Exploration and Evaluation of PLX Floating-point Instructions and Implementations for 3D Graphics

Xiao Yang[1], Shamik K. Valia[2], Michael J. Schulte[2], and Ruby B. Lee[1]
[1]Department of Electrical Engineering, Princeton University, Princeton, NJ 08544
[2]Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI 53706

***Abstract*—PLX FP is a floating-point instruction set architecture (ISA) extension to PLX that is designed for fast and efficient 3D graphics processing. In this paper, we explore the implementation and performance of the fundamental functional unit for PLX FP, the floating-point multiply-accumulate (FMAC) functional unit. We present simulation and synthesis results for several implementations with increasingly powerful sets of instructions, to compare area and delay tradeoffs. We also evaluate the performance tradeoffs with examples taken from the 3D graphics processing pipeline.**

I. Introduction

With the proliferation of computer games, floating-point (FP) intensive 3D graphics processing is rapidly becoming a major component in the workload on multiple computing platforms. To address the needs of 3D graphics, we proposed PLX FP [1][2], a fully subword-parallel floating-point ISA extension to the PLX architecture [3]. It enables fast and efficient 3D graphics processing on PLX, an architecture designed from scratch for fast multimedia processing.

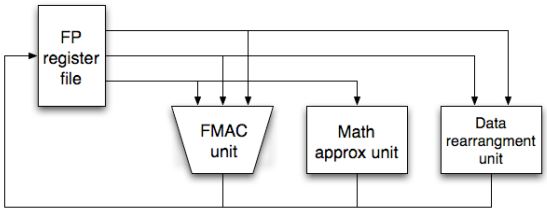

Figure 1 Datapath for PLX FP

Six classes of floating-point instructions are defined in PLX FP; arithmetic, compare, mathematical approximation, data rearrangement, data conversion, and memory instructions [1][2], as shown in Table 1. To execute the PLX FP instructions, a new FP datapath (Figure 1) needs to be added to the base PLX architecture. It includes a separate FP register file, a new set of functional units, and the corresponding databuses. Each of the new functional units handles certain classes of PLX FP instructions. The FMAC unit executes the arithmetic and compare instructions. A novel feature of PLX FP is the introduction of two new types of arithmetic instructions: the FP scale and dot product instructions. These instructions can effectively speed up vector scaling and dot product operations, which are common in 3D graphics. In this paper, we explore the implementations of the FMAC unit to study area and delay tradeoffs, evaluate the incremental cost of the new scale and dot product instructions, and study the performance impact of different FMAC implementations on 3D graphics processing.

Like integer PLX, PLX FP is also datapath scalable. A register can hold one, two, or four subwords, each containing a 32-bit IEEE single-precision FP data. Correspondingly, the datapath size can be 32-bit, 64-bit, or 128-bit, suitable for meeting different cost and performance targets. We assume 128-bit FP datapath in this paper, because the 4-element vector is the most commonly used data type in 3D graphics.

The paper is organized as follows. Section II surveys the past work related to PLX FP and FMAC implementations. Section III studies the implementations of the FMAC unit and presents simulation and synthesis results. Section IV evaluates performance and analyzes the results. Finally we conclude in Section V.

TABLE 1: PLX FP INSTRUCTIONS

Arithmetic (Vector)		**Compare**
padd	padd.neg	pfcmp.rel
psub		fcmp.rel
pmul	pmul.neg	fcmp.rel.pw1
pfmuladd	pfmuladd.neg	**Math approximation**
pfmulsub	pfmulsub.neg	frcpa
pfabs	pfabs.neg	frcpsqrta
pfmax		flog2a
pfmin		fexp2a
pfscale,j	pfscale.neg,j	**Rearrangement**
pfscaleadd,j	pfscaladd.neg,j	fmix.l
pfscalesub,j	pfscalesub.neg,j	fmix.r
pfdp	pfdp.neg	fpermute
pfdp.s	pfdp.s.neg	fextract
Arithmetic (Scalar)		fdeposit
fadd	fadd.neg	**Conversion**
fsub		pfcvti
fmul	fmul.neg	pfcvtu
fmuladd	fmuladd.neg	picvtf
fmulsub	fmulsub.neg	puicvtf
fabs	fabs.neg	fcvti
fmax		fcvtui
fmin		icvtf
Memory		uicvtf
fload	fstore	
floadx	fstorex	
fload.u	fstore.u	
floadx.u	fstorex.u	

Proceedings of the Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, November 7-10, 2003.

II. Past Work

3D graphics is dominated by short vector operations. The majority of the data processed, such as coordinates, normals and colors, can be represented with short vectors of three or four elements. Existing 3D-oriented ISA extensions for general-purpose processors, including SSE and SSE-2 [4] for Intel IA-32, 3DNow! [5] for AMD x86, AltiVec [6] for PowerPC, 3D-ASE [7] for MIPS64, and Intel IA-64 [8], can effectively exploit the subword parallelism [9] in the vector operations by using FP microSIMD [10] instructions. While being able to achieve good performance speedup for 3D graphics processing [11][5][6][7], they still incur the overhead of their base architectures, which are targeted for business and scientific computing. PLX FP, on the other hand, is based on the PLX architecture, which is designed specifically for fast and efficient multimedia processing, and thus does not suffer from the overhead of a large base architecture.

The FMAC unit is the essential functional unit for PLX FP. Also known as multiply-add-fused (MAF) units, FMAC units are used in many contemporary general-purpose processors, such as IBM POWER series [12], HP PA-RISC [13] and Intel IA-64 [8], for executing arithmetic and compare instructions. They are also employed in embedded media processors and dedicated 3D graphics processors such as the Hitachi SH4 [14] and Sony Emotion Engine for PlayStation 2 [15] for fast 3D graphics processing. Quach and Flynn investigated the designs of general-purpose FMAC units and compared three strategies with different degrees of overlap between FP multiply and FP add operations: MAC (multiply-add chained), Greedy, and SNAP (Stanford Nanosecond Arithmetic Processor) [16]. MAC is the simplest, with the smallest area but the longest delay, while SNAP is the most complex, with the largest area and the shortest delay. Our FMAC unit implementation is similar to MAC. Unlike general-purpose FMAC units, an FMAC unit for 3D graphics does not need to guarantee correct rounding of the result and support all four rounding modes defined in the IEEE-754 standard [17]. In the case of the FMAC unit for PLX FP, the addition of the scale and dot product instructions also results in new design and implementation challenges.

III. Implementations of FMAC Unit for PLX FP

We examine the cost of the entire FMAC unit in terms of latency and area for three sets of FP instructions: a minimal set, a baseline set, and a performance set, which is the proposed PLX FP set with the new scale and dot product instructions (Table 2). We evaluate the potential impact on the processor's cycle time versus the incremental number of pipeline stages for the FMAC unit. The latency of an FMAC unit is the number of cycles to complete its longest operation.

The minimal set of FP instructions consists of parallel FP add, subtract, multiply, multiply-add, and multiply-subtract instructions. This covers the basic functions of a SIMD-controlled set of FMAC units. The baseline set adds parallel FP absolute, maximum and minimum instructions, and the scalar and negated versions of all the applicable instructions. It also includes the three compare instructions: parallel compare, compare, and "compare with parallel write" of a predicate register. The baseline set represents a typical set of instructions supported by modern 3D graphics ISA extensions with some minor enhancements (scalar instructions and compare with parallel write). The performance-optimized set further includes some novel features of PLX FP: the scale, scale and add/subtract, dot product and dot product with saturation instructions, and their negations.

TABLE 2: THREE FP INSTRUCTION SETS FOR EXPLORING THE FMAC UNIT IMPLEMENTATIONS

Minimal FP set (5 instructions)	**Baseline FP set** (29 instructions)	**Performance set PLX FP** (39 instructions)
a) pfadd pfsub pfmul pfmuladd pfmulsub	Minimal FP set plus: b) pfabs c) pfmax, pfmin d) *Scalar versions of a, b and c*: fadd, *etc*. e) *Negation of a, b and d*: pfadd.neg, fadd.neg, *etc*. f) pfcmp.rel fcmp.rel fcmp.rel.pw1	Baseline FP set plus g) pfscale,j pfscaleadd,j pfscalesub,j h) pfdp i) pfdp.s j) *Negation of g, h and i*

A. FMAC units for three FP sets

The FMAC unit for the minimal set is the easiest to implement. Figure 2 shows a 128-bit FMAC unit, consisting of four minimal FMACs. Each minimal FMAC is constructed by chaining an FP multiplier and an FP adder, without the rounding logic.

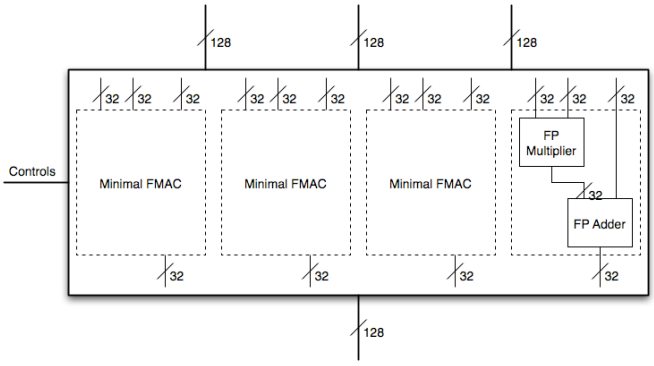

Figure 2 FMAC unit for minimal FP set

The FMAC unit for the baseline set can also be implemented with an array of four FMACs (Figure 3). However, these FMACs are enhanced over the minimal FMACs in order to execute the additional instructions in the baseline set. We need new logic to test the conditions for the compare instructions, a group of multiplexers to select the output among regular adder output, min/max, and parallel compare, and additional wires and pipeline registers. Two small pieces of logic are also added to perform absolute and negation. Notice for the minimal and baseline sets, each FMAC only needs to receive three 32-bit operands.

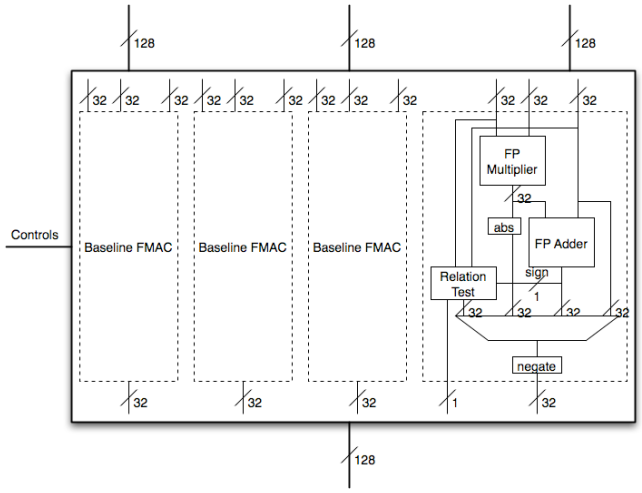

Figure 3 FMAC unit for baseline FP set

The operations of the new scale and dot product instructions in PLX FP are illustrated in Figure 4. Figure 5 shows the modifications needed for the FMAC unit. For the scale instructions, a row of 4-1 muxes are needed in front of each FMAC to select a specified subword in the first source register in the instruction to be the first input to all the FMACs. This requires extra wires to broadcast the entire 128 bits of the first register to all the FMACs. The dot product instructions require the addition of the results of four multiplications. This addition can be implemented with three FP adders in two levels. The first level reuses two existing adders within the FMACs, which requires some modifications to the FMACs. The four FMACs are divided into two groups, each containing two adjacent FMACs. Extra wires are added to the left FMAC in each group to move the result of its FP multiplier to the right FMAC in the same group. Muxes are added to the right FMAC at the right input of the FP adder to select between the multiplier output from the left FMAC and the original input. The second level of addition for the dot product instructions needs an extra FP adder to add the partial sums from the two groups. This adder also needs to support saturation to zero, i.e., set the result to 0.0 if it is less than 0.0. Finally, muxes are used to choose between the original output from the rightmost FMAC and the output from the new adder.

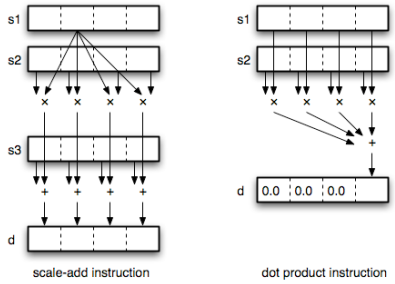

Figure 4 Illustration of `pfscaladd` and `pfdp` instructions

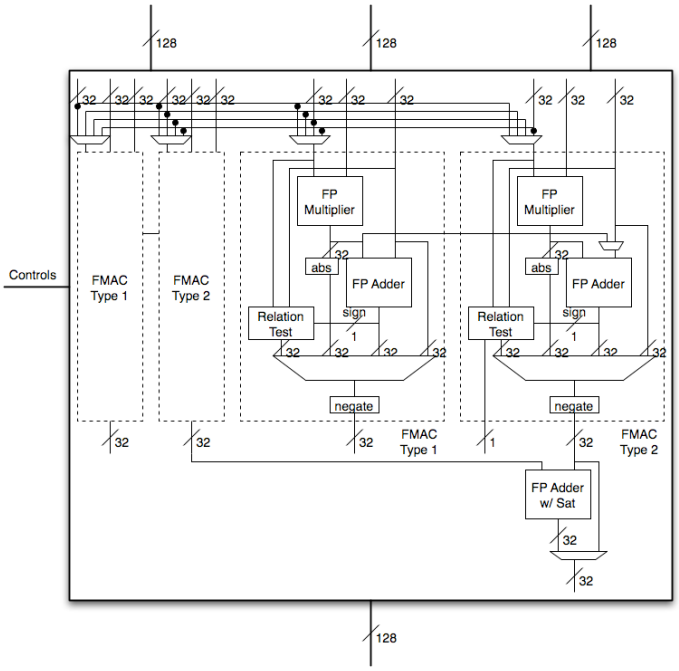

Figure 5 FMAC unit for PLX FP set

B. *Synthesis and simulation results*

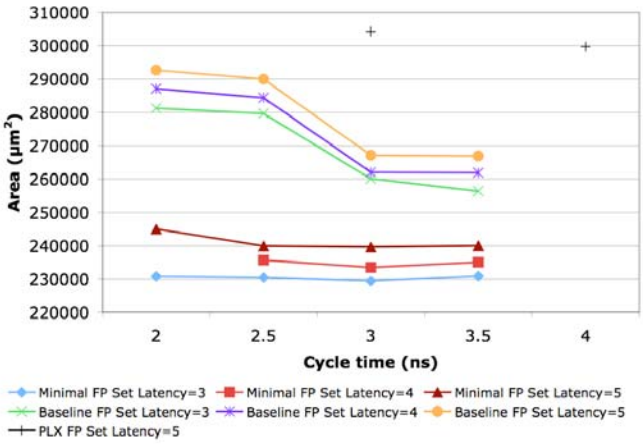

Figure 6 FMAC unit synthesis results for three FP Sets

The synthesis results are shown in Figure 6, where the designs are optimized to achieve a target clock cycle time with minimum area. The synthesis is done using the LSI Logic Gflx-P 0.11μm CMOS standard cell library [18] and Synopsys Design Compiler. For implementations with comparable cycle time and latency (3ns and 5-cycle latency), the baseline set requires about 11.5% more area than the minimal set, while the PLX FP set takes about 13.9% more area than the baseline set. Two trends can be observed from the results. For implementations of the same FP set, increasing latency or pipeline stages, while keeping cycle time constant, only causes small increase in area (less than 3%). This is because pipeline registers only account for a small percentage of the area in the

FMAC unit and fewer buffers are needed when the FMAC unit has a longer latency. Increasing cycle time under constant latency results in a drop in area. This is because the circuit can be more serialized and requires fewer buffers with the relaxed cycle time, and thus takes less area. This levels off, indicating a possible space-time optimum for a given circuit, 2.5ns for the minimal set and 3ns for the baseline set.

Table 3 shows the minimum cycle time that can be achieved for the three FP sets for different latencies and their corresponding area. The minimum cycle times for the minimal and baseline sets are very similar, although the latter demands much larger area. The minimum cycle time for the PLX FP set suffers greatly due to the additional broadcast and select circuitry and especially the extra FP adder. The minimum cycle time decreases when the latency increases. By increasing the latency of the FMAC unit for the PLX FP set, it can achieve comparable minimum cycle time to the FMAC unit for the minimal or baseline set with shorter latency. For example, a 5-cycle implementation for the PLX FP set has a minimum cycle time of 1.94ns, which is close to the 1.79ns or 1.7ns cycle time of the implementations for the baseline and minimal sets with a latency of 3 cycles per FP operation. Clearly, the more powerful PLX FP instructions like scale and dot product impact either cycle time or latency (in cycles) quite significantly, and we need to see if this pays off in performance due to the decrease in the number of instructions executed.

TABLE 3: MINIMUM CYCLE TIME AND CORRESPONDING AREA

	Latency (cycles)	Cycle Time (ns)	Area (μm^2)
Minimal FP Set	3	1.7	219776
	4	1.51	277276
	5	1.39	297987
Baseline FP Set	3	1.79	297475
	4	1.51	342383
	5	1.4	355323
PLX FP Set	3	2.82	300991
	4	2.26	320087
	5	1.94	342339

IV. PERFORMANCE AND DISCUSSIONS

A. Performance evaluation of PLX FP instructions

3D graphics is most commonly handled by a 3D graphics processing pipeline [19], which consists of a geometry processing phase and a rendering phase. The geometry phase is always done in floating-point and is the target of optimization for all existing 3D-oriented ISA extensions. We measure the performance gain from the new scale and dot product instructions by comparing the instruction counts (Table 4) for five basic operations and one processing kernel taken from the geometry processing phase in the 3D graphics pipeline:

- Matrix-vector transform: Multiply a 4-element vector by a 4×4 matrix;
- Normalization: Normalize a 3-element vector;
- Perspective division: Divide the first three elements of a 4-element vector by its fourth element;
- Dot product: compute the dot product of two 4-element vectors;
- Cross product: compute the cross product of two 3-element vectors;
- Transform and lighting: the most important processing kernel in geometry processing, which loads a vertex, does model-view transform, computes lighting, performs perspective transform, and finally stores the processed vertex. This kernel contains many vector scaling and dot product operations, and is a good benchmark for the new scale and dot product instructions.

For reference purposes, we also include AltiVec and IA-64 in the comparison because they have the most advanced instruction sets for 3D graphics in existing microprocessors. The minimal set is omitted from the comparison because it lacks the compare instructions required to implement most of the operations and kernels. Five instruction sets are compared: baseline set, baseline set plus the scale instructions, PLX FP set, AltiVec, and IA-64.

We assume the other classes of PLX FP instructions are supported for both baseline and PLX FP sets. Notice that a scale (or scale-add/subtract) instruction can be simulated with two instructions, one from the baseline set and one subword rearrangement instruction:

```
pfscale,j       Fd,Fs1,Fs2
```

equals to

```
fpermute,jjjj   Ft,Fs1
pfmul           Fd,Ft,Fs2
```

where the first `fpermute` instruction broadcasts subword j in register Fs1 to all subword locations in register Ft. A dot product instruction can be simulated with three instructions from the baseline set and two rearrangement instructions:

```
pfdp            Fd,Fs1,Fs2
```

equals to

```
pfmul           Ft1,Fs1,Fs2
fpermute,-3-1   Ft2,Ft1
pfadd           Ft3,Ft1,Ft2
fpermute,---2   Ft4,Ft3
fadd            Fd,Ft3,Ft4
```

The first `fpermute` instruction moves two products for the first level addition, and the second `fpermute` instruction moves a partial sum for the final addition. In the case of dot product with saturation, an extra `fmax` instruction is needed. The functionality of the scale and dot product instructions are orthogonal, and their performance gains are additive.

The first five rows in Table 4 show the instruction counts for the five operations. The baseline set performs comparably with AltiVec, which is also a 128-bit ISA extension. The higher instruction counts of IA-64 are largely due to its 64-bit FP datapath. The scale instructions reduce instruction counts greatly for the transform, and also for the normalization and perspective division operations. The dot product instructions

speed up the normalization and dot product operations significantly. Cross product operation cannot take advantage of subword parallelism, thus all instruction sets perform similarly.

TABLE 4: PERFORMANCE COMPARISON OF DIFFERENT INSTRUCTION SETS

	Baseline	Baseline + scale	PLX FP	AltiVec	IA-64
Transform	8	4	4	8	12
Norm	8	7	3	7	10
Pers div	3	2	2	3	4
Dot prod	5	5	1	5	7
Cross prod	6	6	6	6	10
TL no lts	10	6	6	10	16
TL 1 direct	70	55	37	90	112
TL 1 point	90	73	47	108	129
TL 1 spot	101	84	53	142	160
TL	39+31*L*	27+28*L*	23+14*L*	38+52*L*	58+54*L*
LD+*MP*+*NS*	+51*M*+62*N*	+46*M*+57*N*	+24*M*+30*N*	+70*M*+104*N*	+71*M*+102*N*

The speedups in the basic operations can lead to major performance gain in 3D graphics processing kernels, as illustrated in the last five rows in Table 4. They correspond to the transform and lighting kernels with different lighting configurations: no light sources, with one direct source, with one point source, with one spotlight source, and with *L* direct sources, *M* point sources, and *N* spotlights. The PLX FP set takes roughly 47% fewer instructions than the baseline set, with the scale instructions accounting for about 17% to 21%, and the dot product instructions for the rest. Notice that the baseline set already performs noticeably better than either AltiVec (about 17%~30% reduction) or IA-64 (about 30%~60% reduction). This performance advantage mainly comes from the exponentiation approximation instructions defined in PLX FP, which are not discussed in this paper. AltiVec and IA-64 do not have such instructions and have to use many instructions to compute exponentiation.

Instruction count is a good indicator of performance for heavily parallel applications such as 3D graphics. There are typically a large number of primitives to be processed with the same kernel. By interleaving the processing of multiple primitives, we can hide multi-cycle instruction latencies and achieve an effective throughput of approximately *n* cycles per primitive, where *n* is the number of instructions in the kernel. Assuming, the FMAC units have the same cycle time for all the instruction sets, then lower instruction count is equivalent to higher performance. For example, a 47% reduction in instruction count corresponds to a speedup of about 100/53≈1.9.

B. *Performance impact of latency and cycle time*

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = x \begin{pmatrix} m_{00} \\ m_{10} \\ m_{20} \\ m_{30} \end{pmatrix} + y \begin{pmatrix} m_{01} \\ m_{11} \\ m_{21} \\ m_{31} \end{pmatrix} + z \begin{pmatrix} m_{02} \\ m_{12} \\ m_{22} \\ m_{32} \end{pmatrix} + w \begin{pmatrix} m_{03} \\ m_{13} \\ m_{23} \\ m_{33} \end{pmatrix}$$

Figure 7 Matrix-vector transform operation

As seen in Section III.B, for a given ISA, there are several design and implementation choices with different latency and cycle time. Now we look at how the cycle time and latency of the FMAC unit can affect the performance of 3D graphics processing. As an example, the matrix-vector transform shown in Figure 7 can be done with four PLX FP instructions:

```
pfscale,3     F3,F2,F10
pfscaleadd,2  F3,F2,F11,F3
pfscaleadd,1  F3,F2,F12,F3
pfscaleadd,0  F3,F2,F13,F3
```

where registers F10-F13 hold the four columns of the transform matrix, F2 contains the input vector, and the output vector is stored in register F3. Assuming the FMAC unit has a *k*-cycle latency and is fully pipelined. Due to the serial data dependencies in this sequence, it takes 4*k* cycles to complete the operation, leaving *k*-1 bubbles between each pair of adjacent instructions. We can fill the bubbles by interleaving the execution of several independent transforms. For example, with a 3-cycle FMAC unit, we can interleave three transforms

```
pfscale,3     F3,F2,F10      ;V1
pfscale,3     F5,F4,F10      ;V2
pfscale,3     F7,F6,F10      ;V3
pfscaleadd,2  F3,F2,F11,F3   :V1
pfscaleadd,2  F5,F4,F11,F5   ;V2
pfscaleadd,2  F7,F6,F11,F5   ;V3
pfscaleadd,1  F3,F2,F12,F3   ;V1
pfscaleadd,1  F5,F4,F12,F5   ;V2
pfscaleadd,1  F7,F6,F12,F5   ;V3
pfscaleadd,0  F3,F2,F13,F3   ;V1
pfscaleadd,0  F5,F4,F13,F5   ;V2
pfscaleadd,0  F7,F6,F13,F5   ;V3
```

The whole sequence takes 14 cycles to execute, resulting in an effective throughput of 4.67 cycles per transform. For a 5-cycle FMAC unit, we can hide all the latencies by interleaving five transforms, which consumes 24 cycles total or 4.8 cycles per transform. In general, when executing an *n*-instruction processing sequence on a processor with a *k*-cycle maximum latency, interleaving the execution of *m* such sequences costs a total number of *mn*+*k*-1 cycles, for *m*≥*k*. This translates into an effective throughput of *n*+(*k*-1)/*m*. For a *k*-cycle FMAC unit, as long as we have sufficient degree of interleaving (*m*≥*k*), different latencies (values of *k*) have negligible effect on performance when the cycle time is kept constant.

The cycle time of the FMAC unit, on the other hand, has significant impact on performance because execution time is proportional to cycle time. Slower cycle time results in lower performance. Table 3 suggests that the implementation of an FMAC unit can achieve shorter cycle time with longer latency. In 3D graphics processing, since the latencies can be completely hidden with sufficient degree of interleaving, an implementation with shorter cycle time can achieve better performance despite the longer latency. The longer minimum cycle time for the PLX FP set can undermine its benefit. For example, the 3-cycle FMAC unit implementation of the PLX FP set has a minimum cycle time of 2.82ns, while the baseline FMAC unit can reach 1.79ns. Suppose the PLX FP set uses 47% fewer instructions than the baseline set for the same task, when taking the cycle time into account, the performance

speedup drops to about 1.9*1.79/2.82≈1.21 instead of 1.9. We can compensate for the cycle time disadvantage of the PLX FP set and improve its performance by increasing the FMAC latency. For example, the minimal cycle time of the PLX FP FMAC unit drops to 1.94ns when its latency is increased to 5-cycle. The speedup over the 3-cycle baseline FMAC unit rises to about 1.9*1.79/1.94≈1.75.

V. CONCLUSIONS

In this paper, we study the FMAC unit implementations for three sets of PLX FP instructions and evaluate the performance of the new PLX FP scale and dot product instructions. The results show that with about 14% area increase over the baseline set, the addition of the new scale and dot product instructions in the PLX FP set results in about 47% instruction reduction for the transform and lighting kernel. This corresponds to a speedup of about 1.9, assuming the FMAC unit implementations have the same cycle time. However, the increase in minimum cycle time caused by the new instructions can partially offset their performance gain. This can be ameliorated by a longer latency for the FMAC unit for the PLX FP set. We also showed that in highly parallel applications like 3D graphics, the performance is not sensitive to increased latency, in cycles, of an instruction. Interleaving the processing of different primitives can hide this latency.

In summary, the most important factors affecting the performance of 3D graphics appear to be the number of instructions needed for frequently executed graphics kernels and the cycle time of the processor. To maximize performance, more powerful instructions that reduce the number of instructions per kernel should be implemented with more cycles of latency rather than causing the cycle time to increase.

REFERENCES

[1] X. Yang and R. B. Lee, "Adding 3D Graphics Support for PLX", *Proceedings of the 2003 IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, August 2003.

[2] X. Yang and R. B. Lee, "PLX FP: An Efficient Floating-Point Instruction Set for 3D Graphics", *Proceedings of the 2004 IEEE International Conference on Multimedia and Expo (ICME 2004)*, June 2004.

[3] R. B. Lee and A. M. Fiskiran, "PLX: A Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing", *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME 2002)*, pp. 117-120, August 2002.

[4] Intel Corporation, Intel Architecture Software Developer's Manual, 2003.

[5] S. Obeman, G. Favor, and F. Weber, "AMD 3Dnow! Technology: Architecture and Implementations", *IEEE Micro*, vol. 19(2):37-48, April 1999.

[6] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec Extension to PowerPC Accelerates Media Processing", *IEEE Micro*, vol. 20(2):85-95, April 2000.

[7] R. Thekkath, M. Uhler, C. Harrell, and W. Ho, "An Architecture Extension for Efficient Geometry Processing", *Proceedings of Hot Chips 11 - A Symposium on High Performance Chips*, August 1999.

[8] Intel Corporation, IA-64 Application Developer's Architecture Guide, May 1999.

[9] R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors", *IEEE Micro*, vol. 15(2):22-32, April 1995.

[10] D. Zucker, R. Lee, and M. Flynn, "Achieving Subword Parallelism by Software Reuse of the Floating-Point Data Path", *Proceedings of SPIE: Multimedia Hardware Architectures 1997*, pp. 51-64, January 1997.

[11] W. Ma and C. Yang, "Using Intel Streaming SIMD Extensions for 3D Geometry Processing", *Proceedings of the 3rd IEEE Pacific-Rim Conference on Multimedia*, December 2002.

[12] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit", *IBM J. Res. Develop.*, vol. 34(1):59-70, January 1990.

[13] C. Heikes and G. Colon-Bonet, "A Dual Floating Point Coprocessor with an FMAC Architecture", *ISSCC Dig. Tech. Papers*, pp. 354-355, Feburary 1996.

[14] F. Arakawa, O. Nishii, K. Uchiyama, and N. Nakagawa, "SH-4 RISC Microprocessor for Multimedia", *Proceedings of Hot Chips 9 - A Symposium on High Performance Chips*, August 1997.

[15] A. Kunimatsu *et. al*, "Vector Unit Architecture for Emotion Synthesis", *IEEE Micro*, vol. 20(2):40-47, April 2000.

[16] N. Quach and M. Flynn, "Suggestions for Implementing a Fast IEEE Multiply-Add-Fused Instruction", Standford Computer Systems Laboratory Technical Report CSL-TR-91-483, July 1991.

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

[18] LSI Logic Corp., Gflx-p Cell-Based ASIC Products Databook for Internal Macrocells, December 2002.

[19] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. Computer Graphics, Principles and Practice, Second Edition in C. *Addison-Wesley*, 1997.