PLX: A FULLY SUBWORD-PARALLEL INSTRUCTION SET ARCHITECTURE FOR FAST SCALABLE MULTIMEDIA PROCESSING

Ruby B. Lee and A. Murat Fiskiran

Princeton Architecture Laboratory for Multimedia and Security (PALMS)[*]
Princeton University
{rblee,fiskiran}@princeton.edu

ABSTRACT

PLX is a small, fully subword-parallel instruction set architecture designed for very fast multimedia processing, especially in constrained environments requiring low cost and power such as handheld multimedia information appliances. In PLX, we select the most useful multimedia instructions added previously to microprocessors. We also introduce a few novel features: a new definition of predication requiring very few bits in each predicated instruction, and datapath scalability from 32-bit to 128-bit words, which allows different degrees of subword parallelism without any changes to the ISA. Performance results from basic multimedia kernels testify to PLX's superiority for multimedia processing.

1. INTRODUCTION

Multimedia processing involves compute-intensive operations and constitutes an increasingly greater fraction of the general-purpose processor's workload [1]. To achieve better multimedia performance, instruction set architectures (ISAs) have added multimedia extensions [2,3], such as MAX-2 [4] to PA-RISC processors [5], MMX [6] to IA-32 processors, and a superset of these to IA-64 [7] processors. These ISAs exploit the following two properties of multimedia applications:

- Huge amounts of data parallelism
- Extensive use of low-precision data

These two properties are exploited well by the use of *subword parallelism,* also called *microSIMD* parallelism [2,8]. In a subword-parallel architecture, the processor's datapath is partitioned into multiple lower-precision segments called the *subwords*, and the instructions operate in parallel on these subwords (Figure 1).

PLX is a fully subword-parallel ISA designed for very fast media processing [9]. We introduce the PLX architecture along with some examples that highlight some of its features, such as low-cost multiplication, a new definition of predication, and datapath scalability.

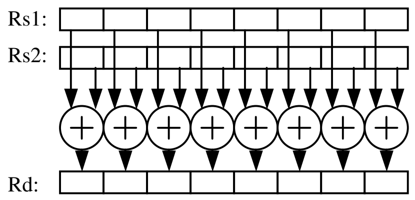

Figure 1: Parallel add instruction operating simultaneously on multiple subwords

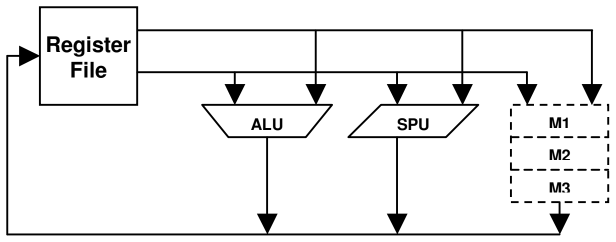

Figure 2: PLX processor with three functional units: ALU, Shift and Permute Unit (SPU), and an optional pipelined multiplier

2. PLX INSTRUCTIONS

PLX instructions can be classified into three major groups based on the functional unit responsible for their execution: ALU instructions, shift and permute instructions, and multiply instructions (Figure 2). All instructions are 32-bits long and subword sizes are 1, 2, 4 and 8 bytes.

Basic ALU instructions shown in Table 1 include parallel add and subtract (with modular or saturation arithmetic), parallel shift and add, parallel average, parallel maximum and minimum, logical and compare instructions (Section 3).

[*] PALMS research is supported in part by HP, NSF and Kodak.

2.1. Low-cost multiplication

`Pshift [left|right] add` instructions allow low-cost integer and fixed-point multiplication in the ALU without requiring a separate multiplier. Since the shift amounts are limited to 1, 2 or 3 bits to the right or left, they are realized by a small pre-shifter added to the ALU [8,10]. Because multiplications can be performed efficiently and inexpensively in the ALU, a separate integer multiplier becomes optional for very low-cost and low-power PLX implementations (as indicated by the dotted lines in Figure 2).

Table 1: ALU instructions*

Instruction	Description
`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`	$c_i = average(a_i, -b_i)$
`pshift left add`	$c_i = (a_i << n) + b_i$
`pshift right add`	$c_i = (a_i >> n) + b_i$
`pmaximum`	$c_i = \max(a_i, b_i)$
`pminimum`	$c_i = \min(a_i, b_i)$
logical operations `(and,or, not,xor,and complement)`	$c = a\ op\ b$, where *op* is one of the logical operations
`cmp` (compare)	$P_{d1} = rel(a,b); P_{d2} = !P_{d1}$
`cmp.pw1` (compare parallel write one)	*see Section 3*

* Variables c_i, a_i and b_i, correspond to the 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 given in the instruction word. The function *rel(a,b)* compares a and b for a relation specified in the instruction word. If this relation is true, *rel(a,b)* returns 1, otherwise it returns 0. P_{d1} and P_{d2} are destination predicate registers in compare instructions.

2.2. Full multiplication

While they are low-cost and effective, the `pshift [left|right]` instructions only allow multiplication by constants. Therefore PLX also includes instructions to multiply two registers (Table 2). These instructions are handled by a separate optional multiplier unit.

`Pmultiply shift right` right-shifts the products before writing the lower-order half of the bits to the destination register. This allows selection of the desired 16-bits of each product. `Pmultiply odd` and `pmultiply even` only multiply the odd or even indexed subwords of the source registers, producing full-length products.

2.3. Shift and permute instructions

PLX has parallel shift and subword permute instructions, implemented in the shift and permute unit (Table 3). The parallel shift instructions shift the subwords in a register to the left or to the right by any amount specified either in an immediate field or in a register. The `shift right pair` instruction, first introduced in PA-RISC processors, is very useful for bit fields spanning two registers [5,7]. This instruction concatenates two source registers and shifts this value to the right. The lower half of the shifted value is placed in the destination register. Rotation is achieved when both source operands are the same register.

Table 2: Multiply instructions

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

Table 3: Shift and permute instructions

Instruction	Description
`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*

Subword permutation instructions are used to reorder the subwords in a register. `Mix` instructions described in [2-4,7] are very useful for performing matrix transposition of subwords packed into multiple registers. The `permute` instruction works on 1-byte and 2-byte subwords, and performs a small set of carefully selected permutation primitives [11,12]. The `permute variable` instruction uses a second source register to specify the permutation control bits, and hence can perform any arbitrary permutation of 1-byte or 2-byte subwords, with or without repetitions of any subword.

3. PREDICATION

All PLX instructions are predicated. PLX has 128 1-bit predicate registers organized into 16 predicate register sets of 8 predicate registers each. At any given time, only one of these predicate register sets is active and the registers in this set are numbered P0 through P7. The active predicate register set is changed in software.

The predicate registers P1 to P7 can be set and cleared using compare instructions (P0 is always true). This definition of predication requires only three bits in each instruction to specify a predicate register compared to the seven bits that would be required if the 128 predicate registers were addressed directly.

Two types of compare instructions set the predicate registers in PLX. They are illustrated below, comparing two registers, R1 and R2, for equality.

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

The first type of compare is useful for implementing *if-then-else* statements without conditional branch instructions. The second type differs from the first because it writes the predicate registers only if the relation specified in the `rel` field is true. This allows multiple `cmp.pw1.rel` instructions to be executed in the same cycle, targeting the same predicate registers, to speedup complex conditional expressions. The values in the predicate registers must be initialized before using `cmp.pw1.rel` instructions.

PLX also has load, store and jump instructions, as needed for a stand-alone processor.

4. DATAPATH SCALABILITY

PLX can be implemented as a 32-bit, 64-bit or 128-bit architecture without any changes to the ISA. To allow this, PLX instructions are designed to work for these different word sizes. All subword sizes of 1, 2, 4 and 8 bytes are supported, up to the larger of the word size or 8 bytes: a 32-bit PLX does not support 8 byte subwords and a 128-bit PLX does not support 16-byte subwords.

Compared to a 64-bit PLX, a 32-bit implementation has a lower performance, but also a lower cost. On the other hand, doubling the datapath width to 128 bits effectively doubles the subword parallelism, but at a lower cost compared to a superscalar implementation with an equivalent degree of operation parallelism.

5. EXAMPLES AND PERFORMANCE

Performance of PLX is verified by simulating three commonly used multimedia algorithms in four different setups: 1) using a basic RISC-like 64-bit ISA without subword parallelism or predication; 2) using 64-bit MMX instructions; 3) using 64-bit PLX; and 4) using 128-bit PLX to demonstrate datapath scalability.

In all cases, the algorithms are hand-coded and optimized in their respective assembly languages. To emphasize the effects of ISA features, we keep the microarchitecture as simple as possible by using a single-issue pipeline and assuming a perfect memory system, where all loads and stores take a single cycle. Execution latencies are properly accounted for, with single-cycle ALU and SPU instructions and three-cycle multiply instructions. Whenever possible, instructions are scheduled to eliminate pipeline stalls caused by data dependencies.

The simulation software used is part of a comprehensive ISA research toolbox developed under the PLX project [9]. In addition to a cycle-accurate customizable simulator, it includes an assembler and a compiler as well as other auxiliary tools for workload characterization and performance analysis.

Performance results are shown in Table 4, as speedups of the second, third and fourth setups over the first one.

5.1. Digital filtering

The most common DSP kernel is the digital filter. Applications include frequency-domain alterations of signals; low-pass, high-pass and band-pass filtering; audio equalization, adaptive filtering and speech compression. We simulate a 4-tap finite impulse response (FIR) filter that uses fixed-point numbers for both input data and coefficients. This algorithm benefits most from subword-parallelism and the low-cost multiplication that is offered by the `pshift [left|right] add` instructions.

The 64-bit PLX is 4.48 times faster than the basic ISA, and also 4.07 times faster than MMX. A 128-bit PLX doubles the performance of a 64-bit PLX.

5.2. Discrete cosine transform

The Discrete Cosine Transform (DCT) and its inverse (IDCT) are commonly used code kernels in image and video compression such as JPEG, MPEG and H.261. We run simulations for an 8x8 2-dimensional DCT using the AAN [13] algorithm.

The most time critical operations in the IDCT algorithm are matrix transposition and multiplication by fractional constants. Using `mix` instructions, the transposition of an 8x8 matrix of 16-bit IDCT coefficients is achieved very efficiently. The average number of parallel shift and add (and other) instructions required per multiplication in AAN DCT is only 3.5. Since four 16-bit multiplications are done in parallel in a single-issue 64-bit PLX, this is more than one multiplication per cycle, using just a single ALU (and no multiplier). A superscalar implementation with multiple ALUs can achieve even higher multiplication performance using the parallel shift and add instructions.

5.3. Median filter

Median filter is an image-processing algorithm used for noise reduction. Its most compute-intensive step is to find the median of nine 8-bit pixels enclosed within a 3x3 box that is stepped across the whole image. To illustrate the PLX compare instructions and predication feature, we use a bubble-sort algorithm to sort the nine pixels, and then take the center value in the sorted list as the desired median.

We show how the evaluation of conditionals in the sorting algorithm is accelerated with `cmp.pw1` instructions. Eight pairs of registers are compared for equality. Decisions are made based on whether the equality holds for all the comparisons or not. Without predication, these equality tests require at least eight serial comparisons, interspersed with conditional jump instructions. In PLX, the `cmp.pw1.ne` (compare, parallel write one, not equal) instruction is used to evaluate the comparisons in parallel as follows:

Optimized Comparison Subroutine in Median Filter
(R1-R8 are compared to R11-R18 respectively for equality.)

```
P0 cmp.ne     R0,R0,P1,P0;;  # Init P1 to 0
P0 cmp.pw1.ne R1,R11,P1,P0
P0 cmp.pw1.ne R2,R12,P1,P0
P0 cmp.pw1.ne R3,R13,P1,P0
   ...
P0 cmp.pw1.ne R8,R18,P1,P0;;
P1 jmp        sometarget;;
```

The double semi-columns are used to separate instruction groups that must be executed in different cycles. The theoretical limit for the execution of this sequence is three cycles. For a two-way superscalar implementation, six cycles are required.

The median filter was further optimized by the use of `shift right pair`, `pmaximum` and `pminimum` instructions, for an overall speedup of about 10x over the non-subword-parallel implementation. A pair of `pmaximum` and `pminimum` instructions can sort 8 bytes in parallel on a 64-bit PLX, and 16 bytes in parallel on a 128-bit PLX.

Table 4: Speedups over the basic ISA

	MMX (64 bits)	64-bit PLX	128-bit PLX
FIR Filter	1.10	4.48	9.83
AAN DCT	1.97	3.10	6.17
Median Filter	6.50	10.66	19.53

In each case, 64-bit PLX is much faster than MMX (which has the same degree of subword parallelism), and 128-bit PLX provides a further 2x speedup.

6. CONCLUSIONS

The PLX architecture is capable of delivering very high multimedia performance at only a fraction of the complexity of existing microprocessors with multimedia extensions. The 32-bit instructions of PLX result in a higher code density compared to architectures with longer instructions such as IA-64. In addition PLX has a novel definition of predication that allows all instructions to be predicated with 128 predicate registers, while only consuming three bits in each instruction. We plan to investigate more thoroughly the usefulness of predication in reducing branch penalties in media programs. Another novel property of PLX is datapath scalability, which allows processor implementations with different datapath sizes using the same ISA. This gives extra flexibility in balancing complexity versus cost.

Our results show that very high multimedia performance can be achieved with a simple and low-cost ISA like PLX. This makes PLX especially suitable for constrained environments such as wireless multimedia information appliances, where high multimedia performance and low cost and power are required.

7. REFERENCES

[1] R.B. Lee and M. Smith, "Media Processing: A New Design Target," *IEEE Micro*, Vol. 16, No. 4, pp. 6-9, Aug. 1996.

[2] R.B. Lee 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, pp. 91-145, Marcel Dekker Inc., ISBN 0-8247-0647-1, 2002.

[3] R.B. Lee, "Multimedia Extensions for General-Purpose Processors," *Proc. IEEE SIPS 97*, pp. 9-23, Nov. 1997.

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

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

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

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

[8] R.B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, pp. 22-32, Apr. 1995.

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

[10] Z. Luo and R.B. Lee, "Cost-Effective Multiplication with Enhanced Adders for Multimedia Applications," *Proc. IEEE International Symposium on Circuits and Systems*, Vol. 1, pp. 651-654, May 28-31, 2000.

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

[12] R.B. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures," *Proc. IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 3-14, Jul. 10-12, 2000.

[13] Y. Arai, T. Agui and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *Trans. IEICE*, E 71(11):1095, Nov. 1988.