Refining Instruction Set Architecture for High-Performance Multimedia Processing in Constrained Environments

Ruby B. Lee, A. Murat Fiskiran, Zhijie Shi and Xiao Yang

Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering
Princeton University
rblee@ee.princeton.edu

Abstract

Multimedia processing in software has been significantly accelerated by the addition of subword-parallel instructions to the instruction set architectures (ISAs) of modern microprocessors. While some of these multimedia instructions are simple and effective, others are very complex, requiring large, special-purpose functional units that are not practical for constrained environments such as handheld multimedia information appliances. For such environments, low-power and low-cost are as important as the high performance required for real-time multimedia processing and the general-purpose programmability required to support an ever growing range of applications. In this paper, we introduce PLX, a concise ISA that selects the most useful features from the first two generations of multimedia instructions added to microprocessors, and explores new ISA features for high-performance yet low-cost multimedia processing with small footprint processors. PLX is unique in that it is designed from scratch as a fully subword-parallel architecture with novel features like datapath scalability from 32-bit to 128-bit words, and a new definition of predication for reducing conditional branches. We illustrate the use of PLX's architectural features with four frequently used multimedia kernels: discrete cosine transform, pixel padding, clip test and median filter. Our performance results show that a 64-bit PLX implementation achieves significant speedups compared to a basic 64-bit RISC processor and to IA-32 processors with MMX and SSE multimedia extensions. PLX's datapath scalability feature often provides an additional 2x speedup in a cost-effective way.

1. Introduction

This paper starts with the premise that information processing in the 21st century will involve significant amounts of multimedia processing. Hence all programmable processors will need to support fast processing of multimedia data. Two characteristics distinguish multimedia information processing from earlier general-purpose workloads: large amounts of data parallelism and use of low-precision data [1,2]. *Subword parallelism* [1,3] exploits these properties by partitioning the processor's functional units into multiple lower-precision segments called *subwords*, and operating on subwords in parallel using *subword-parallel* instructions. Subword parallelism has also been called *packed parallelism* or *microSIMD parallelism* [4].

All modern microprocessors have now added multimedia instructions to their base instruction set architectures (ISAs). These include: MAX [1] and MAX-2 [3,5] added to Hewlett-Packard's PA-RISC; MMX [6], SSE and SSE-2 [7] to Intel's IA-32; VIS [8] to Sun's UltraSparc; and

Ruby B. Lee, A. Murat Fiskiran, Zhijie Shi and Xiao Yang, "Refining Instruction Set Architecture for High-Performance Multimedia Processing in Constrained Environments," *Proceedings of the 13th International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2002), pp. 253-264*, July 2002.

AltiVec [9] to Motorola's PowerPC. Intel's newest ISA, IA-64 [10,11], also includes multimedia instructions as an integral part of the ISA. These multimedia architectures are compared in [12].

Some of the instructions found in these multimedia extensions are simple and effective while others are very complex, having multi-cycle execution latencies and requiring large functional units. Even for the minimalist multimedia extensions like MAX and MAX-2, the base architecture may have more complexity than needed for the most cost-effective processors for constrained environments like multimedia personal digital assistants and 3GPP (3^{rd} Generation Partnership Project) wireless multimedia devices [13]. Here, low-power and low-cost are as important as the high performance required for real-time multimedia processing and the general-purpose programmability required for supporting an ever growing range of applications. In general, microprocessor architectures with multimedia extensions can provide high multimedia performance, but are unnecessarily complicated for use in wireless multimedia devices.

Media processors have been designed specifically for multimedia processing, but they are still more complicated than needed, often containing special-purpose circuitry for specific multimedia kernels. Typical examples are the MAP and MAP-CA processors by Equator [14,15]. These processors use a VLIW (Very Long Instruction Word) architecture with a large instruction set and special functional units tuned to perform common image processing kernels in hardware. Contrasting with these approaches, we believe that high-performance and low-cost multimedia processing can be achieved by using a concise ISA for a RISC-like, general-purpose processor that is designed from scratch to support very fast subword-parallel processing.

In this paper, we introduce PLX [16], a fully subword-parallel ISA where every instruction can operate on subwords of different sizes. The subword sizes are 1, 2, 4 and 8-byte subwords packed into 32-bit, 64-bit or 128-bit words. In only a few instructions, some subword sizes are not supported because the extra cost is not warranted by the expected low use.

2. PLX instruction set architecture

Figure 1 shows the datapath of a PLX processor. PLX has fixed-length 32-bit instructions that require at most two operands and one result per instruction. Three types of functional units implement these instructions: the Arithmetic Logic unit (ALU), the Shift Permute unit (SPU), and the optional integer Multiplier unit (MUL), which is shown here with three pipelined stages. The instructions executed by each of these functional units are shown in Tables 1, 2 and 3, respectively.

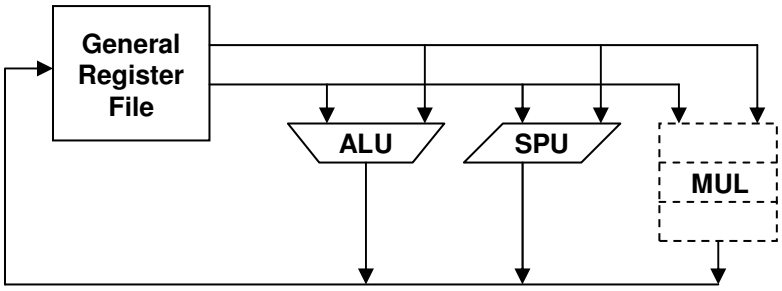

Figure 1. PLX Processor with three functional units

2.1. Datapath scalability

While all instructions in PLX are 32 bits long, the word size and hence the datapath can be 32, 64 or 128 bits wide. We call this variability in datapath size *datapath scalability*. PLX can support such different word sizes without any changes to the ISA. The "sweet spot" around which

the PLX ISA is optimized is a word size of 64 bits, which we call PLX-64. Datapath scalability gives more flexibility to a designer in balancing the cost of a system versus its performance.

Subwords can be 1, 2, 4 or 8 bytes. Maximum subword parallelism is achieved when sixteen 1-byte subwords are packed into a single register in a 128-bit PLX implementation. This allows sixteen subwords to be processed in parallel, resulting in a potential 16x speedup if the program can fully utilize this degree of subword-parallel execution. This performance is attained at only a fraction of the cost and complexity of a superscalar implementation with the same degree of operation parallelism, since subword parallelism requires only minor modifications to the processor's functional units and datapaths [1,2]. In particular, subword-parallel instructions save on register ports, data buses, bypass paths, and instruction dispatch logic when compared to a superscalar or VLIW processor with the same degree of parallelism.

2.2. ALU instructions

Table 1. ALU instructions*

Instruction	Description	MAX-2	MMX + SSE-2
`padd`	$c_i = a_i + b_i$	•	•
`padd w/ saturation`	$c_i = a_i + b_i, \quad c_i \in [L,H]$	•	•
`psubtract`	$c_i = a_i - b_i$	•	•
`psubtract w/ saturation`	$c_i = a_i - b_i, \quad c_i \in [L,H]$	•	•
`paverage`	$c_i = average(a_i, b_i)$	•	•**
`psubtract average`	$c_i = average(a_i, -b_i)$		
`pshiftadd left`	$c_i = (a_i << n) + b_i$	•	
`pshiftadd right`	$c_i = (a_i >> n) + b_i$	•	
`pmax`	$c_i = \max(a_i, b_i)$		•**
`pmin`	$c_i = \min(a_i, b_i)$		•**
logical operations (`and,or, not,xor,and complement`)	$c = a\ op\ b$, where *op* is one of the logical operations	•	• (all except `not`)
`pcmp.rel` (parallel compare)	c = [(rel(a_m,b_m),..,rel(a_1,b_1), rel(a_0,b_0)]		• (subword bit-masks are generated)
`cmp.rel` (compare)	P_{d1} = rel*(a,b)*; P_{d2} = $!P_{d1}$	N/A***	N/A
`cmp.rel.pw` (compare parallel write)	see Section 2.5	N/A	N/A

* Variables c_i, a_i and b_i, are subwords in the destination and source registers respectively. (If no subscript is given, the entire register is used as source or destination.) L and H represent the low and high saturation limits when saturation arithmetic is used. If used, n represents an immediate value in the instruction word. The function rel*(a,b)* compares a and b for a relation specified in the instruction. If true, rel*(a,b)* returns 1, otherwise it returns 0. P_{d1} and P_{d2} are destination predicate registers in compare instructions.

** These instructions come from either SSE or SSE-2.

*** Since PA-RISC 2.0 and IA-32 do not have predication, these predicate setting instructions are not applicable to MAX-2 and MMX + SSE-2.

Table 1 summarizes the ALU instructions in PLX and shows whether a given instruction is also found in the MAX-2 extensions for PA-RISC 2.0 processors [3,5] or in the MMX, SSE and SSE-2 extensions for IA-32 processors [6,7]. MAX-2 is shown since PLX can also be considered "MAX-3", except in a stand-alone processor rather than as an extension of the PA-RISC processor. MMX+SSE-2 is shown as the multimedia ISA available in the dominant microprocessor, IA-32, for desktop and notebook computers.

PLX has parallel add and parallel subtract instructions with modular arithmetic or saturation arithmetic. Parallel add and subtract are the basic subword parallel arithmetic instructions, supported by all multimedia instruction sets [1-3, 5-12]. `Paverage` is used to compute the average of two source subwords and write the result to the destination subword. `Psubtract average` is used to compute the average of the first source subword and the negative of the second source subword. `Pshiftadd left` (or `pshiftadd right`) instructions shift the first source subword left (or right) by one, two or three bits before adding it to the second subword. The shift is implemented by the ALU using a small pre-shifter before the first ALU input. With these instructions, fixed-point and integer multiplication by constants can be done in the ALU without a separate multiplier [1,17]. Therefore, for some very low cost environments, the integer multiplier in a PLX processor may be omitted.

`Pmax` and `pmin` instructions write the greater and smaller of the two source subwords to the destination subword respectively. These instructions are very useful for sorting algorithms since a single `pmax` and `pmin` pair can perform a swap operation for multiple pairs of subwords in a single cycle. Five logical operations included are `and`, `and complement`, `or`, `xor` and `not`.

PLX defines two types of comparison instructions. The first type, `pcmp`, compares pairs of subwords from the two source registers for a relation and generates a 1 if the relation is true, or a 0 if the relation is false. This sequence of m bits, one for each of the m pairs of subwords compared, is written to the low order m bits of the destination register. (In MMX, a bit-mask of all 1s (if `rel` true) or all 0s (if `rel` false) is generated for each destination subword, rather than a single bit per comparison.) The second type of compare instruction compares only the rightmost pair of subwords and it will be discussed in Section 2.5 with predication.

PLX also supports immediate versions of many operations. For example, `add immediate` adds a source register and an immediate value given in the instruction to get the result.

2.3. Shift and permute instructions

Table 2. Shift and permute instructions

Instruction	Description	MAX-2	MMX + SSE-2
`pshift left`	$c_i = a_i << n$	•	•
`pshift left variable`	$c_i = a_i << b$		•
`pshift right`	$c_i = a_i >> n$	•	•
`pshift right variable`	$c_i = a_i >> b$		•
`shift right pair`	$c = [[a,b] >> n]_{lowerhalf}$	•	
`mix [left|right]`	see text	•	
`permute`	see text	•	•**
`permute variable`	see text		

Basic shift instructions, `pshift [left|right]`, are used to shift the subwords in a register to the left or right by an immediate amount specified in the instruction (Table 2). To specify the shift amounts in a register at runtime, `pshift variable` instructions are used.

In the `shift right pair` instruction (Figure 2), two source registers are concatenated and shifted right. The lower half of the shifted result is placed in the destination register. This instruction is very useful for entities that span two registers. Also, when the same register is used as both operands, the result is a rotation of that register.

Subword permutation instructions reorder the subwords in a register. `Mix` instructions, which come in two variants, write alternating (odd or even) subwords from the two source registers to the destination register (Figure 2). The `permute variable` instruction can perform any

arbitrary permutation of 1-byte or 2-byte subwords, with or without repetitions of any subword. Mapping specifications for the permutation are provided in the second source register. For fixed permutations, the `permute` instruction can perform a selected subset of permutations [3,18].

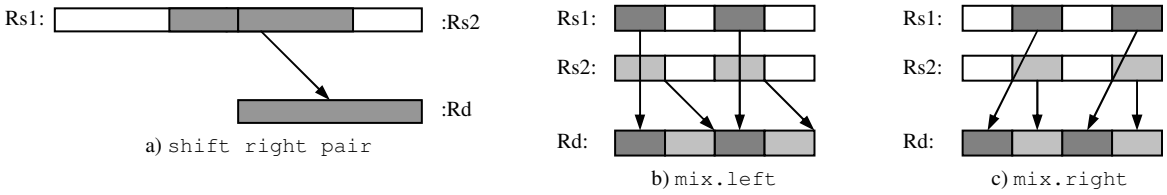

Figure 2. `Shift right pair`, `mix left` and `mix right`

2.4. Multiply instructions

Table 3. Multiply instructions

Instruction	Description	MAX-2	MMX + SSE-2
`pmultiply even`	$[c_{2i}, c_{2i+1}] = a_{2i} * b_{2i}$		
`pmultiply odd`	$[c_{2i}, c_{2i+1}] = a_{2i+1} * b_{2i+1}$		
`pmultiply shift right`	$c_i = [(a_i * b_i) >> n]_{lowerhalf}$		

While many multimedia algorithms only need multiplication by constants (Section 2.2), some require multiplication of two variables. The optional multiplier unit (Figure 1) supports this. `Pmultiply [even|odd]` instructions only multiply the even or odd subwords of the source registers respectively, generating full-sized products. The `pmultiply shift right` instruction allows four half-size (16-bit) products to be generated, by shifting the intermediate 32-bit products right by 0, 8, 15 or 16 bits and selecting the lower 16 bits.

Currently, only 16-bit subwords are supported by these multiply instructions for cost reasons. 16-bit subwords are the most common in multimedia applications needing such multiplication. A 64-bit PLX implementation can choose to implement all three multiply instructions with one, two or four 16-bit multipliers. Since the `pmultiply even` (and `pmultiply odd`) instruction involves only two 16-bit multiplications on a 64-bit PLX, it can be implemented using two 16-bit pipelined multipliers. If only one such multiplier is available, then the second 16-bit multiplication can be started with a single-cycle delay, and the whole instruction can be completed with one extra cycle of latency. Similarly, the `pmultiply shift right` instruction involves four 16-bit multiplications and can be completed with the minimum number of cycles of latency with four 16-bit multipliers, or with only one extra cycle if two pipelined multipliers are implemented, or with three extra cycles if only one multiplier is implemented.

2.5. Predication

PLX allows all instructions to be predicated. There are 128 1-bit predicate registers, grouped into 16 predicate register sets of 8 predicate registers each. Only one set is active at any time, and the 8 predicate registers in that set are addressed P0 through P7. The active predicate register set is changed in software. This definition of predication is novel to PLX and requires only three bits in each instruction to specify a predicate register compared to the seven bits that would be required if 128 predicate registers were addressed directly. Of the eight predicate registers in the active set, P0 always returns true. Any instruction predicated with P0 is unconditionally executed. The remaining seven predicate registers, P1-P7, can be set and cleared using compare instructions.

The first type of compare instruction is `cmp.rel`, which takes two general registers as sources, and two predicate registers as destinations, and compares the two source registers for the relation specified in the `rel` field. If the relation is true, 1 is written to the first predicate register and 0 to the second one; or if the relation is false, 0 is written to the first predicate register and 1 to the second one.

The second type of compare instruction is `cmp.rel.pw1`, where the `pw1` modifier stands for *parallel write one* and indicates that only a 1 can be written to the first destination predicate register and not a 0. This instruction is identical to `cmp.rel` except that nothing is written to either of the predicate registers if the relation is false. This means that the initial values of the two predicate registers are preserved. The program should initialize the predicate registers to known values before executing the `cmp.rel.pw1` instruction. This definition allows multiple `cmp.rel.pw1` instructions targeting the same predicate registers to be executed simultaneously. Examples for both types of instructions are given below.

Type 1: `cmp.rel` (`rel` field specifies the relation to be tested.)
Example: `cmp.eq R1,R2,P1,P2`
Operation: If R1==R2, P1 ← 1 and P2 ← 0, else P1 ← 0 and P2 ← 1.

Type 2: `cmp.rel.pw1` (`pw1` stands for *parallel write one.*)
Example: `cmp.eq.pw1 R1,R2,P1,P2`
Operation: If R1==R2, P1 ← 1 and P2 ← 0, else P1 and P2 are unchanged.

2.6. Other instructions

PLX has load and store instructions for accessing memory, with base plus displacement addressing. Loads also have indexed addressing. Program flow can be changed with jump instructions. This includes jump and link instructions for procedure calls. Conditional branches are achieved with predicated jump instructions. Ideally, a PLX program attempts to eliminate most of the conditional branches with the in-line predicated instruction execution feature to reduce performance penalties for pipeline stall cycles due to conditional branches.

3. Examples and performance

The examples below are chosen because they represent important code kernels in multimedia applications, and because they illustrate the use of the PLX instructions. We evaluate the performance of PLX by running simulations in four different setups using:

1) A basic 64-bit RISC-like ISA without subword parallelism or predication, but otherwise using optimized code. Results from this setup are used as a baseline.
2) IA-32 with MMX and SSE instructions. Since the default datapath size of the first setup and of PLX is 64-bits, we do not use SSE-2 instructions, which require a 128-bit datapath. The IA-32 with MMX and SSE setup is chosen to represent the dominant processor architecture in notebook processors.
3) 64-bit PLX architecture including all PLX-specific optimizations.
4) 128-bit PLX architecture. Compared to the 64-bit PLX, this shows the speedup due to the datapath scalability feature.

All performance results are reported in Table 4 as speedups of the second, third and fourth setups over the first one. All results assume single-cycle instructions on a single-issue pipeline, and single-cycle loads and stores. Instructions are scheduled to minimize pipeline stalls due to

data dependencies. A detailed listing of the instruction frequencies for each code example is given in the Appendix.

3.1. Discrete cosine transform

Discrete cosine transform (DCT) and its inverse (IDCT) are extensively used in image and video compression applications, such as JPEG and MPEG. We simulate an 8x8 2-D DCT on 16-bit pixels using the method described by Arai, Agui and Nakajima in [19] (AAN DCT), which minimizes the number of multiplications needed.

The most time-critical steps of the AAN DCT algorithm are transposition of 8x8 blocks, and multiplication by fixed-point constants [1]. Matrix transposition uses `mix` instructions extensively. An 8x8 block of 16-bit subwords can be transposed in 32 instructions [2,3], which can execute in 16 cycles on a 2-way superscalar machine or 8 cycles on a 4-way superscalar machine. Multiplication by constants is efficiently achieved with `pshiftadd right` instructions. The PLX instructions used to perform two of the four different fixed-point multiplications used in the AAN DCT are shown below. An average of 3.5 instructions are needed per multiplication of a register by the corresponding fractional constant. Therefore, in a 64-bit PLX processor, four 16-bit multiplications can be done simultaneously in 3.5 cycles on the average. This performance, achieved using the adder with subword parallelism [1,17], is even better than using one or two 16-bit integer multipliers, each multiplication taking at least 3 cycles of execution latency.

Input: R1 is the source register that contains the four subwords to be multiplied.
Output: R2 contains the product of R1 and the fractional constant multiplier.

1st coefficient = $(0.70711)_{10}$ = 0.10110101

```
pshiftadd.2.right R2,R1,R1       # R2 = 1.01           * R1
pshiftadd.3.right R2,R2,R2       # R2 = 1.01101        * R1
pshifti.2.right   R3,R1,8        # R3 = 0.00000001     * R1
pshiftadd.1.right R2,R2,R3       # R2 = 0.10110101     * R1
```

2nd coefficient = $(0.54120)_{10}$ = 0.10001010

```
pshiftadd.2.right R2,R1,R1       # R2 = 1.01           * R1
pshifti.2.right   R2,R2,5        # R2 = 0.0000101      * R1
pshiftadd.1.right R2,R1,R2       # R2 = 0.1000101      * R1
```

The most important factors contributing to performance for this algorithm are suitability of the algorithm for 4-way or 8-way subword parallelism, low-cost but high performance multiplication due to `pshiftadd` instructions, and fast matrix transposition due to `mix` instructions.

3.2. Pixel padding

Pixel padding [20] is used in MPEG-4 [21]. MPEG-4 differs from previous video compression standards like MPEG-1 and MPEG-2 in that it works on structures called Video Object Planes (VOPs) rather than on frames. The VOP structure permits arbitrary shaped video objects. Padding of an entire 8x8 block is done in four phases: first horizontal padding, first transposition, second horizontal padding, and second transposition. Horizontal padding of the pixels is a simple operation that is efficiently handled by logical instructions. Transposition of blocks is done by the `mix` instructions. A special case in the algorithm requires averaging of pixel values, which can be performed in parallel using the `paverage` instruction.

Subword parallelism, subword permutation (`mix`) instructions and the `paverage` instruction are the main factors that contribute to the speedup for pixel padding. Unlike the case for AAN

DCT, 128-bit PLX offers no further speedup (Table 4) over 64-bit PLX since pixel padding works on irregularly spaced non-consecutive 8x8 blocks. Each of these blocks fit well into eight 64-bit registers, and the extra space offered by 128-bit registers remains unutilized.

3.3. Clip test in 3D graphics processing

In this example, we explore the use of predication and compare instructions. In 3D graphics processing, primitive objects (mostly triangles) need to be clipped before they are rendered [22]. A triangle consists of three vertices (`v1`, `v2` and `v3`), each of which is represented by its spatial coordinates `(x,y,z,w)`. The bounding volume for each vertex `(x,y,z,w)` is defined by $-w \le x \le w$, $-w \le y \le w$, $-w \le z \le w$. Clip test is performed for each triangle to determine its relationship with its bounding volume. If the triangle is completely inside the bounding volume, it is accepted and sent to the next processing stage. If it is completely outside, it is discarded. If only a part of the triangle is inside the bounding volume and part of it is outside, it must be clipped.

By definition, a triangle is completely inside its bounding volume if all its vertices are inside their bounding volumes. It is completely outside if all its vertices are outside the same plane of their bounding volumes. Otherwise the triangle intersects its bounding volume and needs to be clipped. This test can be written as follows:

Clip test (for a single triangle – This test is repeated for all triangles in consideration.)

```
if (!((v1.x < -v1.w && v2.x < -v2.w && v3.x < -v3.w) ||
      (v1.x >  v1.w && v2.x >  v2.w && v3.x >  v3.w) ||
      (v1.y < -v1.w && v2.y < -v2.w && v3.y < -v3.w) ||
      (v1.y >  v1.w && v2.y >  v2.w && v3.y >  v3.w) ||
      (v1.z < -v1.w && v2.z < -v2.w && v3.z < -v3.w) ||
       (v1.z >  v1.w && v2.z >  v2.w && v3.z >
v3.w))) {

  if (!(v1.x < -v1.w || v2.x < -v2.w || v3.x < -v3.w ||
        v1.x >  v1.w || v2.x >  v2.w || v3.x >  v3.w ||
        v1.y < -v1.w || v2.y < -v2.w || v3.y < -v3.w ||
        v1.y >  v1.w || v2.y >  v2.w || v3.y >  v3.w ||
        v1.z < -v1.w || v2.z < -v2.w || v3.z < -v3.w ||
        v1.z >  v1.w || v2.z >  v2.w || v3.z >  v3.w));

  else clip();
  next_stage();
}
```

The first set of conditions in this pseudocode evaluates whether a triangle is completely outside its bounding volume. If it is not, we evaluate the second set of conditions, which tests whether the triangle is completely inside. These nested `if-then` statements can be accelerated greatly by using predication and the `compare parallel write` instructions described in Section 2.5. If sufficient hardware resources are present, each of the triplets in the first `if` statement can be computed simultaneously in a single cycle by using `cmp.rel.pw1` instructions. This is possible because multiple `cmp.rel.pw1` instructions can target the same destination predicate register simultaneously. Therefore, the speedups shown in Table 4 will be even more pronounced for superscalar PLX implementations than for the single-issue implementation upon which our results are based.

The most important factor that contributes to the speedup for clip test is the use of predication for the resolution of complicated `if-then` statements shown above. Dependence of this

algorithm on predication is verified by the fact that only 64% of the fetched instructions are actually executed; the rest were predicated false and therefore did not execute.

3.4. Median filter

Median filter is an example of an algorithm that initially seems to require conditional execution, and hence is likely to benefit from predication. However, as for other multimedia processing examples, it can be optimized to eliminate any need for conditional execution (and predication).

A median filter is used for noise reduction in image processing. A 3x3 box is moved across the image and the center pixel value is replaced with the median value of the nine pixels enclosed by the 3x3 box. If the value of the center pixel is abnormally above or below the value of its neighbors (because of distortion by noise), it would be eliminated in this process. Our simulations are for a median filter that works on 8-bit pixels. The algorithm is as follows:

1. Place the 3x3 box at the beginning of the image.
2. Find the median of the nine pixels in the box.
3. Replace the center pixel with the median value from step 2.
4. If the end of image is reached stop. Otherwise move the box to the next position and go to step 2.

Step 2 above is the most computationally intense of the four. The subroutine used for this step uses a `p-sort` operation that conditionally swaps two pixels, P1 and P2, as follows:

```
# define p-sort(P1,P2) {if (P1>P2) P1 ↔ P2;}
```

In a 64-bit PLX, eight instances of this `p-sort` operation can be done using just two instructions. For instance if the pixels P1 and P2 are initially in the lowest order bytes of registers R1 and R2 respectively, the following two instructions will perform the swap (if needed) and write the results to R3 and R4. After these instructions, the lowest order byte of R3 will contain the smaller of P1 and P2, and the lowest order byte of R4 will contain the larger.

```
pmin.1 R3,R1,R2;     pmax.1 R4,R1,R2;
```

The '1' modifier in these instructions indicates that the operations are performed on 1-byte subwords, each of which corresponds to one 8-bit pixel. Since the 64-bit registers R1-R4 each contain eight pixels, eight pixel pairs are sorted in parallel with only two instructions. For a 128-bit PLX, 16 pixel pairs can be similarly sorted. Since this algorithm can fully utilize subword parallelism, the speedup figures in Table 4 are proportional to the number of subwords in one register (i.e. 8x for 64-bit PLX, and 16x for 128-bit PLX). Below, we show how 19 `p-sort` operations can be optimally used to complete the step 2 above. After the last `p-sort`, P4 will be the median for the nine pixels P0-P8.

```
01. p-sort(P1,P2); 02. p-sort(P4,P5); 03. p-sort(P7,P8);
04. p-sort(P0,P1); 05. p-sort(P3,P4); 06. p-sort(P6,P7);
07. p-sort(P1,P2); 08. p-sort(P4,P5); 09. p-sort(P7,P8);
10. p-sort(P0,P3); 11. p-sort(P5,P8); 12. p-sort(P4,P7);
13. p-sort(P3,P6); 14. p-sort(P1,P4); 15. p-sort(P2,P5);
16. p-sort(P4,P7); 17. p-sort(P4,P2); 18. p-sort(P6,P4);
19. p-sort(P4,P2);
```

In PLX, eight medians can be computed simultaneously with the `pmix` and `pmax` instructions. The pathlength savings this achieves and the amenability of this algorithm for

subword parallel implementation are the main factors for speedup. Note that since IA-32 instructions overwrite one source register, an additional move operation is needed to save this register when its initial value must be preserved, as in the `p-sort` operation.

Table 4. Speedups over the basic 64-bit ISA

	Basic 64-bit RISC	IA-32 with 64-bit MMX and SSE	64-bit PLX	128-bit PLX
AAN DCT	1	1.1	4.6	9.1
Pixel padding	1	4.9	7.9	7.9
Clip test	1	0.5	1.9	1.9
Median filter	1	5.3	8.0	16.0

For all algorithms, Table 4 shows an increasing speedup as we move from the basic 64-bit RISC to IA-32 with 64-bit MMX and SSE, to 64-bit PLX. The only exception is that IA-32 is slower than a basic 64-bit RISC for the clip test because of a paucity of registers and additional memory accesses. 64-bit PLX is consistently faster than IA-32/MMX/SSE. For algorithms like clip test, if we compare superscalar implementations rather than single-issue implementations, PLX will have even greater speedup compared to IA-32/MMX/SSE since PLX allows different comparisons targeting the same predicate register to execute in the same cycle. For algorithms like the AAN DCT and median filter where the data parallelism of adjacent pixels can be further exploited with a wider subword-parallel functional unit, the datapath scalability feature of 128-bit PLX provides an additional 2x speedup over the 64-bit PLX.

4. Conclusions

An ISA targeted for use in constrained environments such as handheld wireless multimedia appliances must have high performance for multimedia processing, low cost and low power. Existing ISAs fall short of meeting all three criteria simultaneously. Based on our study of first and second-generation multimedia ISAs [2,12,23], we have selected the most useful multimedia instructions whose implementations are expected to be low in cost and power. To this set, we added new architectural features for even higher performance. The result is PLX, intended to be a minimalist ISA for high-performance multimedia processing. PLX is also used as a test bed for hands-on teaching and research in ISA and microarchitecture design and analysis at Princeton University. This paper presents PLX 1.1.

PLX is designed as a concise RISC-like instruction set, where every instruction takes a single cycle to execute. The only exceptions are the multiply by variable instructions, where a pipelined multiply functional unit could take multiple cycles to implement. This is an optional functional unit, and may be omitted for cost reasons. It has to be included in environments where multiply by variables is frequently needed. Otherwise, PLX supports low-cost multiply by constants using the ALU, with `pshiftadd` instructions, achieving very good performance due to subword parallelism. In our multimedia kernels (including many not presented in this paper), constant multiplication is needed much more frequently than variable multiplication. The fast subword permutations, especially `mix`, have been shown very useful in two of the code kernels. The agility of the other `permute` instructions in PLX will be demonstrated in subsequent papers. Other instructions shown to be useful are parallel averaging and the parallel sorting achieved with the `pmin` and `pmax` instructions. New features in PLX, like efficient predication and datapath scalability, have also proved useful. However, although the clip test example benefited from predication, we plan to investigate predication further to determine its importance in multimedia processing. We do not advocate unlimited datapath scalability, but feel that the scalability from 64 bits down to 32 bits is desirable for cost and power reasons, and the scalability up to 128 bits may be justified from a performance-cost basis.

Our simulation results indicate that critical multimedia kernels benefit from at least one of the key features of PLX: subword parallelism, subword permutations, low-cost multiplication, datapath scalability and efficient predication. Overall, our results show that high-performance can be achieved with a concise ISA like PLX without incurring the complexity costs of larger ISAs in both general-purpose microprocessors and mediaprocessors.

Future ISA work will include further explorations of predication and subword permutation operations, and the design of floating-point instructions for graphics and high-fidelity audio processing. We are also creating models of latency, area and power for architectural tradeoff studies of different PLX features and new ISA proposals.

Acknowledgements

We thank the students of the ELE-475 and ELE-572 classes at Princeton University who worked on the PLX project. PLX is being developed at the Princeton Architecture Laboratory for Multimedia and Security (PALMS). We acknowledge the support for PALMS research from Hewlett-Packard, National Science Foundation and Kodak – A.M. Fiskiran is a Kodak Fellow.

References

[1] Lee, R.B., "Accelerating Multimedia with enhanced Microprocessors", IEEE Micro, Vol. 15, No. 2, pp. 22-32, April 1995.

[2] Lee, R.B., "Multimedia Extensions for General-Purpose Processors", invited paper, Proceedings of SiPS 1997 IEEE Workshop on Signal Processing Systems Design and Implementation, pp. 9-23, November 1997.

[3] Lee, R.B., "Subword Parallelism with MAX-2", IEEE Micro, Vol. 16, No. 4, pp. 51-59, August 1996.

[4] Lee, R.B., "Efficiency of MicroSIMD Architectures and Index-Mapped Data for Media Processors", invited paper, Proceedings of Media Processors 1999, IS&T/SPIE Symposium on Electric Imaging: Science and Technology, pp. 34-46, January 25-29, 1999.

[5] Kane, G., "PA-RISC 2.0 Architecture", Prentice Hall, ISBN 0-13-182734-0, 1996.

[6] Peleg, A. and U. Weiser, "MMX Technology Extension to the Intel Architecture", IEEE Micro, Vol. 16, No. 4, pp. 42-50, August 1996.

[7] Intel, "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference", Order Code 243191, 1999.

[8] Tremblay, M., J.M. O'Connor, V. Narayanan and H. Liang, "VIS Speeds New Media Processing", IEEE Micro, Vol. 16, No. 4, pp. 10-20, August 1996.

[9] Motorola, "AltiVec Technology Programming Environments Manual", Revision 0.1, Order Code ALTIVECPEM/D, November 1998.

[10] Intel, "IA-64 Architecture Software Developer's Manual, Vol. 3: ISA Reference", Rev. 1.1, ID. 245319-002, July 2000.

[11] Lee, R.B., A.M. Fiskiran and A. Bubshait, "Multimedia Instructions in IA-64", Proceedings of ICME 2001 IEEE International Conference on Multimedia and Expo, August 22-25, 2001.

[12] Lee, R.B. and A.M. Fiskiran, "Multimedia Instructions in Microprocessors for Native Signal Processing", Programmable Digital Signal Processors: Architecture, Programming, and Applications, edited by Yu Hen Hu, Marcel Dekker Inc., ISBN 0-8247-0647-1, pp. 91-145, 2002.

[13] Elsen, I., F. Hartung, U. Horn, M. Kampmann and L. Peters, "Streaming Technology in 3G Mobile Communication Systems", IEEE Computer, Vol. 34, No. 9, pp. 46-52, September 2001.

[14] Basoglu, C., R. Gove, K. Kojima, and J. O'Donnell, "A Single-chip Processor for Media Applications: the MAP1000", International Journal of Imaging Systems and Technology, Vol. 10, pp. 96-106, 1999.

[15] Basoglu, C., K. Zhao, K. Kojima and A. Kawaguchi, "The MAP-CA VLIW Based Media Processor", white paper from Equator Technologies Inc. and Hitachi Ltd., March 2000.

[16] Lee, R.B., et al., "PLX-Project at Princeton University", http://palms.ee.princeton.edu/plx.

[17] Luo, Z. and R.B. Lee, "Cost-Effective Multiplication with Enhanced Adders for Multimedia Applications", Proceedings of IEEE International Symposium on Circuits and Systems, Vol. 1, pp. 651-654, May 2000.

[18] Lee, R.B., "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures", Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 3-14, July 2000.

[19] Arai, Y., T. Agui and M. Nakajima, "A Fast DCT-SQ Scheme for Images", Trans. IEICE, E71(11):1095, November 1988.

[20] Edirisinghe, E.A., J. Jiang and C. Grecos, "A Novel Shape Padding Technique for Improving MPEG-4 Compression Efficiency", IEEE Electronic Letters, Vol. 35, No. 17, pp. 1453-1454, August 1999.
[21] ISO/IEC JTC 1/SC 29/WG 11, "Information Technology — Coding of audio-visual objects, Part 1: Systems, Part 2: Visual, Part 3: Audio", FCD 14 496, December 1998.
[22] Liang, Y. and B.A. Barsky, "An Analysis and Algorithm for Polygon Clipping", Communications of the ACM, Vol. 26, November 1983.
[23] Lee, R.B., "Instruction Set Architecture for Multimedia Signal Processing", invited chapter, Computer Engineering Handbook, edited by Vojin Oklobdzija, CRC Press, ISBN 0-8493-0885-2, January 2002.

Appendix: Instruction frequencies for simulated algorithms

	AAN DCT	Pixel padding	Clip test	Median filter
Instruction	**Percent**	**Percent**	**Percent**	**Percent**
padd.2	**15.62%**	0.00%	0.00%	0.00%
psubtract.2	**13.12%**	0.00%	0.00%	0.00%
psubtract.4	0.00%	0.00%	**5.17%**	0.00%
paverage.1	0.00%	**6.50%**	0.00%	0.00%
pshiftadd.1.right	**3.75%**	0.00%	0.00%	0.00%
pshiftadd.2.right	**20.62%**	0.00%	0.00%	0.00%
pshiftadd.3.right	**1.25%**	0.00%	0.00%	0.00%
pmax.1	0.00%	0.00%	0.00%	**27.78%**
pmin.1	0.00%	0.00%	0.00%	**27.78%**
and complement	0.00%	**11.38%**	0.00%	0.00%
or	0.00%	**26.00%**	0.00%	0.00%
cmp.4.eq.pw1	0.00%	0.00%	**12.07%**	0.00%
cmp.4.gt.pw1	0.00%	0.00%	**31.03%**	0.00%
cmp.4.lt.pw1	0.00%	0.00%	**31.03%**	0.00%
cmp.eq	0.08%	0.00%	0.00%	0.00%
cmp.gt	1.41%	0.81%	1.72%	0.00%
changepr.ld	0.00%	0.00%	1.72%	0.00%
add immediate	0.78%	0.00%	0.00%	0.00%
subtract immediate	1.48%	0.81%	1.72%	0.00%
srli	0.00%	0.00%	**10.34%**	0.00%
srai	0.00%	0.00%	0.00%	1.85%
shift right pair	0.00%	0.00%	0.00%	**12.96%**
pshifti.2.right	**1.88%**	0.00%	0.00%	0.00%
mix.2.left	**6.25%**	**13.00%**	0.00%	0.00%
mix.2.right	**6.25%**	**13.00%**	0.00%	0.00%
mix.4.left	**6.25%**	**13.00%**	0.00%	0.00%
mix.4.right	**6.25%**	**13.00%**	0.00%	0.00%
load immediate	0.31%	0.00%	0.01%	1.85%
load.8	**5.00%**	0.05%	0.00%	**16.67%**
store.8	**7.50%**	0.00%	0.00%	3.70%
jump	1.52%	0.82%	1.73%	1.85%
jump.link	0.00%	0.81%	1.72%	1.85%
jump.reg	0.00%	0.81%	1.72%	1.85%
Total	100%	100%	100%	100%
Instruction count	2559	31506	14338	5601