Adding 3D Graphics Support to PLX

Xiao Yang and Ruby Lee
Department of Electrical Engineering
Princeton University
{xiaoyang, rblee}@princeton.edu

***Abstract*—PLX is a compact, fully subword-parallel instruction set architecture (ISA) for very fast multimedia processing. This paper adds floating-point instructions to PLX for 3D graphics processing, which is essential for applications such as games and digital content creation. Based on an analysis of the 3D graphics pipeline from an ISA point of view, we show the operations and data types needed. We present the FP ISA and demonstrate its use and performance with code examples from the 3D graphics pipeline.**

Keywords—processor architecture, instruction-set architecture (ISA), floating-point, multimedia, 3D graphics

I. INTRODUCTION

Multimedia information processing is an important part of the workload for today's computing platforms, ranging from high-end desktop workstations to low-end portable devices. High-performance multimedia processing is one of the key design goals for contemporary microprocessors. Common operations for media processing are part of the core ISA of processors instead of being merely ISA extensions, as in the past. PLX [1], developed at Princeton University, is a fully subword-parallel ISA designed specifically for very fast multimedia processing. The initial release of PLX includes only integer instructions targeting integer-based media applications, such as image and video processing. In this paper, we present the floating-point (FP) part of PLX designed for fast 3D graphics processing.

3D graphics is dominant in areas such as gaming, digital content creation, and simulation. Most 3D graphics applications are built on top of standard 3D graphics libraries, such as OpenGL [2] and Direct3D [3]. These libraries provide a set of APIs for the applications to specify the composition of the scenes and how they are to be rendered. Underlying these libraries is a 3D graphics processing pipeline, which takes the scene descriptions from the applications, processes them, and draws them into 2D images to be displayed on a screen. The performance of this 3D processing pipeline, which is floating-point intensive, determines the performance of 3D applications. The ultimate goal of 3D graphics processing is to render photo-realistic scenes in real-time. Traditionally, the operations of the 3D pipeline are handled either by general purpose processors, with or without ISA extensions for 3D, or by specialized graphics hardware. Both approaches have their shortcomings. The general purpose processor approach lacks high performance, while the specialized graphics hardware approach is deficient in programmability. By adding 3D graphics support to PLX, we can achieve higher performance 3D processing than general purpose processors with 3D extensions. This processor ISA approach could also inform the design of specialized 3D graphics processors, where programmability is becoming increasingly important.

In Section 2, we survey past work on ISA for 3D graphics processing. In Section 3, we investigate the operations of a 3D graphics pipeline and their characteristics. In Section 4, we present our PLX floating-point ISA for 3D graphics processing. In Section 5, we show code examples written with the new instructions. Section 6 concludes the paper.

II. PAST WORK

In the early days of real-time 3D graphics processing, this capability was only available on high-end systems with expensive dedicated hardware, such as the SGI Reality Engine system [4]. These systems employ a fixed function pipeline. Since mid-1990s, graphics hardware gradually became available to consumer desktops. However, these hardware only handle basic drawing functions rather than the full spectrum of the 3D pipeline. Since then, the development of 3D graphics processing has forked into two directions. One direction adds basic operations for 3D graphics into general purpose processor ISAs as extensions, such as SSE-2 extension to Intel x86 [5], AltiVec extension to PowerPC [6], 3DNow! in AMD x86 [7], and MIPS-3D extension to MIPS64 [8]. However, these processors are more targeted for traditional workloads such as business and scientific applications. IA-64 [9] is a big step forward in that it includes instructions useful to 3D graphics from the beginning. The other direction is to continue using a specialized graphics processor while improving its functionality and speed. This approach led to the migration of more of the 3D pipeline operations from the CPU to the graphics processor, and finally, to the implementation of the entire 3D processing pipeline on a graphics processor. The first such product is the GeFORCE 256 processor by nVIDIA [10]. Up until then, the pipeline implemented was mostly fixed function with little programmability restricted to some degree of configurability. As high-quality, photo-realistic graphics becomes the center of attention, many new algorithms have been developed, and programmability becomes a key goal of 3D processor design. The fixed-function pipeline model is gradually being abandoned and ISA-like programmability is added. The programmable shader [3] in Microsoft Direct3D is the latest attempt. It defines a set of operations that can be used to implement different algorithms. However, it does not encompass the entire pipeline, and certain functionalities are

This work was supported in part by a research gift from Hewlett-Packard Laboratories.

still implemented by fixed-function hardware. More operations are being added as more algorithms are devised. The advantage of the specialized 3D processor approach is higher performance compared with the general purpose processor approach; the downside is limited general purpose applicability. PLX with 3D graphics addresses the shortcomings on both sides: higher performance than general-purpose architectures with full programmability.

III. Operations of a 3D Graphics Pipeline and Their Characteristics

A 3D graphics pipeline is divided into two major phases: geometry processing and rendering. The inputs to the graphics pipeline are scene descriptions. The top level of scene descriptions consists of objects, light sources, materials, and textures. Each object is composed of primitives such as points, lines, and polygons, where the most commonly used primitives are triangles. A primitive is represented with a collection of vertices. Each vertex is associated with a set of properties, including coordinates (*xyzw*), normal (*xyz*), material index, and multiple sets of texture coordinates (*rstq*), most of which are FP quantities. Aside from the vertices, each object is associated with a set of transform matrices to be applied onto every vertex in that object. Light sources and materials are also each associated with a set of properties represented with FP or fixed point numbers. Textures are essentially one, two or three dimensional tables referenced by the texture coordinates. Each entry of the table contains a color value (which can also be interpreted in other ways) normally consisting of three (*RGB*) or four (*RGBA*) integer numbers. Managing the scenes and deciding which objects are to be shown are done by the applications. Processing and drawing of the scenes are done by the 3D graphics pipeline.

The data flow of a typical 3D pipeline is shown in Figure 1. In the geometry processing phase, the vertices (coordinates and normals) are first transformed to eye space and lit, then transformed to clip space and assembled into triangles. The triangles are clipped against the view frustum, and finally the clipped triangles are transformed to the screen space. In the rendering phase, the triangles from the previous stage are rasterized into groups of pixels. During the rasterization step, the parameters associated with the vertices of a triangle are interpolated to obtain the parameters for each pixel in the triangle. These parameters include screen coordinates, depth, colors, texture coordinates, etc. The generated pixels then go though texture mapping, fogging, a series of tests, and alpha blending on a per-pixel basis. Finally the pixels are written to the frame buffer and form the final output image.

Nowadays the implementation of the entire 3D pipeline is being moved to FP, since error accumulation in integer calculation can produce serious visual artifacts [11]. Single-precision floating-point data representation is used in the 3D graphics processing pipeline due to its large dynamic range compared to fixed-point or integer data with the same number of bits. In this paper, we do not focus on the algorithmic aspects of the 3D pipeline, but rather its computational characteristics and their implications for ISA design. Therefore, we do not show the exact computations in 3D graphics, since these can change with different algorithms; rather we discuss the types of operations which are common across different algorithms.

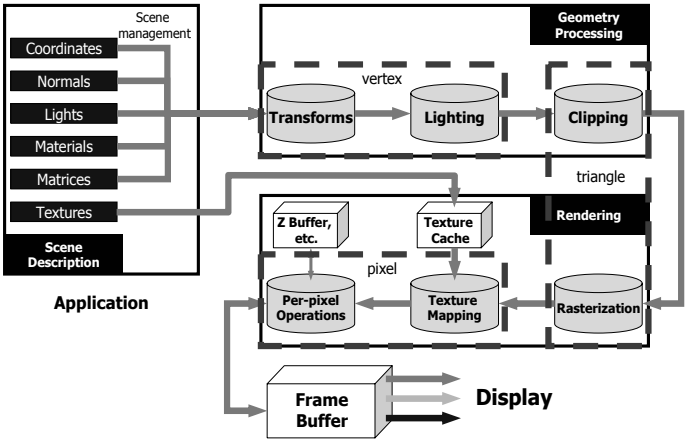

Figure 1 3D processing pipeline

We classify the basic operations in a 3D pipeline into the following categories. **Data-parallel arithmetic operations** perform the same calculation on multiple sets of data in parallel. Examples are parallel add, subtract, and multiply. Most operations on coordinates, normals, and colors fall into this category. A very common operation is the multiplication of a 4×4 matrix with a 4-element vector. **Scalar arithmetic operations** like add, subtract and multiply are used when computing coefficients in lighting and rasterization. **Advanced math operations** like reciprocal, reciprocal square root, and exponentiation are needed at various precisions. For example, computation of coordinates needs high precision in order to avoid visual artifacts, while color calculation requires lower accuracy. These operations are expensive to implement, therefore carrying out each of them in full precision is not necessary if lower precision is sufficient and less costly. **Compare and conditionally executed operations** are important in 3D graphics processing, although not very common. During the clipping step, vertex coordinates of vertices are compared to their bounding volume to check whether clipping is needed. New algorithms also allow vertices or pixels to be handled differently basing on their values, where compare and conditional execution is necessary. Conditional execution may be implemented with predicated execution, which is available in PLX. **Memory access operations** are intensive in the 3D graphics pipeline. They are used to import data into the pipeline, output processed triangles to the rendering phase, load textures, read and write the depth buffer and the frame buffer. The data in these accesses can be either FP or integer data. **Data rearrangement operations** do not alter the values of data, but rather change the order of data. They are convenient when reordering of data is needed to facilitate parallel computation. **Data conversion operations** are needed because when data are imported into the pipeline, they are not always in FP representation. For example, colors and textures are usually of integer types. The final rendered image is also in integer format. Therefore, proper data conversions need to be performed in various places in the 3D processing pipeline to convert integer to FP and vice versa.

IV. PLX FP INSTRUCTIONS FOR 3D GRAPHICS

The FP instructions of PLX are classified as follows: FMAC instructions, FP compare instructions, FP math approximation instructions, FP memory access instructions, FP data rearrangement and data conversion instructions. All instructions are 32 bits and the subword size is always 4 bytes, to represent a 32-bit single-precision FP number. There are 32 FP registers. Register F0 is hardwired to 0.0 and register F1 is hardwired to 1.0. Figure 2 shows the PLX FP datapath. The FP ISA continues the word size scalability feature of PLX, where the size of the registers can be 32, 64 or 128 bits, denoted PLX-32, PLX-64 and PLX-128, respectively. PLX-32 has no subword-parallelism, PLX-64 has two subwords in each register, and PLX-128 has four. PLX-128 is the preferred implementation, because the register size matches the size of a 4-component FP vector, which is the most commonly used data type in the 3D graphics pipeline.

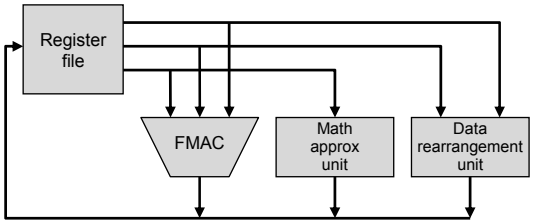

Figure 2 PLX FP datapath with three functional units: FMAC, math approximation unit, and data rearrangement unit

A. FMAC instructions

The floating-point multiply-accumulate unit (FMAC) is the basic functional unit in FP processing. It can perform FP operations such as add, subtract, multiply and multiply-add. All the instructions in this class generate full-precision results. Table I shows the instructions and their descriptions. The mnemonics of subword-parallel instructions start with a "p". For most subword-parallel instructions, the same operation is performed on every corresponding set of subwords in the registers, with the exception of `pfscale` instructions, where the first operands are always the *j*-th subword in the first source register. In `pfdp` instructions, the results are written to the rightmost subword in the destination register.

There are also scalar versions of these instructions (not shown), where only the rightmost subwords in each register are used as operands. These instructions have the same mnemonics as the subword-parallel instructions but without the leading "p". Notice that the scalar versions of `pfscale` instructions and `pfmul` instructions are equivalent. Therefore we only define `fmul` instructions for scalar operations. There are also no scalar versions for `pfdp` instructions. Less common instructions such as `pfscale` and `pfdp` instructions are included due to their usefulness in vector operations. `pfscale` and `pfdp` can be used to scale vectors and to take the dot product of two vectors, respectively, both of which are common in 3D graphics processing. `pfscale` and `pfdp` can be implemented with slight modifications to standard FMACs. `pfabs` instructions are useful in clipping. `pfmin` and `pfmax` instructions are useful in pixel operations, and are also easy to implement.

TABLE I. FMAC INSTRUCTIONS

Instruction	Description	Mnemonic
pfadd	$d_i=a_i+b_i$	`pfadd`
pfadd negate	$d_i=-(a_i+b_i)$	`pfadd.neg`
pfsubtract	$d_i=a_i-b_i$	`pfsub`
pfmultiply	$d_i=a_i\times b_i$	`pfmul`
pfmultiply negate	$d_i=-a_i\times b_i$	`pfmul.neg`
pfscale	$d_i=a_j\times b_i$	`pfscale,j`
pfscale negate	$d_i=-a_j\times b_i$	`pfscale.neg,j`
pfdotproduct	$d=\sum a_i\times b_i$	`pfdp`
pfdotproduct w/ satuation	$d=\sum a_i\times b_i$, $d\geq 0$	`pfdp.s`
pfdotproduct negate	$d=-\sum a_i\times b_i$	`pfdp.neg`
pfdotproduct neg w/ sat	$d=-\sum a_i\times b_i$, $d\geq 0$	`pfdp.neg.s`
pfabsolute	$d_i=\lvert a_i\rvert$	`pfabs`
pfabsolute negate	$d_i=-\lvert a_i\rvert$	`pfabs.neg`
pfmax	$d_i=\max(a_i, b_i)$	`pfmax`
pfmin	$d_i=\min(a_i, b_i)$	`pfmin`
pfmultiply-add	$d_i=a_i\times b_i+c_i$	`pfmuladd`
pfmultiply-add negate	$d_i=-(a_i\times b_i+c_i)$	`pfmuladd.neg`
pfmultiply-subract	$d_i=a_i\times b_i-c_i$	`pfmulsub`
pfmultiply-subract negate	$d_i=-(a_i\times b_i-c_i)$	`pfmulsub.neg`
pfscale-add	$d_i=a_j\times b_i+c_i$	`pfscaleadd,j`
pfscale-add negate	$d_i=-(a_j\times b_i+c_i)$	`pfscaladd.neg,j`
pfscale-subtract	$d_i=a_j\times b_i-c_i$	`pfscalesub,j`
pfscale-subtract negate	$d_i=-(a_j\times b_i-c_i)$	`pfscalesub.neg,j`

B. Compare instructions

Compare instructions are used to generate conditions for later conditional executions. Compare instructions (see Table II) can also be implemented with the FMAC functional unit. Their definition parallels the integer compare instructions in PLX. The relation *rel* in the `fcompare` instructions can be any of `eq`(=), `ne`(≠), `ge`(≥), `gt`(>), `le`(≤), and `lt`(<).

TABLE II. FP COMPARE INSTRUCTIONS

Instruction	Description	Mnemonic
fcompare rel single	$P_{d1}=rel(a, b)$, $P_{d2}=!P_{d1}$	`fcmp.rel`
fcompare rel single pw1	if $rel(a, b)$ = TRUE, then $P_{d1}=1$, $P_{d2}=0$	`fcmp.rel.pw1`

Both `fcompare` instructions compare the rightmost subwords in the two source registers and write complementary results to a pair of predicate registers. The difference between the two is that the `pw1` version only updates predicates when *rel* is true. This enables multiple compares to write to the same pair of predicates in parallel. The destination predicate registers should be preset before an `fcmp.rel.pw1` instruction is executed.

Conditional execution is implemented with predicated execution. Instructions guarded with a predicate that is true will be executed, and nullified otherwise. Conditional branches that are not eliminated by predicated execution are done with predicated jump instructions.

C. FP math approximation instructions

Full-precision versions of mathematical functions such as reciprocal, reciprocal square root, and exponentiation are very slow and expensive to implement. As full-precision results are normally not needed in 3D graphics processing, approximations are used instead. These instructions are defined in Table III.

TABLE III. FP MATH APPROXIMATION INSTRUCTIONS

Instruction	Description	Mnemonic
freciprocal approx	$d \approx 1/a$	`frcpa`
freciprocal sqrt approx	$d \approx 1/(a^{1/2})$	`frcpsqrta`
flog base 2 approx	$d \approx \log_2 a$	`flog2a`
fexp base 2 approx	$d \approx 2^a$	`fexp2a`

These instructions operate on the rightmost subwords in the source and destination registers. Exponentiation calculation is decomposed into two instructions, `flog2a` and `fexp2a`, because function a^b is not convergent. The decomposition is based on the equality $a^b = 2^{b \times \log_2 a}$. All these instructions generate partial-precision results only. The accuracy in the results is greater than 8 bits. Higher-precision to full-precision results can be obtained with the FMAC by doing successive iterations based on the Newton-Raphson method [12]. With these instructions, we can control the accuracy and cost of the program. When lower precision is sufficient, we use coarser but faster approximation. When higher accuracy is required, we perform the slower high-precision calculation.

D. Memory access instructions

Memory access operations include loads and stores. A full register (or full word) is loaded or stored by each instruction shown in Table IV.

TABLE IV. FP MEMORY ACCESS INSTRUCTIONS

Instruction	Description	Mnemonic
fload	*d*=mem[*a*+*im*]	`fload`
fload index	*d*=mem[*a*+*b*]	`floadx`
fload update	*d*=mem[*a*+*im*], *a*=*a*+*im*	`fload.u`
fload index update	*d*=mem[*a*+*b*], *a*=*a*+*b*	`floadx.u`
fstore	mem[*a*+*im*]=*d*	`fstore`
fstore index	mem[*a*+*b*]=*d*	`fstorex`
fstore update	mem[*a*+*im*]=*d*, *a*=*a*+*im*	`fstore.u`
fstore index update	mem[*a*+*b*]=*d*, *a*=*a*+*b*	`fstorex.u`

There are also scalar versions of these instructions (not shown), where the data operand *d* is the rightmost subword in the respective register. In the table, *im* stands for immediate offset, *b* is register offset. We include both load/store word instructions and load/store single FP number instructions for flexibility. Since textures are mostly integer data, access to textures can be handled with integer loads.

E. Data rearrangement and data conversion instructions

Data rearrangement is useful for changing the order of data to make subsequent subword-parallel calculation easier. We define two instructions for data rearrangement. The `fmix` instruction is similar to the `mix` instruction defined in HP MAX-2 [13] and IA-64 [9], used to rearrange even (or odd) subwords from two registers into one register. Two variants, `fmix.l` (for even) and `fmix.r` (for odd) are defined. This instruction is very useful for matrix transposition. The `fpermute` instruction is similar to the `permute` instruction [13] and the `mux` instruction [9], used to obtain any permutations of up to four subwords in one register.

Data conversion is required at several places in a 3D graphics pipeline. We define six data conversion instructions, converting from 32-bit floats to 32-bit, 16-bit or 8-bit integers, and vice versa.

V. CODE EXAMPLES

We use 128-bit registers (PLX-128) in the examples below, where each register can hold a 4-component vector.

A. 4×4 matrix transform

This operation transforms a 4-component vector **V** into another vector **V′** by multiplying it with a 4×4 matrix **M**:

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

$$x' = m_{00}x + m_{01}y + m_{02}z + m_{03}w \qquad \text{similarly for } y', z', w'$$

This operation forms the basis for coordinate and normal transform. Assume **V** is a vertex with four components (*xyzw*) stored in memory. The address of **V** is stored in integer register R1. The four columns of matrix **M** are stored in registers F10-F13. The transformed vertex **V′** is stored in register F3. The above operation can be implemented with the following code:

```
fload.u       F2, R1, 16
pfscale,0     F3, F2, F10
pfscaleadd,1  F3, F2, F11, F3
pfscaleadd,2  F3, F2, F12, F3
pfscaleadd,3  F3, F2, F13, F3
```

The first instruction loads the vertex **V** into register F2 and advances the address R1 to the next vertex, assuming vertices are stored sequentially in memory in 16-byte chunks. The next instruction multiplies the first column of matrix **M** by the first component of vertex **V**, which is *x*, and stores the result into register F3. The next three instructions multiply the second, third, and fourth columns of **M** by *y*, *z*, and *w* accordingly, and accumulate the results into F3. At the end, the first subword in F3 contains $m_{00}x+m_{01}y+m_{02}z+m_{03}w$, which matches the equation for x' above. The results for y', z', and w' are similarly obtained in the other subwords of F3. This example shows the use of `pfscale(add)` instructions. The same routine on IA-64 takes 14 instructions, due to the fact that the FP datapath in IA-64 is only 64 bits wide and it does not have a `pfscale(add)` instruction. Even after factoring in the datapath width advantage, PLX is still faster (5 versus 14/2=7 instructions).

B. Perspective division

Perspective division is needed for each vertex when a triangle is rasterized. The first three components of the vertex coordinates (*xyz*) are divided by the fourth (*w*). To get a low-precision result, we only need to compute the reciprocal approximation of *w* and multiply it with *x*, *y*, and *z*. Higher-precision results can be obtained by progressively refining the initial approximation. The "dirty method" by Markstein [12] is one way to achieve sufficient precision for 3D graphics:

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

The first two steps can be used to obtain the coarsest approximation $\mathbf{V}'$ (approximately 8 bits accuracy), and the last two steps are the progressive refinements. The accuracy of $\mathbf{V}'''$ can satisfy the precision of single-precision FP numbers. Assume F2 contains $\mathbf{V}$=(*xyzw*), and $\mathbf{V}'''$ is to be placed in F12, we have:

```
frcpa          F3,  F2
pfscale,3      F10, F3, F2
fmulsub.neg    F4,  F3, F2, F1
pfscaleadd,3   F11, F4, F10, F10
pfscaleadd,3   F12, F4, F11, F10
```

The first two instructions compute $\mathbf{V}'$. If further precision is required, we then use the third instruction to calculate *q*. The last two instructions compute $\mathbf{V}''$ and $\mathbf{V}'''$, accordingly. This example shows the ability of PLX to make tradeoffs between precision and cost. By comparison, IA-64 needs 11 instructions to get $\mathbf{V}'''$.

C. Calculation of spotlight factor in lighting

This calculation is needed when there are spotlights in the scene. The actual computation performed is:

$$\text{if} \quad \mathbf{A} \bullet \mathbf{B} \geq \cos(a), spot = (\mathbf{A} \bullet \mathbf{B})^e$$
$$\text{else} \quad spot = 0.0$$
$$\text{where } \mathbf{A} \bullet \mathbf{B} = \min\left(\sum_{i=0}^{3} A_i \times B_i, 0.0\right)$$

In the above equation, **A** is the normalized vector from the light source to the vertex, **B** is the direction of the spot source, *e* is the spotlight exponent, and *a* is the open angle of the spotlight, cos(*a*) is a constant which has already been calculated. The dot product of **A** and **B** is a saturated dot product, in that no result is less than zero. Assume **A**, **B**, *e*, and cos(*a*) are stored in registers F2-F5, accordingly. **A** and **B** occupy whole registers, while *e* and cos(*a*) only use the rightmost subwords. The result *spot* is to be placed in the rightmost subword of register F10. Register F6 is used as a temporary register.

```
     pfdp.s    F6,  F2, F3
     fcmp.ge   F6,  F5, P1, P2
(P1) flog2a    F6,  F6
(P1) fmul      F6,  F4, F6
(P1) fexp2a    F10, F6
(P2) fsub      F10, F0, F0
```

The `pfdp.s` instruction calculates the saturated dot product, while the `fcmp.ge` instruction does the comparison and sets the predicates. The next three instructions perform the exponentiation operation if the comparison result is true. This uses the formula $a^b = 2^{b \times \log_2 a}$ described earlier. The last instruction sets *spot* to zero when the comparison is false. Since this code is used in color calculation, we use the approximation instructions for exponentiation without doing further iterations to get more precision. This example demonstrates the use of dot product, fast exponentiation approximation, compare, and conditional execution. The same routine on IA-64 would take tens of instructions because IA-64 does not have `flog2a`, `fexp2a`, or `pfdp` instructions.

VI. Conclusion

We have defined a set of PLX floating-point instructions targeted for 3D graphics processing by extracting the common operations in the 3D graphics pipeline. The `pfscaleadd` instruction with subword-parallelism is an effective primitive that allows very fast and efficient implementation of the 4×4 matrix transform, which is a fundamental operation in the 3D graphics pipeline. The math approximation instructions can obtain low-precision results with low cost. They can also be used as the seed operations to get higher precision with instructions such as `fmulsub` and `pfscaleadd`. The predication mechanism for integer PLX can also be used on the FP side for conditional execution in 3D graphics. PLX FP achieves better performance than general purpose processors with 3D extensions, as indicated by the examples. It also offers a more flexible solution than the specialized graphics processor approach since every stage in the 3D graphics pipeline can be easily rewritten if an algorithm is updated.

VII. References

[1] Ruby B. Lee and A. Murat 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.

[2] Mark Segal and Kurt Akeley, The OpenGL Graphics System: A Specification (Version 1.4), 2002.

[3] Microsoft Corporation, DirectX 9.0 SDK Documentation, http://www.microsoft.com/directx.

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

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

[6] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing", IEEE Micro, 20(2):85-95, April 2000.

[7] Stuart Obeman, Greg Favor, Fred Weber, "AMD 3Dnow! Technology: Architecture and Implementations", IEEE Micro, Vol. 19(2):37-48, April 1999.

[8] MIPS Technologies Inc., MIPS-3D ASE: 3D Graphics Application Specific Extension, 2000.

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

[10] NVIDIA Corporation, GeFORCE 256: The World's First GPU, 1999.

[11] NVIDIA Corporation, High-Precision Graphics: Studio-Quality Color on the PC, 2002.

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

[13] Ruby B. Lee, Subworld Parallelism with MAX-2, IEEE Micro, 16(4):51-59, August 1996.