Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures

Ruby B. Lee
Princeton University
rblee@ee.princeton.edu

Abstract

   MicroSIMD architectures incorporating subword parallelism are very efficient for application-specific media processors as well as for fast multimedia information processing in general-purpose processors. This paper addresses the unsolved problem of the need to permute the subwords packed in registers for maximum parallelism performance, especially for two-dimensional (2-D) multimedia algorithms. We propose a new systematic approach for identifying the fundamental data rearrangement needs in current and future 2-D pixel processing programs based on the hierarchical decomposition of frames and objects into atomic 2-D structures. We define new subword permutation instructions, Check, Excheck, Exchange, and Permset, that achieve these data rearrangements across multiple registers. We also define an alphabet of subword permutation primitives, including these new instructions and the Mix instruction defined for PA-RISC MAX-2 and IA-64, which supports the data rearrangement needs of 2-D frames and objects. We show the sufficiency and efficiency of this alphabet for achieving all possible permutations of hierarchical 2-D blocks.

1. Introduction

   Multimedia information processing can be considered an increasing part of the general-purpose workload or a special-purpose application area. In this paper, we consider new instructions for accelerating multimedia processing in any programmable processor, whether general-purpose or application-specific. The focus is on simple, single-cycle instructions, which can be used to construct any type of permutations needed in two-dimensional (2-D) multimedia processing.

   Multimedia extensions have been added to general-purpose processors to accelerate the processing of different media types [1-7,15,16]. The types of application-specific processors we target in this paper are designed to execute various multimedia programs, rather than just one. They include digital signal processors [8], video signal processors [9, 10], and mediaprocessors [11, 12].

   Subword parallelism [1,4] is now widely deployed by multimedia instructions in microprocessor architectures [1-7, 15,16] and in media processors [12] to accelerate the processing of lower-precision data, like 16-bit audio samples or 8-bit pixel components. We also call this *microSIMD architecture* [13], since it applies SIMD (Single Instruction Multiple Data) parallel processor techniques [14] within a single processor. A subword-parallel (or microSIMD) instruction performs the same operation in parallel on multiple pairs of subwords packed into two registers, which are typically 32 to 128 bits wide in today's microprocessors and mediaprocessors (see figure 1). For example, a 64-bit word-oriented datapath can be

partitioned into eight 8-bit subwords, or four 16-bit subwords, or two 32-bit subwords. Substantial performance improvements have been realized using subword parallel instructions, at very low cost compared to other forms of parallelism, like superscalar, VLIW or parallel processor organizations, for the same degree of operation parallelism [13].
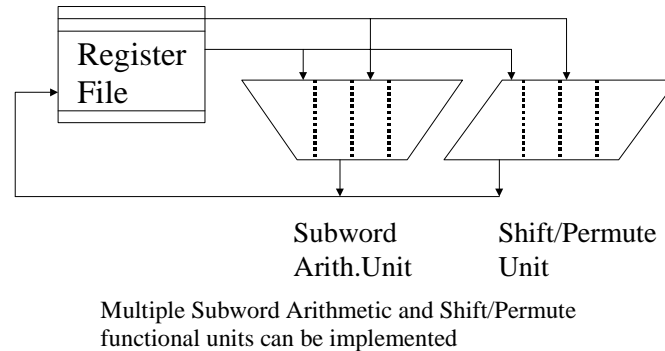

Subword
Arith.Unit

Shift/Permute
Unit

Multiple Subword Arithmetic and Shift/Permute
functional units can be implemented

Figure 1: microSIMD subword parallelism leveraging word datapaths

With packed subwords in registers, we now need to be able to re-arrange subwords within a register, and between registers. This is necessary to achieve the maximum parallelism for subsequent processing. Unfortunately, subword permutation operations are not understood as clearly as subword arithmetic operations. They require moving several fields (subwords) in parallel. Conventional **shift** and **rotate** instructions move all the bits in a register by the same amount. **Extract** and **deposit** instructions, found in instruction-set architectures like PA-RISC [17], move one field using one or two instructions. Early subword permutation instructions like **mix** and **permute** [4] in the PA-RISC MAX-2 multimedia instructions are a first attempt to find efficient and general-purpose subword permutation primitives. However, the sufficiency or efficiency of these permutation primitives in achieving any arbitrary permutation has not been demonstrated. The problem is further complicated by the fact that image, video or graphics processing require mapping two-dimensional objects onto subwords in multiple registers, and then permuting these subwords between registers. In addition, since permutations have not been easily achieved by programmable processors, algorithm designers may not have optimized algorithms using permutations. Hence, one cannot just comb through all the common multimedia algorithms to determine what permutations are used and the performance impact they have: one would often need to re-think algorithms to see if efficient permutations would help improve the performance. Furthermore, in designing subword permutation primitives, we need to project the permutation needs of future, yet-to-be defined multimedia algorithms, and this seems to be an intractable problem. In this paper, we propose a systematic solution to this unsolved problem of finding generic subword permutation primitives for both current and future algorithms for processing two-dimensional multimedia data. We also define a small set of subword permutation primitives, and show that this is both a sufficient and an efficient set.

In section 2, we describe how 2-dimensional frames can be mapped into the packed subwords of microSIMD architectures. We also show that two-dimensional objects can be decomposed into smaller blocks or polygons, and ultimately into atomic 2x2 matrices and triangles. In section 3, we review the subword permutation instructions that have been defined in the multimedia instructions MAX-2 for PA-RISC processors [4] and for IA-64 EPIC processors [15], especially the **mix** instruction. We show an example of how a permutation on a 2-D object can be decomposed into hierarchical permutations on 2x2 matrices. In section 4, we investigate the subword permutation needs of atomic 2-D structures, and postulate that

these are generic primitives since all 2-D frames and objects can be decomposed into these atomic 2-D structures. In section 5, we propose small subsets of subword permutation primitives that are sufficient and also efficient for different performance and cost levels. Section 6 summarizes and concludes the paper.

2. Mapping and decomposition of 2-D blocks

To use microSIMD architectures for maximum performance, we need to map multimedia data into packed subwords in a way that permits maximum parallel execution, SIMD style. Pixel-oriented multimedia data in images, graphics, video or animation, are two-dimensional (2-D) in nature. How should 2-D blocks of data be mapped into the packed subwords of micro-SIMD architectures?

A 2-D array of pixels in memory is normally stored in row-major format: elements of row one are stored sequentially in successive memory locations, followed by elements of row two, and so forth. When words are loaded into registers from memory, this translates into mapping the first row into a set of registers, mapping the second row into another set of registers, and so forth. This is called *area-mapping* [13] of a 2-D block – different rows of the 2-D block are held in different registers.

A 2-D image or frame is easily decomposed into smaller 2-D blocks. The smallest 2-D block is a 2x2 block (or matrix). A 2-D object within a frame can also be decomposed into smaller blocks, where again the smallest 2-D rectangular block is a 2x2 matrix of pixels. For example, an 8x8 matrix used in DCT or IDCT can be decomposed into four 4x4 matrices, each stored in four 64-bit registers, as shown in Figure 2a, where each element is a 16-bit subword. Each such 4x4 matrix can be further decomposed into four 2x2 matrices (Figure 2b). Matrices with dimensions that are a power of two can be successively decomposed into smaller matrices, and ultimately into the smallest 2x2 matrix.

```
(a) Area mapping of a 4x4 matrix:
    R1 =  a00  a01  a02  a03
    R2 =  a10  a11  a12  a13
    R3 =  a20  a21  a22  a23
    R4 =  a30  a31  a32  a33

(b) Decomposition into four 2x2 matrices:
    R1 =  a00  a01  b00  b01
    R2 =  a10  a11  b10  b11
    R3 =  c00  c01  d00  d01
    R4 =  c10  c11  d10  d11
```

Figure 2: Area mapping and decomposition of 2-D blocks

Non-rectangular objects may more accurately be decomposed into non-rectangular polygons, the smallest of which is a triangle. Since all 2-D frames and objects can be decomposed into atomic 2-D units like the 2x2 matrix and the triangle, we postulate that if we can determine the permutation needs of these atomic units, they can serve as permutation primitives for the entire frame or object. At the lowest level, we permute the four pixels of a 2x2 matrix. At the next higher level, we again permute a 2x2 matrix, where each element is now itself a 2x2 matrix.

3. Mix permutation instruction

For microprocessor multimedia instructions, only PA-RISC MAX-2[4], IA-64[15] and the PowerPC Altivec[16] have a few instructions designed for general-purpose subword permutation. Because our focus in this paper is on 2-D multimedia processing, and area-mapped 2-D objects span at least two registers, we seek permutation primitives that reorder subwords from two source registers. We describe **mix**, defined by the author for MAX-2 and IA-64, which is currently the only subword permutation instruction with two source registers.

3.1. Definition of Mix instruction

The **mix** operation selects either all even elements, or all odd elements, from the two source registers [4,15]. The pair of **mixL** and **mixR** operations is defined as follows:

- **mixL**: interleave the corresponding "even" elements from the two source registers, starting from the leftmost elements in each register
- **mixR**: interleave the corresponding "odd" elements from the two source registers, ending with the rightmost elements in each register

Table 1 defines these **mix** instructions, for three different subword sizes: 8 bits, 16 bits and 32 bits. Each letter in the register contents represents an 8-bit subword, and each register holds a total of 64 bits.

Table 1: Definition of Mix instruction

	Register Contents:
	R1 = a b c d e f g h
	R2 = A B C D E F G H
Instruction:	**Definition:**
mixL,8 R1,R2,R3	R3 = a A c C e E g G
mixR,8 R1,R2,R3	R3 = b B d D f F h H
mixL,16 R1,R2,R3	R3 = a b A B e f E F
mixR,16 R1,R2,R3	R3 = c d C D g h G H
mixL,32 R1,R2,R3	R3 = a b c d A B C D
mixR,32 R1,R2,R3	R3 = e f g h E F G H

3.2. Example of decomposable subword permutations.

A common permutation of a 2-D object is matrix transpose, where the matrix is flipped along its diagonal: rows become columns, and columns become rows. This is a decomposable permutation. For example, an 8x8 matrix of 16-bit elements stored in 16 registers can be decomposed into four 4x4 matrices (Figure 2a), each of which can be further decomposed into four 2x2 matrices (Figure 2b). By transposing each of the 2x2 matrices, then transposing the larger 2x2 matrix, where each element is itself one of these 2x2 matrices, we obtain the matrix transpose of a 4x4 matrix (see Figure 3). The **mix** instructions can perform these hierarchical 2x2 matrix transpositions. The **mixL** and **mixR** instructions are used in pairs at the level of a

subword size equal to the matrix element size. Then, they are used at the size of subwords that are twice as large. Repeating this on each of the four 4x4 matrices completes the transpose of the original 8x8 matrix.
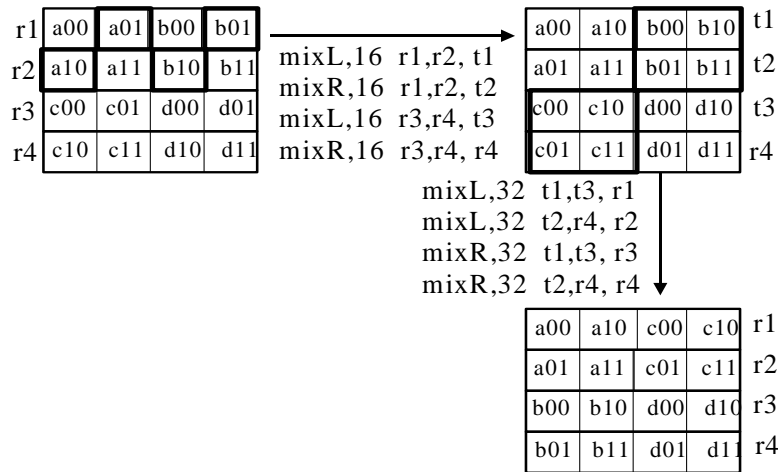

r1	a00	a01	b00	b01
r2	a10	a11	b10	b11
r3	c00	c01	d00	d01
r4	c10	c11	d10	d11

mixL,16 r1,r2, t1
mixR,16 r1,r2, t2
mixL,16 r3,r4, t3
mixR,16 r3,r4, r4

a00	a10	b00	b10	t1
a01	a11	b01	b11	t2
c00	c10	d00	d10	t3
c01	c11	d01	d11	r4

mixL,32 t1,t3, r1
mixL,32 t2,r4, r2
mixR,32 t1,t3, r3
mixR,32 t2,r4, r4

a00	a10	c00	c10	r1
a01	a11	c01	c11	r2
b00	b10	d00	d10	r3
b01	b11	d01	d11	r4

Figure 3: Hierarchical Decomposition of Matrix Transpose Permutation

4. Fundamental data rearrangements in 2-D blocks

We propose that a systematic approach to finding a set of permutation primitives for current and future 2-D multimedia programs can be based on decomposing images and objects into atomic units, then finding the permutations desired for these 2-D building blocks. The subword permutation instructions for these 2-D building blocks are also defined for larger subword sizes at successively higher hierarchical levels. We propose studying the permutations of a 2x2 matrix, and the permutations of the four triangles contained within this 2x2 building block. What are the useful data rearrangements in a 2x2 matrix and its four embedded triangles (section 4.1)? What are permutation primitives that can perform these data rearrangements (section 4.2)? Are these permutation primitives sufficient and efficient (section 4.3)? Can they be generalized (section 4.4)?

4.1. Characterization of 2-D data rearrangements

The first set of data rearrangements likely to be needed in a 2x2 matrix is to be able to swap elements vertically, horizontally and diagonally. This is based on observing that nearest neighbor interactions are perhaps the most common 2-D pixel operations. The eight nearest-neighbor movements for a pixel in a 2-D frame are shown in Figure 4a. Figure 4b expresses the 9-element matrix of Figure 4a as four 2x2 matrices (outlined in bold). Here, an element of a 2x2 matrix can move to its right (or left) neighbor, its downward (or upward) neighbor, or its diagonal right (or left) neighbor. Figure 4c shows all possible nearest neighbor movements, for one or two pairs of elements for a 2x2 matrix.

The four elements of a 2x2 matrix can also be rotated clockwise by 1, 2 or 3 positions (Figure 5a). This is equivalent to rotating counter-clockwise by 3, 2 or 1 position. Also, rotating by 2 positions is equivalent to swapping both the diagonal and anti-diagonal elements,

as already covered in Figure 4c. Hence, we need only consider rotating clockwise or anti-clockwise by 1 position.
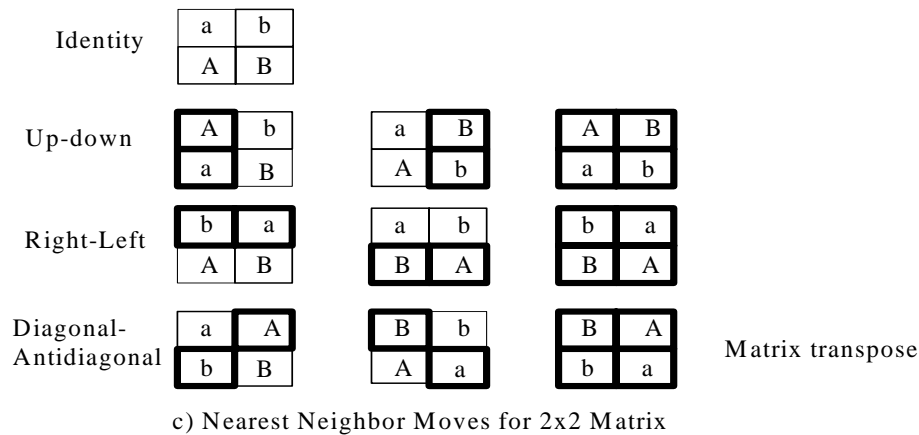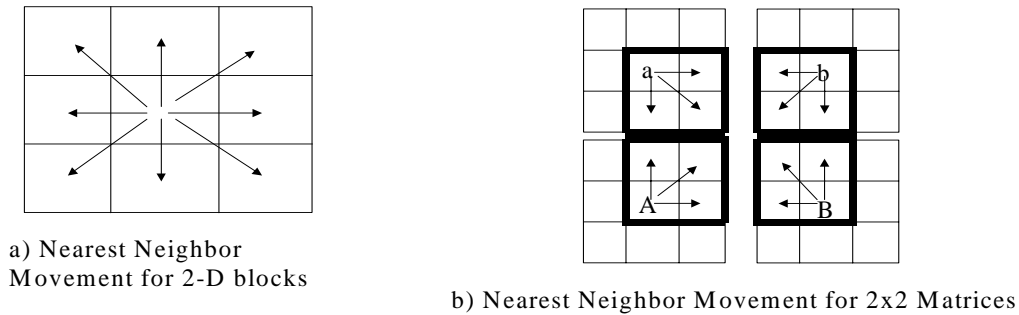
a) Nearest Neighbor
Movement for 2-D blocks

b) Nearest Neighbor Movement for 2x2 Matrices

Identity

Up-down

Right-Left

Diagonal-Antidiagonal

Matrix transpose

c) Nearest Neighbor Moves for 2x2 Matrix

Figure 4: Nearest Neighbor Permutations

clockwise

Anti-clockwise

Rotate by 2 elements
= swap diagonal and
antidiagonal elements

a) Rotation of 2x2 Matrix

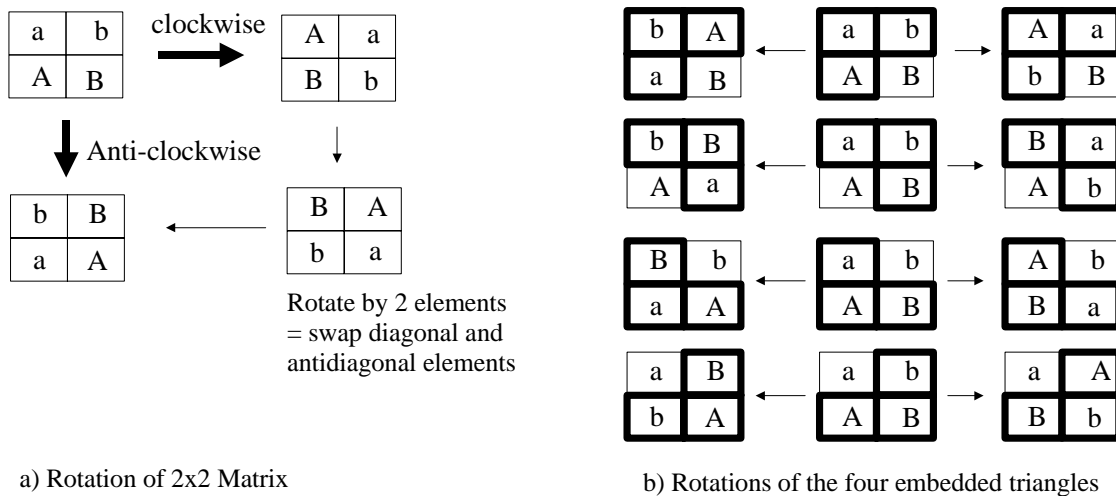b) Rotations of the four embedded triangles

Figure 5: Rotations of a 2x2 matrix and its embedded triangles

A 2x2 matrix contains four triangles, each of which can be rotated clockwise or anti-clockwise by 1 position. This results in 8 different permutations of the 2x2 matrix, as shown in Figure 5b. Triangles are useful for representing non-rectangular shapes.

We postulate that these permutations of 2x2 matrices and triangles should be efficiently supported, at all subword sizes (powers of 2), for use in decomposable permutations of 2-D objects. What subword permutation instructions can achieve these common data rearrangements?

4.2. Check, Exchange and Excheck operations

To achieve these common data rearrangements, we only need to define three new subword permutation primitives (see Table 2). The **check** instruction allows the downward and upward swapping of elements, the **exchange** instruction allows the right and left movement, while the **excheck** instruction allows the rotation of a triangle of three elements within a 2x2 matrix. The **mixL** and **mixR** operations, defined earlier, achieve the swapping of diagonal elements.

The **check** instruction performs a checkerboard pattern: it selects alternately from the corresponding subwords in the two source registers, for each position in the result register (see Table 2). **Exchange** is an operation on a single source register: it swaps adjacent subwords in each pair of consecutive subwords. **Excheck** can be described as a composite operation: it performs a **check** on the two source registers, followed by an **exchange** operation on the result.

Table 2: Definition of Check, Exchange and Excheck

	Register Contents:
	R1 = a b c d e f g h
	R2 = A B C D E F G H
Instruction:	**Definition:**
check,8 R1,R2,R3	R3 = a B c D e F g H
check,16 R1,R2,R3	R3 = a b C D e f G H
check,32 R1,R2,R3	R3 = a b c d E F G H
exchange,8 R1,R3	R3 = b a d c f e h g
exchange,16 R1,R3	R3 = c d a b g h e f
exchange,32 R1,R3	R3 = e f g h a b c d
excheck,8 R1,R2,R3	R3 = B a D c F e H g
excheck,16 R1,R2,R3	R3 = C D a b G H e f
excheck,32 R1,R2,R3	R3 = E F G H a b c d

4.3. Sufficiency and efficiency of permutation primitives

In Table 3, we systematically enumerate the permutations of area-mapped 2x2 matrices, to verify that the subword permutation instructions defined above can indeed perform all these permutations efficiently. R1 and R2 contain four 2x2 matrices. It is easier to follow just the leftmost matrix (in bold), which is labeled as in figures 4-6, initially "a b" in R1 and "A B" in R2. The permutations are enumerated as follows: each of the 4 elements in a resulting 2x2 matrix can be in the top left corner in R3. Thereafter, each of the 3 remaining elements can be in the top right corner in R3. This gives 12 possibilities for the top row, which is used for the numeric numbering of the cases. The two remaining elements of each 2x2 matrix are in the bottom row in R4, and their two possible orderings give the (a) and (b) numbering in Table 3.

Table 3: All Permutations of Four Area-Mapped 2x2 Matrices

Operand registers:	R1 = **a b** c d e f g h R2 = **A B** C D E F G H		
	Result Registers:	**Instructions Used:**	**Type of Data Movement:**
1(a) a at top left	R3 = **a b** c d e f g h R4 = **A B** C D E F G H	;R3=R1 ;R4=R2	identity permutation
1(b)	R3 = **a b** c d e f g h R4 = **B A** D C F E H G	;R3=R1 ;R4=exchange(R2)	swap bottom row elements right-left
2(a)	R3 = **a B** c D e F g H R4 = **A b** C d E f G h	;R3=check(R1,R2) ;R4=check(R2,R1)	swap right column elements up-down
2(b)	R3 = **a B** c D e F g H R4 = **b A** d C f E h G	;R3=check(R1,R2) ;R4=excheck(R2,R1)	rotate bottom-right triangle anti-clockwise
3(a)	R3 = **a A** c C e E g G R4 = **b B** d D f F h H	;R3=mixL(R1,R2) ;R4=mixR(R1,R2)	swap diagonal elements = transpose
3(b)	R3 = **a A** c C e E g G R4 = **B b** D d F f H h	;R3=mixL(R1,R2) ;R4=mixR(R2,R1)	rotate bottom-right triangle clockwise
4(a) b at top left	R3 = **b a** d c f e h g R4 = **A B** C D E F G H	;R3=exchange(R1) ;R4=R2	swap top row elements right-left
4(b)	R3 = **b a** d c f e h g R4 = **B A** D C F E H G	;R3=exchange(R1) ;R4=exchange(R2)	swap both rows' elements right-left
5(a)	R3 = **b B** d D f F h H R4 = **A a** C c E e G g	;R3=mixR(R1,R2) ;R4=mixL(R2,R1)	rotate top-right triangle anti-clockwise
5(b)	R3 = **b B** d D f F h H R4 = **a A** c C e E g G	;R3=mixR(R1,R2) ;R4=mixL(R1,R2)	rotate anti-clockwise 1 element
6(a)	R3 = **b A** d C f E h G R4 = **a B** c D e F g H	;R3=excheck(R2,R1) ;R4=check(R1,R2)	rotate top-left triangle anti-clockwise
6(b)	R3 = **b A** d C f E h G R4 = **B a** D c F e H g	;R3=excheck(R2,R1) ;R4=excheck(R1,R2)	
7(a) A at top left	R3 = **A a** C c E e G g R4 = **b B** d D f F h H	;R3=mixL(R2,R1) ;R4=mixR(R1,R2)	rotate top-left triangle clockwise
7(b)	R3 = **A a** C c E e G g R4 = **B b** D d F f H h	;R3=mixL(R2,R1) ;R4=mixR(R2,R1)	rotate clockwise 1 element
8(a)	R3 = **A b** C d E f G h R4 = **a B** c D e F g H	;R3=check(R2,R1) ;R4=check(R1,R2)	swap left column elements up-down
8(b)	R3 = **A b** C d E f G h R4 = **B a** D c F e H g	;R3=check(R2,R1) ;R4=excheck(R1,R2)	rotate bottom-left triangle clockwise
9(a)	R3 = **A B** C D E F G H R4 = **a b** c d e f g h	;R3=R2 ;R4=R1	swap left and right column elements up-down
9(b)	R3 = **A B** C D E F G H R4 = **b a** d c f e h g	;R3=R2 ;R4=exchange(R1)	
10(a) B at top left	R3 = **B a** D c F e H g R4 = **A b** C d E f G h	;R3=excheck(R1,R2) ;R4=check(R2,R1)	rotate top-right triangle clockwise
10(b)	R3 = **B a** D c F e H g R4 = **b A** d C f E h G	;R3=excheck(R1,R2) ;R4=excheck(R2,R1)	
11(a)	R3 = **B b** D d F f H h R4 = **a A** c C e E g G	;R3=mixR(R2,R1) ;R4=mixL(R1,R2)	rotate bottom-left triangle anti-clockwise
11(b)	R3 = **B b** D d F f H h R4 = **A a** C c E e G g	;R3=mixR(R2,R1) ;R4=mixL(R2,R1)	swap anti-diagonal elements
12(a)	R3 = **B A** D C F E H G R4 = **a b** c d e f g h	;R3=exchange(R2) ;R4=R1	
12(b)	R3 = **B A** D C F E H G R4 = **b a** d c f e h g	;R3=exchange(R2) ;R4=exchange(R1)	swap diagonal and anti-diagonal elements =rotate clockwise by 2

The subword permutation instructions used to achieve each of the 2x2 block permutations are shown. Only the 5 subword permutation primitives defined earlier are needed: **mixL**, **mixR**, **exchange**, **check**, and **excheck**. If the processor has at least two permutation units, then

each case in Table 3 can be executed in one cycle, since there are no dependencies in generating R3 and R4. This establishes the efficiency of these permutation primitives.

Each 2x2 matrix permutation is also labeled with one of the 20 data movements (including identity) described in Figures 4c, 5a and 5b. There are four permutations in Table 3 that are not labeled with a data movement described earlier. They correspond to more esoteric data rearrangements of a 2x2 matrix, described best as changing rows into diagonals, and changing diagonals into columns (Figure 6). Even though these four permutations were not initially identified as data rearrangements to be supported, the permutation primitives we defined efficiently support them. This supports the thesis that if we can define permutation primitives that somehow form a basis set, they can be used to implement other permutations that may be needed in algorithms yet to be invented.
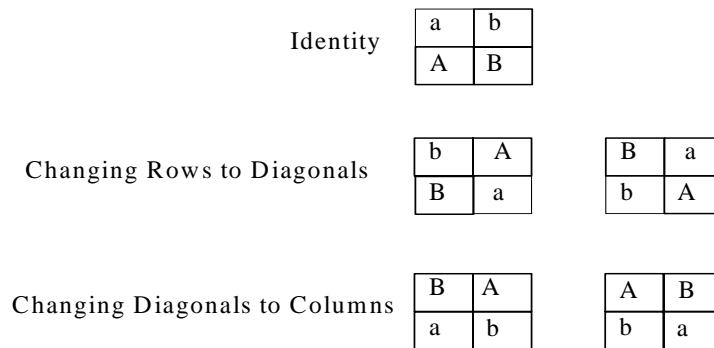
Identity

a	b
A	B

Changing Rows to Diagonals

b	A
B	a

B	a
b	A

Changing Diagonals to Columns

B	A
a	b

A	B
b	a

Figure 6: Four unlabeled permutations of a 2x2 matrix

4.4. Repeating permutations on smaller subsets of subwords

The **exchange** instruction can be replaced by a more general **permset** instruction, which repeats a permutation on a subset of elements over the rest of the elements in the register. **Permset** is also a generalization of the **permute** instruction in MAX-2 [4]. The subwords in the source register are numbered, and **permute** specifies the new ordering desired in terms of this numbering. The **mux** instruction in IA-64 [15] and the **vperm** instruction in Altivec [16] are similar. Table 4 gives examples of this **permute** instruction on 8-bit and 16-bit subwords.

Table 4: Examples of Permute Instruction on 8-bit and 16-bit Subwords

Operand register:	R1 = a b c d e f g h	
permute Instruction	**Result register contents**	**Type of Permutation**
permute,8,01234567 R1, Rt	Rt = a b c d e f g h	identity permutation
permute,8,10325476 R1, Rt	Rt = b a d c f e h g	exchange
permute,8,66666666 R1, Rt	Rt = g g g g g g g g	broadcast
permute,8,76543210 R1, Rt	Rt = h g f e d c b a	reverse
permute,8,05276341 R1, Rt	Rt = a f c h g d e b	arbitrary permutation
permute,8,55000366 R1, Rt	Rt = f f a a a d g g	permutation with repetitions
permute,16,0213 R1, Rt	Rt = a b e f c d g h	permuting four 16-bit subwords

There is a limit to the efficiency of the **permute** instruction for permuting many subwords, since the control bits quickly exceed the number of bits permuted. Permuting four subwords requires only 8 control bits, which can be encoded in the **permute** instruction itself [4, 15]. Beyond four elements and up to sixteen elements, any arbitrary permutation can still be

performed with one instruction, by providing the control bits for the permutation in a second source register [16], rather than in the 32-bit instruction. Permuting 32 elements requires 160 bits, and permuting 64 elements requires 384 bits (n*log n bits). Hence, permuting more than 16 elements cannot be achieved by a single instruction with two source registers, using this method of specifying permutations.

To permute more subwords without increasing the number of control bits required, we define a new **permset** instruction which permutes a subset of m subwords, where m is less than the number of subwords in the register. The same permutation is repeated on consecutive subsets of m subwords. If the total number of subwords in the register is not a multiple of m, we can pad this last set of subwords with zeros.

Table 5: Replacing 8-element Permute with 4-element Permset instructions

Permute example	Equivalent Permset instructions	Type of permutation
permute,8,01234567 R1, Rt	permset, 8,4,0123 R1, Rt	identity
permute,8,10325476 R1, Rt	permset, 8,4,1032 R1, Rt	**exchange**
permute,8,66666666 R1, Rt	permset, 8,4,2222 R1, Rt permset,16,4,2222 Rt, Rt	broadcast
permute,8,76543210 R1, Rt	permset, 8,4,3210 R1, Rt permset,16,4,2301 Rt, Rt	reverse

A **permute** instruction can be turned into a **permset** instruction, by inserting a new parameter which specifies the number of elements to be permuted in each set. In Table 5, this can be a second parameter, inserted between the two existing parameters of subword size and permutation control bits. Using this new **permset** instruction, the first four permutations in Table 4 can also be specified as permutations on sets of 4 elements, as shown in Table 5. The identity and **exchange** operations can be replaced by exactly one such **permset** instruction. The broadcast and reverse operations each need two **permset** instructions, with 4-element permute sets. The next two **permute** instructions in Table 4 cannot be accomplished in 1 or 2 instructions, because of the lack of symmetry in the permutation done on consecutive sets of 4 elements. So, while the **permset** instruction with 4-element sets is not as general as the full **permute** instruction on 8 elements, it can specify all possible permutations of 2x2 matrices, with lower implementation cost.

5. Alphabet of Subword Permutation Primitives

An *alphabet* is a small set of basic primitives from which words, phrases, sentences, paragraphs and stories can be built. Many of these stories and words were not even conceived when the alphabet was designed. We propose an *alphabet of fundamental permutation primitives*, which are simple yet powerful enough to express all data rearrangement needs of current and future 2-D media processing programs.

The **mix** operations appear to be truly fundamental, selecting fairly between elements across the width of both source registers, embodying the powerful even-odd paradigm. Although the **check** instruction can be derived from the **mix** operation, it can also be considered a fundamental permutation since it embodies the checkerboard pattern. The **exchange** operation, while a useful permutation primitive in itself, can be replaced by the more general **permset** instruction, as described above.

An initial alphabet of subword permutation primitives is shown in Figure 7, including **mixL**, **mixR**, **permset**, **check** and **excheck**, defined on 8, 16 and 32 bit subwords. For very low cost

implementations, at slightly reduced performance, a minimal alphabet could exclude **check** and **excheck**. **Check** may be excluded from a minimal set, because a **Shift_Left** of the second operand, followed by a **mixL** instruction can accomplish it. **Excheck** is the composition of **check** followed by **exchange**, so it may also be omitted from a minimal set of fundamental permutations. They are included in the initial alphabet for efficiency and uniformity in performance, so that every permutation of a basic 2x2 matrix, as enumerated in Table 3, can be done in a single cycle (or a single step).

Minimal Alphabet:
mixL, **mixR** on 8, 16 and 32 bit subwords
permset on 8, 16 and 32 bit subwords, with 4-element sets

Additional Primitives:
check on 8, 16 and 32-bit subwords
excheck on 8, 16 and 32-bit subwords

Figure 7: Alphabet A of Subword Permutation Primitives

The minimal set of **mixL**, **mixR** and **permset** may be further reduced depending on the size of the registers in the processor. For example, if registers are only 64 bits wide, then permutation instructions for the two 32-bit subwords may not be needed, since they can easily be specified as permutations on the four 16-bit subwords. These permutation instructions may also be extended down to subwords of 4 bits, 2 bits and 1 bit, especially if it is also desired to support permutations for cryptography efficiently.

6. Summary

MicroSIMD architecture incorporating subword parallelism is very efficient for application-specific media processors, as well as for fast multimedia information processing in general-purpose microprocessors. This is because, in the large majority of cases, microSIMD architectures can exploit the data-parallelism present in multimedia programs as efficiently as other more expensive parallel architectures. The reduced complexity in register ports and register bypassing in microSIMD architectures results in faster cycle times, less area and less design complexity for the same degree of parallelism as other parallel architectures like VLIW, superscalar, or conventional SIMD or MIMD parallel processor architectures [13].

We pose the problem of finding a small set of fundamental subword permutation operations that can be used efficiently for current and future two-dimensional multimedia programs. Such a subword permutation instruction rearranges data between subword tracks in microSIMD architectures, performing a function like that of interconnection networks which move data between parallel processors in conventional SIMD or MIMD parallel processor architectures. While this initially appears to be an intractable problem, this paper describes a novel approach to solving this problem systematically.

We first describe how 2-dimensional objects are loaded into registers as packed subwords in area-mapped format, corresponding to how 2-dimensional data is usually stored in memory. We use the 2x2 matrix as a basic building block to which 2-dimensional frames of pixels and 2-D objects can be hierarchically decomposed. We then characterize the interesting permutation operations of this basic 2x2 matrix, as well as the four triangles that it contains. These are vertical, horizontal, diagonal, and rotational rearrangements of various kinds.

We define new subword permutation primitives: **check, exchange, excheck**, and **permset**. The **check** instruction allows the downward and upward swapping of elements, the **exchange**

instruction allows the right and left movement, while the **excheck** instruction allows the rotation of triangles. The **mixL** and **mixR** operations defined earlier [4] achieve the swapping of diagonal elements. **Permset** allows the permutation of a smaller set of subwords to be repeated on other subwords in the source register, enabling symmetric permutations to be specified on many more elements, without increasing the number of permutation control bits. **Exchange** is one example of the **permset** instruction.

We then define an initial alphabet (Alphabet A) of subword permutations which contains **mix, permset, check** and **excheck**. Processors designed for high performance can implement Alphabet A, while very cost sensitive processors can choose to implement an even smaller set - a minimal alphabet of only **mix** and **permset** instructions. The omitted instructions, **check** and **excheck** in Alphabet A, can be composed from **mix** and **permset**. That this minimal set is essentially equivalent to the set consisting of **mix** and **permute** in MAX-2 is a partial validation of the sufficiency of the subword permutation instructions chosen for MAX-2 [4]. We verify that all the 24 permutations of a 2x2 matrix can be obtained using only instructions from Alphabet A, in a single cycle, in a processor with at least two permutation units.

Just as subword parallelism is useful beyond multimedia processing for accelerating all forms of data-parallel computations on lower precision data, we expect that subword permutations will be equally useful. The problem is that there are so many possible rearrangements of a rectangular grid of pixels of arbitrary size that it is extremely difficult to select a set of fundamental permutation primitives, from which all other permutations can be built. This paper has proposed a systematic approach to solving this problem, and has proposed a very small alphabet of fundamental subword permutation primitives for existing and future two-dimensional processing in microSIMD architectures.

7. References

1. Ruby Lee, "Accelerating Multimedia with Enhanced Microprocessors", IEEE Micro, Vol. 15 No. 2, April 1995, pp. 22-32.
2. Marc Tremblay, J. O'Connor, V. Narayanan, and L. He, "VIS Speeds New Media Processing", IEEE Micro, Vol. 16 No. 4, August 1996, pp. 10-20.
3. Alex Peleg and Uri Weiser, "MMX Technology Extension to the Intel Architecture", IEEE Micro, Vol. 16 No. 4, August 1996, pp. 42-50.
4. Ruby Lee, "Subword Parallelism with MAX-2", IEEE Micro, Vol. 16 No. 4, August 1996, pp. 51-59.
5. Ninth Annual Microprocessor Forum, October 21-24, 1996, San Jose, California (Mips and Alpha multimedia).
6. Ruby Lee, "Multimedia Extensions for General-Purpose Processors", IEEE Workshop on Signal Processing Systems SiPS97 Design and Implementation, November 3-5, 1997, Leicester, United Kingdom, pp. 9-23.
7. S. Oberman, F. Weber, N. Juffa, G. Favor, "AMD 3Dnow! Technology and the K6-2 Microprocessor", Hot Chips 10 Symposium on High-Performance Chips, August 16-18, 1998, Palo Alto, California, pp. 245-254.
8. J. Golston, "Single-Chip H.324 Videoconferencing", IEEE Micro, Vol. 16 No. 4, August 1996, pp. 21-33.
9. S. Dutta, K. O'Connor, W. Wolf, and A. Wolfe, "A Design Study of a 0.25-um Video Signal Processor", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 8 No. 4, August 1998.
10. Karl Guttag et al, "A Single-Chip Multiprocessor for Multimedia: the MVP", IEEE Computer Graphics and Applications, Vol. 12 No. 6, November 1992, pp.53-64.
11. P. Foley, "The Mpact Media Processor Redefines the Multimedia PC", Proc. Compcon, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 311-318.
12. C. Basoglu, W. Lee, J. O'Donnell, "The MAP1000A VLIW Mediaprocessor", Equator Technologies Inc.
13. Ruby Lee, "Efficiency of microSIMD architectures and index-mapped data for media processing", Proceedings of IS&T/SPIE Symposium on Electric Imaging: Media Processors 99, January 1999, San Jose, California.
14. Michael Flynn, "Very High-Speed Computing Systems", Proceedings of IEEE, Vol. 54 No. 12, 1966, pp. 1901-1909.
15. IA-64 Application Developer's Architecture Guide, Intel Corporation, Order Number: 245188-001, May 1999. http://developer.intel.com/design/ia64.
16. AltiVec Extension to PowerPC Instruction Set Architecture Specification. Motorola, Inc., May 1998. http://www.motorola.com/AltiVec.
17. Ruby Lee, "Precision Architecture", *IEEE Computer,* Vol. 22, No. 1, Jan 1989, pp.78-91.