Efficiency of microSIMD architectures and index-mapped data for media processors

Ruby B. Lee

Department of Electrical Engineering, Princeton University, Princeton, NJ 08544

ABSTRACT

We show that microSIMD architectures are more efficient for media processing than other parallel architectures like SIMD or MIMD parallel processor architectures, and VLIW or superscalar architectures. We define alternative mappings of data onto subwords, and show that the index mapping is an ideal mapping for achieving maximal subword parallelism with minimal revamping of the original serial loop code. We show an example where packed data loaded directly into registers from memory can be interpreted as index-mapped data rather than area-mapped data. This allows increased use of the subword parallelism provided by the microSIMD architecture, by exploiting data parallelism across loop iterations rather than within a loop. We also show how to convert rapidly between data mappings by using the Mix permutation instructions, first defined in the MAX-2 multimedia extensions for PA-RISC processors. We propose a new instruction, MixPair, which cuts by half the cost of parallel Mix functional units, while achieving maximum subword permutation performance.

Keywords: multimedia, media processing, media processor, subword parallelism, SIMD, multimedia extensions, Mix, subword permutation, instruction set architecture, data parallelism

1. INTRODUCTION

Media processors are programmable processors, designed for processing different types of multimedia information such as images, video, graphics, animation, audio, and text. Many different architectures have been proposed for media processors. All these architectures attempt to exploit the parallelism in media processing to increase its performance. In this paper, we show that micro-SIMD architectures are the most cost-effective among the parallel architectures that have been proposed. These architectures support the subword parallelism[1,2] defined in the multimedia extensions for general-purpose processors[1,2,3,4,5].

Section 2 discusses alternative parallel architectures such as MIMD and SIMD[6] parallel processor organizations[7], superscalar and VLIW[8,9] parallel function organizations, and the microSIMD organization. We show that the microSIMD architecture has a lower overall cost and complexity, especially in register file organization and instructions needed, without sacrificing any performance. In fact, the lower complexity can result in shorter cycle times, further improving performance. MicroSIMD parallelism can also be combined with the other forms of parallelism, for increased flexibility and performance.

Section 3 describes different mappings of 1-dimensional (1-D) and 2-dimensional (2-D) multimedia data into the packed subwords of microSIMD architectures. This lays down the terminology for discussing the benefits of different data mappings in exposing different forms of parallelism. The "natural mappings" of data in memory are the linear mapping for 1-D data and the area mapping for 2-D data. We define the index mapping, and show it to be a canonically ideal mapping for exploiting subword parallelism, without having to revamp the serial code drastically. We also define the linear-projection mapping which reduces the number of registers needed to hold 2-D building blocks, like a 2x2 matrix. This in turn reduces the number of load instructions and hence the performance degradations due to potential cache misses associated with the load instructions.

Section 4 describes means and methods for converting rapidly between data mappings. We review the definition of Mix, a subword permutation operation defined earlier by the MAX-2 Multimedia Acceleration eXtensions for the PA-RISC general-purpose processors[2]. These single-cycle instructions move data rapidly between subword parallel tracks and across registers, performing work done by an interconnection network in moving data between parallel processors. We show how Mix may be used to convert rapidly between the different mappings of 2-D objects. We also define MixPair, a more powerful version of the Mix instructions, which cuts by half the number of Mix instructions and parallel functional units needed for maximum subword permutation performance. Section 5 summarizes the paper.

2. EFFICIENCY OF MICROSIMD ARCHITECTURES

We propose that microSIMD architectures, incorporating subword parallelism, are more efficient than other parallel architectures for the design of media processors.

2.1. Parallel architecture alternatives

Figure 1: MIMD or SIMD Parallel Processor Architecture

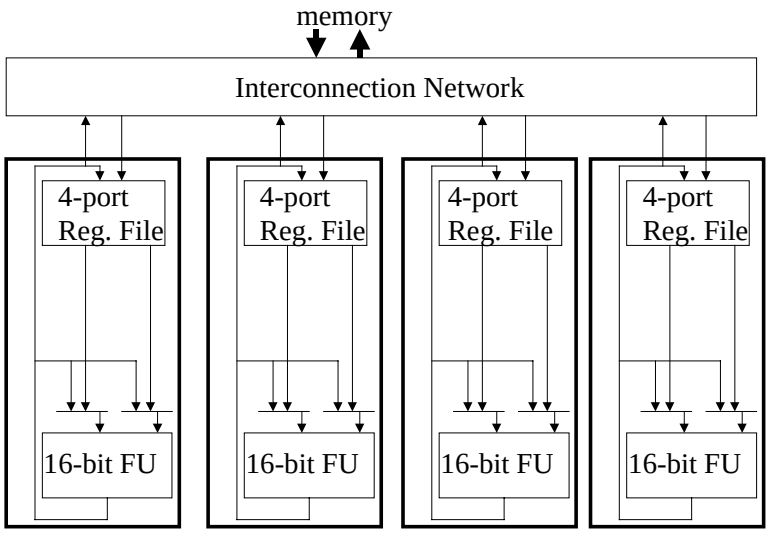
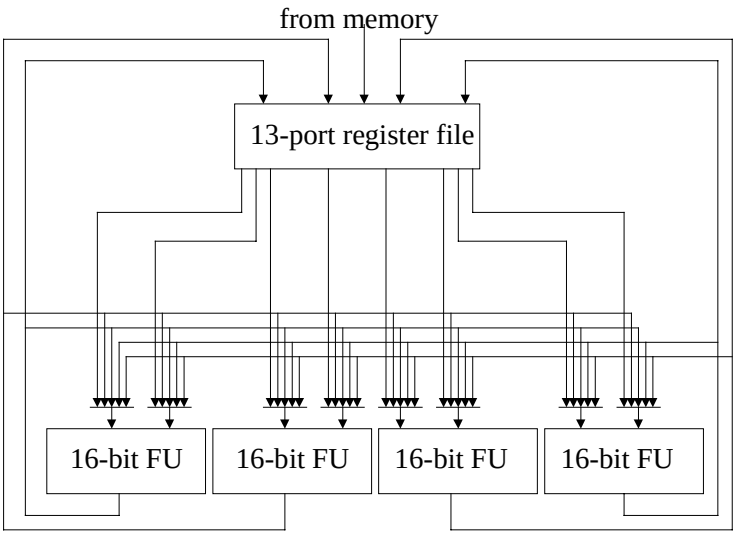


Figure 1 shows a multiprocessor architecture approach, which may be on a single chip[7]. Such multiprocessors are usually of a MIMD (Multiple Instruction Multiple Data) nature. This means that each processor can execute a different instruction in each cycle. Each processor has its own register file. Some sort of interconnection network between the register files is needed to move data between the processors. Otherwise, data has to be stored back to memory from one register file and loaded into a different register file in order to move data from one processor to another. For four processors, four instructions

Figure 2: Superscalar or VLIW Parallel Function Architecture



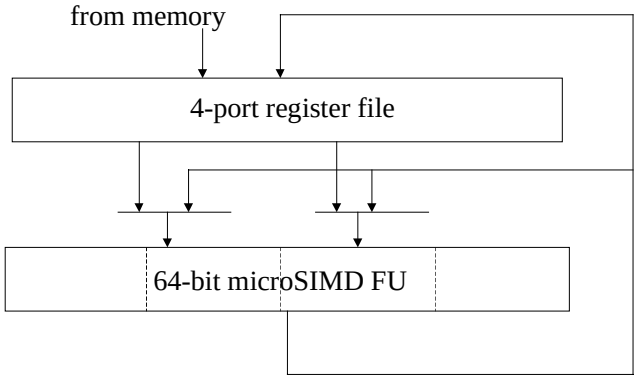
must be issued. A SIMD (Single Instruction Multiple Data) architecture has the same datapaths as the MIMD architecture, except that a single instruction is issued to all the processors in a cycle (see also figure 4).

Figure 2 shows a superscalar architecture for media processors. In this case, the register file is shared between M parallel functional units. In each cycle, N different instructions are issued, where N is less than or equal to M. Because of the single register file, and the scalar connection to memory, this is denoted a scalar processor or uniprocessor rather than a vector or parallel processor. Because of its parallel functional units and N parallel instructions issued per cycle, it is called an N-way superscalar architecture.

VLIW (Very Long Instruction Word) architectures [8,9] look like figure 2, except that only a single instruction is issued each cycle. However, this single instruction has up to M different operation fields in it, one for each of the parallel functional units. In figure 2, M=N=4 parallel functional units, each operating on 16-bit data. The functional units are clustered around a common register file.

Figure 3 shows a microSIMD architecture with a 64-bit functional unit operating out of a single register file in a single processor. The register file has four 16-bit subwords packed into each register, and the functional unit has been slightly modified so that it can operate either in 64-bit mode or in parallel 16-bit mode. If desired, the 64-bit functional unit can also process other subword sizes such as two 32-bit subwords, eight 8-bit subwords, sixteen 4-bit subwords, thirty-two 2-bit subwords or sixty-four 1-bit subwords, determined on an instruction by instruction basis. This microSIMD architecture incorporates *subword parallelism* [1,2,5], a key characteristic of the multimedia extensions added to general-purpose processors to accelerate the processing of different media types[1-5].

Figure 3: microSIMD Parallel Subword Architecture



2.2. Register Ports and Bypassing

The parallel functional unit approach of figure 2 is more efficient than the parallel processor approach of figure 1. Sharing the register file reduces the overhead of having to move data between the register files via an interconnection network or crossbar switch. However, the shared register file is now more complicated with 13 ports rather than four 4-ported register files. Since four independent 16-bit functional units share the register file in figure 2, it needs two read ports and one write port for each functional unit, plus one more write port for loading data from memory. In addition, it has to have four times as many registers, to hold the equivalent amount of data in registers.

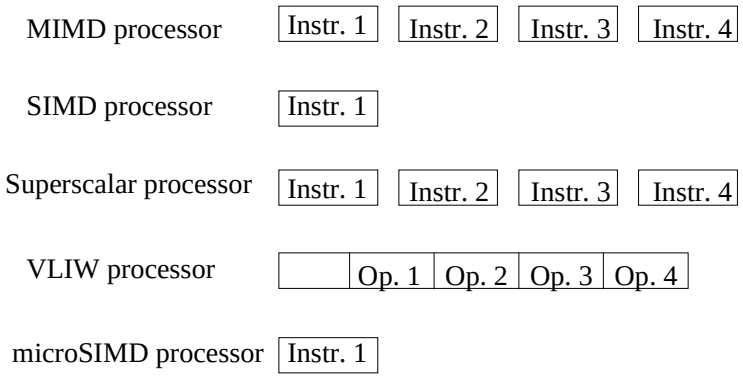
The microSIMD approach in figure 3 is more efficient than the parallel function approach of figure 2, because the register file has been simplified considerably without losing any parallelism. One might question whether building a wider (e.g., 64 bit) functional unit is a more efficient approach than having multiple narrower (e.g., 16 bit) functional units, since smaller functional units are more economical in area, and 16-bit arithmetic operations can be executed faster than 64-bit ones.

Unfortunately, the number of register ports, and the complexity of register bypassing, is much more of a cycle time limiter and source of complexity than the functional units. In addition to needing 8 read ports and 5 write ports in figure 2, the result register writes must be bypassed to each of the eight read ports, so that the result generated in the previous cycle can be used as a source operand in the current cycle. In figure 3, the register file only needs 4 ports: two read ports and one write port for the 64-bit functional unit, plus one additional write for loading data from memory. Register bypassing is required from only one result write port to the two read ports, in order to allow 16-bit results generated in the previous cycle to be used in the current cycle. Hence, the microSIMD architecture has a much simpler, and hence faster, register file and bypassing design: four versus thirteen ports, and simple 2-to-1 register multiplexing versus 5-to-1 multiplexing for selecting each read register operand.

2.3. Number of Instructions

Figure 4 shows the number of instructions needed to achieve the same degree of parallelism in the different parallel architectures. Both SIMD and microSIMD architectures require only one instruction, whereas MIMD multiprocessors and superscalar processors require 4 instructions for 4-way parallelism. While VLIW architectures require only one instruction, this contains 4 different operation fields. The reduced number of instructions of microSIMD architectures, or reduced size of its instructions with respect to VLIW architectures, is a cost reduction since this reduces the instruction memory requirements. It is also a performance benefit, since potential cache misses during instruction fetches are also reduced.

Figure 4: Instructions Needed Per Cycle for Parallelism of degree 4



2.4. Register Capacity and Instruction Encoding Efficiency

Each instruction (or operation in a VLIW instruction) typically needs three register addresses: two for register reads and one for the register to be written with the result of the operation. Thus if there are t registers, then $3*\log_2(t)$ bits are needed to specify the registers for each operation. With 32-bit instructions, where at least 12 bits should be reserved for the operation code and conditions, this leaves a maximum of 20 bits for three register addresses, limiting each register address to 6 bits, or 64 addressable registers.

In Table 1, we show the register address fixed at 5 bits, allowing 32 registers per register file. The MIMD and SIMD parallel processor architectures each has four such register files, with 128 total registers, each capable of holding a 16-bit operand. Their area requirements are proportional to the total number of bits in all four register files, with an overhead “d” per register file, and an addressing overhead “e” per register. The microSIMD architecture holds as many 16-bit operands in one quarter the number of registers, since these are packed as four 16-bit subwords per 64-bit register. Hence, it has slightly less area requirements due to lower area overhead for the registers and register file than the MIMD or SIMD architectures.

The superscalar and VLIW architectures hold only one-fourth the number of 16-bit operands, at a lower area. To hold the same number of 16-bit operands as the other architectures, superscalar and VLIW architectures have to have four times as many registers in the register file. This translates to 6 extra bits for register addresses in each superscalar instruction, and 24 extra bits for register addresses in each VLIW instruction with 4 operation fields. This is very costly and often infeasible for many instruction sets.

Table 1: Storage Capacity and Area Requirements with Fixed Number of Bits per Register Address

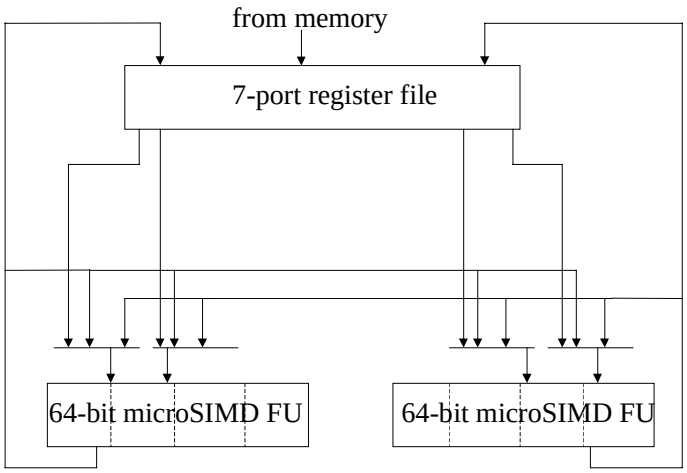
Parallel Architecture	Number of Register Files	Number of Registers, t, per Register File	Total Registers	Max. Number of 16-bit operands	Width of Register	Approximate Area for all Registers
MIMD	4	32	128	128	16 bits	F(4*32*16) + 4(d+32e)
SIMD	4	32	128	128	16 bits	F(4*32*16) + 4(d+32e)
Superscalar	1	32	32	32	16 bits	F(32*16) + d+32e
VLIW	1	32	32	32	16 bits	F(32*16) + d+32e
MicroSIMD	1	32	32	128	64 bits	F(4*32*16) + d+32e

2.5. Flexibility and Performance

The microSIMD architecture is also more flexible in its ability to efficiently support different data sizes simultaneously. This is especially useful for multimedia programs, where a stream of 16-bit audio samples may be processed together with a frame of 8-bit pixels. The microSIMD arithmetic functional units, once partitioned for 8-bit subwords, can also be partitioned for 16-bit or 32-bit subwords at essentially no additional cost.

A microSIMD architecture is less flexible than a MIMD, superscalar or VLIW architecture in the different instructions that can be issued in a single cycle. However, because pixel-oriented computations have high degrees of data parallelism, filling four or eight subword parallel slots for microSIMD execution is easy, and linear or better speedup is attainable[10, 1-5]. Beyond this small degree of operation parallelism, where SIMD operation is sufficient and most efficient, the ability to issue different types of instructions in the same cycle can be achieved with multiple microSIMD functional units, organized in a superscalar

Figure 5: Superscalar-microSIMD Architecture



or VLIW manner. We call this a *superscalar-microSIMD* or *VLIW-microSIMD* architecture. It illustrates the ability to combine other forms of instruction level parallelism (ILP), such as superscalar, VLIW or MIMD parallelism, together with the microSIMD *intra-instruction* parallelism. In general, the subword parallelism in figure 3 can be combined with the functional unit parallelism of a uniprocessor in figure 2, and with the processor parallelism in figure 1, for even higher degrees of parallelism and performance.

3. MAPPING MULTIMEDIA DATA INTO PACKED SUBWORDS

This section provides a foundation for talking about why subwords packed into registers need to be re-arranged, from time to time, and what the resulting subword mapping looks like. We try to give interpretations of these subword mappings, relative to the data in the program code, and the data parallelism in the algorithm. We define different subword mappings of 1-D and 2-D data, and describe the benefits of each. We also define an optimal mapping of data into packed subwords, where maximal subword parallelism can be exploited, without major re-structuring of the algorithm or the program. Identification of this optimal mapping, and easy conversion to it (described later in section 4), enables us to determine that the subword parallelism provided by microSIMD architectures can be fully exploited.

Visual multimedia data, whether representing two-dimensional or three-dimensional objects or scenes, are eventually mapped into 2-D frames of pixels (images and graphics), or a sequence of these 2-D frames (video and animation). This 2-D aspect of multimedia processing complicates the mapping into registers with packed subwords, which are 1-D structures. Hence, we first define alternative mappings of 2-D data into packed subwords. Then, we list the mappings of 1-D structures, and relate these to the 2-D mappings. We conclude the section with an example of a 3x3 box filter, illustrating the differences between area mapping and index mapping, and the performance advantage of the latter.

3.1. Area mapping

The natural mapping of a 2-D array of pixels in memory is to store a whole row in successive memory byte locations, followed by the second row, 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 second set of registers, and so forth. The end of each row may have to be padded with null elements, so that a new row starts in a new register. We call this *area mapping* of a 2-D block, defined as different rows of the 2-D block held in different registers.

The smallest 2-D unit is that of a 2x2 block. The first two rows in figure 6a show four such 2x2 blocks (or matrices) tiled across two registers, R1 and R2, in area-mapped format. Each register is assumed to be wide enough to hold 8 packed subwords, each subword representing an element of one of the four matrices, a, b, c, and d. This is repeated in the next two registers, R3 and R4, for a total of eight area-mapped 2x2 matrices in four registers.

Figure 6: Area mapping of 2-D blocks

a. Eight 2x2 matrices

R1 =	a00	a01	b00	b01	c00	c01	d00	d01
R2 =	a10	a11	b10	b11	c10	c11	d10	d11
R3 =	e00	e01	f00	f01	g00	g01	h00	h01
R4 =	e10	e11	f10	f11	g10	g11	h10	h11

b. Two 4x4 matrices

R1 =	a00	a01	a02	a03	c00	c01	c02	c03
R2 =	a10	a11	a12	a13	c10	c11	c12	c13
R3 =	a20	a21	a22	a23	c20	c21	c22	c23
R4 =	a30	a31	a32	a33	c30	c31	c32	c33

Larger square matrices can be built up of smaller 2x2 building blocks. For example, a 4x4 matrix can be built up of four 2x2 matrices, in four registers, as shown in figure 6b. Here the 2x2 matrices have been re-labeled to give the new 4x4 “a” and “c” matrices. Similarly, an 8x8 matrix can be built from four 4x4 matrices packed into 8 registers. Matrices with dimensions that are a power of two can be successively decomposed into smaller matrices, and ultimately into the smallest 2x2 matrix. Matrices with dimensions, which are not powers of two, can also be decomposed into basic 2x2 blocks. Hence, we describe subword mappings of basic building blocks like the 2x2 matrix, and use this to compose mappings of larger blocks.

3.2. Linear projection mapping

A *linear projection mapping* places consecutive rows of the 2-D block into the same register. For a 2x2 matrix, it places the two elements of the second row of the matrix immediately after the two elements of its first row, in the same register. If the register can hold eight elements, then two full 2x2 matrices can be mapped into a single register, as shown in each register of

figure 7a. Comparing with the 4 registers needed for an area-mapped 4x4 matrix in figure 6b, each linear-projected 4x4 matrix can be held in only 2 registers in figure 7b, although two 4x4 matrices require 4 registers in either mapping.

Figure 7: Linear projection mapping of 2-D blocks

a. Eight 2x2 matrices

R1 =	a00	a01	a10	a11	c00	c01	c10	c11
R2 =	b00	b01	b10	b11	d00	d01	d10	d11
R3 =	e00	e01	e10	e11	g00	g01	g10	g11
R4 =	f00	f01	f10	f11	h00	h01	h10	h11

b. Two 4x4 matrices

R1 =	a00	a01	a02	a03	a10	a11	a12	a13
R2 =	a20	a21	a22	a23	a30	a31	a32	a33
R3 =	c00	c01	c02	c03	c10	c11	c12	c13
R4 =	c20	c21	c22	c23	c30	c31	c32	c33

Linear projection mappings of 2-D blocks can reduce the number of memory load instructions needed to access a particular 2-D block, which in turn reduces any performance penalties due to cache misses associated with the load instructions.

3.3. Index mapping

An *index mapping* of a 2-D object places each element of the object in a separate register. For example, index mapping of multiple 2x2 matrices means that the top left elements of each 2x2 matrix is contained in one register, the top right elements in another register, the bottom left elements in a third register, and the bottom right elements in a fourth register, as shown in figure 8 for eight 2x2 matrices, a_{ij} though h_{ij}.

Figure 8: Index mapping of eight 2x2 matrices

R1 =	a00	b00	c00	d00	e00	f00	g00	h00
R2 =	a01	b01	c01	d01	e01	f01	g01	h01
R3 =	a10	b10	c10	d10	e10	f10	g10	h10
R4 =	a11	b11	c11	d11	e11	f11	g11	h11

For a 4x4 matrix, as in figure 6b, the direct definition of an index mapping would require 16 registers, each holding one of the 16 elements of the matrix. If there were at least eight of these 4x4 matrices that have to be processed, then the index mapping is very efficient. However, if only two 4x4 matrices are to be processed, then each index-mapped register would only contain two useful subwords, an inefficient use of the subword parallelism. In this case, one would try to see if the 4x4 matrix can be processed as four 2x2 matrices in parallel. Index mapping the 2x2 matrices would make fuller use of the subword parallelism available. In any case, the achievable parallelism is always the lower of the data parallelism in the program and the subword parallelism available in the hardware. We just have to be careful to expose the appropriate level of data parallelism, and this is hardly a problem when processing millions of pixels in a 2-D frame.

3.4. Linear and Index Mappings of 1-D structures

A *linear mapping* of a 1-D structure packs the first N elements into one register, the next N elements in another register, and so forth, where each element maps into a subword track. This matches the "natural' mapping of 1-D structures in memory.

Index mapping of a single 1-D structure is not practical, since this would only use one subword in a register that can hold N subwords. It is useful when there are multiple 1-D structures, e.g., m structures, where m is greater than or equal to N, and N is the number of subwords that can be packed into a register. In an *index mapping* of multiple 1-D structures, the first elements of N 1-D structures go into one register, the second elements of these same N 1-D structures go into another register, and so forth (see figure 9). The first elements of another N 1-D structures go into a different register, the second elements of these same N 1-D structures go into another register, and so forth. If m does not divide N exactly, the last register in each of these sets of m/N registers may only be partially packed with subwords.

Figure 9 shows seven 1-D structures, A, B, C, D, E, F and G, each containing a long sequence of elements, in linear mapped and index mapped formats. We assume that each element is now 16 bits and each register is 64 bits wide.

Figure 9: Linear and Index Mappings of Multiple 1-D Structures

a. Linear-mapped multiple 1-D structures:

```
R1 = A1 A2 A3 A4
R2 = B1 B2 B3 B4
R3 = C1 C2 C3 C4
R4 = D1 D2 D3 D4
        …
R5 = A5 A6 A7 A8
R6 = B5 B6 B7 B8
R7 = C5 C6 C7 C8
R8 = D5 D6 D7 D8
        …
R9 = E1 E2 E3 E4
R10= F1 F2 F3 F4
R11= G1 G2 G3 G4
      Etc.
```

b. Index-mapped multiple 1-D structures:

```
R1 = A1 B1 C1 D1
R2 = A2 B2 C2 D2
R3 = A3 B3 C3 D3
R4 = A4 B4 C4 D4
        …
R5 = A5 B5 C5 D5
        …
R9 = E1 F1 G1 0
R10= E2 F2 G2 0
      Etc.
```

If we take N registers, each containing N elements from multiple linear-mapped 1-D structures, then we have an NxN matrix, which is area-mapped or index mapped (see R1 through R4 in figure 9a and b). So, the conversions between mappings of multiple 1-D structures are like those for equivalent 2-D blocks.

3.5. Optimality of index mapping

The index mapping is a canonically ideal mapping for exploiting data parallelism without major restructuring of the serial code. It takes advantage of the fact that data parallelism is often expressed in a program as different iterations of a loop, where each loop operates on an independent but identically structured set of data. Since the same sequence of loop instructions is performed on different sets of data, SIMD (Single Instruction Multiple Data) parallelism is achieved by executing multiple iterations of the loop in parallel. MIMD parallelism of the same degree would not provide any additional parallelism or performance advantages. MicroSIMD parallelism is achievable if the corresponding elements of the independent sets of data operated on by different loop iterations are packed into the subwords of the source registers used by the instructions in the loop. An index mapping of 2-D blocks, or multiple 1-D structures, does exactly this type of subword packing.

In many algorithms, the same sequence of operations is performed on each 2-D block, although very different operations may be performed on different elements within the block. Index mapped 2-D blocks allow parallel execution of N loop iterations, each operating on one 2-D block, where N is the number of elements packed as subwords in a register. For area-mapped 2-D blocks to achieve the same degree of parallel execution, the same sequence of operations has to be performed on all elements of a given row of the 2-D blocks. For linear-projection mapping to achieve the same parallelism, the same sequence of operations has to be performed on all elements of all rows of the 2-D blocks mapped into a single register. The latter two cases are less frequent, resulting in less than maximum performance from subword parallelism.

The same is true for multiple 1-D structures. Often, each 1-D structure is operated upon by a different iteration of a program loop. That is, the same sequence of operations is performed on each 1-D structure, although different operations may be performed on different elements in a 1-D structure. For the linear mapping to achieve a parallelism of N, the same sequence

of operations would have to be performed on N elements of a 1-D structure. This is less common, and often cited erroneously as a limitation of SIMD parallelism.

The change in a serial program loop to a subword-parallel loop, with N index mapped subwords is just:

Original code: For I = 1 step 1 until 800 do F(a(I), b(I), …, z(I));
Subword-Parallel Code: For I = 1 step N until 800 do F(a(I), b(I), …, z(I));

3.6. Example: 3x3 Box Filter

Sometimes, index mapping and area mapping are just different ways of interpreting the subwords packed into a register, and no conversion between mappings is actually needed. For example, consider a 3x3 box filter that is used to smooth every pixel in a frame of pixels by taking a weighted average of its eight nearest neighbors with itself. In figure 10, the two boxes outlined in bold show two adjacent 3x3 matrices labeled a_{ij} and d_{ij}. The box outlined with dotted lines is an overlapping 3x3

Figure 10: Overlapping Area-Mapped 3x3 Matrices

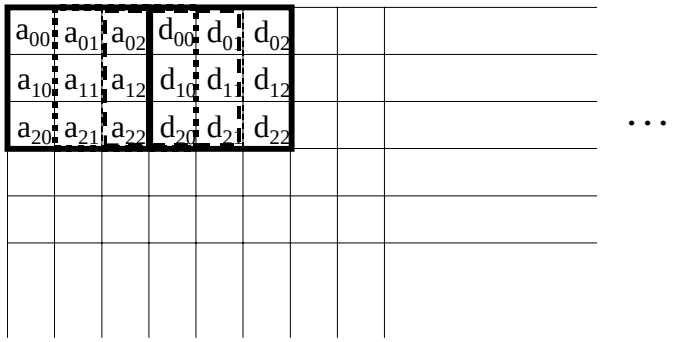
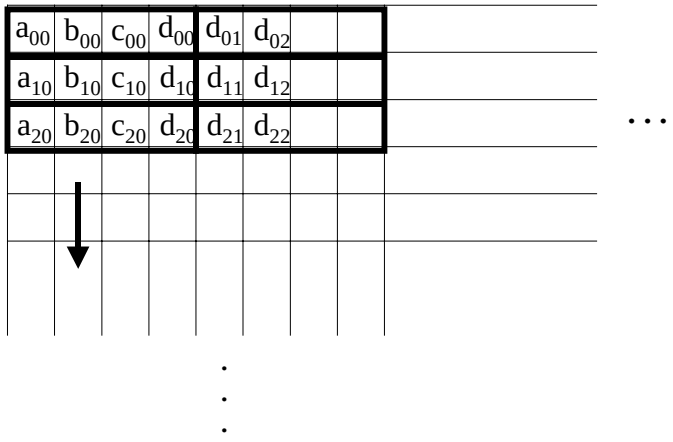


Figure 11: Interpreting Overlapping Area-Mapped 3x3 matrices as Index-Mapped 3x3 matrices



matrix, where element a_{01} is now interpreted as the top left element b_{00} of the next 3x3 matrix. Similarly, element a_{02} can also be interpreted as the top left element c_{00} of the third overlapping 3x3 matrix, shown outlined with dashed lines.

Interpreting the packed subwords of a register as area-mapped 3x3 matrices does not permit maximum subword parallelism. For example, the weights applied to a_{00}, a_{01} and a_{02} may be different, so a register containing these three elements cannot be multiplied by a single number to apply the desired weightings. Furthermore, if the register holds four subwords, then the fourth subword is wasted. However, if we interpret the same packed subwords in the registers as index-mapped 3x3 matrices, then we can exploit the full subword parallelism of degree four.

Figure 11 shows the same elements as figure 10, but now interpreted as index-mapped 2-D objects, packed into six registers (in bold outlines). The three registers on the left contain the index-mapped format of the elements of the leftmost column of four overlapping 3x3 matrices, a , b, c and d. To get the index-mapped registers of the other six elements, the three registers on the right have to be used as additional source registers. For example to get a register to contain the top row's middle elements, (a_{01}, b_{01}, c_{01}, d_{01}), we concatenate the top left and top right registers in figure 11, and shift left by one subword. Performing a subsequent left shift of the same two registers gives a register containing the top right elements (a_{02}, b_{02}, c_{02}, d_{02}) of the corresponding 3x3 matrices. This can be repeated for the second and third rows of the 3x3 matrices.

This interpretation of the packed subwords as index-mapped 3x3 matrices exploits the maximal subword parallelism by executing four iterations of the box-filter loop simultaneously. To maximize re-use of the nine index-mapped registers, the next four loop iterations should proceed down the frame of pixels, rather than across to the right. This allows re-use of six of the previous nine index-mapped registers, and only requires two load instructions to get the contents of two new source registers[10].

In other algorithms, explicit conversion from area-mapped 2-D structures and linear-mapped 1-D structures to the index-mapped format is required for maximum subword parallelism. In the next section, we describe simple but powerful subword permutation primitives for performing these conversions.

4. CONVERSION BETWEEN DATA MAPPINGS

We describe general-purpose subword permutation operations, and how they may be used to convert between the 2-D mappings defined above. We also define MixPair, a further optimization on subword permutation instructions.

4.1. Mix: a fundamental subword permutation operation

The Mix operation was first defined in MAX-2, the multimedia instruction extensions for PA-RISC processors[2,5]. It combines subwords from two source registers to produce one result register. There are twice as many subwords as would fit into a single result register, and too many possible combinations to be able to specify all of them. So, Mix selects either all even elements, or all odd elements, alternately from the source registers. To avoid ambiguity, the "L" and "R" suffixes for "Left" and "Right" are used instead of "Even" and "Odd". Elements may be labeled even or odd, depending on whether the elements are numbered from 0 or from 1, starting from the left (big-endian) or from the right (little-endian).

Table 2: Definition of MixL and MixR for 8, 16 and 32 bit Packed Subwords

	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)	Rt = a A c C e E g G
MixR,8(R1,R2)	Rt = b B d D f F h H
MixL,16(R1,R2)	Rt = a b A B e f E F
MixR,16(R1,R2)	Rt = c d C D g h G H
MixL,32(R1,R2)	Rt = a b c d A B C D
MixR,32(R1,R2)	Rt = e f g h E F G H

Table 2 defines this pair of Mix instructions, for three different subword sizes: 8 bits, 16 bits and 32 bits. Each letter in the contents of a register represents an 8-bit quantity. MixL means select the *left* subword from each adjacent pair of subwords, alternately from each source register. *MixR* means select the *right* subword from each adjacent pair of subwords, alternately from each source register. Mix has been implemented in a single cycle by simple modifications to the Shifter unit in PA-RISC processors.

4.2. Area mapped to index mapped 2-D objects

To convert eight area-mapped 2x2 matrices to index mapping, four Mix instructions are sufficient (see figure 12b). We assume that the order of the index-mapped elements in the subword tracks is irrelevant, since the same sequence of operations is performed on every subword. The reverse transformation, from index mapping to area mapping, is done with the same four Mix instructions (see figure 12c), except that the programmer has to mentally rename R11 with R12, and vice versa.

Figure 12: Mix transforms area mapping to index mapping

a. Area-Mapped 2x2 Matrices in Source Registers

```
R1 = a00  a01  b00  b01  c00  c01  d00  d01
R2 = a10  a11  b10  b11  c10  c11  d10  d11
R3 = e00  e01  f00  f01  g00  g01  h00  h01
R4 = e10  e11  f10  f11  g10  g11  h10  h11
```

b. Transform eight area-mapped 2x2 matrices to index-mapping:

```
R6 = MixL,8(R1,R3) =  a00  e00  b00  f00  c00  g00  d00  h00
R7 = MixR,8(R1,R3) =  a01  e01  b01  f01  c01  g01  d01  h01
R8 = MixL,8(R2,R4) =  a10  e10  b10  f10  c10  g10  d10  h10
R9 = MixR,8(R2,R4) =  a11  e11  b11  f11  c11  g11  d11  h11
```

c. Transform eight index-mapped 2x2 matrices in b. to area-mapping:

```
R10= MixL,8(R6,R7) =  a00  a01  b00  b01  c00  c01  d00  d01
R11= MixR,8(R6,R7) =  e00  e01  f00  f01  g00  g01  h00  h01
R12= MixL,8(R8,R9) =  a10  a11  b10  b11  c10  c11  d10  d11
R13= MixR,8(R8,R9) =  e10  e11  f10  f11  g10  g11  h10  h11
```

d. Transform four area-mapped 2x2 matrices to double-precision index-mapping:

```
R6 = MixL,8(R0,R1) =   0   a00   0   b00   0   c00   0   d00
R7 = MixR,8(R0,R1) =   0   a01   0   b01   0   c01   0   d01
R8 = MixL,8(R0,R2) =   0   a10   0   b10   0   c10   0   d10
R9 = MixR,8(R0,R2) =   0   a11   0   b11   0   c11   0   d11
```

To convert only four area-mapped 2x2 matrices in R1 and R2 to index mapping, only four 8-bit elements are to be held in a result register which has a capacity for eight such elements. Hence, we can use extended precision of 16-bit subwords (see figure 12d), using Mix with R0 to supply zeros. However, 4 Mix instructions and 4 result registers are still needed. The extended precision for intermediate results is highly desirable. However, this case does illustrate that the achieved parallelism can never be more than the data parallelism in the program. In figure 12d, the data parallelism is 4, and the microSIMD subword parallelism is 8. Fortunately, the data parallelism in processing frames of pixels (e.g. 480,000 pixels in an 800x600 frame) is much higher than the subword parallelism of 4 or 8, supplied by a microSIMD architecture.

4.3. Linear projected to index mapped 2-D objects

The conversion of eight linear-projected 2x2 matrices to index mapping can be done with 8 Mix instructions, as shown in figure 13b. This can be done in 2 cycles, with four Mix functional units. Again, the reverse conversion can be done symmetrically, with the same 8 Mix instructions.

Figure 13: Linear projection mapping to Index Mapping of 2x2 matrices

a. Eight linear-projected 2x2 matrices:

```
R1 =  a00  a01  a10  a11  c00  c01  c10  c11
R2 =  b00  b01  b10  b11  d00  d01  d10  d11
```

```
                R3 =  e00  e01  e10  e11  g00  g01  g10  g11
                R4 =  f00  f01  f10  f11  h00  h01  h10  h11

       b. Transform to eight index-mapped 2x2 matrices:
R5 = MixL,8(R1,R2)  =  a00  b00  a10  b10  c00  d00  c10  d10
R6 = MixR,8(R1,R2)  =  a01  b01  a11  b11  c01  d01  c11  d11
R7 = MixL,8(R3,R4)  =  e00  f00  e10  f10  g00  h00  g10  h10
R8 = MixR,8(R3,R4)  =  e01  f01  e11  f11  g01  h01  g11  h11

R9 = MixL,16(R5,R7) =  a00  b00  e00  f00  c00  d00  g00  h00
R11= MixL,16(R6,R8) =  a01  b01  e01  f01  c01  d01  g01  h01
R10= MixR,16(R5,R7) =  a10  b10  e10  f10  c10  d10  g10  h10
R12= MixR,16(R6,R8) =  a11  b11  e11  f11  c11  d11  g11  h11
```

4.4. Area mapped to linear projected 2-D objects

The conversion of area-mapped 2x2 matrices (as in R1 and R2 in figure 12a) to linear-projected ones can also be done by using a pair of Mix operations on 16-bit packed subwords, as shown in figure 14.

Figure 14: Area Mapping to Linear Projection Mapping of 2x2 Matrices

```
      MixL,16 (R1,R2) =  a00  a01  a10  a11  c00  c01  c10  c11
      MixR,16 (R1,R2) =  b00  b01  b10  b11  d00  d01  d10  d11
```

The reverse conversion of linear-mapped 2x2 matrices to area-mapped matrices can be accomplished symmetrically, by the same two Mix operations.

4.5. MixPair optimization

The mapping conversions in the above cases can all be done in one or two cycles, depending on data dependencies, if there are four parallel functional units capable of performing Mix instructions. Since the MixL and MixR instructions are almost always used as a pair, we can cut the cost in half by creating a new *MixPair* instruction that combines a pair of MixL and MixR instructions. This requires that two result registers be written for each MixPair instruction, which is too costly for most microprocessors, but quite feasible for specially designed media processors. Such a 2-read, 2-write MixPair instruction may be more efficient for a media processor than doubling the number of Mix functional units and associated register port requirements, in order to achieve maximum subword permutation performance.

In fact, we can achieve this cost savings without having to go to an instruction format with 4 register addresses. Although the most flexible solution is to have two separate addresses for each result register in the MixPair instruction, a sufficient solution is to have the results written into an even-odd pair of registers, with just one result register address. For example, no matter whether the result register address is even or odd, the MixL result word goes into the even register, and the MixR result word goes into the odd register. This is simple to implement, since the pair of result registers is selected by ignoring the least significant bit of the result register address. In table 3, either Rs or Rt can be specified as the result register of the MixPair instruction, where Rs and Rt have the same address bits except for the last bit.

Table 3: New MixPair instruction definition

	Register Contents:
	R1 = a b c d e f g h
	R2 = A B C D E F G H
Instruction:	Definition:
MixPair, 8(R1,R2)	Rs = a A c C e E g G and Rt = b B d D f F h H
MixPair,16(R1,R2)	Rs = a b A B e f E F and Rt = c d C D g h G H
MixPair,32(R1,R2)	Rs = a b c d A B C D and Rt = e f g h E F G H

5. CONCLUSIONS

We show that a microSIMD architecture incorporating subword parallelism is more efficient for a media processor, than a MIMD or SIMD parallel processor architecture, or a superscalar or VLIW architecture. This is because microSIMD architectures can exploit the data-parallelism present in media processing programs as well as the other parallel architectures can, at a much lower complexity of register ports, register bypassing, and interconnection networking. This results in faster cycle times, less area and less design complexity for microSIMD architectures with the same degree of operation parallelism. In addition, the number of static and dynamic instructions needed for a given parallel computation is reduced. For the same number of addressable registers, microSIMD architectures hold more multimedia data than superscalar or VLIW architectures. MicroSIMD architectures also have the flexibility of being able to support different subword sizes on an instruction by instruction basis. By combining microSIMD with superscalar or VLIW parallelism, the flexibility of executing different instructions in one cycle is also achieved, at a lower cost.

Next, we lay the foundation for systematically mapping data into microSIMD registers, to exploit fully the subword parallelism provided. We defined alternative mappings of 1-D and 2-D multimedia objects into the packed subwords in microSIMD registers. We use the 2x2 matrix as a basic building block to which 2-dimensional frames of pixels can be decomposed. Area-mapped 2-D blocks and linear-mapped 1-D structures most closely match how data is usually stored in memory. The linear-projection mapping minimizes the number of registers needed for holding a 2-D block. Index-mapped data structures can achieve the maximum subword parallelism provided by the microSIMD architecture, with only minor program code modifications.

The example of the 3x3 box filter shows how the same packing of subwords in registers can be interpreted as area-mapped or index-mapped pixels. This is equivalent to trying to find SIMD parallelism amongst instructions in one loop iteration, versus across multiple loop iterations. By interpreting the data as index-mapped pixels, the full subword parallelism provided by the microSIMD hardware can be exploited. This is because media processing programs tend to have large amounts of data parallelism, and loops are typically iterated much more than four or eight times, which is the degree of subword parallelism provided in a typical microSIMD instruction.

In other media processing programs, explicit conversion to and from the index mapping may be necessary to achieve the maximum performance from microSIMD architectures. We show that the Mix instructions defined for the MAX-2 multimedia acceleration extensions of the PA-RISC 2.0 processors[2,5] can be used to convert rapidly between the different mappings of subwords in microSIMD architectures. We also propose a new MixPair instruction, which can achieve the same permutation performance with half the number of Mix instructions and functional units.

REFERENCES

1. Lee R., "Accelerating Multimedia with Enhanced Microprocessors", *IEEE Micro*, Vol. 15 No. 2, April 1995, pp. 22-32.
2. Lee R., "Subword Parallelism with MAX-2", *IEEE Micro*, Vol. 16 No. 4, August 1996, pp. 51-59.
3. Tremblay M., O'Connor J., Narayanan V., and He L., "VIS Speeds New Media Processing", *IEEE Micro*, Vol. 16 No. 4, August 1996, pp. 10-20.
4. Peleg A., and Weiser U., "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, Vol. 16 No. 4, August 1996, pp. 42-50.
5. Lee R., "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.
6. Flynn M., "Very High-Speed Computing Systems", *Proceedings of IEEE*, Vol. 54 No. 12, 1966, pp. 1901-1909.
7. Guttag K. et al, "A Single-Chip Multiprocessor for Multimedia: the MVP", *IEEE Computer Graphics and Applications*, Vol. 12 No. 6, November 1992, pp.53-64.
8. Rathnam S. and Slavenvenburg, "An Architectural Overview of the Programmable Multimedia Processor, TM-1", *Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 319-326.
9. Foley P., "The Mpact Media Processor Redefines the Multimedia PC", *Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 311-318.
10. Lee R., and McMahan M., "Mapping of Application Software to the Multimedia Instructions of General-Purpose Microprocessors", *Proceedings of IS&T/SPIE Symposium on Electric Imaging: Multimedia Hardware Architectures 1997*, Feb 10-14, 1997, San Jose, California, pp. 122-133.