Journal of VLSI Signal Processing 40, 85–108, 2005

PLX: An Instruction Set Architecture and Testbed for Multimedia Information Processing

RUBY B. LEE AND A. MURAT FISKIRAN
Department of Electrical Engineering, Princeton University, Princeton, NJ 08544, USA

Received February 13, 2003; Revised April 28, 2004; Accepted April 28, 2004

Abstract. PLX is a concise instruction set architecture (ISA) that combines the most useful features from previous generations of multimedia instruction sets with newer ISA features for high-performance, low-cost multimedia information processing. Unlike previous multimedia instruction sets, PLX is not added onto a base processor ISA, but designed from the beginning as a standalone processor architecture optimized for media processing. Its design goals are high performance multimedia processing, general-purpose programmability to support an ever-growing range of applications, simplicity for constrained environments where low power and low cost are paramount, and scalability for higher performance in less constrained multimedia systems. Another design goal of PLX is to facilitate exploration and evaluation of novel techniques in instruction set architecture, microarchitecture, arithmetic, VLSI implementations, compiler optimizations, and parallel algorithm design for new computing paradigms.

Key characteristics of PLX are a fully subword-parallel architecture with novel features like wordsize scalability from 32-bit to 128-bit words, a new definition of predication, and an innovative set of subword permutation instructions. We demonstrate the use and high performance of PLX on some frequently-used code kernels selected from image, video, and graphics processing applications: discrete cosine transform, pixel padding, clip test, and median filter. Our results show that a 64-bit PLX processor achieves significant speedups over a basic 64-bit RISC processor and over IA-32 processors with MMX and SSE multimedia extensions. Using PLX's wordsize scalability feature, PLX-128 often provides an additional 2× speedup over PLX-64 in a cost-effective way. Superscalar or VLIW (Very Long Instruction Word) PLX implementations can also add additional performance through inter-instruction, rather than intra-instruction parallelism. We also describe the PLX testbed and its software tools for architecture and related research.

Keywords: multimedia, instruction set architecture, ISA, processor architecture, media processing

1. Introduction

Multimedia information processing is the processing of integrated digital video, audio, images, graphics, animation, and text by a programmable processor. It is also called media processing [1], and is a significant and increasing fraction of the general-purpose workload in desktop and mobile computers and information appliances. In this paper, we describe a canonical processor architecture optimized for media processing. Such an architecture must be suitable for very constrained environments. Design goals include flexible general-purpose programmability and very high multimedia performance with small footprint, low power, and low cost. The architecture should also allow scalability for higher performance in environments that are less constrained.

Previously, multimedia instructions were added to microprocessor instruction set architectures (ISAs) to accelerate multimedia processing. Examples include MAX [2] and MAX-2 [3, 4] added to Hewlett-Packard's PA-RISC; MMX [5], SSE and SSE-2 [6] to Intel's IA-32; VIS [7] to Sun's UltraSparc; and AltiVec [8] to Motorola's PowerPC. Intel's new 64-bit ISA,

IA-64 [9, 10], also includes multimedia instructions as an integral part of the ISA. Some of these multimedia instruction sets are minimalist, while others are very large and complex [11–13], but all incur the cost of a large base microprocessor ISA. This paper describes a concise ISA, designed from scratch for a standalone processor optimized for media processing. This ISA can be implemented with very small area, cost, and power requirements, as necessary for the most cost-effective processors for constrained environments like multimedia personal digital assistants, wireless mobile multimedia devices [14], and wearable information appliances. Here, low power and low cost are just 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.

Digital signal processing (DSP) chips have been used very successfully for high-performance and low-cost audio processing. Extending them to processing all types of multimedia has been less successful because of the difficulty in their programming. Special-purpose video processing chips have been designed that implement some specific video processing algorithms, like MPEG for video compression and decompression, and H.261 or H.263 for video conferencing. Efforts to process multiple types of multimedia data with a single processor have resulted in media processors. Typical examples are the MAP, MAP-CA [15], and TriMedia [16] processors. MAP processors use a VLIW architecture with a large instruction set and special functional units tuned to perform common image processing kernels in hardware [15]. Although specifically designed for multimedia processing, these processors are often rather large and complex.

Multimedia information processing has matured sufficiently for us to begin the refinement process for a minimalist multimedia ISA [17]. We believe that high-performance and low-cost multimedia processing can be achieved by using a concise ISA in a RISC-like general-purpose processor that is designed from the beginning to support very fast subword-parallel processing. PLX [17, 18] is such an ISA, where every instruction can operate on multiple subwords in parallel. Each instruction therefore provides an inherent degree of operation parallelism. Although a basic PLX implementation can already achieve very high performance, further performance can be achieved by architectural techniques like superscalar or VLIW processor implementations, which exploit instruction-level parallelism. This paper describes PLX version 1.2.

Another goal of the PLX project is to encourage the design exploration of new architectural features for multimedia processing as well as the rigorous evaluation of proposed features. Instruction set architecture, as the native language of a machine, should be both efficient and effective for new computing paradigms, including multimedia. Hence, the PLX instruction set architecture includes some promising new architectural features like wordsize scalability (or datapath scalability [17]), novel subword permutation primitives, and instruction predication, which provide opportunities for interesting performance and implementation evaluations. The PLX project also includes the development of a testbed for such architectural explorations, as well as the exploration of related VLSI chip designs and compiler optimizations.

In Section 2, we define *full subword parallelism* and *wordsize scalability*, two key features in the design of PLX. In Section 3, we describe the PLX instruction set architecture. In Section 4, we describe some important multimedia kernels to illustrate how the PLX instructions are used, and to demonstrate the performance of the architecture. In Section 5, we describe the PLX architecture testbed, which can be used for research and education in instruction set architecture, microarchitecture, computer arithmetic, VLSI design, low-power design, compiler optimizations, and multimedia program optimizations.

2. Full Subword Parallelism and Wordsize Scalability

Two key characteristics that differentiate PLX from other instruction set architectures are full subword parallelism and wordsize scalability.

Subword parallelism is motivated by the two distinctive properties of multimedia information processing: large amounts of data parallelism and frequent use of low-precision data [1–3]. Subword parallelism, which is also called packed parallelism [5] or microSIMD parallelism [19], accelerates multimedia information processing by exploiting both properties.

Subword parallelism involves partitioning the processor's datapath into units smaller than a word, called *subwords*, and then processing these subwords in parallel with a single instruction. A parallel add instruction is shown in Fig. 1, where eight pairs of bytes are added in parallel with a single instruction in a single

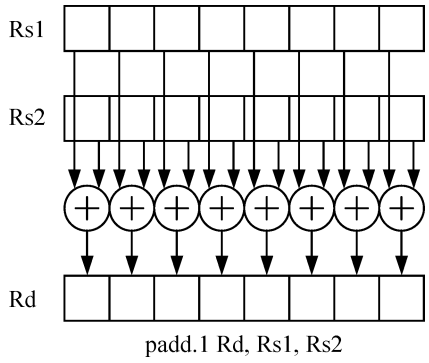

Figure 1. Parallel add on 1-byte subwords in PLX-64.

cycle, using a slightly modified version of an ordinary 64-bit adder. Subword parallelism has been shown to be a key feature contributing to high-performance multimedia capability [2, 3, 5, 7, 10], therefore PLX has been designed from scratch as a fully subword-parallel ISA.

We define a *fully subword-parallel ISA* as one that supports all useful subword sizes, for every computation instruction. In PLX, there are 1-byte, 2-byte, 4-byte, and *wordsize* subwords. A wordsize subword is one that spans an entire register, and is also called a full word. As we will explain below, the register size can vary with different PLX implementations, and therefore the size of a wordsize subword is also variable. Almost every computation instruction in PLX supports parallel execution on all four subword sizes. A few are defined for fewer subword sizes because the cost of supporting these instructions for other subword sizes is relatively large compared to their low expected utility in multimedia processing.

Another key characteristic of PLX is *wordsize scalability*, or *datapath scalability.* While the instruction size in PLX is fixed at 32 bits, the size of a word is variable from one PLX implementation to another. A given implementation can have a 32-bit, 64-bit, or a 128-bit wordsize, denoted PLX-32, PLX-64, and PLX-128 respectively. The default implementation is PLX-64. A wordsize subword, or full word, is 4 bytes, 8 bytes, and 16 bytes respectively for PLX-32, PLX-64, and PLX-128. Combined with subword parallelism, wordsize scalability allows increased operation parallelism and hence increased performance at a potentially lower cost when compared to other techniques. Wordsize scalability also allows different PLX implementations to trade off performance and cost. Compared to PLX-64, PLX-32 is likely to have lower performance, but also a lower cost. On the other hand, the wider datapath in PLX-128 doubles the subword parallelism, but at a lower cost compared to a superscalar implementation with an equivalent degree of operation parallelism.

Maximum operation parallelism is achieved when 16 1-byte subwords are packed into a single register in PLX-128. This allows 16 subwords to be processed in parallel, resulting in a potential 16× 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 because subword parallelism requires only minor modifications to the processor's functional units and datapath [2, 19]. Figure 2 shows the reduced datapath complexity of a subword-parallel processor compared to a superscalar or VLIW processor, both with 4-way parallelism. The superscalar or VLIW processor has the flexibility of performing four different instructions in a cycle, but this flexibility is not required in many multimedia computations with enormous amounts of data parallelism. Although each ALU in the superscalar processor can be one-fourth as wide as the ALU in the subword-parallel processor, the complexity of additional register ports, data buses, bypass paths, and instruction dispatch logic is much greater. As we will show in Section 4, superscalar implementation techniques can be used to further enhance a subword-parallel ISA like PLX.

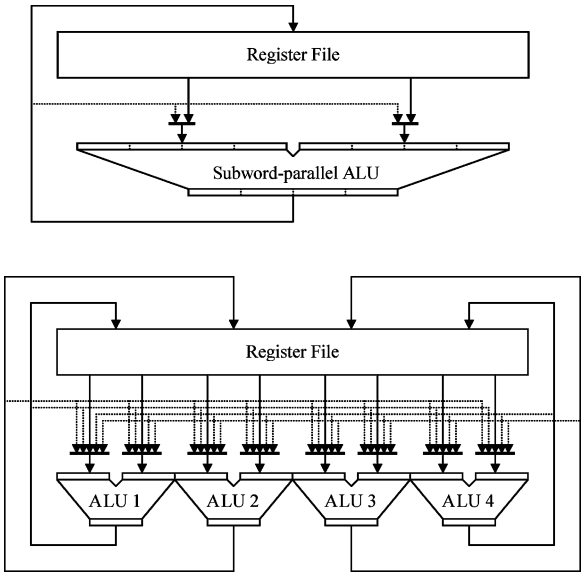

Figure 2. Datapath of 4-way subword-parallel and 4-way superscalar processors.

In addition to full subword parallelism and word-size scalability, PLX also supports low-cost constant multiplication in the ALU (Section 3.1), subword permutations (Section 3.2), optional parallel subword multipliers (Section 3.3), predication (Section 3.4), and saturation arithmetic (Section 3.5).

3. PLX Instruction Set Architecture

PLX is a register-based ISA with 32 general-purpose registers, R0 through R31. R0 always returns 0 when read; writing to it is equivalent to discarding the value written. All PLX instructions are 32 bits long and belong to one of six major instruction formats. The encoding of these is shown in Fig. 3.

Figure 4 shows the datapath of a basic single-issue PLX processor. All PLX computation instructions require at most two source registers and one destination register, and are executed by one of the three functional units: arithmetic logic unit (ALU), shift permute unit (SPU), and the optional integer multiplier (MUL), which is shown here with three pipelined stages. Tables 1–3 show the instructions executed by the ALU, the SPU, and the optional integer multiplier respectively. Instructions that write predicate registers are shown separately in Table 4; memory access and program flow instructions are shown in Table 5. These tables also show whether a given instruction is also found in the MAX-2 extensions for PA-RISC 2.0 processors [3, 4] or in the MMX, SSE, and SSE-2 extensions for IA-32 processors [5, 6]. MAX-2 is shown because PLX can also be considered as its successor in a standalone processor, since both are designed by the same architect, Lee, with the same minimalist goals. MMX+SSE-2 is shown as the multimedia ISA available in the dominant processor architecture, IA-32, for desktop and notebook computers.

3.1. ALU Instructions

Table 1 shows all the instructions executed by the ALU functional unit in Fig. 4. The ALU instructions include the basic *parallel add* and *parallel subtract* instructions, with modular arithmetic and saturation arithmetic. Addition with modular arithmetic means that overflows are ignored, whereas addition with saturation arithmetic means that the result is clamped to the largest or smallest representable number for that subword size. Saturation arithmetic is described further in Section 3.5. The *parallel average* instruction computes the average of two source subwords, using rounding rather than truncation to preserve precision in successive averages. An average is just an addition followed by a right shift of 1 bit. A round-to-odd rounding is easily implemented in hardware by OR'ing the two least significant bits, after the sum is shifted right [2]. The *parallel subtract average* is used to compute the

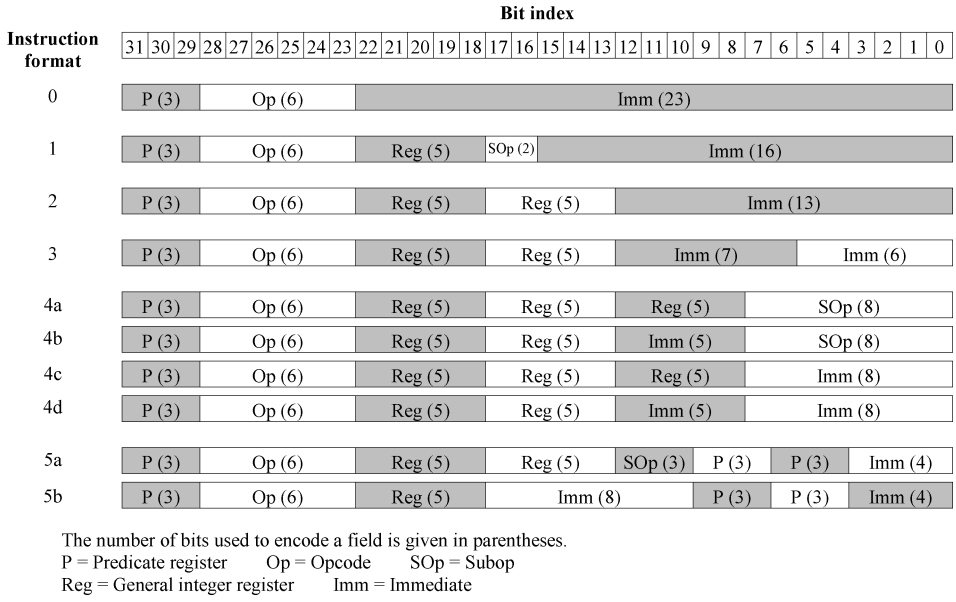

Figure 3. Encoding of PLX instructions.

Table 1. ALU instructions.

Instruction	Mnemonic	Description[1]	Format	PA-RISC 2.0 with MAX-2	IA-32 with MMX+SSE-2
Parallel add	padd	$c_i \leftarrow a_i + b_i$	4a	•	•
Parallel add with signed (or unsigned) saturation	padd.s, padd.u	$c_i \leftarrow a_i + b_i$ if $c_i < c_{\min}$, $c_i \leftarrow c_{\min}$ if $c_i > c_{\max}$, $c_i \leftarrow c_{\max}$	4a	•	•
Parallel subtract	psub	$c_i \leftarrow a_i - b_i$	4a	•	•
Parallel subtract with signed (or unsigned) saturation	psub.s, psub.u	$c_i \leftarrow a_i - b_i$ if $c_i < c_{\min}$, $c_i \leftarrow c_{\min}$ if $c_i > c_{\max}$, $c_i \leftarrow c_{\max}$	4a	•	•
Parallel average	pavg	$c_i \leftarrow (a_i + b_i) \gg 1$	4a	•	•
Parallel subtract average	psubavg	$c_i \leftarrow (a_i - b_i) \gg 1$	4a		
Parallel shift left and add	pshiftadd.l	$c_i \leftarrow (a_i \ll imm) + b_i$, for $imm = 1, 2,$ or 3	4a	•	
Parallel shift right and add	pshiftadd.r	$c_i \leftarrow (a_i \gg imm) + b_i$, for $imm = 1, 2,$ or 3	4a	•	
Parallel maximum	pmax	$c_i \leftarrow \max(a_i, b_i)$	4a		•
Parallel minimum	pmin	$c_i \leftarrow \min(a_i, b_i)$	4a		•
Logic operations: and, or, not, xor, and complement	and, or, not, xor, andcm *respectively*	$c \leftarrow a \,\&\, b$, $c \leftarrow a \mid b$, $c \leftarrow \bar{a}$, $c \leftarrow a \oplus b$, $c \leftarrow a \,\&\, \bar{b}$ respectively	4a	•[2]	•[3]
Parallel compare (for relation rel)	pcmp.rel	$c_{[n-1,0]} \leftarrow (\text{rel}(a_{n-1}, b_{n-1}), \ldots,$ $\text{rel}(a_1, b_1), \text{rel}(a_0, b_0))$	4a		•[4]

[1] c_i, a_i, and b_i represent the subwords in the destination register and the two source registers respectively. The range of i is $[0, n-1]$ for n subwords. If no index is given, the entire register is used. $c_{\min}$ and $c_{\max}$ represent the low and high saturation limits when saturation arithmetic is used. *imm* represents an immediate value given in the instruction word. Operators $\ll$ and $\gg$ denote left and right shifts respectively. $\max(a, b)$ and $\min(a, b)$ return the larger and smaller of a and b respectively. The operators &, |, $\oplus$ denote bitwise AND, OR, and XOR operations respectively. Complement of a is $\bar{a}$. Numbers in subscripted square brackets address the individual bits or bit ranges in a register. For example, $a_{[0]}$ is the least-significant bit of a; $a_{[31,0]}$ is the least-significant 4 bytes of a. $\text{rel}(a, b)$ compares a and b for the relation specified in the compare instruction. If the relation is true, 1 is returned; if the relation is false, 0 is returned.

[2] All logic operations exist except *not*.

[3] All logic operations exist except *and complement*.

[4] Instead of generating one bit per subword pair, the entire destination subword is written with 1's or 0's.

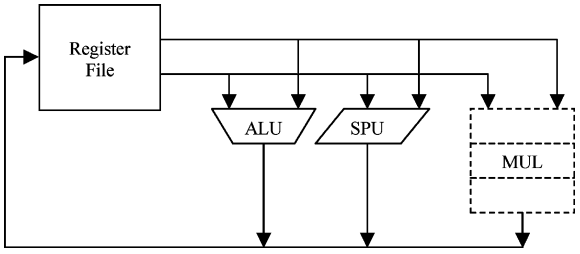

Figure 4. Single-issue PLX processor with 3 functional units.

average of the first source subword and the negative of the second source subword. *Parallel shift left and add* and *parallel shift right and add* instructions shift the first source subword left or right by 1, 2, or 3 bits before adding it to the second source subword. The shift is implemented 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 [2, 20]. Therefore, for some very low cost environments, the integer multiplier in a PLX processor may be omitted. The *parallel maximum* and *parallel minimum* instructions write the greater or smaller, respectively, of the corresponding subwords in the two source registers to the corresponding subword in the destination register. These instructions are very useful for sorting algorithms since a single pair of *parallel maximum* and *parallel minimum* instructions can perform a conditional swap operation for multiple pairs of subwords in a single cycle.

Table 2. Shift and subword permutation instructions.

Instruction	Mnemonic	Description[1]	Format	PA-RISC 2.0 with MAX-2	IA-32 with MMX+SSE-2
Parallel shift left	pshift.l	$c_i \leftarrow a_i \ll b$	4a		•
Parallel shift right	pshift.r	$c_i \leftarrow a_i \gg b$	4a		•
Parallel shift left immediate	pshifti.l	$c_i \leftarrow a_i \ll imm$	4b	•	•
Parallel shift right immediate	pshifti.r	$c_i \leftarrow a_i \gg imm$	4b	•	•
Shift left logical immediate	slli	$c \leftarrow a \ll imm$	2	•	•
Shift right arithmetic (logical) immediate	srai, srli	$c \leftarrow a \gg imm$	2	•	•
Shift right pair	shrp	$c \leftarrow ((a, b) \gg imm)_{\mathrm{L}}$ (Fig. 5)	4c	•	
Mix left, mix right	mix.l, mix.r	Fig. 6	4a	•	
Check, excheck	check, excheck	Fig. 7 and 8	4a		
Alternate left, alternate right	alt.l, alt.r	Fig. 9	4a		
Permute	perm	Fig. 10	4a		
Permute set immediate	permseti	Fig. 11	4d	•[2]	•[2]

[1]Subscript L denotes the lower half of the preceding quantity.
[2]These are the *permh* and *pshufw* instructions in PA-RISC 2.0 and IA-32 respectively. Both instructions are limited to permutations of four 2-byte subwords only, and do not have the repeating capability of *permseti*.

Table 3. Multiply instructions.

Instruction	Mnemonic	Description	Format	PA-RISC 2.0 with MAX-2	IA-32 with MMX+SSE-2
Parallel multiply even	pmul.even	$(c_{2i+1}, c_{2i}) \leftarrow a_{2i} \times b_{2i}$	4a		
Parallel multiply odd	pmul.odd	$(c_{2i+1}, c_{2i}) \leftarrow a_{2i+1} \times b_{2i+1}$	4a		
Parallel multiply and shift right	pmulshr	$c_i \leftarrow ((a_i \times b_i) \gg k)_{\mathrm{L}}$ where $k \in \{0, 8, 15, 16\}$	4a		

Table 4. Predication instructions.

Instruction	Mnemonic	Description[1]	Format
Parallel compare and write predicate	pcmp.wp.rel	(P7, ..., P1, P0) $\leftarrow$ (rel(a_7, b_7), ..., rel(a_1, b_1), rel(a_0, b_0))	4a
Compare	cmp.rel	Pd1 $\leftarrow$ rel(a_0, b_0), Pd2 $\leftarrow \overline{\text{Pd1}}$	5a
Compare parallel write 1	cmp.pw1.rel	If rel(a_0, b_0) = 1, then Pd1 $\leftarrow$ 1, Pd2 $\leftarrow$ 0	5a
Test bit	testbit	Pd1 $\leftarrow a_{[b]}$, Pd2 $\leftarrow \overline{\text{Pd1}}$	5a
Test bit immediate	testbiti	Pd1 $\leftarrow a_{[imm]}$, Pd2 $\leftarrow \overline{\text{Pd1}}$	5b
Change predicate register set, Change predicate set and load	changepr, changepr.ld	See Section 3.4	5b

[1]P0 to P7 are the eight predicate registers in the currently active predicate register set. Pd1 and Pd2 are the destination predicate registers.
Note: Neither PA-RISC 2.0 with MAX-2 nor IA-32 with MMX+SSE-2 support predicated execution.

Table 5. Memory access and program flow instructions.

Instruction	Mnemonic	Description[1]	Format	PA-RISC 2.0 with MAX-2	IA-32 with MMX+SSE-2
Load (base + displacement)	load	$c \leftarrow$ M[$b + imm$]	2	•	
Load update (base + displacement)	load.update	$c \leftarrow$ M[b], $b \leftarrow b + imm$	2	•	
Load indexed	loadx	$c \leftarrow$ M[$a + b$]	4a	•	
Load indexed update	loadx.update	$c \leftarrow$ M[b], $b \leftarrow a + b$	4a	•	
Store (base + displacement)	store	M[$b + imm$] $\leftarrow a_0$	2	•	
Store update (base + displacement)	store.update	$b \leftarrow b + imm$, M[b] $\leftarrow a_0$	2	•	
Load immediate[2]	loadi	$c_t \leftarrow imm$, all other $c_i \leftarrow 0$	1	•[3]	•[3]
Load immediate and keep[2]	loadi.k	$c_t \leftarrow imm$, all other c_i are unchanged	1		
Jump	jmp	PC $\leftarrow$ PC + imm	0	•	•
Call	call	R31 $\leftarrow$ PC + 4, PC $\leftarrow$ PC + imm	0	•	
Return	ret	PC $\leftarrow$ R31	0	•	•

[1] M is the memory array; PC is the program counter.
[2] c_t is one of the four least-significant 16-bit subwords of c. In PLX-32, c_t can be c_0 or c_1; in PLX-64 and PLX-128, c_t can be c_0, c_1, c_2, or c_3.
[3] These instructions write an immediate value to the rightmost bits of the destination register. Unlike PLX, they cannot be used to load an arbitrary subword.

Five logical instructions are included in PLX: *and*, *and complement*, *or*, *xor*, *not*. The *and complement* instruction complements the second operand before AND'ing it with the first operand.

There are also immediate forms of the *add*, *subtract*, *and*, *or*, *xor* instructions, where the second operand comes from an immediate field in the instruction, rather than from a register. These immediate instructions operate on full words, not on parallel subwords.

The *parallel compare* instruction tests 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 n bits, one for each of the n pairs of subwords compared, is written to the rightmost n bits of the destination register.

3.2. Shift and Subword Permutation Instructions

The shift permute unit (SPU) in Fig. 4 is responsible for executing all shift and subword permutation instructions in PLX. These are summarized in Table 2. There are four basic parallel shift instructions: *parallel shift left*, *parallel shift right*, *parallel shift left immediate*, *parallel shift right immediate*. These instructions shift the subwords of the first source register to the left or right by the number of bits specified in a second source register or by an immediate amount specified in the instruction. The right shifts can be specified to be arithmetic or logical. Due to the limited encoding space in 32-bit instructions, the maximum shift amount that can be specified in an immediate field for parallel shift instructions is 31 bits. To perform shifts of up to 127 bits, there are three other instructions: *shift left logical immediate*, *shift right arithmetic immediate*, *shift right logical immediate*. These instructions work on full words instead of subwords.

To select a bit field from a data object that spans two registers, the *shift right pair* instruction is used (Fig. 5). Two source registers are concatenated and shifted right, and then the lower half of the shifted result is placed in the destination register. If both source registers are the same, the result is a rotation of that register.

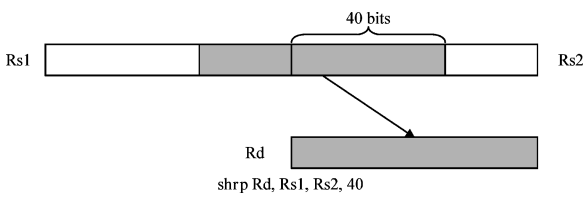

Figure 5. Example of *shift right pair* in PLX-64.

Subword permutation instructions reorder the subwords from one or two source registers. Instructions operating on two source registers deal with twice the number of subwords, and must select only a subset for reordering. These include the *mix*, *check*, *excheck*, and *alt* instructions. *Mix* instructions, which come in two variants, write alternating (odd or even) subwords from two source registers to the destination register (Fig. 6). Transposition of two-dimensional data can be performed very efficiently using *mix* instructions [3, 11]. *Check* instructions [21] traverse the subwords of the two source registers in a checkerboard pattern, and write these subwords to the destination register (Fig. 7). *Excheck* [21] performs the equivalent of a *check* followed by an exchange of the subwords in the pair before writing them to the destination register (Fig. 8). Another new permutation instruction, *alt*, writes alternate subwords of the source registers to the destination register (Fig. 9). This is very useful for sub-sampling in image and video processing. Each of *mix*, *check, excheck*, and *alt* instructions can work on 1-byte, 2-byte, and 4-byte subwords.

To perform arbitrary permutations of subwords from one source register, *permute* and *permute set immediate* instructions are used. Both can work on 1-byte and 2-byte subwords. To specify any permutation of n subwords with or without repetitions, $n \times \log_2(n)$ permutation control bits are required (Table 6). In *permute*,

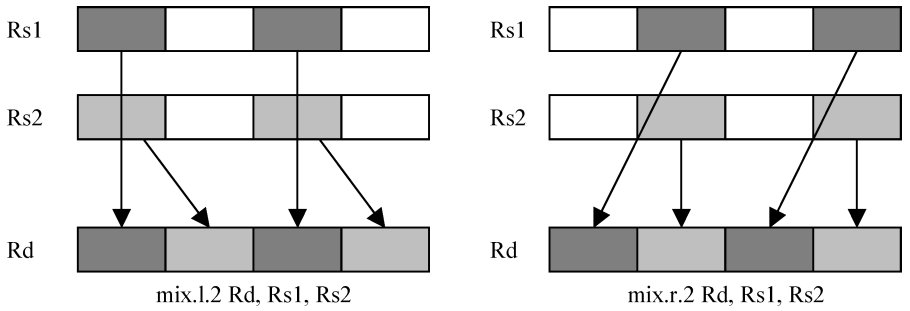

Figure 6. *Mix left* and *mix right* on 2-byte subwords in PLX-64.

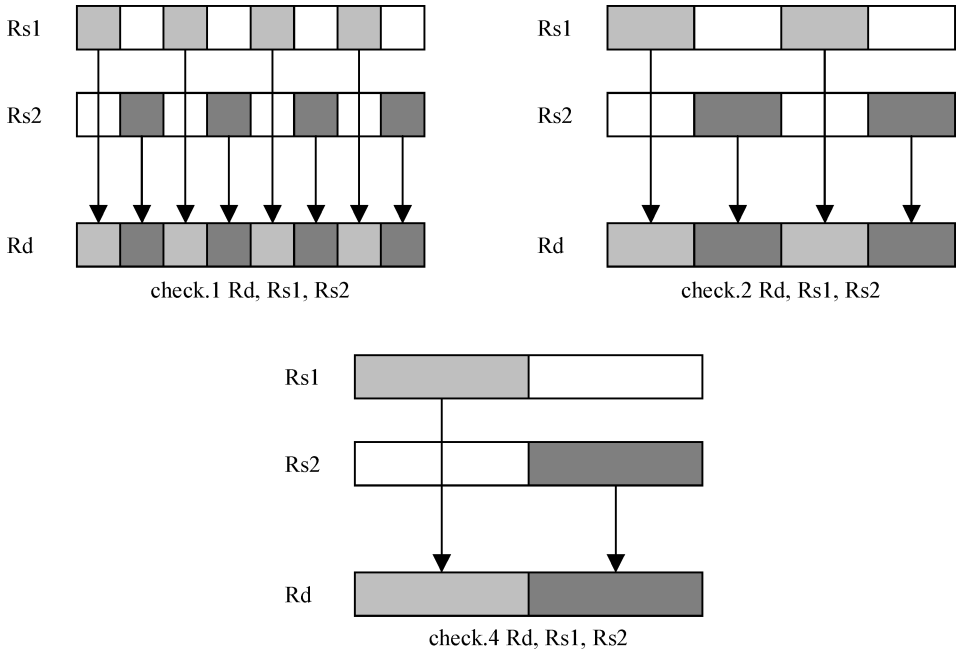

Figure 7. *Check* on 1-byte, 2 byte, and 4-byte subwords in PLX-64.

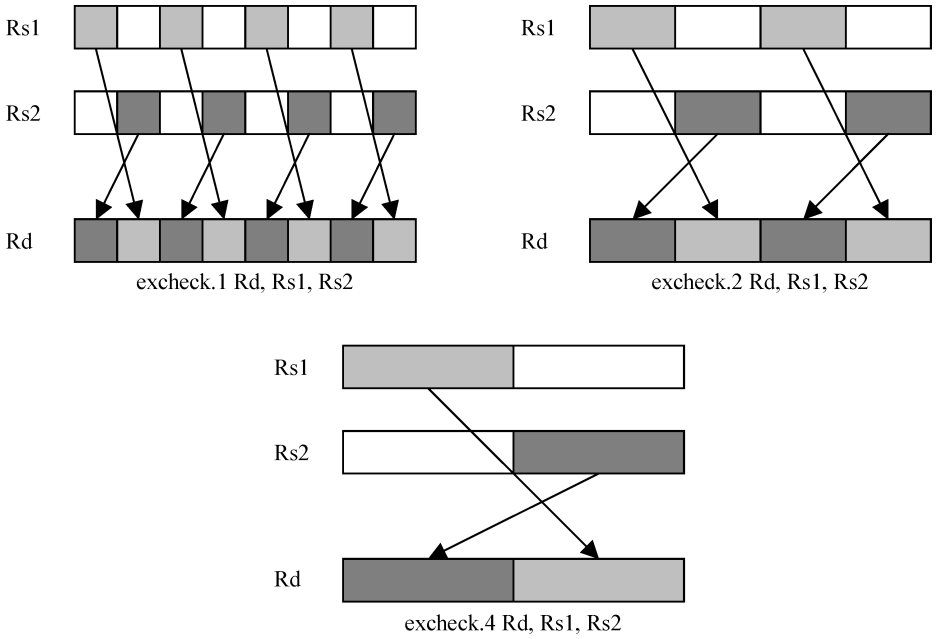

Figure 8. *Excheck* on 1-byte, 2 byte, and 4-byte subwords in PLX-64.

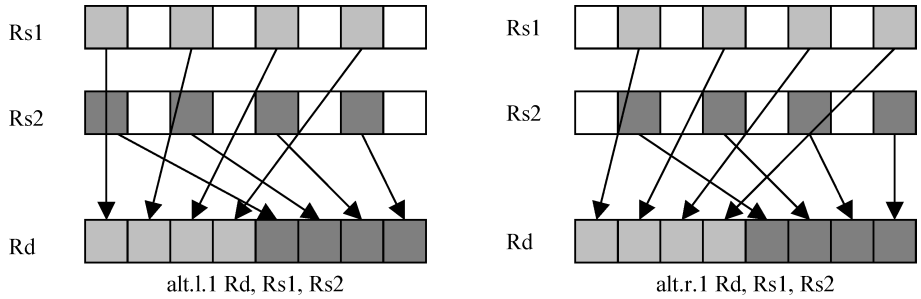

Figure 9. *Alternate left* and *alternate right* on 1-byte subwords in PLX-64.

these control bits are read from a second source register while the data to be permuted is read from the first source register. *Permute* instruction on 2-byte subwords in PLX-64 is shown in Fig. 10. The maximum number of control bits needed by the *permute* instruction is 64, which is used to permute 16 1-byte subwords in a 128-bit register in PLX-128.

Table 6. Number of permutation control bits required to specify an arbitrary permutation.

Number of subwords (n)	Number of permutation control bits ($n \times \log_2(n)$)
2	2
4	8
8	24
16	64
32	160

Permute set immediate differs from *permute* in that the permutation control bits are specified in an immediate field instead of a second source register. The size of this immediate field is 8 bits, which is sufficient to specify permutations of 4 subwords. The source register is first parsed into groups of 4 subwords each. Then, the permutation specified by the 8 immediate bits is performed on each group. Examples of the *permute set immediate* instruction for PLX-64 are shown in Fig. 11, where the permutation control bits are shown mod 4, so that each digit corresponds to the index of one of the four subwords in each group in the source register. *Permute set immediate* was first introduced by Lee in [21]

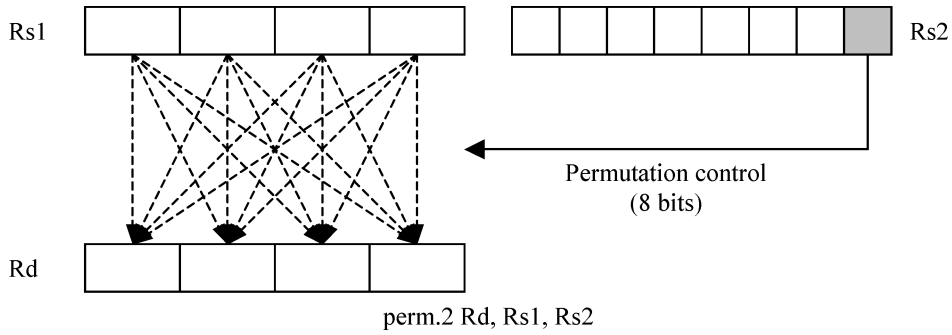

Figure 10. *Permute* on 2-byte subwords in PLX-64.

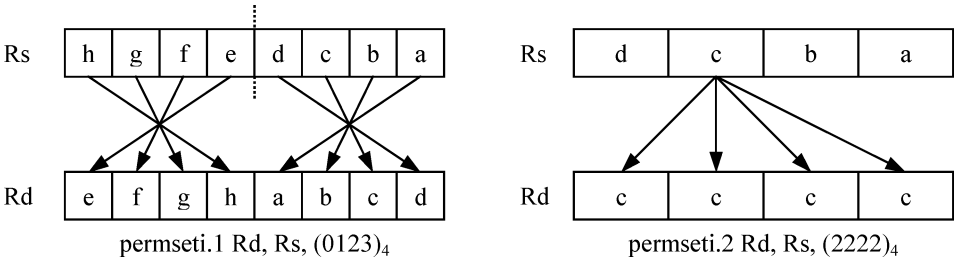

Figure 11. Examples of *permute set immediate* on 1-byte and 2-byte subwords in PLX-64.

to eliminate the dependence on the number of control bits needed to permute many subwords. This allows a smaller set of repeating permutations to be achieved on an unlimited number of subwords.

Mix, *check*, *excheck*, and *permute set immediate*, defined for different subword sizes, have been proposed as efficient subword permutation primitives for performing hierarchical permutations on two-dimensional images [21]. With respect to basic 2×2 blocks of subwords packed in two registers, *mix* transforms columns into rows, while *check* and *excheck* transform diagonals and anti-diagonals into rows. Permutation of subwords in a row can then be accomplished using the *permute* or *permute set immediate* instructions on a single source register.

3.3. Multiply Instructions

While many multimedia algorithms only need multiplication by constants, some require multiplication of two variables. The optional multiplier unit (MUL) in Fig. 4 supports this by executing the multiply instructions summarized in Table 3. *Parallel multiply even* and *parallel multiply odd* instructions only multiply the even or odd subwords of the two source registers respectively, generating full-sized products. The *parallel multiply and shift right* instruction generates four half-size products by shifting the intermediate full-size products right by 0, 8, 15, or 16 bits, and selecting the lower halves of the shifted results. These three multiply instructions are shown in Figs. 12 and 13.

Currently, only 2-byte subwords are supported by these multiply instructions for cost reasons. This is the most common subword size used in multimedia applications that require multiplication of two variables. PLX-64 can choose to implement all three multiply instructions with one, two, or four 16-bit multipliers. Since *parallel multiply even* and *parallel multiply odd* instructions involve only two 16-bit multiplications in PLX-64, they 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 one cycle delay, and the whole instruction can be completed with one extra cycle of latency. Similarly, the *parallel multiply and 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 1 extra cycle if two pipelined multipliers are implemented, or with 3 extra cycles if only one multiplier is implemented.

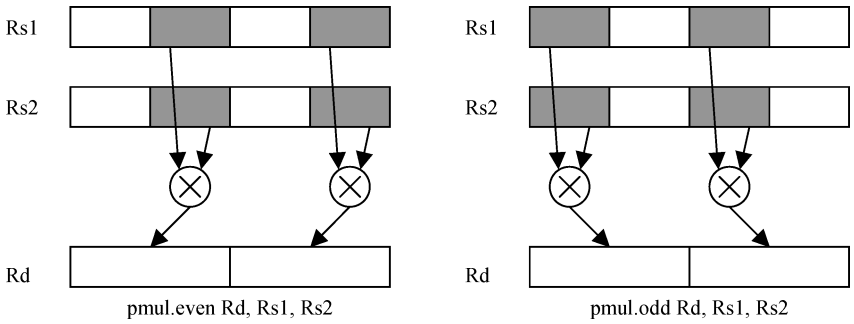

Figure 12. *Parallel multiply even* and *parallel multiply odd*.

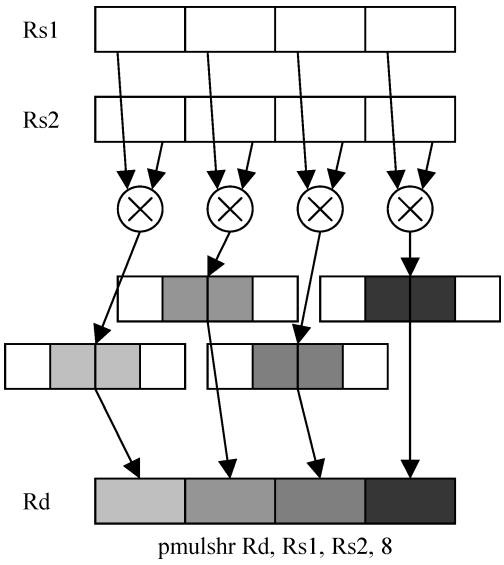

Figure 13. *Parallel multiply and shift right* with an 8-bit right shift in PLX-64.

3.4. Predication

PLX uses predication to reduce the performance degradation caused by conditional branch instructions. All instructions can be predicated using one of the 128 1-bit predicate registers. These registers are grouped into 16 predicate register sets, each containing 8 predicate registers. At any time, only one of these 16 sets is active, and the 8 predicate registers in this active set are addressed P0 through P7. Organizing the predicate registers into multiple groups in this way 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. Since we are exploring efficient predication for multimedia information processing, we first define a large number, 128, of predicate registers, so as not to restrict creative exploitation of predication by algorithms. This large number of predicate registers can be reduced if we find that it is not needed.

To access a predicate register that resides outside the currently active set, the *change predicate register set* instruction (Table 4) is used first to change the active set. A similar *change predicate register set and load* instruction can further initialize predicate registers in the new active set to values given in the immediate field of this instruction. In every predicate set, P0 always returns true, therefore all instructions predicated with P0 execute unconditionally. The remaining seven predicate registers, P1–P7, can be set and cleared using four different instructions.

Parallel compare and write predicate is a variation of the *parallel compare* instruction that writes its result to the active predicate register set instead of a general register. If more than 8 bits are generated as a result, only the bits corresponding to the least-significant 8 subword pairs are written; the remaining bits are discarded. Note that this instruction is identical to the *parallel compare* instruction at the end of Table 1 except that the predicate registers are written rather than a general register.

The basic *compare* instruction compares rightmost subwords of the two source registers for a relation, and writes two predicate registers as destination. If the relation is true, a 1 is written to the first predicate register and a 0 to the second one; if the relation is false, a 0 is written to the first predicate register and a 1 to the second one.

The other type of compare instruction is *compare parallel write 1*. Here, the predicate registers are written only when the relation between the rightmost subwords in the two source registers is true, in which case

a 1 is written to the first predicate register and a 0 to the second one. Nothing is written to either of the predicate registers if the relation is false, and therefore the initial values of the registers are preserved. The program should initialize the predicate registers to known values before executing this instruction. This definition allows multiple *compare parallel write 1* instructions targeting the same predicate registers to be executed simultaneously. There are six relations that can be tested between the source registers: equal, not equal, less, less or equal, greater, greater or equal. The last four relations have two forms; the first is for comparison of signed integers, the second for unsigned integers. The functionality of both types of compare instructions is summarized below.

Type:	compare
Mnemonic:	cmp.rel.4 (rel specifies the tested relation. Subword size is specified as 4 bytes, hence the rightmost 4 bytes are compared.)
Example:	cmp.eq.4 R1, R2, P1, P2
Operation:	If the rightmost 4 bytes of R1 and R2 are equal, then P1 $\leftarrow$ 1, P2 $\leftarrow$ 0, else P1 $\leftarrow$ 0, P2 $\leftarrow$ 1.
Type:	compare parallel write 1
Mnemonic:	cmp.pw1.rel.w (*pw1* stands for *parallel write 1*; *w* indicates that full words are compared.)
Example:	cmp.pw1.eq.w R1, R2, P1, P2
Operation:	If R1 and R2 are equal, then P1 $\leftarrow$ 1, P2 $\leftarrow$ 0, else P1 and P2 are unchanged.

Finally, the *test bit* instruction is used to test the value of an arbitrary bit in a source register. The index of the tested bit is specified in a second source register. If this bit is 1, a 1 is written to the first destination predicate register and a 0 to the second one; if the bit is 0, a 0 is written to the first destination predicate register and a 1 to the second one. Immediate versions of these four instructions are also included in the instruction set.

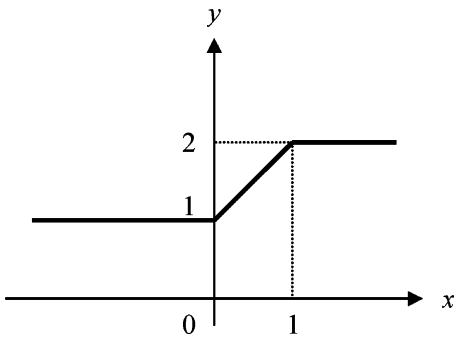

Figure 14. Limiting function in Code 1.

To illustrate how predication is used to eliminate conditional branches, we consider the continuous limiting function shown in Fig. 14. The function maps the input values less than 0 and greater than 1, to 1 and 2 respectively. In between, the function is linear in x with the formula $y = x + 1$. This could be written in C for PLX-64 as shown in Code 1. The input and output data are fixed-point integers in s31.32 format. This means that the most-significant bit of the 64-bit word is the sign bit, the next 31 bits represent the integral part of the integer, and the least-significant 32 bits represent the fractional part.

The corresponding code written in PLX assembly is shown in Code 2. The first two instructions read the function parameters to R20 and R21 from the top of the stack. R29 is the stack pointer; R22 is used as the local copy of the input; R23 is initialized to 1 in the s31.32 format. The first compare instruction clears P2 to 0. The following two compare instructions resolve the if-then-else statements and set the predicate registers accordingly. Instructions 8, 9, and 10 compute the result, which is placed in R24. The store

Code 1. C code for the limiting function in Fig. 14.

```
void limit (int *result, int *input)
{
  int one = 0x0000000100000000;          /* initialize to 1 in s31.32 format */
                                         /* int type is 64 bits in PLX-64    */

  if ( *input < 0 )           { *result = one; }           /* less than 0    */
  else if ( *input < one )    { *result = *input + one; }  /* linear region  */
  else                        { *result = one << 1; }      /* greater than 1 */

  return;
}
```

Code 2 PLX-64 assembly for the limiting function in Fig. 14.

```
limit:
        P0 load      R20, R29,  24      # R20 is pointer to result
        P0 load      R21, R29,  16      # R21 is pointer to input

        P0 load      R22, R21,   0      # R22 is local copy of input
        P0 loadi.2   R23,   1           # R23 is initialized to 1 in s31.32 format

        P0 cmp.eq.1   R0,  R0,  P0, P2  # Init P2 to zero
        P0 cmp.lt.w  R22,  R0,  P1, P3  # Write predicate registers P1 and P3
        P3 cmp.lt.w  R22, R23,  P2, P3  # If 0 < input < 1, then write predicate
                                        # registers P2 and P3

        P1 loadi.2         R24,   1     # Output is 1 if input is less than 0
        P2 padd.w    R24, R22, R23      # Linear region
        P3 slli      R24, R23,   1      # Output is 2 if input is greater than 1

        P0 store.w   R24, R20,   0      # Write result

        P0 ret       R31                # Return from function
```

instruction writes R24 to the destination address, and the *ret* instruction returns from the function.

Conditional branches are replaced by in-line predicated execution. Two compare instructions are sufficient to resolve if-then-else statements with three different outcomes. The subsequent three instructions predicated with different predicate registers can be issued in parallel in a superscalar processor, further enhancing the performance.

3.5. Saturation Arithmetic

To deal with the multiple overflows that may occur during parallel arithmetic operations, PLX uses saturation arithmetic [2, 11, 13]. With saturation arithmetic, the values of both the source and destination operands are restricted to pre-defined numeric ranges. Any result falling outside the allowed range is clipped by hardware to either the upper or the lower range limit. In this way, multiple parallel overflows can be handled efficiently without operating system intervention. Saturation arithmetic must correctly handle the two types of overflows possible: (1) a *positive overflow*, which occurs when the result is larger than the upper limit of the allowed result range, and (2), a *negative overflow,* which occurs when the result is smaller than the lower limit of the allowed result range.

For a given instruction several saturation options may exist based on whether the operands and the result are treated as signed or unsigned integers. For a three-register instruction, eight different saturation options may be defined. Not all of these saturation options are equally useful. For PLX only two saturation options are selected.

Signed saturation: Both source and destination subwords are treated as signed integers, where the most significant bit of the subword indicates its sign. For n-bit subwords, source and destination values are limited to the range $[-2^{n-1}, 2^{n-1}-1]$. If a positive overflow occurs, the result is saturated to $2^{n-1}-1$. If a negative overflow occurs, the result is saturated to -2^{n-1}. In an add instruction using signed saturation, a positive overflow is possible only when both source values are positive. Similarly, a negative overflow is possible only when both source values are negative.

Unsigned saturation: Both source and destination subwords are treated as unsigned integers. Their values are limited to the range $[0, 2^n - 1]$. If a positive overflow occurs, the result is saturated to $2^n - 1$. If a negative overflow occurs, the result is saturated to 0. In an add instruction using unsigned saturation, a negative overflow is not possible because the source subwords are always positive. In a subtract instruction, a negative overflow is possible, and any negative result will be clamped to 0.

In addition to efficient handling of parallel overflows, saturation arithmetic also facilitates several other useful computations. In Table 7, we summarize other arithmetic operations that can be performed using saturation arithmetic [2, 12, 13]. For example, the first two rows show that saturation arithmetic can be used to clip results to arbitrary maximum or minimum values. Without saturation arithmetic these operations could normally take up to five instructions for each subword, which would include instructions to check for upper and lower bounds, and then to perform the clipping. Using saturation arithmetic, this effect can be achieved

Table 7. Clipping and clamping operations using saturation arithmetic.

Operation[1]	Instructions	Explanation
Clip all a_i at an arbitrary maximum value $a_{\max}$, where $0 \le a_{\max} \le 2^{15}-1$.	padd.s.2 a, a, b	Initially all b_i contain $2^{15}-1-a_{\max}$. If $a_i > a_{\max}$, this instruction clips a_i at $2^{15}-1$.
	psub.s.2 a, a, b	a_i is now at most $a_{\max}$.
Clip all a_i at an arbitrary minimum value $a_{\min}$, where $-2^{15} \le a_{\min} < 0$.	psub.s.2 a, a, b	Initially all b_i contain $2^{15}+a_{\min}$. If $a_i < a_{\min}$, this instruction clips a_i at -2^{15}.
	padd.s.2 a, a, b	a_i is now at least $a_{\min}$.
$c_i = \lvert a_i - b_i \rvert$ (Compute the absolute values of the differences between a_i and b_i).	psub.u.2 e, a, b	If $a_i > b_i$, then $e_i = a_i - b_i$, else $e_i = 0$.
	psub.u.2 f, b, a	If $a_i \le b_i$, then $f_i = b_i - a_i$, else $f_i = 0$.
	padd.2 c, e, f	If $a_i > b_i$, then $c_i = a_i - b_i$, else $c_i = b_i - a_i$.

[1]Symbols a through f represent registers. Destination is the first register in an instruction. The letters s and u indicate signed and unsigned saturation respectively.

in two instructions for all subwords packed in a register. This is a speedup of $\frac{5n}{2}$, where n is the number of subwords in a register.

Saturation arithmetic can also be used for in-line conditional execution, reducing the need for conditional branches that can cause significant performance degradation in pipelined processors. For example, the last row of Table 7 and Fig. 15 show how the absolute values of the differences of corresponding pairs of subwords in two registers can be computed in parallel using only three instructions.

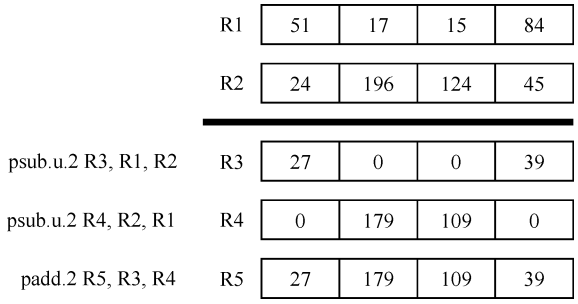

Figure 15. Parallel computing of absolute differences using saturation arithmetic.

Table 8 provides a comparison of PLX versus other multimedia architectures for their register and subword sizes, and saturation options.

3.6. Memory Access and Program Flow Instructions

Table 5 shows the memory access and program flow instructions in PLX. Base plus displacement addressing can be used by both load and store instructions. Load instructions can also use indexed addressing. Only full words are loaded into registers with load instructions. Store instructions can store 1-byte, 2-byte, 4-byte, and full words. To make use of subword parallelism, a full word is typically loaded, and then parallel shift instructions or subword permutation instructions are used to zero out unwanted data, or rearrange the subwords in the register. The load instruction has a variant where the base address register is updated (post-modified) after the load is performed. In a post-modify scheme, the data loaded is at the address pointed by the initial value of the base register. After the load, the base register is incremented by the displacement. This facilitates accessing the elements of an array that are separated from each other by a fixed distance. Similarly, a variant of the

Table 8. Comparison of PLX and other multimedia ISAs.

	PLX	MAX-2	MMX	SSE-2	AltiVec	IA-64
Number of integer registers	32	32	8	8	32	128
Integer wordsize (bits)	32, 64, 128	64	64	128	128	64
Subword sizes (bytes)	1, 2, 4, 8, 16	2	1, 2, 4	1, 2, 4, 8	1, 2, 4	1, 2, 4
Modular arithmetic	•	•	•	•	•	•
Saturation options[1]	sss, uuu	sss, uus	sss, uuu	sss, uuu	sss, uuu	sss, uuu, uus

[1]The letters s and u indicate whether the destination and the two source registers are treated as signed or unsigned integers respectively.

store instruction does a pre-modify of the base register before the store to memory is performed.

Using the *load immediate* instruction, any one of the four 2-byte subwords of a 64-bit register can be loaded with the 16-bits in the immediate field of the instruction. The subword to be loaded is selected using the subword specifier field in the instruction. The remaining three subwords are normally cleared to zero, however they can be left unchanged by using the *load immediate and keep* variant. In PLX-64, a register can be initialized to any arbitrary value using at most four *load immediate* instructions.

Program flow can be changed with *jump* instructions. The basic *jump* instruction changes the program counter by adding to it the value given in the immediate field (displacement) of the instruction. The *call* instruction saves the address of the instruction after it to R31, and then increments the program counter by the displacement given in the instruction word. The *return* instruction changes the program counter to the value in R31, the return address saved by the previous *call* instruction, thus implementing a return from subroutine. Conditional branches in PLX 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.

4. Examples and Performance

The examples below are four frequently-used code kernels selected from image, video, and graphics processing applications. They illustrate the use of PLX's architectural features like subword parallelism, low-cost multiplication, subword permutations, predication, and wordsize scalability. They also illustrate the *parallel average* instruction, and parallel sorting using *parallel minimum* and *parallel maximum* instructions. For each kernel, we perform simulations in four different setups:

(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 we are comparing this with the first and third setups, where the wordsize is 64 bits, we do not use SSE-2 instructions, which require 128-bit words. The IA-32 with MMX and SSE is chosen to represent the dominant processor architecture in notebook processors.
(3) PLX-64 including all PLX-specific optimizations.
(4) PLX-128. Compared to PLX-64, this shows the speedup due to the wordsize scalability feature.

Table 9. Speedup over the basic 64-bit ISA.

	Basic 64-bit RISC (normalized to 1.0)	IA-32 with MMX+SSE	PLX-64	PLX-128
DCT	1.0	1.1	4.6	9.1
Pixel padding	1.0	4.9	7.9	7.9
Clip test	1.0	0.5	1.9	1.9
Median filter	1.0	5.3	8.0	16.0

Simulation results are summarized in Table 9. To emphasize the effect of ISA features on performance, a simple single-issue pipeline is used. All instructions, including loads and stores, have single-cycle execution latencies. 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 Table 10, which shows only the instructions used by at least one example, and highlights those instructions used more frequently.

4.1. Discrete Cosine Transform

Discrete cosine transform (DCT) and its inverse (IDCT) are extensively used in image and video compression standards, such as JPEG and MPEG. We simulate a two-dimensional DCT on 8×8 blocks of 16-bit pixels using the method described in [22], which minimizes the number of multiplications needed.

Each two-dimensional 8×8 DCT can be decomposed into 8 independent one-dimensional DCTs on the rows of the 8×8 block, followed by eight more independent one-dimensional DCTs on the columns. This is implemented by transposing the block after the first set of DCTs on the rows, and then performing the second set of DCTs on the rows of the transposed block. Afterwards, a second transposition is used to restore the initial orientation of the block (Code 3). These transpositions and multiplications by fractional constants are the two most time-critical operations in DCT

Table 10. Instruction frequencies for simulated kernels.

Instruction	DCT (%)	Pixel padding (%)	Clip test (%)	Median filter (%)
padd.2	**15.62**	0.00	0.00	0.00
psub.2	**13.12**	0.00	0.00	0.00
psub.4	0.00	0.00	**5.17**	0.00
pavg.1	0.00	**6.50**	0.00	0.00
pshiftadd.1.r	**3.75**	0.00	0.00	0.00
pshiftadd.2.r	**20.62**	0.00	0.00	0.00
pshiftadd.3.r	**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**
andcm	0.00	**11.38**	0.00	0.00
or	0.00	**26.00**	0.00	0.00
cmp.pw1.eq.4	0.00	0.00	**12.07**	0.00
cmp.pw1.gt.4	0.00	0.00	**31.03**	0.00
cmp.pw1.lt.4	0.00	0.00	**31.03**	0.00
cmp.eq.w	0.08	0.00	0.00	0.00
cmp.gt.w	1.41	0.81	1.72	0.00
changepr.ld	0.00	0.00	1.72	0.00
addi	0.78	0.00	0.00	0.00
subi	1.48	0.81	1.72	0.00
pshifti.r.2	**1.88**	0.00	0.00	0.00
srli	0.00	0.00	**10.34**	0.00
srai	0.00	0.00	0.00	1.85
shrp	0.00	0.00	0.00	**12.96**
mix.l.2	**6.25**	**13.00**	0.00	0.00
mix.r.2	**6.25**	**13.00**	0.00	0.00
mix.l.4	**6.25**	**13.00**	0.00	0.00
mix.r.4	**6.25**	**13.00**	0.00	0.00
load	**5.00**	0.05	0.00	**16.67**
store.w	**7.50**	0.00	0.00	3.70
loadi	0.31	0.00	0.01	1.85
jmp	1.52	0.82	1.73	1.85
jmp.link	0.00	0.81	1.72	1.85
ret	0.00	0.81	1.72	1.85
Total	100	100	100	100
Instruction count	2559	31506	14338	5601

Code 3. Pseudocode for 8 × 8 DCT.

1. For *i* from 0 to 7 do
one-dimensional DCT on row *i*
2. Transpose 8×8 matrix
3. For *i* from 0 to 7 do
one-dimensional DCT on row *i*
4. Transpose 8×8 matrix

[23]. These operations can be accelerated using subword permutation instructions and parallel arithmetic instructions.

Matrix transposition can be performed efficiently using *mix* instructions. Figure 16 shows how a 4×4 matrix of 2-byte elements initially stored in four 64-bit registers can be transposed using only 8 *mix* instructions and 3 temporary registers. The same code can be used four times to transpose an 8×8 block of 16-bit subwords in 32 instructions [11]. This can execute in 32 cycles on a single-issue PLX-64 processor, or 16 cycles on a 2-way superscalar PLX-64 processor, or 8 cycles on a 4-way superscalar processor. By going up to a wordsize of 128 bits, a whole row of 8 16-bit subwords can be held in a single register. A single-issue PLX-128 processor can achieve this 8×8 matrix transpose in $n \times \log_2(n) = 24$ cycles, where $n = 8$. A 2-way superscalar PLX-128 processor can accomplish it in 12 cycles.

Multiplication by fractional constants is efficiently performed with *parallel shift and add* instructions. The instruction sequences used to perform these multiplications are shown in Code 4 for each of the four fractional constants used in the DCT algorithm. An average of 3.5 instructions are needed per multiplication of a register by these four fractional constants. Therefore in PLX-64, four 16-bit multiplications can be done simultaneously in 3.5 cycles on average. This performance, achieved using the adder with subword parallelism [11, 13, 20], is even better than using one or two 16-bit integer multipliers, where each multiplication takes at least 3 cycles of execution latency.

Overall, 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 with *parallel shift and add* instructions, and fast matrix transposition with *mix* instructions (Table 10). Subword parallelism is equivalent to executing 4 iterations of the one-dimensional DCT loop in parallel (Steps 1 and 3 in Code 3) for PLX-64, and 8 iterations in parallel for PLX-128. Hence, increasing the wordsize to 128 bits from 64 bits provides significant additional speedup. When 128-bit words are used, each row of an 8×8 block can be accommodated in a single register, as compared to the 64-bit wordsize where a single row spans two registers. This causes the performance to scale up almost linearly with increasing wordsizes. Table 9 shows that PLX-64 provides a 4.6× speedup over a 64-bit RISC processor without subword parallelism, and PLX-128 almost doubles that to a 9.1×

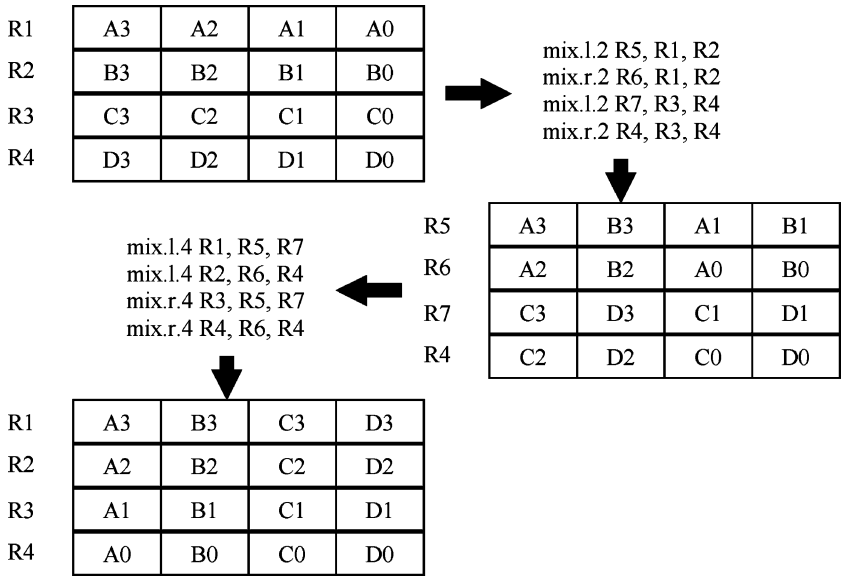

Figure 16. 4 × 4 matrix transposition using *mix* instructions.

Code 4. Parallel multiplication of the subwords of R1 by four fractional constants. R2 is the final destination register.

1st coefficient = $0.70711 = 0.10110101_2$

```
pshiftadd.2.r  R2, R1, R1     # R2 = 1.01₂           × R1
pshiftadd.3.r  R2, R2, R2     # R2 = 1.01101₂        × R1
pshifti.r.2    R3, R1, 8      # R3 = 0.00000001₂     × R1
pshiftadd.1.r  R2, R2, R3     # R2 = 0.10110101₂     × R1
```

2nd coefficient = $0.54120 = 0.10001010_2$

```
pshiftadd.2.r  R2, R1, R1     # R2 = 1.01₂           × R1
pshifti.r.2    R2, R2, 5      # R2 = 0.0000101₂      × R1
pshiftadd.1.r  R2, R1, R2     # R2 = 0.1000101₂      × R1
```

3rd coefficient = $1.30658 = 1.01001110_2$

```
pshiftadd.1.r  R2, R1, R1     # R2 = 1.1₂            × R1
pshiftadd.1.r  R2, R2, R1     # R2 = 1.11₂           × R1
pshiftadd.3.r  R2, R2, R1     # R2 = 1.00111₂        × R1
pshiftadd.2.r  R2, R2, R1     # R2 = 1.01001110₂     × R1
```

4th coefficient = $0.38268 = 0.11000010_2$

```
pshifti.r.2    R2, R1, 7      # R2 = 0.00000010₂     × R1
pshiftadd.1.r  R3, R1, R1     # R3 = 1.1₂            × R1
pshiftadd.1.r  R2, R3, R2     # R2 = 0.11000010₂     × R1
```

speedup. The speedup achieved is greater than the maximum speedup expected due to the degree of subword parallelism alone, which is 4× and 8× in PLX-64 and PLX-128 respectively. This is because PLX-64 (PLX-128) can load four (eight) pixels from the memory with a single load instruction and immediately begin to process these with subsequent subword-parallel instructions. This is possible if the pixels are stored in memory in the same way they are stored in a packed register. In contrast, a non-subword parallel ISA incurs many overhead instructions after the load to first isolate and right-align individual pixels before they can be processed. This overhead is incurred once again at the end of the computations, when the individual pixels spread across multiple registers are packed into a single register before being stored back to the memory.

4.2. Pixel Padding

The MPEG-4 [24] video compression standard differs from previous compression standards like MPEG-1 and MPEG-2 in the use of structures called video object planes (VOPs) rather than video frames. Unlike a video frame, which is always rectangular, a VOP can have any arbitrary shape. When a VOP with an irregular shape is fit into a rectangular block, there will be unused pixels. The adding of new pixels to fill out a rectangular block is called pixel padding [25].

Pixel padding on an 8×8 block is performed in four stages: (1) padding of the rows; (2) first transposition of the block; (3) padding of the rows of the transposed block, which corresponds to padding of the columns of the original block; (4) a final transposition.

Padding of rows is a simple operation that is performed efficiently by logical instructions. Transposition of blocks is performed by *mix* instructions. A special case in the algorithm requires averaging of pixel values, which is performed using the *parallel average* instruction.

PLX-64 provides a 7.9× speedup over a basic 64-bit RISC processor. Unlike the DCT example, PLX-128 offers no additional speedup over PLX-64 since pixel padding is performed on irregularly spaced non-consecutive 8×8 blocks, and therefore the extra parallelism offered by 128-bit registers remains unutilized.

4.3. Clip Test in Three-Dimensional Graphics Processing

In three-dimensional graphics processing, primitive objects, which are generally triangles, need to be clipped before they are rendered [26]. A triangle consists of three vertices (**v1**, **v2**, **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 $-\mathbf{w} \leq \mathbf{x} \leq \mathbf{w}$, $-\mathbf{w} \leq \mathbf{y} \leq \mathbf{w}$, $-\mathbf{w} \leq \mathbf{z} \leq \mathbf{w}$. 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 each of its vertices is within its bounds; and it is completely outside if all its vertices are outside the same plane of their bounding volume. Otherwise the triangle intersects its bounding volume and needs to be clipped. This test can be written as shown in Code 5.

The first set of conditions in Code 5 evaluates whether a triangle is completely outside its bounding volume. If so, the triangle is discarded. Otherwise, we evaluate the second set of conditions, which tests whether the triangle is completely inside. These nested if-then-else statements can be accelerated greatly by using predication. 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 *compare parallel write 1* instructions. This is possible because multiple *compare parallel write 1* instructions can target the same destination predicate register simultaneously. Therefore, the speedups will be even more pronounced for superscalar PLX implementations than for the single-issue implementation upon which our results in Table 9 are based. Dependence of this algorithm

Code 5. Pseudocode for the clip test for a single triangle.

```
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    ))
    {
          accept_triangle();   /* because the triangle is completely inside  */
    }
    else clip_triangle();      /* because the triangle is neither completely */
                               /* inside, nor completely outside             */
}
else discard_triangle();       /* because the triangle is completely outside */
```

Table 11. Speedup of superscalar PLX-64.

	Single-issue PLX-64 (normalized to 1.0)	2-way PLX-64 (with 2 memory ports)	4-way PLX-64 (with 2 memory ports)	Single-issue PLX-128
Clip test	1.0	1.7	2.7	1.0
Median filter	1.0	1.7	2.4	2.0

on predication in its PLX implementations 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.

Table 11 shows the additional speedup that can be obtained in 2-way and 4-way superscalar PLX-64 processors as compared to a single-issue PLX-64 processor. For the clip test, these speedups are 1.7× and 2.7× respectively. This is due mainly to the *compare parallel write 1* instructions. Even in highly serial code like the clip test, additional speedup is achieved with 2-way and 4-way superscalar implementations (with 2 or 4 ALUs) because PLX allows different comparisons targeting the same predicate register to execute in the same cycle. Doubling the wordsize to 128 bits does not help in clip test because of the serial nature of the testing process.

4.4. Median Filter

A median filter is used for noise reduction in image processing [27]. A pixel at the center of a $k \times k$ box is replaced with the median value of the $k \times k$ pixels enclosed in the box. If the value of a center pixel was significantly above or below the value of its neighbors, perhaps because it was distorted by noise, it would be eliminated in this process. Typically k is a small odd integer, giving 3×3, 5×5, or 7×7 median filters.

In this example, we describe a 3×3 median filter. The pseudocode is given in Code 6. Step 2 of the code, where the median of the nine pixels is computed, is the most computationally intense of the four. One simple way to perform this is to sort the nine pixels, and then take the value in the center. However, this approach involves more work than what is required to find the median, which can be done without sorting the pixels completely. It is possible to swap the nine pixels in pairs in order to bring the median pixel to the center, with all the pixels above the median to one side of it, and all the pixels below the median to the other side. The pixels above and below the median are not sorted among themselves, which saves significant computation time as compared to fully sorting the pixels. We first define the *pixel swap* operation, which conditionally swaps two pixels, Px and Py, so that Px is always less than or equal to Py after the operation.

```
# define pixel-swap (Px, Py)
```

$$\{\text{if } (Px > Py)\ Px \leftrightarrow Py;\}$$

The sequences of the *pixel swap* operation used to find the median of 3, 5, and 9 pixels are shown in Fig. 17 [28]. The vertical bars in the figure correspond to the pixel swap operation, where the top and bottom pixels correspond to Px and Py respectively in the definition above. For 9 pixels, 19 *pixel swap* operations are sufficient to complete Step 2 in the median filter. The pseudocode corresponding to this case is shown in Code 7.

In PLX multiple *pixel swap* operations can be completed in only two instructions. For example, if pixels P1 and P2 are initially in the lowest order bytes of the 64-bit registers R1 and R2 respectively, the following two instructions will perform the swap, and write the results to R3 and R4. The lowest order byte of R3 will then 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;
```

Since each register contains eight pixels, eight pixel pairs are swapped (sorted) simultaneously. In

Code 6. Pseudocode for the median filter.

1. Place the 3×3 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.

Code 7. Finding the median of nine pixels using the **pixel swap** operation.

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

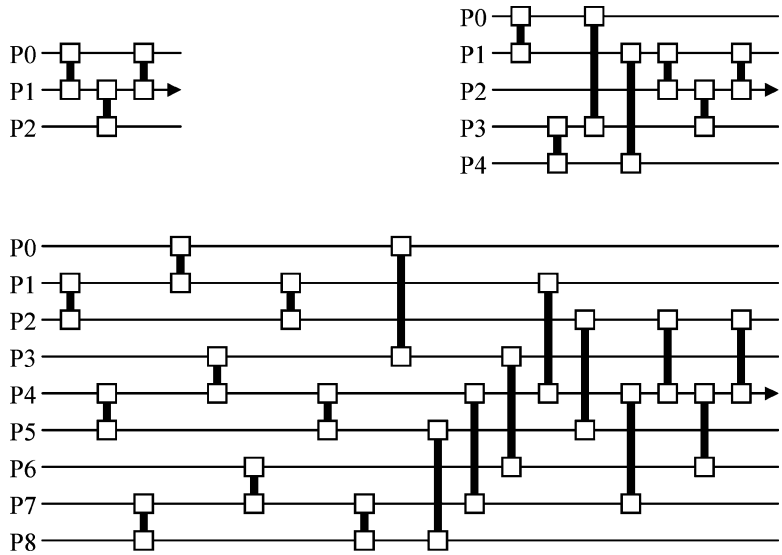

Figure 17. Finding the median of 3, 5, and 9 pixels using the pixel swap operation.

PLX-128, 16 pixel pairs can be similarly sorted in parallel. In IA-32 with MMX+SSE, an additional move instruction is needed to save one register before each *parallel minimum* and *parallel maximum* because these instructions overwrite one source register.

Since the *pixel swap* operation can fully utilize subword parallelism, the speedups in Table 9 are proportional to the number of subwords in one register (i.e. 8× for PLX-64 and 16× for PLX-128).

Table 11 shows the speedups obtained with superscalar PLX-64 implementations and single-issue PLX-128. In the median filter, pairs of *parallel minimum* and *parallel maximum* instructions in the same *pixel swap* operation have no data dependencies. Hence, they can be executed in parallel in superscalar processors. In a 2-way superscalar PLX-64 processor, an additional 1.7× speedup is obtained. A less significant speedup of 2.4× is obtained in a 4-way superscalar processor because while one pair of *parallel maximum* and *parallel minimum* instructions can be executed in a single cycle, two pairs usually cannot be executed together due to the data dependencies between consecutive *pixel swap* operations. For this example, we see that doubling the subword parallelism with wordsize scalability from PLX-64 to PLX-128 provided better performance at a lower cost than doubling the instruction parallelism from single-issue PLX-64 to 2-way superscalar PLX-64.

For all algorithms, Table 9 shows an increasing speedup as we move from the basic 64-bit RISC to IA-32 with MMX and SSE, to PLX-64. The only exception is that IA-32 is slower than a basic 64-bit RISC for the clip test because it only has 8 registers as compared to 32 registers, and requires additional memory accesses. PLX-64 is consistently faster than IA-32 with MMX+SSE. For algorithms like the DCT and median filter, where the data parallelism of adjacent pixels can be further exploited with a wider functional unit, PLX-128 provides an additional 2× speedup over PLX-64. This is made possible by the wordsize scalability feature.

In addition, Table 11 shows that sometimes doubling the instruction parallelism to 2-way superscalar

PLX-64 provides better performance improvement than doubling the wordsize with PLX-128, but other times, the reverse is true. Both methods have the same degree of operation parallelism per cycle. For the median filter, we had to unroll the loop and essentially use the superscalar processor as a wider subword-parallel processor, in order to get the speedups shown. However, register pressure builds up especially in the 4-way superscalar processor, preventing further speedup improvements.

5. PLX Architecture Testbed

In addition to designing the architecture, the PLX project [18] also involves developing a testbed for research and education in instruction set architecture, microarchitecture, arithmetic, VLSI chip design, compiler optimizations, multimedia algorithm optimizations, subword-parallel programming techniques, and performance evaluation. Currently, we have developed the following basic software tools:

- PLX assembler
- PLX simulator
- Workload analysis tools
- PLX compiler
- Media loops library

The PLX assembler can be used to convert code files written in PLX assembly to the corresponding PLX binary. The PLX simulator is then used to simulate the PLX binary generated by the assembler, and to generate timing results for various microarchitectural settings. Workload analysis tools are used with the simulator to generate detailed workload characteristics of the simulated code. The PLX compiler is currently an extended version of the Lcc C compiler [29]. It is used to generate PLX assembly from standard C code, which then can be assembled using the PLX assembler. The media loops library consists of the PLX-optimized versions of key multimedia kernels from image, video, graphics, and audio applications.

We have hierarchical simulators for PLX. A fast instruction-level simulator can be used when accurate timing data is not required, such as in code debugging or instruction frequency analysis. In the latter case, the dynamic instruction count is sufficient, and the accurate cycle count is not essential. A slower cycle-accurate simulator is also available for performance studies needing accurate cycle counts for the execution time. For this, we have used pipeline and microarchitecture models from the SimpleScalar [30] simulator. This allows the simulation of superscalar implementations, in-order or out-of-order execution, and various other microarchitecture features. Each simulator allows the states of the memory, register file, and the predicate registers to be observed at any time during the program execution. The simulator and the workload analysis tools generate several simulation results. These include the total number of instructions executed (the program's pathlength), total cycle counts (execution time), individual instruction counts, distribution of instructions by major instruction classes, percentage of instructions that are predicated true and false, and the percentage of instructions that exhibit subword parallelism.

Since one of the goals of the PLX project is to facilitate architecture research, we designed the assembler, simulator, and compiler to make it easy to add new instructions, and evaluate their performance on existing or new code. For example, the assembler can generate either standard PLX binary code or PLX intermediate code. PLX binary code is a sequence of 32-bit PLX instructions, with the bit encoding of each instruction exactly as defined in the PLX ISA document. PLX intermediate code is a sequence of 8 integer fields for each instruction. If a field is not used by a particular instruction, it is assigned a value of 0. The first field contains an instruction sequence number, which uniquely identifies each instruction and its location in the code. The second field corresponds to the predicate register used by the instruction. The third field is the opcode of the instruction concatenated with any subop fields, which completely determines the instruction's functionality. The remaining fields correspond to the target register, source registers, and target predicate registers. An example of an intermediate code segment, which corresponds to the PLX-64 assembly in Code 2, is given in Code 8.

Simulation of PLX intermediate code is faster than simulation of the binary code. For example in the binary code, the operation to be simulated has to be looked up in a table after assembling the bits in non-adjacent opcode and subop fields in the instruction. But in the intermediate code, the operation is just the third integer, and can be used immediately in the simulator. The intermediate code also simplifies adding new instructions and instruction types to the simulator, since it does not depend on finding and finalizing the exact encodings of these potential new instructions. This in turn speeds up the evaluation of new instructions for ISA research.

Code 8 Intermediate code output of the assembler for Code 2.

```
unsigned int i_code[] =
{                                       /* limit:                                  */
   0, 0, 0x1300, 20, 29, 24, 0, 0,      /* P0 load      R20, R29,  24              */
   1, 0, 0x1300, 21, 29, 16, 0, 0,      /* P0 load      R21, R29,  16              */
   2, 0, 0x1300, 22, 21,  0, 0, 0,      /* P0 load      R22, R21,   0              */
   3, 0, 0x0402, 23,  1,  0, 0, 0,      /* P0 loadi.2   R23,   1                   */
   4, 0, 0x0800,  0,  0,  0, 2, 0,      /* P0 cmp.eq.1   R0,  R0,  P0, P2          */
   5, 0, 0x0802, 22,  0,  1, 3, 0,      /* P0 cmp.lt.w  R22,  R0,  P1, P3          */
   6, 3, 0x0802, 22, 23,  2, 3, 0,      /* P3 cmp.lt.w  R22, R23,  P2, P3          */
   7, 1, 0x0402, 24,  1,  0, 0, 0,      /* P1 loadi.2   R24,   1                   */
   8, 2, 0x3003, 24, 22, 23, 0, 0,      /* P2 padd.w    R24, R22, R23              */
   9, 3, 0x2500, 24, 23,  1, 0, 0,      /* P3 slli      R24, R23,   1              */
  10, 0, 0x1B00, 24, 20,  0, 0, 0,      /* P0 store.w   R24, R20,   0              */
  11, 0, 0x0200, 31,  0,  0, 0, 0,      /* P0 ret       R31                        */
};
```

New instructions can also be added to the compiler. Because it is difficult for a compiler to generate fully-optimized subword-parallel code, this can be aided by the programmer through the use of intrinsics. Every PLX instruction can be invoked within a C program using intrinsics. The subsequent instruction stream can then go through other compiler optimizations, before being simulated.

6. 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 [11–13], we have designed PLX by selecting the most useful multimedia instructions whose implementations are expected to be low in cost and power. We also added new architectural features for even higher performance. The result is a minimalist ISA for high-performance multimedia processing. PLX is also used as a testbed for hands-on teaching and research in ISA and microarchitecture design space exploration and analysis at Princeton University and several other universities. This paper describes PLX version 1.2.

PLX is designed as a concise RISC-like instruction set, where every instruction takes a single cycle to execute. The only exceptions are for the multiplication of two registers, where a pipelined multiplier could have a multi-cycle execution latency. This is an optional functional unit, and may be omitted for cost reasons. It has to be included in environments where such multiplication by variables is frequently needed. Otherwise, PLX supports low-cost multiplication by constants using the ALU with *parallel shift and add* instructions, achieving very good performance due to subword parallelism. In our multimedia kernels, including many not presented in this paper, multiplication by constants is needed much more frequently than multiplication of two registers. 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 average* and the parallel sorting achieved with the *parallel maximum* and *parallel minimum* instructions. New features in PLX, like efficient predication and wordsize scalability, have also proved useful. The graphics clip test example showed the benefits of predication and the *compare parallel write 1* instruction. We have demonstrated how the wordsize can be scaled up from 64 to 128 bits for performance reasons (and scaled down from 64 bits to 32 bits for cost and power reasons). For certain application environments, scaling up to wordsizes of 256 bits or larger may be beneficial.

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, wordsize 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.

We have also described the various components of the PLX testbed. One of the design goals for the PLX testbed is to provide a complete toolkit for research and education in instruction set architecture, microarchitecture, computer arithmetic, VLSI design, low-power design, compiler optimizations, and multimedia program optimizations. We designed the instruction-level simulator to make it easy to add new instructions, and

evaluate new ISA features. We also have an assembler, cycle-accurate simulators, and performance tools that allow adding or deleting new ISA features.

Future work on the PLX testbed may include the development of lower-level simulators that can simulate the hardware implementations of PLX functional units at the logic gate, standard cell, or custom circuit design levels. In addition, we are interested in collaborations to generate PLX hardware implementations for different microarchitectures. Also, much work can be done to optimize the PLX C compiler, which currently generates basic code without optimizations for many of the PLX architectural features.

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 exploring the ideal architectural structures for the memory access patterns common in processing video and audio streams. Future work also includes creating accurate models of latency, area, and power for architectural tradeoff studies of different PLX features and new ISA proposals. We will also use the PLX testbed to explore architectural features for fast software cryptography for secure information processing.

Acknowledgments

PLX is a project of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). We thank PALMS colleagues Zhijie Shi and Xiao Yang for the median filter and clip test examples respectively. We also thank the students of Professor Lee's computer architecture classes, ELE-475 and ELE-572, at Princeton University for their contributions to the PLX testbed. A. Murat Fiskiran is a Kodak Fellow.

References

1. R.B. Lee and M.D. Smith, "Media Processing, a New Design Target," *IEEE Micro*, vol. 16, no. 4, 1996, pp. 6–9.
2. R.B. Lee, "Accelerating Multimedia With Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, 1995, pp. 22–32.
3. R.B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, 1996, pp. 51–59.
4. G. Kane, *PA-RISC 2.0 Architecture*, Prentice Hall, 1996.
5. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, 1996, pp. 42–50.
6. Intel, IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference, available at <http://www.intel.com>, 2002.
7. M. Tremblay, J.M. O'Connor, V. Narayanan, and H. Liang, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, 1996, pp. 10–20.
8. Motorola, AltiVec Technology Programming Environments Manual Revision 2.0, available at <http://www.motorola.com>, 2002.
9. Intel, Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference – Revision 2.1, available at <http://www.intel.com>, 2002.
10. R.B. Lee, A.M. Fiskiran, and A. Bubshait, "Multimedia Instructions in IA-64," in *Proc. IEEE Int. Conf. Multimedia and Expo (ICME)*, Aug. 2001, pp. 281–284.
11. R.B. Lee, "Multimedia Extensions For General-Purpose Processors," *IEEE Workshop on Signal Processing Systems—Design and Implementation (SIPS)*, Nov. 1997, pp. 9–23.
12. R.B. Lee and A.M. Fiskiran, "Multimedia Instructions in Microprocessors for Native Signal Processing," in *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, Yu Hen Hu (ed.), Marcel Dekker, 2002, pp. 91–145.
13. R.B. Lee, "Instruction Set Architecture for Multimedia Signal Processing," in *Computer Engineering Handbook*, Vojin Oklobdzija (ed.), CRC Press, Jan. 2002, pp. 39–1 to 39–38.
14. I. Elsen, F. Hartung, U. Horn, M. Kampmann, and L. Peters, "Streaming Technology in 3G Mobile Communication Systems," *IEEE Computer*, vol. 34, no. 9, 2001, pp. 46–52.
15. C. Basoglu, R. Gove, K. Kojima, and J. O'Donnell, "Single-Chip Processor For Media Applications: The MAP1000," *Int. Journal of Imaging Systems and Technology*, vol. 10, no. 1, 1999, pp. 96–106.
16. S. Rathnam and G. Slavenburg, "Processing the New World of Interactive Media," *IEEE Signal Processing Magazine*, vol. 15, no. 2, 1998, pp. 108–117.
17. R.B. Lee, A.M. Fiskiran, Z. Shi, and X. Yang, "Refining Instruction Set Architecture for High-Performance Multimedia Processing in Constrained Environments," in *Proc. Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, July. 2002, pp. 253–264.
18. R.B. Lee, et al., PLX Project at Princeton University, <http://palms.ee.princeton.edu/plx>.
19. R.B. Lee, "Efficiency of MicroSIMD Architectures and Index-Mapped Data for Media Processors," in *Proc. Media Processors IS&T/SPIE Symp. Electric Imaging: Science and Technology*, Jan. 1999, pp. 34–46.
20. Z. Luo and R.B. Lee, "Cost-Effective Multiplication with Enhanced Adders for Multimedia Applications," in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, vol. 1, 2000, pp. 651–654.
21. R.B. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures," in *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, July. 2000, pp. 3–14.
22. Y. Arai, T. Agui, and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *Trans. IEICE*, vol. E71, no. 11, 1988, pp. 1095–1097.
23. V. Bhaskaran, K. Konstantinides, R.B. Lee, and J.P. Beck, "Algorithmic and Architectural Enhancements for Real-Time MPEG-1 Decoding on a General Purpose RISC Workstation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 5, 1995, pp. 380–386.
24. IEC 14496-2, "Coding of Audio-Visual Objects: Visual, Final Draft International Standard ISO/IEC JTCI/SC29/WG11 N2502," Oct. 1998.

25. E.A. Edirisinghe, J. Jiang, and C. Grecos, "Object Boundary Padding Technique for Improving MPEG-4 Compression Efficiency," *IEEE Electronics Letters*, vol. 35, no. 17, 1999, pp. 1453–1455.
26. Y. Liang and B.A. Barsky, "An Analysis and Algorithm for Polygon Clipping," *Communications of the ACM*, vol. 26, no. 11, 1983, pp. 868–877.
27. J.C. Russ, *The Image Processing Handbook*, CRC Press, 2002.
28. P. Kolte, R. Smith, and W. Su, "A Fast Median Filter Using AltiVec," in *Proc. Int. Conf. Computer Design (ICCD)*, Oct. 1999, pp. 384–391.
29. C.W. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995.
30. D. Burger and T.M. Austin, "The SimpleScalar Tool Set Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June. 1997.

Ruby B. Lee is the Forrest G. Hamrick Professor of Engineering and Professor of Electrical Engineering at Princeton University, with an affiliated appointment in the Computer Science department. She is the founder and director of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Her current research is in rethinking computer architecture for high-performance but low-cost security and multimedia processing. Prior to joining the Princeton faculty in 1998, Dr. Lee served as chief architect at Hewlett-Packard, responsible at different times for processor architecture, multimedia architecture, and security architecture for e-commerce and extended enterprises. She was a key architect in the initial definition and the evolution of the PA-RISC processor architecture used in HP servers and workstations. As chief architect for HP's multimedia architecture team, Dr. Lee led an inter-disciplinary team focused on architecture to facilitate pervasive multimedia information processing using general-purpose computers. She introduced innovative multimedia instruction set architecture (MAX and MAX-2) in microprocessors, resulting in the industry's first real-time, high-fidelity MPEG video and audio player implemented in software on low-end desktop computers. Dr. Lee also co-led an HP-Intel multimedia architecture team for IA-64, released in Intel's Itanium microprocessors. Concurrent with full-time employment at HP, Dr. Lee also served as Consulting Professor of Electrical Engineering at Stanford University. Dr. Lee has a Ph.D. in Electrical Engineering and a M.S. in Computer Science, both from Stanford University, and an A.B. from Cornell University, where she was a College Scholar. She is a Fellow of ACM, a Fellow of IEEE, and a member of IS&T, Phi Beta Kappa, and Alpha Lambda Delta. She has been granted 115 U.S. and international patents, with several patent applications pending.
rblee@princeton.edu

A. Murat Fiskiran is a Ph. D. student at the Department of Electrical Engineering at Princeton University. He is a member of the Princeton Architecture Laboratory for Multimedia and Security (PALMS) and a Kodak Fellow. His research interests include computer architecture and computer security.
fiskiran@princeton.edu