A Low-power, High-speed Implementation of a PowerPC™ Microprocessor Vector Extension

Martin S. Schmookler, Michael Putrino, Charles Roth, Mukesh Sharma

IBM Corporation
11400 Burnet Rd.,
Austin, TX 78758, USA

Anh Mather, Jon Tyler, Huy Van Nguyen, Mydung N. Pham, Jeff Lent

Motorola Incorporated
6200 Bridgepoint Pkwy, Bldg.4
Austin, TX 78730, USA

Abstract

The AltiVec™ technology is an extension to the PowerPC architecture™ which provides new computational and storage operations for handling vectors of various data lengths and data types. The first implementation using this technology is a low cost, low power processor based on the acclaimed PowerPC 750™ microprocessor. This paper describes the microarchitecture and design of the vector arithmetic unit of this implementation.

1. Introduction

As chosen CMOS technologies reduce in size, conventional microprocessor logic sometimes shrinks to the point where chip size is limited by I/O pins. Often a microprocessor architect is left with considerable chip area in which to "fill". Microprocessor designers in the desktop computing market typically add co-processor implementations such as floating-point units and multimedia execution units to their designs when provided with such opportunities.

The chip size of today's PowerPC microprocessor implementations have been reduced to the point where it has become feasible for the addition of hardware to implement a vector unit executing a multimedia Single Instruction Multiple Data (SIMD) extension of the PowerPC architecture, jointly developed by the IBM Corporation, Motorola, Inc., and Apple Computer, Inc., and announced by Motorola, Inc. as the AltiVec technology [1]. The technology has been designed in such a way as to provide RISC-like "tools" (instructions) in which to build complex networking, multimedia, video, audio, and graphics applications that will execute with high speed.

The first processor having such an implementation is one based on the PowerPC 750 microprocessor microarchitecture. Although the chip area occupied by the AltiVec unit is considerable, it does not greatly impact power and cost.

2. Implementation objectives/constraints

The objective of the design team was to produce a new 32-bit microprocessor that offers a single-chip solution with the combined PowerPC architecture and AltiVec technology. Because the AltiVec enhancement introduces 156 new instructions oriented to highly parallel operations with varying data types, its inclusion posed a challenge to the designers to meet area constraints (the AltiVec hardware must not occupy more than 25% of the total chip area) without impacting the improved cycle time required by a next-generation microprocessor.

The nature of networking, multimedia, video, audio, and graphics applications implies the need for a low-latency, non-stalling pipeline; therefore, a further objective was to ensure that the AltiVec instructions only stall the pipeline when waiting for input operands, and that instructions have no more than four cycles of latency, with many having no more than one.

3. Data types and operations

The operands of the AltiVec instructions consist of 128-bit vectors which, depending on the instruction type, are 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. Each of the instructions operates on one, two, or three operand input vectors from the vector register file (VRF—32, 128-bit registers) and produces an operand output vector destined to the VRF. Each element of the vectors is computed in parallel with the same operation depicted by the OP Code of the instruction (except in some special cases where the operation may apply to the full 128-bit operand or a single portion—one byte, halfword, or word).

Operations on integers or fixed-point numbers include 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; element 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.

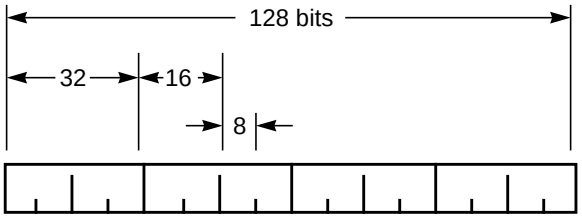

Figure 1. Vector operand format.

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; $\log_2$ 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.

It can be seen that there is no divide instruction; however, the reciprocal estimate and reciprocal square root estimate instructions offer much higher precision than their PowerPC scalar counterparts—both producing 12 bits of precision, which is sufficient for many media and graphics applications. Furthermore, one Newton-Raphson iteration would essentially yield a full single-precision result [4], although in some cases, it may not be properly rounded. These instructions may be used by in-line software to provide divide and square root functions [5] which, in a vector environment, would give improved performance over hardware instructions by permitting the interleaving of instructions for other vector elements, thereby avoiding the effects of inter-operation dependencies. New implementation techniques were developed to obtain the 12-bit precision [6], without much additional area for hardware while maintaining a one-cycle pipeline rate.

The $\log_2$ estimate and 2 raised to the exponent instructions are included for performing some graphics applications, and can be used together to calculate x^y. They each only produce approximately four bits of precision; however, when the input operand is a power of two, corresponding to when the fraction is zero, the $\log_2$ result is exact, being equal to the unbiased exponent of the input operand. Similarly, when the input operand value is an exact integer within the single-precision exponent range, the 2 raised to the exponent instruction result is also exact.

Other operations, in addition to fixed- and floating-point operations, include moving data to and from the vector status and control register (VSCR); pack (16 and 32 bit only); unpack high or low (eight or 16 bit only); pack pixel; unpack pixel high or low; merge high or low; splat; splat immediate; and permute.

For detailed descriptions of each instruction in the AltiVec technology, see [1].

4. AltiVec unit organization

The AltiVec 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 (ALU) as shown in Figure 2. Vector instructions are dispatched to the AltiVec Unit in parallel with instructions dispatched to the scalar fixed-point, scaler floating-point, or load/store units. Data for dispatched AltiVec instructions is stored in the VRF, separate from the scaler register files, and results are placed on separate vector rename buses.

By including a separate register file, the 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.

Both the Permute and Vector ALU units may receive instructions simultaneously, increasing performance. Instructions dispatched to the Vector ALU can be routed to one of three separate sub-execution units: the Vector Simple-fixed Unit (VSFX), the Vector Complex-fixed Unit (VCFX), or the Vector Floating-point Unit (VFPU).

Each execution unit places its results on one of the Vector Rename Buses assigned at dispatch time. 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.

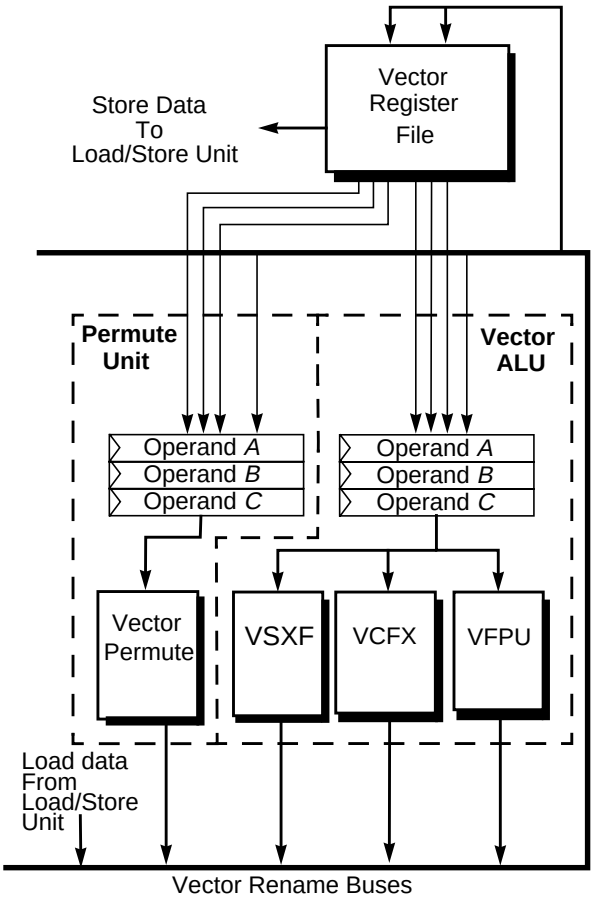

Figure 2. AltiVec unit block diagram.

5. The vector permute unit

A complication of SIMD designs is the fact that data must be properly positioned within the vector operand prior to any arithmetic execution acting on it. The Permute unit performs complex data movement and manipulation to meet the positional requirements of SIMD. The unit executes its instructions in one cycle and 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 multiplexor 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 multiplexor.

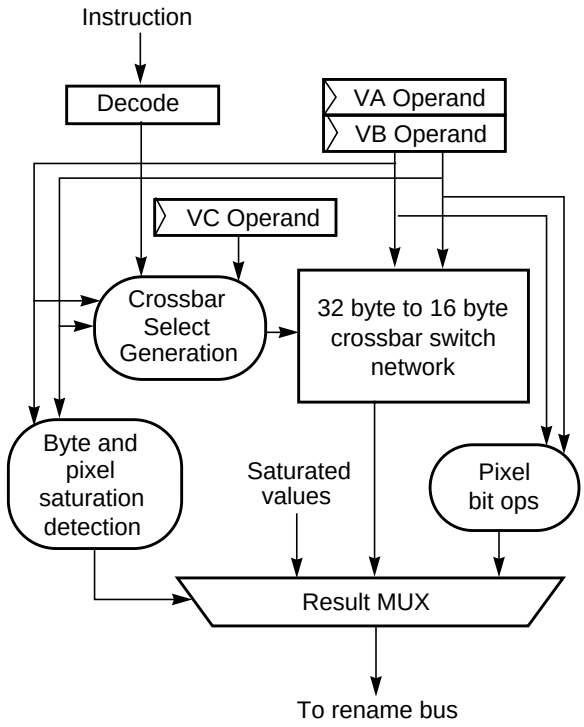

Figure 3. Permute unit block diagram.

6. The vector simple fixed-point unit

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 4 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.

A challenge associated with the VSFX hardware design is the ability to handle different data lengths (byte, halfword, word) as defined by the AltiVec technology. To han-

dle 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.

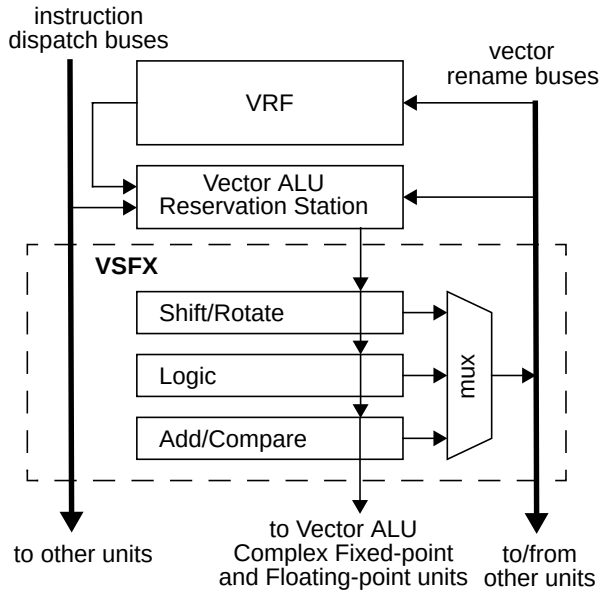

Figure 4. VSFX block diagram.

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 5. 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.

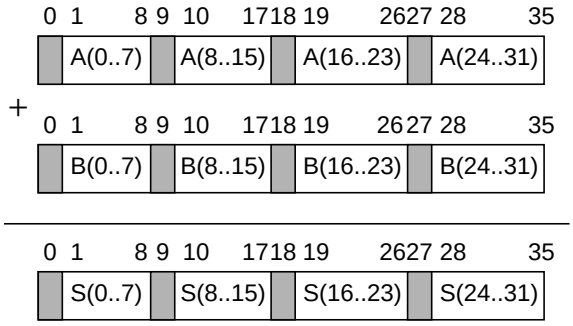

Figure 5. 36-bit addition for byte, halfword, and word operations (one word slice).

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.

7. The vector complex fixed-point unit

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 6 and 7 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 [2]. Carry save adder (CSA) trees combine these partial products for all three data paths. In Figures 6 and 7, the CSAs are shown to be all separate. In reality, common terms are shared among the 8x8 and 16x16 multiply data paths.

The mux/add blocks of Figures 6 and 7 perform the majority of the controlled dataflow for most instructions executing in the unit. Receiving sum and carry vector results from the multiply arrays (8x8 odd, 8x8 even, and 16x16) and several addend terms, the even mux/add block outputs sum and carry vectors for the following.

16x16(multiply halfword)
16x16 + *C*(multiply add)
8x8 + 8x8 + *C*(forward to even halfword structure for multiply sum byte)
A + *A*(forward to even halfword structure for full sum across)

The even mux/add block sum and carry vector output is routed to the odd halfword datapath where it is applied to the input of the odd mux/add block (the only place where data is shared between halfwords), enabling inter-element operations such as multiply sum and sum across.

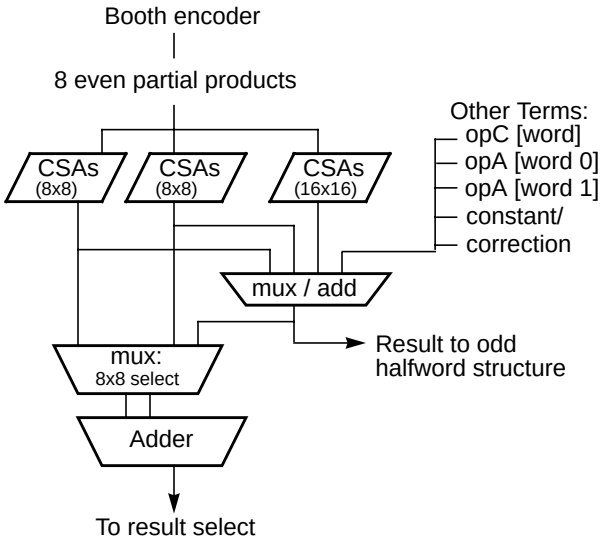

Figure 6. Even halfword structure.

The odd halfword structure produces results similar to the even halfword structure; however, its results can also be added together with the even halfword mux/add results, to perform the following operations.

8x8 + 8x8 + 8x8 + 8x8 + *C*(multiply-sum byte)
16x16 + 16x16 + *C*(multiply-sum halfword)
A + *A* + *A* + *A* + *B*(full sum across)
16x16(multiply halfword)
16x16 + *C*(multiply add)

Partial sum across functions are implemented by using the multiply-add data path and forcing constant values into some operands before the Booth encode. For example, the vsum4shs instruction [1] is implemented the same as the vmsumshs instructions [1] with the constant “one” applied to each halfword multiplicand.

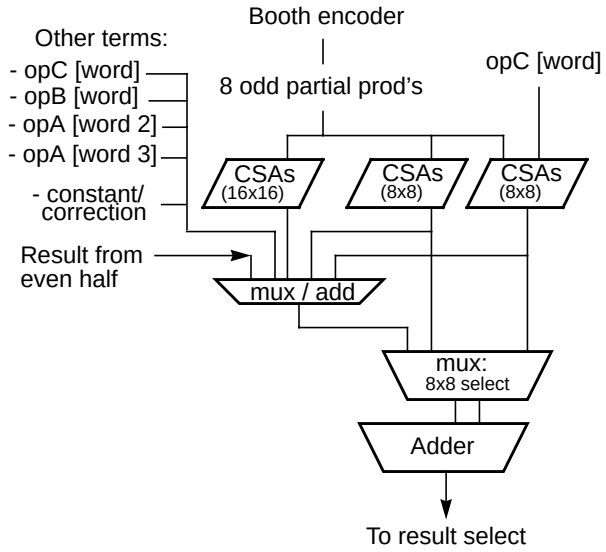

Figure 7. Odd halfword structure.

The input to the final adder, depending on the operation, is either the result sum & carry outputs from the mux/add block, or the even and odd 8x8 multiply sum and carry partial products. The adder is a full carry look-ahead adder with carry blocking capability. Depending on the instruction type, the adder can produce a full word or two half word results as needed.

Each halfword dataflow produces a one-word result (may consist of two concatenated halfwords or four concatenated bytes); therefore, the result selector chooses one word out of these two to be forwarded to the saturation detection logic. The selection, prior to saturation, is from odd halfwords, even halfwords, odd words, or even words, and some combinations of rounding and shifting. Lastly, saturation detection is performed; the results are saturated, as necessary, and are driven onto a result bus.

8. The vector floating-point unit

The VFPU supports four simultaneous, single-precision floating-point operations conforming to a subset of the IEEE floating-point standard [3]. The unit operates as if all exceptions are disabled and the round-to-nearest rounding mode is selected. The unit has a latency of four cycles and a throughput of one cycle and will only stall when waiting for input operands.

Two modes of operation are available which control how denormalized numbers are handled by the VFPU. For

applications requiring Java or IEEE compatibility, the Java Compliant mode (or Java mode) is provided. In the Java mode, operations on denormalized operands, and operations producing denormalized results, may be supported by a combination of hardware and software. For the current implementation, the hardware detects such operands and invokes a trap which completes the operation in software.

For most media and graphics applications, the non-Java mode is provided. The non-Java mode allows all instructions to execute at a one-cycle pipelined rate. In this mode, no traps are taken when denormalized numbers are encountered. Instead, denormalized input operands are treated as zeros in most cases, and denormalized results are forced to zeros. Two cases where denormalized inputs are not treated as zeros include multiplication of a denormalized number by infinity, which produces a result of infinity, and compare instructions which are easily performed without first normalizing the operands.

No IEEE exception status flags or traps are provided in either mode of operation; invalid operations produce NaNs, and exponent overflow produces infinity. In Java mode, exponent underflow produces a denormalized number derived from a software trap.

In order to speed up results of compare operations, including the possible update of the Condition Register, the compare instructions are not executed in the VFPU, but in the VSFX unit where they are performed in one cycle. In addition, the floating-point maximum and minimum instructions are also executed in the VSFX unit.

The VFPU contains four engines optimized for the Multiply-Add Fused (MAF) [7] primitive as the core of the vector floating-point micro-architecture. 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 8.

The multiply stages perform the main multiply function using a combination of a modified-Booth [8] re-coding and Wallace-tree [9] multiplier array to generate the accumulated partial product in the sum-and-carry format. Exponent logic calculates the B operand alignment shift count required for the add function. The mantissa of the *B* operand is shifted right for alignment with the product. In addition, the result exponent is computed.

A 23-bit incrementer concatenated with a 52-bit carry propagate adder with end-around carry [10] is used in the Add stage to compute the sum of the multiply partial products which includes the aligned *B* operand. A 73-bit Leading Zero Anticipator (LZA) [11, 12] is used to estimate the position of the most-significant leading one of the adder result. With accuracy of +1, 0, or -1 bit position, the LZA computes both the shift amount required for exponent adjustment, and the shift controls for the normalization operation performed in the next pipe stage.

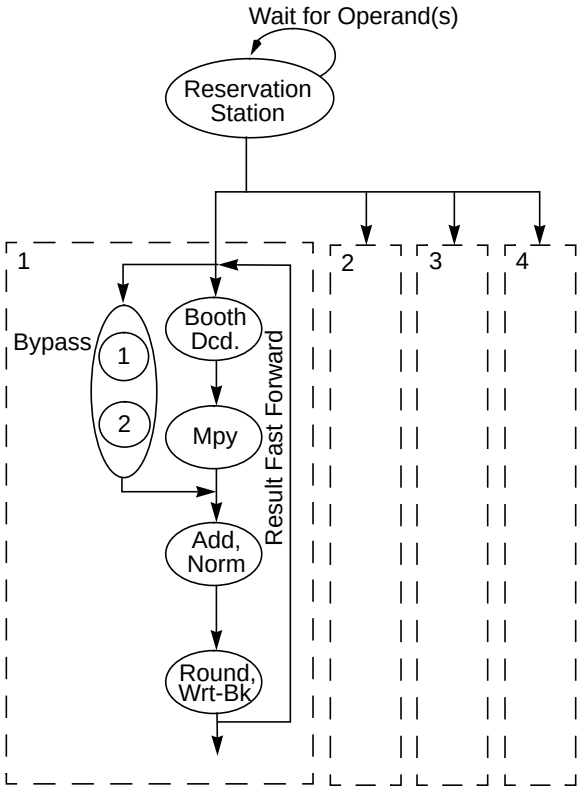

Figure 8. VFPU execution stages.

Full single-cycle normalization is performed by left shifting the result from the Add stage by the correct number of leading zeros calculated by the LZA. Concurrently, the sticky-bit is calculated. Because of the inaccuracy of the LZA, a correction is required for the final normalizer multiplexor step. The computed correction is used to adjust both the final normalizer shift stage and associated exponent calculations as well as the sticky-bit computation and underflow/overflow detection logic.

Following normalization, the result is rounded. All rounding is performed in the IEEE round-to-nearest mode, except for the convert to integer instruction which only rounds to zero, and the round to floating-point integer instruction, which allows any of the four IEEE rounding modes to be specified by the instruction itself. It should be noted that the instructions which convert between floating point numbers and 32-bit fixed point integers also provide for scaling the floating point number, a feature which is not included in the scalar floating point conversion instructions. After rounding, the four final results are written back to one of the six pre-assigned, 128-bit rename busses.

The Bypass stages operate in parallel with the Multiply and Add stages. The result of the Bypass stages is used in

lieu of the Multiply/Add stage result for the following cases:

- detecting invalid operations and preparing for the defaulted result,
- producing results for estimate instructions,
- handling the round and conversion instructions,
- and any other special cases requiring a result that the Multiply-Add-Write Back engine cannot handle easily.

When the Bypass logic encounters one of the above conditions in parallel with the Multiply stage, the adjusted result or a defaulted result is created and forwarded to the input of the pipeline register between the Add and Write Back stages. The result computed by the Multiply and Add stages is ignored.

Each of the four floating point sub-units has its own lookup tables and other supporting data path circuits in order to maintain a one-cycle pipelined execution rate for the reciprocal estimate and reciprocal square root estimate instructions. Direct table lookup for 12 bits of precision could become prohibitively expensive, considering that eight separate tables, each with 2048 entries, are required. Instead, a linear interpolation is used which requires 32-word tables—each word containing two values, thereby reducing the total table size by a factor of 32. Furthermore, the smaller tables can be accessed with much less delay. To further reduce area, the tables are implemented using synthesized standard cells, eliminating the need for custom-designed PLAs or ROMs.

Use of linear interpolation and linear approximation for reducing table size has been described in several papers [4, 5, 6]. In the present implementation, the fraction of the input operand B is partitioned into two parts. The high-order bits B_h specify an interval within the range of values from 1.0 to 2.0, as illustrated in Figure 9. Because the exponent is processed separately, a limited range need only be considered. For each interval, the function is approximated by a straight line segment. The values along each segment are computed using the value y_0 of that segment at its left-hand edge, the slope m of the segment, and the displacement along the horizontal axis. The displacement is equal to the low-order fraction bits B_l of the input B operand, just to the right of B_h. The B_h bits are used to access the table which contains the y_0 values and the absolute values of m. Both of these estimate functions have negative slopes for all positive values of B; therefore, to obtain the value of the function along the line segment, the product of m and B_l must be subtracted from y_0.

To minimize the size of the required table, each line segment is placed so as to minimize the error within the interval. For the reciprocal estimate instruction, B_h has five bits resulting in a 32-word table. Each word contains a 15-bit value for y_0 and a 10-bit value for the slope m. To obtain 12 bits of accuracy, only the high order 10 bits of B_l are needed; however, in the present implementation, the full single-precision, multiply-add execution unit is used, so all of B_l participates in the calculation, resulting in slightly improved precision.

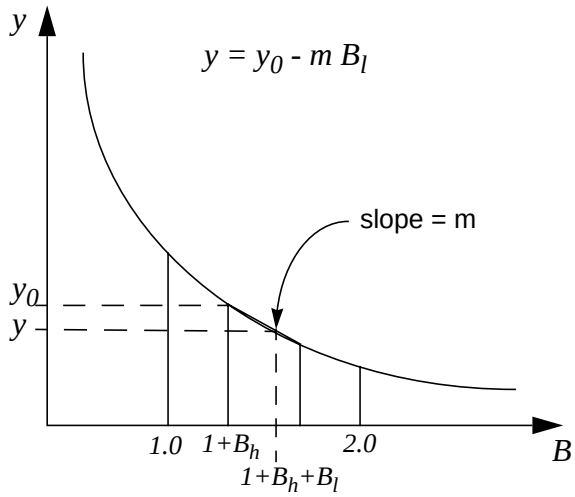

Figure 9. Linear approximation of 1/*B* or 1/√*B*.

For the reciprocal square root estimate instruction, the 32-word table can be thought of as consisting of two separate 16-word tables: one for even exponents and one for odd exponents. The least-significant bit of the exponent selects the table, and the word within the table is addressed by a 4-bit B_h. The values for y_0 consist of 16 bits each, and 11-bit values are required for m. Again, all of the bits of B_l are used, although only ten are required.

The estimate tables are accessed during the first pipeline stage of the VFPU. Then, the table output values are multiplexed with the input operands for the multiply-add instructions. Thus, the full single-precision multiply-add mantissa dataflow is available for the calculation.

Very little additional area is required for the $\log_2$ and 2 raised to the exponent estimate instructions. For the $\log_2$ estimate instruction, the value of the result consists of a signed integer part and a positive fractional part. The integer value is the unbiased exponent of the input operand. The fractional value is obtained from a very small table accessed by the high-order bits of the input operand. The two parts concatenated produce the value of the result in a fixed-point format. This is then converted to a single-precision floating-point value using hardware techniques similar to those used for fixed-to-float conversion operations.

The 2 raised to the exponent estimate instruction is

essentially the reverse operation of the $\log_2$ estimate instruction. Techniques similar to those used for float-to-fixed conversion without rounding to integer, provide an intermediate result consisting of a signed integer part and a positive fractional part. A table is used to translate the fractional part to the corresponding fraction of the final result. The signed integer part is added to the single-precision exponent bias to obtain the exponent of the result.

9. Comparisons

Table 1 shows a comparison of the feature sets of MMX w/KNI (Intel Corp.), MMX w/3DNow (AMD), and AltiVec [13]. It can be seen that AltiVec has incorporated a richer set of operations as compared to the other multimedia architectures. A good example is the floating point multiply-add instruction used in many common DSP applications. The permute operation in AltiVec also offers a more flexible version of the data reorganization than is offered by the other multimedia architectures.

Table 1: Multimedia architecture comparison.

Feature	MMX w/KNI	MMX w/ 3DNow	PowerPC AltiVec
Registers (width)	8 fp+8 int 128bit/64bit	8int/fp 64bits	32int/fp 128bits
FP precision	Single	Single	Single
Multiply-add	No	No	Yes fp & int
No. of source operands	2	2	3
Data reorganization	Limited to Pack/Merge	None	Complex permutations

The enhanced feature set of AltiVec provides improved performance in many of today's applications. In audio applications a speedup of 3.6x is achieved when compared to optimized C code used for computing the inverse FFT (FP) [14]. A similar speedup is achieved for a forward DCT used in Video applications. And, performing a matrix-matrix multiplication, used in many graphics applications, provides a speedup of 6.2x when compared to optimized C code [14].

10. Conclusion

The AltiVec technology extension to the PowerPC architecture was implemented on a microprocessor based on the PowerPC 750 microprocessor microarchitecture. Its design resulted in a next generation microprocessor with multimedia capabilities that compete well against MMX with KNI and MMX with 3DNow technology [13]. It was shown how the broad array of AltiVec instructions were implemented having restricted area constraints, low-latency and minimal pipeline stalls.

References

[1] *AltiVec Programming Environments Manual*, http://www.mot.com/SPS/PowerPC/tecsupport/teclibrary/manuals/altivec_pem.pdf

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

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

[4] 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.

[5] M. Ito, N. Takagi and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 2-9, July 1995.

[6] D. Das Sarma and D. W. Matula, "Faithful Interpolation in Reciprocal Tables," *Proc. 13th IEEE Symp. Computer Arithmetic*, pp. 82-91, July 1997.

[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] A. D. Booth, "A Signed Binary Multiplication Technique," *Qt. J. Mech. Appl. Math*., Vol. 4, Part 2, 1951.

[9] C. S. Wallace, "A suggestion for fast multipliers," *IEEE Trans. Electron. Comput.*, Vol. EC-13, pp. 14-17, Feb. 1964.

[10] S. Vassiliadis, D. S. Lemon, and M. Putrino, "S/370 Sign-magnitude Floating-point Adder," *IEEE Journal of Solid-State Circuits*, Vol. 24, No. 4, pp. 1062-1070, Aug. 1989.

[11] E. Hokenek, R. K. Montoye, "Leading-zero Anticipator (LZA) in the IBM RISC System/6000 Floating-point Execution Unit," IBM J. Res. Develp., Vol. 34, No. 1, pp. 71-77, 1990.

[12] S. Knowles, "Arithmetic Processor Design for the T9000 Transputer," in *SPIE*, Vol. 1566, *Advanced Signal Processing Algorithms, Architectures, and Implementations II*, 1991.

[13] K. Diefendorff, "Katmai Enhances MMX," Microprocessor Report, pp. 1, 6-7, 9, Oct. 5, 1998.

[14] *Executive Summary*, Apple Corp. developer Web Page, http://developer.apple.com/hardware/altivec/summary.html