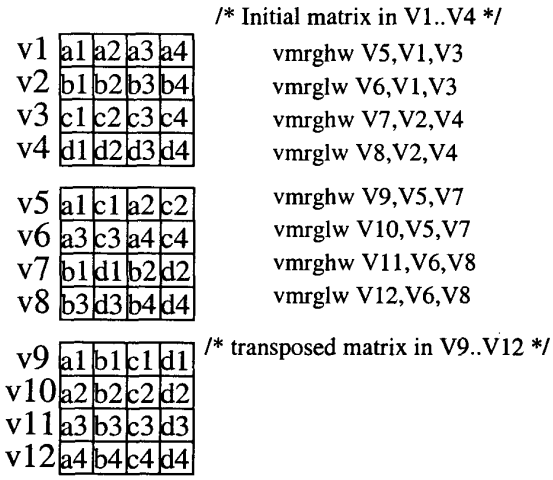


Figure 5: Matrix Transpose

4.The Vector Simple Fixed-point Unit (VSFX)

Except for vector multiply and sum across instructions, the VSFX implements all integer instructions as defined by the AltiVec technology. In addition, it implements the vector floating-point compare and vector minimum/maximum floating-point instructions because these instructions can use the same dataflow as used for

their vector integer counterparts. All instructions execute in the VSFX in a single cycle. Figure 6 shows a block diagram of one 32-bit slice of the VSFX unit (the unit is implemented as four separate 32-bit datapaths).

The VSFX unit shares the reservation station with the VCFX and VFPU units. The VSFX unit comprises a Shift/Rotate block which executes vector integer bit-wise shift/rotate instructions, a Logic block which executes vector integer logical instructions, and an Add/Compare block which executes the remaining add, subtract, and compare instructions. Once all operands are available in the reservation station, the VSFX unit executes the instruction in one of the three blocks and writes the result to one of the six vector rename buses at the end of the execution cycle.

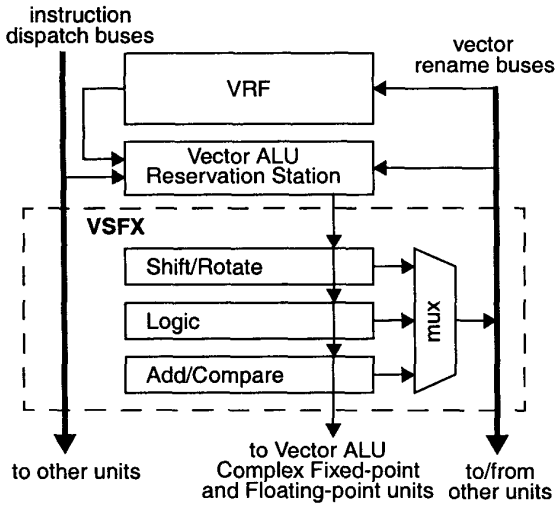


Figure 6: VSFX Block Diagram

A challenge associated with the VSFX hardware design is the ability to handle different data lengths (byte, half-word, word) as defined by the AltiVec technology. To handle different data lengths, the Shift/Rotate block uses separate 8-bit, 16-bit, and 32-bit shifters for byte, half-word, and word operations respectively, because different data portions in vector integer shift/rotate instructions can have different shift amounts. Depending on the data length of the current vector integer shift/rotate instruction, the results from the separate shifters are multiplexed producing the final result of the Shift/Rotate block.

The Logic block does not require any special extra hardware to handle different data lengths because all vector integer logical instructions are bit-wise operations.

In the Add/Compare block, it is possible to use separate 8-bit, 16-bit, and 32-bit adders for byte, half-word, and word operations respectively, but the required chip area would be expensive. To meet area constraints, each word slice of the Add/Compare block uses a single 36-bit adder to handle different data lengths. The 36-bit adder inputs are divided into four segments of nine bits each as shown in Figure 7. Each 9-bit segment contains the 8-bit *A* or *B* operand data and an extra bit used to either block or propagate the carry to the next segment, depending on which data length the instruction operates on.

For example, if the instruction operates on byte data, then the extra bit, forced to a binary 0, blocks the carry from one byte segment to the next; if the instruction operates on word data, then the extra bit on the intra-word byte boundaries, forced to a binary 1, propagates the carry. The extra bits are also used to force carries into a data segment in subtraction or comparison operations. The extra bits are easily and quickly generated at instruction dispatch time.

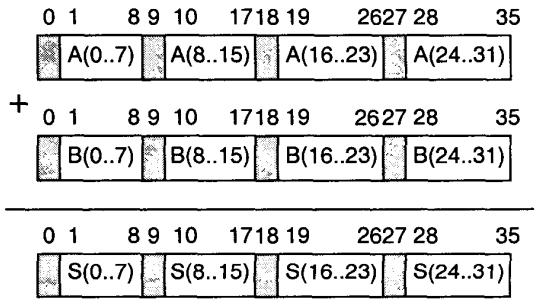


Figure 7: 36-bit Addition for Byte, Half-Word, and Word Operations (One Word Slice)

To meet the chip cycle-time design constraint, the 36-bit adder is designed using dynamic circuitry. In addition, it is fully customized to generate very fast carry outputs at segment borders needed in timing-critical saturation and greater-than/less-than detection logic. The critical timing path of the VSFX hardware is one which passes through the Add/Compare block.

The VSFX unit performs many intra-element operations such as addition, subtraction, average, minimum, maximum, comparison, shift, rotate, and logical operations. These operations support both saturation and modulo arithmetic. They also support both signed and unsigned integers. These flexibilities, along with wide datapaths and wide field operations, help make VSFX instructions powerful in many 3D graphics, audio and video kernels.

One programming example of using VSFX instruc-

tions is the sum of absolute differences (SAD) function, which is used in several video coder implementations such as MPEG2 and H.263. This function compares the similarity of two blocks of pixels in the motion estimation algorithms. When the sum of absolute differences between the respective pixels in two blocks is zero, the blocks are then identical, and a motion vector is established to indicate where to obtain this block from the previous frame. In video coding, motion estimation is often very computationally intensive; therefore, reducing the time to compare two blocks can significantly increase the application performance. For 16x16 blocks of 8-bit pixels, the SAD is computed by

$$SAD(U, V) = \sum_{x=0}^{15} \sum_{y=0}^{15} |U(x, y) - V(x, y)|$$

where

x, y are spatial coordinates in the pixel domain,
U, V are arbitrary 16x16 blocks in adjacent picture frame.

Although AltiVec technology does not provide a dedicated instruction for SAD, it can be computed very quickly by using a few general purpose VSFX instructions as follows:

```
/* V1 contains SAD */
/* V2 contains U(x, y) */
/* V3 contains V(x, y) */

/* Repeat this 16 times for a 16x16 block */
vmaxub    V4, V2, V3    /* larger of U, V */
vminub    V5, V2, V3    /* smaller of U, V */
vsububm   V6, V4, V5    /* absolute difference */
vsum4ubs  V1, V6, V1    /* accumulate result */
```

This is graphically illustrated in figure 3.

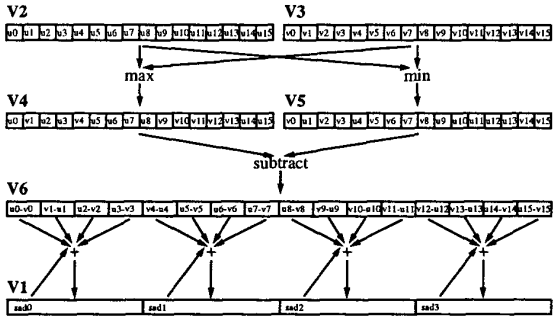


Figure 8: Sum of Absolute Differences

Alternatively, motion estimation can be done using a sum of absolute differences squared algorithm. In this case, vmsumubs can be used to do the multiplication and summation, providing a faster overall function [5].

Another programming example of using VSFX instructions in video coder implementations is the quantization function. Quantization for a H.263 video encoder can be implemented as follows:

```
q  =  sign(prev_q)*((abs(prev_q)-ql/2)/(2*ql)
   =  (prev_q-sign(prev_q)*ql/2)/(2*ql)
   =  prev_q/(2*ql)-sign(prev_q)/4
```

where

q is the quantized value (-127 <= q <= 127),
prev_q is the output of the previous DCT stage multiplied by 4,
ql is the quantization level,
sign(prev_q) is -1 if prev_q <=0; otherwise, it is 1.

The above quantization function can be implemented with VSFX instructions as follows:

```
/* V2 contains ql value */
/* V3 contains input vector */
/* V5 contains constant 2 */
/* V12 contains the constant -128 */
/* V10 contains constant 0 */
/* V0 contains constant 1 */

/* Repeat this 8 times for 8 lines */
vsubshs     V20,V10,V3
vmaxsh      V21,V20,V3       /* V21=abs(V3) */
vmhraddshs  V7,V21,V2,V10  /* prev_q*(2/ql) */
vcmpgtuh    V22,V7,V10
vaddshs     V7,V7,V22   /* add -1 for non-zero */
vcmpgtsh    V4,V10,V3
vsubshs     V1,V4,V0
vor         V4,V4,V1       /* V4=sign(prev_q) */
vsrah       V9,V1,V5        /* scale result by 4 */
vmladduhm   V9,V9,V4,V10  /* mul with sign */
```

Another powerful usage of VSFX instructions is the combination of compare and select instructions to mask and replace data elements across the entire 16-byte field of vector registers. For example:

```
vcmpgtfp  V10, V0, V1          /* V10=V0>V1 */
vcmpgtfp  V11, V1, V2          /* V11=V1>V2 */

/* V20=(V0>V1)? V1, V2 */
/* V21=(V1>V2)? V1, V0 */
vsel      V20, V2, V1, V10
vsel      V21, V0, V1, V11
```

This is graphically illustrated in figure 4.

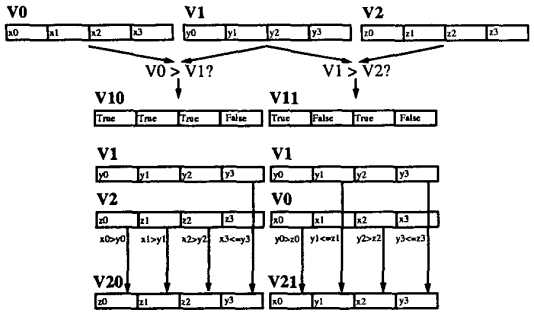


Figure 9: Combination of compare and select

This technique can be used for video masking and 3D clipping functions. In addition, compilers can also use this to eliminate some branch instructions by computing both paths of the branch in parallel. Then, the correct results can be selected using the select instruction.

5: The Vector Complex Fixed-point Unit (VCFX)

The VCFX unit performs the SIMD multiply, multiply-add, multiply-sum and sum across type operations. These instructions operate on eight, 16, and/or 32 bit widths and execute with a three-cycle latency and a one-cycle throughput. The inter-element operations, including multiply-sum and sum across, allow for elements within a single register to be summed in combination with a separate accumulation register, useful in many common vector code sequences, including dot-product algorithms.

The VCFX is implemented as four 32-bit datapaths. Each of these datapaths contains a separate multiply-add structure for the even and odd halfwords. Figures 10 and 11 show the odd and even data paths, respectively, for one word of the complex unit. All four words are similar, with the exception that words one and three have extra inputs for operand *A*, used in the sum across function.

Each of the halfword multiply structures is able to perform simultaneous dual even and odd byte multiplies (8x8) or a full halfword multiply (16x16). The results of the multiplies are selectively added together with several other terms to give the many combinations of multiply, multiply-add and sum functions. The partial product array within each halfword multiply structure is assembled as set forth in [6]. Carry save adder (CSA) trees combine these partial products for all three data paths.

Partial results are shared between halfwords, allowing operations such as summation. Final results can be either saturated or modulated, depending on instruction type.

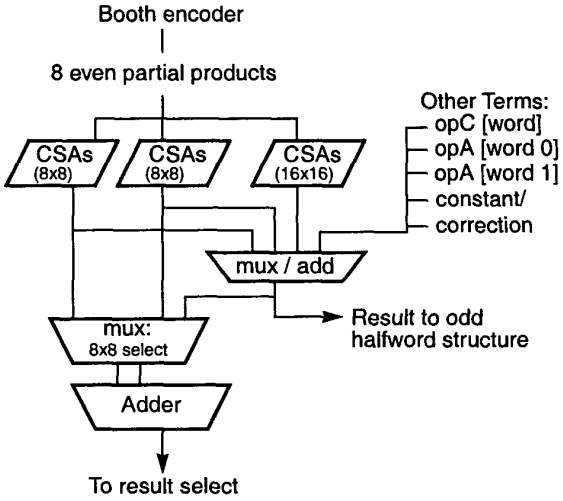


Figure 10: Even Halfword Structure

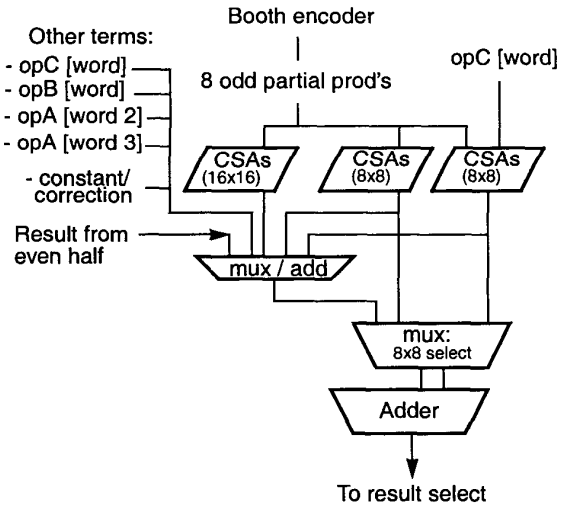


Figure 11: Odd Halfword Structure

The complex unit provides a wide variety of instructions, many of which take the place of a large number of scalar instructions: vmsumubm can do the work of 16 multiplies and 16 adds! This sort of power allows entire routines to be replaced by one or two instructions. In the sum of absolute differences example described earlier, 256 values needed to be summed in the scalar version. With AltiVec technology, this can be computed with just 17 instructions (16 vsum4ubs

instructions to accumulate the partial sums, and one vsumsws to combine them at the end). Since the complex unit has the ability to sum across elements of the same vector, it eliminates the need for permute instructions for many common operations.

6: The Vector Floating Point Unit (VFPU)

The VFPU can perform four simultaneous single-precision floating point operations. Similar to the scalar floating point unit, the VFPU is organized around a Multiply-Add Fused primitive [7]. The unit has a four cycle latency for all instructions except compares, and it is fully pipelined for a one cycle throughput. Floating point compares are done in the VSFX unit and have only a one cycle latency and throughput.

The VFPU operates according to a subset of the IEEE floating point standard [8]. It treats all arithmetic results as round to nearest, and it disables all special floating point exceptions. Denormalized numbers are properly handled in Java-mode by a special trap routine, at the cost of extra execution time. However, for most media and graphics applications, the level of accuracy provided by denormalized numbers is not necessary. For these cases, a non-Java mode is provided, which treats denormalized inputs as zero and forces denormalized results to zero. In non-Java mode, all VFPU instructions take exactly 4 cycles to execute, and there are no special trap or stall conditions.

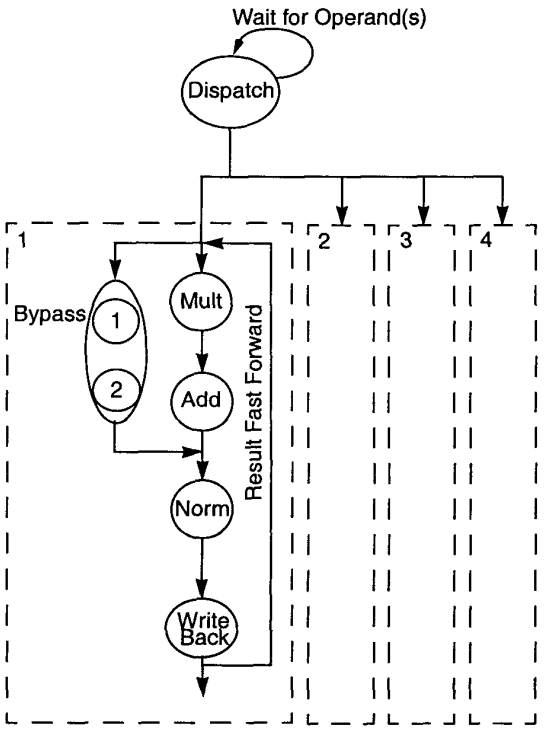


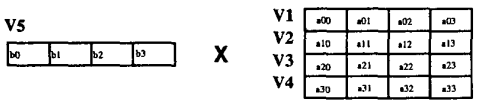
Figure 12: VFPU execution stages

Most of the arithmetic instructions are performed in the Multiply-Add-Normalize-Write Back dataflow engine. The remaining floating-point instructions and special cases are handled outside of the MAF flow. The state diagram of the VFPU is shown in Figure 12.

Floating point code is generally more regular than integer code, and it usually operates on larger data sets and matrices. For these reasons, converting floating point code to vector form is often an easier task than with integer code.

As an example, 4x4 matrix multiplied by a vector can be implemented with vspltw and vmaddfp, and gives a performance gain of over 3.5 times its scalar

counterpart.



Which is equivalent to the following four sums:

b0 * a00	b0 * a01	b0 * a02	b0 * a03
+ (b1 * a10)	+ (b1 * a11)	+ (b1 * a12)	+ (b1 * a13)
+ (b2 * a20)	+ (b2 * a21)	+ (b2 * a22)	+ (b2 * a23)
+ (b3 * a30)	+ (b3 * a31)	+ (b3 * a32)	+ (b3 * a33)
c1	c2	c3	c4

```
/* V0 contains the constant zero */
/* V5 contains the vector b */
/* V1..V4 contain the rows of the matrix a */
/* result will be in V20 */

vspltw    V10, V5, 0 /* make vec b0 b0 b0 b0*/
vmaddfp   V20,V10,V1,V0
vspltw    V11, V5, 1 /* make vec b1 b1 b1 b1*/
vmaddfp   V20,V11,V2,V20
vspltw    V12, V5, 2 /* make vec b2 b2 b2 b2*/
vmaddfp   V20,V12,V3,V20
vspltw    V13, V5, 3 /* make vec b3 b3 b3 b3*/
vmaddfp   V20,V13,V4,V20
```

Figure 13: Multiply Vector x Matrix

The vector floating point unit also has several other instructions very useful to graphics: Reciprocal estimate has only a 4 cycle latency. Only one Newton-Raphson iteration is required to produce a full precision answer [9]. Thus vector divide provides a much faster alternative than its scalar counterpart.

2^x and log2(x) estimate functions are expected to be useful in fast estimation of lighting effects in many 3D graphics applications. These operations are all very low latency. This coupled with the fact that four operations execute in parallel for each instruction make them extremely powerful.

7: Conclusion

AltiVec technology can speed up many applications. It has been tested with great success in many desktop computer applications, including MPEG-type video encoding, Dolby AC-3 audio, 3D graphics and video/graphics manipulation routines. Vector loads and stores, along with new data stream touch instructions, provide a more powerful way for the processor to move data around, and can speed up functions such as memory copies, string compares and page clears.

AltiVec technology is designed to perform very well in embedded systems, where one PowerPC microprocessor with AltiVec technology can potentially replace banks of DSPs or custom ASIC chips. Applications such as IP telephony gateways, multi-channel modems, speech processing systems, echo cancelers, image and video processing systems, scientific array processing systems, internet routers and virtual private network servers will all benefit from the additional processing capabilities possible from AltiVec technology.

References

[1] Fuller, Sam, "Motorola's AltiVec Technology," http://mot-sps.com/sps/General/altivec_wp.pdf

[2] J. Alvarez, E. Barkin, C.-C. Chao, B. Johnson, M. D'Addeo, F. Lassandro, C. Nicoletta, P. Patel, P. Reed, D. Reid, H. Sanchez, J. Siegel, M. Snyder, S. Sullivan, S. Taylor, M. Vo, "A 450MHz PowerPC Microprocessos with Enhanced instruction Set and Copper Interconnect," ISSCC, Feb. 1999.

[3] Gwennap, Linley, "G4 Is First PowerPC with AltiVec," *Microprocessor Report,* pp 17-19, Nov 16, 1998.

[4] *AltiVec Programming Environments Manual,* http://mot-sps.com/sps/General/altivec_pem.pdf

[5] *Mips Extention for Digital Media with 3D*, http://www.mips.com/Documentation/isa5_tech_brf.pdf, pp 20-21.

[6] S. Vassiliadis, M. Schwarz, B. M. Sung, "Hard-Wired Multipliers with Encoded Partial Products," *IEEE Trans. on Comp.*, pp. 1181-1197, Nov. 1991.

[7] E. Hokenek, R. K. Montoye, and P. W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, pp. 1207-1213, Oct. 1990.

[8] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985.

[9] M. J. Schulte, J. Omar, and E. E. Swartzlander, Jr., "Optimal Initial Approximations for the Newton-Raphson Division Algorithm," *Computing*, vol.53, pp. 233-244, 1994.