PLX FP: An Efficient Floating-Point Instruction Set for 3D Graphics

Xiao Yang and Ruby B. Lee
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Princeton University
{xiaoyang, rblee}@princeton.edu

Abstract

3D graphics is an important component in the workload of today's computing platforms. Many ISA extensions for 3D graphics have been proposed and implemented. We describe PLX FP, a new floating-point extension to the PLX architecture, designed to support very efficiently the essential operations needed for the 3D graphics pipeline. Very high performance floating-point 3D graphics processing is achieved, using a low-cost PLX processor.

1. Introduction

The importance of multimedia processing on general-purpose computing platforms has prompted processor designers to add multimedia instructions to microprocessor instruction set architectures (ISAs). These include MAX-2 for the PA-RISC architecture [1], MMX, SSE and SSE-2 for the Intel IA-32 architecture [2], and a superset of these to the Itanium IA-64 architecture [3]. Although these multimedia instructions may be very effective, they still incur the overhead of their base microprocessor ISA. PLX [4] is a new ISA designed from scratch for fast and efficient multimedia processing. Prior work has demonstrated its effectiveness for integer media applications [4].

This paper describes the new floating-point ISA for PLX version 1.3, designed to enable support for very fast 3D graphics. With the proliferation of 3D games, it is highly desirable to support fast 3D graphics with the same media processor used for integer media types like images, video and audio. Although 3D graphics has traditionally been handled by separate graphics processors and boards, this is often infeasible for handheld computers. PLX FP provides a more economical solution for 3D graphics in such cases.

2. Past work

Figure 1 shows a 3D graphics pipeline used to perform 3D graphics rendering, as defined for example, by the OpenGL [5] standard. The 3D graphics pipeline is implemented in many different ways. On some high-end platforms like the SGI RealityEngine system [6], it is implemented with specialized graphics rendering boards. However, such dedicated hardware is expensive and not flexible because it executes fixed functions. One alternative is to use pure software running on general-purpose processors, such as Mesa3D [7]. However, the performance is very low because the processors are not optimized for 3D graphics processing. The addition of 3D graphics ISA extensions, such as SSE and SSE-2 [2], IA-64 [3], AMD 3DNow! [8], AltiVec in PowerPC [9], and recently, ARM VFP [10], partially alleviates the problem. They boost the speed of certain 3D graphics operations significantly, but the performance still falls short of that demanded for real-time rendering.

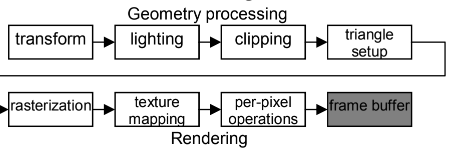

Figure 1: A simple 3D graphics pipeline

The common approach for desktop 3D graphics is to use both software and hardware, where part of the pipeline is executed in software, and the rest is done with a dedicated graphics processor or board. Early generations of the graphics boards only perform simple texture mapping and drawing in the back-end of the graphics pipeline, with the majority of the 3D operations performed in software. The newer graphics processors, such as the GeForce series by nVIDIA [11] and the Radeon series by ATI [12], support more geometry processing operations at the front-end of the 3D graphics pipeline with more FP capability and increased programmability. This approach provides good leverage of the host processor and the graphics processor. However, the graphics processor cannot be used for other tasks, and having a separate graphics processor is not economical in constrained environments like PDAs.

3. PLX FP ISA

PLX FP introduces 32 new FP registers F0-F31, with register F0 hardwired to the value 0.0. Like the PLX integer ISA, the PLX FP ISA is also datapath scalable and fully subword-parallel [4]. Datapath scalability means that the PLX floating-point registers and functional units can

0-7803-8603-5/04/$20.00 ©2004 IEEE.

be 32, 64 or 128 bits in a given PLX implementation. The recommended datapath width is 128 bits, corresponding to a 4-component FP vector, which is the most common data type in 3D graphics. The FP datapath of PLX is independently scalable from the integer datapath. For example, a PLX implementation can have 64-bit integer registers and 128-bit floating-point registers. Unlike the integer ISA, PLX FP has only one subword size of 32 bits, corresponding to a single-precision FP number. The subwords in a register are numbered from right to left, with the rightmost subword labeled 0.

There are six classes of floating-point instructions. Subword-parallel FMAC instructions, shown in Table 1, can be executed on the basic floating-point multiply-accumulate (FMAC) functional unit. All these instructions, except `scale`-related and `pfdp` instructions, also have a scalar version in addition to the subword-parallel version shown in the table, which operates on the rightmost subwords of the operands.

Table 2 shows the FP approximation instructions. These perform the mathematical functions with reduced precision (8 or more bits) on the rightmost subword in the source register, which is much faster than computing these functions with full precision. This is very useful in many 3D graphics applications where speed is more important than very accurate results. When more accuracy is required, these instructions can also act as seeds for iterative refinement methods such as the Newton-Raphson method as shown in [13] to generate higher-precision results.

FP data rearrangement instructions are shown in Table 3. `fpermute` provides a general way to re-order the subwords within a single register, while `fmix`, `fextract`, and `fdeposit` can reorganize subwords from multiple registers.

Table 4 shows the FP compare instructions. The two `fcmp` instructions compare the rightmost subwords in the source operands and update a pair of predicates. The `pcmp` instructions compare all pairs of subwords and write all 1's or 0's to the respective subwords in the destination register, depending on the outcomes of the comparisons.

PLX FP also has data conversion instructions which convert FP data to integers of various sizes and vice versa, and FP load/store instructions which move data between FP registers and memory. PLX FP also shares the branch instructions already defined for integer PLX (PLX 1.0 to 1.2), thus eliminating the need for new branch instructions.

4. Implementing 3D graphics kernels

We now demonstrate a few of the more unusual and powerful instructions in PLX FP in implementing some of the most important kernel operations in the 3D graphics pipeline. Assume a processor with 128-bit FP registers and single-issue instruction execution.

Table 1: FMAC instructions

Instruction	Description	Mnemonic
add	$d_i=a_i+b_i$	`Pfadd`
add negate	$d_i=-(a_i+b_i)$	`pfadd.neg`
subtract	$d_i=a_i-b_i$	`pfsub`
multiply	$d_i=a_i\times b_i$	`pfmul`
multiply negate	$d_i=-a_i\times b_i$	`pfmul.neg`
scale, j	$d_i=a_j\times b_i$	`pfscale,j`
scale negate, j	$d_i=-a_j\times b_i$	`pfscale.neg,j`
dot product	$d_0=\sum a_i\times b_i$	`pfdp`
dot product w/ sat	$d_0=\sum a_i\times b_i$, $d\geq 0$	`pfdp.s`
dot product negate	$d_0=-\sum a_i\times b_i$	`pfdp.neg`
dot product neg w/ sat	$d_0=-\sum a_i\times b_i$, $d\geq 0$	`pfdp.s.neg`
absolute	$d_i=\|a_i\|$	`pfabs`
absolute negate	$d_i=-\|a_i\|$	`pfabs.neg`
max	$d_i=\max(a_i,b_i)$	`pfmax`
min	$d_i=\min(a_i,b_i)$	`pfmin`
multiply-add	$d_i=a_i\times b_i+c_i$	`pfmuladd`
multiply-add negate	$d_i=-(a_i\times b_i+c_i)$	`pfmuladd.neg`
multiply-subract	$d_i=a_i\times b_i-c_i$	`pfmulsub`
multiply-subract negate	$d_i=-(a_i\times b_i-c_i)$	`pfmulsub.neg`
scale-add, j	$d_i=a_j\times b_i+c_i$	`pfscaleadd,j`
scale-add negate, j	$d_i=-(a_j\times b_i+c_i)$	`pfscaladd.neg,j`
scale-subtract, j	$d_i=a_j\times b_i-c_i$	`pfscalesub,j`
scale-subtract negate, j	$d_i=-(a_j\times b_i-c_i)$	`pfscalesub.neg,j`

Table 2: FP math approximation instructions

Instruction	Description	Mnemonic
reciprocal approx	$d_0\approx 1/a_0$	`frcpa`
reciprocal sqrt approx	$d_0\approx 1/(a_0^{1/2})$	`frcpsqrta`
log base 2 approx	$d_0\approx\log_2 a_0$	`flog2a`
exp base 2 approx	$d_0\approx 2^{a}{}_0$	`fexp2a`

Table 3: FP data rearrangement instructions

Instruction	Description	Mnemonic
permute	Any i^i permutation of the i subwords in 1 source reg	`fpermute`
mix {left, right}	Interleave {odd, even} subwords from 2 regs	`fmix.l` `fmix.r`
extract, j	$d_o=a_j$	`fextract,j`
deposit and zero, j	$d_j=a_0$, $d_{i,i\neq j}=0$	`fdeposit.z,j`
deposit and keep, j	$d_j=a_0$	`fdeposit.k,j`

Table 4: FP compare instructions

Instruction	Description	Mnemonic
compare rel single	$P_{d1}=rel(a_0,b_0)$, $P_{d2}=!P_{d1}$	`fcmp.rel`
compare rel single pw1	If $rel(a_0,b_0)$=TRUE, then $P_{d1}=1$, $P_{d2}=0$	`fcmp.rel.pw1`
compare rel parallel	if $rel(a_i,b_i)$=TRUE, then $c_i=1$, else $c_i=0$	`pfcmp.rel`

4.1. 4×4 matrix-vector transform

4×4 matrix-vector transform, shown below, is used for geometry transforms in the 3D graphics pipeline. This operation is normally performed multiple times on each vertex, making it one of the most frequently used operations in the graphics pipeline:

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

$$x' = m_{00}x + m_{01}y + m_{02}z + m_{03}w \quad \text{Similarly for } y', z', w'.$$

Based on the following observation, the whole operation can be done with 4 PLX FP instructions:

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

```
pfscale,3       F2, F1, F10
pfscaleadd,2    F2, F1, F11, F2
pfscaleadd,1    F2, F1, F12, F2
pfscaleadd,0    F2, F1, F13, F2
```

F1 contains the source vector, F2 holds the result, and F10-F13 hold the four columns of the matrix. The `pfscale` instruction computes the first term by multiplying the first column of **M** by subword 3 in F1, which is *x*. Each of the subsequent `pfscaleadd` instructions computes one other term and accumulates the result from the previous instruction. This causes data dependencies between adjacent instructions. If the instructions have multi-cycle latencies, then transforming a vector takes more than four cycles. To solve this problem, the transformation of several vectors can be interleaved to achieve a very fast throughput of one vector every four cycles.

4.2. Vector normalization

Vector normalization is an important operation since several steps in lighting computations require normalized direction vectors. This consists of a dot product, a reciprocal square root and vector scaling, achievable with just 3 PLX FP instructions:

$$\mathbf{V}' = \begin{pmatrix} x' \\ y' \\ z' \\ 0 \end{pmatrix} = \mathbf{V}/|\mathbf{V}| = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} / \sqrt{x^2 + y^2 + z^2}$$

```
pfdot           F3, F1, F1
frcpsqrta       F3, F3
pfscale,0       F2, F3, F1
```

The `pfdot` multiplies corresponding subwords in the two source registers and adds these, giving the dot product. The `frcpsqrta` approximates the reciprocal square root of the operand to more than 8 bits. This is sufficient for color calculation where full precision is not required, and avoids the long latency otherwise needed for full precision.

4.3. Perspective division

In the 3D graphics pipeline, perspective division is used to convert clip coordinates to normalized device coordinates needed for rendering. This operation divides each of the coordinates by *w*:

$$\mathbf{V}' = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \mathbf{V}/w = \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

The accuracy required of this operation varies with the targeted display resolution. The following PLX FP code demonstrates how the desired precision can be achieved:

```
frcpa           F10, F1
pfscale,0       F2, F10, F1
fmulsub.neg     F11, F3, F1, F12
pfscaleadd,0    F3, F11, F2, F2
pfscaleadd,0    F4, F11, F3, F2
```

This sequence of code implements the following [13]:

$$t \approx 1/w = \mathtt{frcpa}(w)$$
$$\mathbf{V}' = t\mathbf{V}$$
$$q = 1.0 - tw$$
$$\mathbf{V}'' = \mathbf{V}' + q\mathbf{V}'$$
$$\mathbf{V}''' = \mathbf{V}' + q\mathbf{V}'' = \mathbf{V}' + q\mathbf{V}' + q^2\mathbf{V}'$$

F1 contains the vector **V** and F12 contains the value 1.0. If low precision is sufficient, we only need the first two instructions: `frcpa` gives an 8-bit approximation of the reciprocal of *w* in F10, while `pfscale,0` multiplies the vector in F1 with this approximate value, yielding $\mathbf{V}'$ in F2. When more precision is required, the third instruction computes *q* and stores it in F11. With the last two instructions, we can obtain the full-precision result $\mathbf{V}'''$ in F4. F3 and F4 contain progressive refinements of the initial result in F2. Thus, a less accurate result is achieved quickly, and more accurate results with more cycles.

4.4. Cross-product of 3-component vectors

Cross product of two 3-component direction vectors is necessary when doing backface culling and texture-space lighting, etc. The operation is given below:

$$\mathbf{V}' = \begin{pmatrix} x' \\ y' \\ z' \\ 0 \end{pmatrix} = \mathbf{V}_1 \times \mathbf{V}_2 = \begin{pmatrix} y_1z_2 - y_2z_1 \\ z_1x_2 - z_2x_1 \\ x_1y_2 - x_2y_1 \\ 0 \end{pmatrix}$$

To implement this, we can first use the `fpermute` instruction to rearrange the components of vectors $\mathbf{V}_1$ and $\mathbf{V}_2$, and then perform the multiplications and subtraction, as shown below:

```
fpermute,2130  F4, F1
```

```
fpermute,1320  F5, F1
fpermute,2130  F6, F2
fpermute,1320  F7, F2
pfmul          F8, F5, F6
pfmulsub       F3, F4, F7, F8
```

F1 and F2 contain the source vectors ($x_1y_1z_10$) and ($x_2y_2z_20$), respectively. `fpermute` rearranges the subwords in the operand register according to the order described by the subop field: the rightmost subword is labelled 0 and the leftmost is labelled 3 in a 128-bit register with four 32-bit subwords. For example, after the first two `fpermute` instructions, F4 contains ($y_1z_1x_10$) and F5 contains ($z_1x_1y_10$).

4.5. Exponentiation

Exponentiation $d=a^b$ is a key operation for doing specular lighting and spot lighting. As for many other operations in lighting, an exact result is not needed. Normally, exponentiation is a very expensive operation; we use approximation instructions in PLX FP to obtain an approximate result quickly, using the equation $a^b=2^{b\log_2 a}$. The rightmost subwords of F1 and F2 have the values a and b in the following:

```
flog2a   F4, F1
fmul     F5, F4, F2
fexp2a   F3, F5
```

4.6. Performance with datapath scalability

Table 5 shows the instruction counts for the above examples with different FP word sizes.

Table 5: Performance for different word sizes

	xform	norm	div	cross	exp
FP128	4	3	2	6	3
FP64	8	6	3	10	3
FP32	16	7	5	6	3
RISC32	28	13	4[1]	9	>10

1. Using four full-precision division instructions taking ~17 cycles each.

RISC32 represents a typical RISC FP ISA with 32-bit FP registers and no special instructions for 3D graphics support. We use it as a baseline for comparison. Even our smallest PLX FP implementation, FP32, offers significant performance advantage over RISC32. In general, the performance trend is FP128 > FP64 > FP32. In the transform case where subword parallelism is the most abundant, the performance scales linearly. In other cases, where only one or three subwords are used per vertex, the performance grows less than linearly. Cross product is an extreme case: for FP32, neither data rearrangement nor under-utilization of subword parallelism occurs. For FP128, data rearrangement is done per register. FP64 is the most inefficient because data rearrangement needs to be done across registers.

5. Comparison with other FP extensions

Table 6 briefly summarizes the feature set of PLX FP and notes whether these features are present in other FP ISA extensions for 3D graphics (see Section 2). PLX FP provides advanced features others do not.

Table 6: Feature set of PLX FP

Subword-parallelism	All except ARM VFP
Datapath scalability	PLX FP
Vector scaling, dot product	PLX FP
Approximation instructions	PLX FP, AltiVec, 3DNow!, IA-64[1]
Permutation within a register	PLX FP, AltiVec, IA64[2]
Predicated execution	PLX FP, IA-64

1. 3D Now! and IA-64 have reciprocal and reciprocal square root approximations only.
2. IA-64 can swap two 32-bit FP subwords in one register.

6. Conclusions

3D graphics is increasingly popular, creating a demand for floating-point processing with characteristics different from scientific floating-point processing. Our proposed PLX FP instruction set addresses the need for economical yet high-performance 3D graphics. The PLX datapath scalability feature allows a range of performance and cost targets. PLX FP's rich feature set makes it a highly versatile and effective ISA for 3D graphics.

7. References

[1] R. B. Lee, "Subworld Parallelism with MAX-2", *IEEE Micro*, 16(4):51-59, August 1996.

[2] Intel Corporation, Intel Architecture Software Developer's manual, 2003.

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

[4] 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.

[5] M. Segal and K. Akeley, The OpenGL Graphics System: A Specification (Version 1.4), 2002.

[6] K. Akeley, "RealityEngine Graphics", Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 109-116, September 1993.

[7] B. Paul, Mesa 3D project, http://www.mesa3d.org.

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

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

[10] D. R. Lutz and C. N. Hinds, "Accelerating Floating-Point 3D graphics for Vector Microprocessors", *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[11] nVIDIA Corporation, GeForce family of graphics processors, http://www.nvidia.com

[12] ATI Technologies Inc., Radeon family of graphics processors, http://www.ati.com

[13] P. Markstein, IA-64 and Elementary Functions: Speed and Precision, Prentice Hall, 2000.