Subword Sorting with Versatile Permutation Instructions

Zhijie Shi and Ruby B. Lee
Department of Electrical Engineering, Princeton University
{zshi, rblee}@ee.princeton.edu

Abstract

Subword parallelism has succeeded in accelerating many multimedia applications. Subword permutation instructions have been proposed to efficiently rearrange subwords in or among registers. Bit-level permutation instructions have also been proposed recently for their importance in cryptography. However, some important algorithms, especially ones with lots of conditional control dependencies such as sorting, have not exploited the advantage of subword parallel instructions. In this paper, we show how one of the bit permutation instructions, GRP, can be used for fast sorting. In the process, we demonstrate the versatility of this permutation instruction for uses other than bit permutations. This versatility is important in considering the addition of a new instruction to a general-purpose processor. The results show that our sorting methods have a significant speedup even when compared with the fastest sorting algorithms. We also discuss the hardware implementation of the GRP instruction and compare its latency to a typical processor's cycle time.

1. Introduction

General-purpose processors are optimized for word-oriented computation. Hence, their instruction set architecture (ISA) provides limited support for the manipulation of data items smaller than a word. Logical operations AND, OR, and NOT along with SHIFT are common bit-level operations. Today, most processor ISAs support subword parallel computation [1-6]. Subwords are data items that are smaller than a word. Multiple subwords can be packed into one register, so that one instruction can perform an operation on multiple data items simultaneously. Efficient rearrangement of subwords is often necessary to put subwords into proper positions in registers so that operations can be applied to all subwords at the same time. Without such fast arrangement of subwords, called *subword permutation*, the performance gain achieved by subword parallelism may be diminished.

This material is based upon work supported by the National Science Foundation under grant No. 0105677.

The PERMUTE and MIX instructions are proposed in the MAX-2 multimedia instructions for PA-RISC 2.0 processors to address subword permutations [2]. The PERMUTE instruction can perform any arbitrary permutation of 16-bit subwords from one source register, with a single instruction. The MIX instruction combines even or odd subwords from two source registers. While these are the first general-purpose subword permutation primitives introduced into microprocessors, both PERMUTE and MIX dealt only with 16-bit subwords in MAX-2 [2]. IA-64 extended MIX to support 8-bit subwords and added five variants of byte permute instructions called MUX [17]. However, it has not been shown whether these permutation primitives can efficiently generate all desired subword rearrangements. Lee further proposed new subword permutation primitives for 2-dimensional rearrangements of subwords packed into registers [7]. A minimal canonical set includes MIX and PERMSET, defined for subword sizes that are powers of two. PERMSET repeats a permutation on a small set of subwords over the entire set of subwords in a register.

While most subword permutation instructions are proposed to handle subwords of at least 8 bits in size, instructions for permutations on subwords of 1 bit (also called *bit permutations*) have recently been proposed [13-16]. Bit permutations are useful for achieving diffusion [8, 9] in symmetric-key algorithms such as DES, Twofish and Serpent [10-12]. The new instructions, GRP [13,15], OMFLIP [14,15], and SIEVE [16], can efficiently perform arbitrary bit permutations, which are very slow on existing word-oriented processors. These new instructions are designed to have two operands and one result, to fit in the datapath of a typical processor. Each of them has different strong and weak points [15]. Any one is sufficient to achieve bit-level permutations efficiently. Which one to choose depends on its cost (in area and latency), and its versatility, not only for bit permutations, but also for other functions. In this paper, we demonstrate the versatility of one of these bit permutation instructions, GRP, showing the large speedup achievable in sorting subwords packed in one or more registers.

Although subword instructions have been used to implement many algorithms, such as IDCT [2], with

Zhijie Shi and Ruby B. Lee, "Subword Sorting with Versatile Permutation Instructions", *Proceedings of ICCD 2002 International Conference on Computer Design, pp. 234- 341,* September 2002

significant speedups, it has not been exploited by some control-intensive algorithms such as sorting. Most fast sorting algorithms, such as quicksort, are control intensive [18, 19]. In such sorting algorithms, since the subwords in a register affect the control flow in different ways, it appears difficult to exploit the performance benefits of the SIMD-style operation provided by subword-parallel instructions. A few attempts have been made to show how conditional branches may be eliminated in doing parallel compare-and-swap operations in the implementation of median filters, but not for general-purpose sorting [22]. In this paper, we show another way in which control flow dependencies can be eliminated, and subword parallelism at the subword or bit levels fully exploited, to sort subwords packed into one or more registers. In the process, we also show the versatility and effectiveness of a bit permutation instruction, GRP, for accelerating subword sorting.

In section 2, we describe the GRP instruction, and show how to use it to do subword sorting. In Section 3, we describe a hardware implementation of the GRP operation. The performance is compared in Section 4. Section 5 summarizes the paper.

2. Sorting subwords with the GRP instruction

Sorting rearranges the data items in a list by its value in a monotonically increasing, or decreasing, order. Many algorithms such as binary search require the input data to be sorted. In this section, we focus on sorting a small number of positive integers. We assume these values are stored contiguously in memory.

2.1. The GRP instruction

The GRP instruction was first proposed to do arbitrary permutations of n bits [13] very quickly. It uses a typical two-operand, one-result instruction format:

```
GRP     R1, R2, R3
```

R1 and R2 are the source registers, and R3 is the destination register. If R1[i] represents the i^{th} bit of R1, the function of the GRP instruction can be described with pseudo code shown in Figure 1.

```
j = 0;
for (i = 0; i < n; i ++)
        if (R2[i] == 0)
                R3[j ++] = R1[i]
for (i = 0; i < n; i ++)
        if (R2[i] == 1)
                R3[j ++] = R1[i]
```

Figure 1: The GRP instruction

The GRP instruction divides the bits in the source R1 into two groups according to the bits in R2. For each bit in R1, we check the corresponding bit in R2. If the bit in R2 is 0, the bit in R1 is put into the first group. Otherwise the bit in R1 is put into the second group. During this process we do not change the relative position of bits within each group. Finally, putting the first group to the left of the second group, we get the result in R3. Figure 2 shows how the GRP instruction works on 8-bit registers.

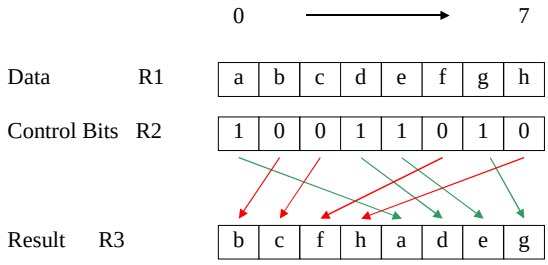


Figure 2: The GRP instruction on 8-bit registers

It has been shown that on n-bit systems, at most lg(n) GRP instructions can perform any n-bit permutations [13]. Figure 3 shows how a permutation is done on an 8-bit system. The permutation (7,6,5,4,3,2,0,1) indicates that bit 0 of the output is bit 7 of the input; bit 1 of the output is bit 6 of the input; and so forth. Comments right before an instruction show the register contents before the execution of that instruction. The underlined bits have a control bit of 1 in R2, thus will be moved to the right after the execution of the instruction. After the execution of the third instruction, R1 contains the desired result.

```
; R2=00101010      R1=(0,1,2,3,4,5,6,7)
GRP     R1, R2, R1
; R3=11010010      R1=(0,1,3,5,7,2,4,6)
GRP     R1, R3, R1
; R4=10101100      R1=(3,7,2,6,0,1,5,4)
GRP     R1, R4, R1
;                 R1=(7,6,5,4,3,2,0,1)
```

Figure 3: GRP instructions used to do the permutation specified by (7,6,5,4,3,2,0,1)

2.2. Sorting subwords within a register

The GRP instruction can be described as sorting bits in the first operand according to the bit values (0 or 1) in the second operand. When used in radix sort [18], it can sort multi-bit subwords in a register. Radix sort takes the least significant bit (LSB) of each subword, sorts the subwords by that bit maintaining the order of subwords that have the same value for that bit, and repeats the sorting with the next more significant bit. During this process, all bits in a subword should be moved together. This requires the control bits have the same value for all

bits in a subword. To quickly generate control bits in this format, we introduce a new instruction BroadcastBit:

BroadcastBit,s,i R1, R2

BroadcastBit has one source operand R1 and one destination R2. It broadcasts a bit in a subword to all bits in that subword. *s* specifies the subword size in bits, and *i* specifies which bit in each subword is broadcast. After the instruction is executed, all bits in each subword in R2 have the same value, the value of bit *i* of the corresponding subword in R1. Table 1 shows the results of some BroadcastBit instructions when R1 = 0x01234567. The first instruction broadcasts the LSB of each 4-bit subword, giving a result of alternating 0s and 1s. The second instruction broadcasts bit 1 of each 4-bit subword. The third instruction sets all the bits in R2 to 1 because the LSB of all the 8-bit subwords (i.e. 0x01, 0x23, 0x45, 0x67) is 1.

Note: R1 and R2 are 32-bit words, and R1 = 0x01234567

Instruction	Result in R2
BroadcastBit,4,0 R1, R2	0x0F0F0F0F
BroadcastBit,4,1 R1, R2	0x00FF00FF
BroadcastBit,8,0 R1, R2	0xFFFFFFFF

Table 1: Example of the BroadcastBit instruction

The BroadcastBit instruction can be added to an ISA with little overhead because it requires only small control modifications to the functional unit that does parallel addition with saturation, which is included in almost all subword instruction sets [1-5, 17]. When saturation occurs during an addition, all bits in a subword are set to 0s or 1s. When doing BroadcastBit, we force bits in a subword to all 0s or all 1s, determined by whether bit *i* of the source subword is 0 or 1. Note that BroadcastBit can be considered a special case of PERMSET [7].

BroadcastBit and GRP can sort subwords packed in a register. For every bit *i* in a subword from the LSB to the most significant bit (MSB), repeat the following: broadcast bit *i* in each subword; then perform the GRP operation on the subwords, using the result of broadcasting as the control bits. A BroadcastBit instruction uses the result of the previous GRP instruction, except for the first BroadcastBit which uses the original input. An example of sorting 8-bit subwords in R1 is shown in Figure 4, using 16 instructions.

The program in Figure 4 can be used to sort eight 8-bit subwords in a 64-bit register, or sixteen 8-bit subwords in a 128-bit register, or thirty-two 8-bit subwords in a 256-bit register. Although the time complexity of radix sort is O(*kn*), which depends on both the key length *k* and the number of items *n*, the number of instructions required in our method is 2*k*, where *k* is the number of bits in a subword. For example, sorting four 16-bit subwords in a 64-bit register requires 32 instructions.

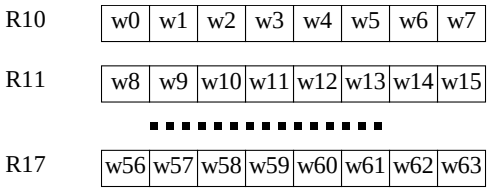
```
;The subword size is 8 bits
;The register size is 64 bits
;R1 contains the subwords to be sorted
;The sorted subwords are also placed back into R1

BroadcastBit,8,0    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,1    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,2    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,3    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,4    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,5    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,6    R1, R2
GRP                 R1, R2, R1
BroadcastBit,8,7    R1, R2
GRP                 R1, R2, R1
```

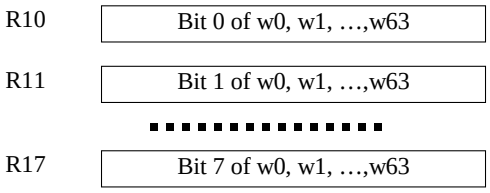
Figure 4: Sorting subwords in a register with the GRP and BroadcastBit instructions

2.3. Sorting subwords in multiple registers

A problem with the above method is that only a few subwords can be packed within a register, and thus sorted by GRP and BroadcastBit. In this section, we will introduce another method to sort more subwords packed in multiple registers. We rearrange the bits so that each register has only one bit from each subword, then sort *n* subwords simultaneously where *n* is the number of bits in the registers.



a) All 8 bits in each subword are together



b) Each bit of each subword is in a different register

Figure 5: Arrangement of bits in subwords

Figure 5 shows two different storage formats for subwords. Suppose registers are 64 bits in width, and subwords are 8 bits. If we can keep all 8 bits in a subword together, as in Section 2.2, each register can hold eight subwords, and 64 subwords require 8 registers, R10, R11, …, R17. This is shown in Figure 5a. Alternatively, we can rearrange bits so that the first register, R10, holds bit 0 for all 64 subwords, the second register, R11, holds bit 1 for all 64 subwords, and so on. This is shown in Figure 5b. The two formats require the same number of registers to store 64 subwords.

We can easily apply radix sort to sort subwords in Figure 5b: use the GRP instruction to sort all the registers R10, R11, …, R17 by each bit from the LSB (R10) to the MSB (R17). The code is shown in Figure 6. R10, the LSB, is first used to sort each register, R10 through R17. Then R11, the second least significant bit, is used to sort the eight registers, and so forth. Thus, 8x8 = 64 instructions can sort 64 8-bit subwords.

```
GRP     R17, R10, R17
GRP     R16, R10, R16
GRP     R15, R10, R15
GRP     R14, R10, R14
GRP     R13, R10, R13
GRP     R12, R10, R12
GRP     R11, R10, R11
GRP     R10, R10, R10

GRP     R17, R11, R17
GRP     R16, R11, R16
GRP     R15, R11, R15
GRP     R14, R11, R14
GRP     R13, R11, R13
GRP     R12, R11, R12
GRP     R10, R11, R10
GRP     R11, R11, R11
……

GRP     R16, R17, R16
GRP     R15, R17, R15
GRP     R14, R17, R14
GRP     R13, R17, R13
GRP     R12, R17, R12
GRP     R11, R17, R11
GRP     R10, R17, R10
GRP     R17, R17, R17
```

Figure 6: Sort subwords after the pre-transpose

The subwords, however, are normally not in the format shown in Figure 5b when stored in the main memory. Instead, they are more likely in the format shown in Figure 5a. To convert the subwords into the format we need, we consider the 8 bytes in the same column of 8 registers as an 8x8 bit matrix. Thus, the 64 subwords shown in Figure 5a can be considered as eight 8x8 bit matrices. We transpose these matrices so that R10 will have bit 0 of w0, w8, …, w56, w1, w9, …, w57, …, w7, w15, …, w63. Similarly, R11 will have bit 1 of w0, w8, …, w56, w1, w9, …,w57, …, w7, w15, …, w63, and so on. We call this the *pre-transpose* step. Although the bit order after this pre-transpose is different from that shown in Figure 5b, the subwords can be sorted in the same way.

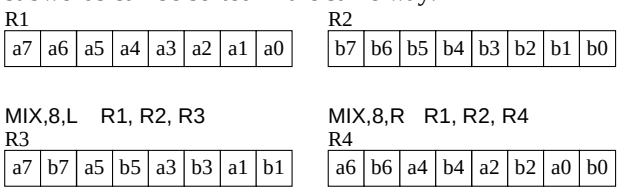


Figure 7: The MIX instruction

Lee showed that the MIX instruction can do matrix transpose efficiently [2]. Each MIX instruction takes two registers as input, each of which consists of *r* subwords. Subwords are further divided into pairs, and each pair has a left subword and a right subword. "MIX, L" selects the left subwords alternately from each register; "MIX, R" similarly selects the right subwords. The subword size is specified in the MIX instruction. For example, in Figure 7, "MIX, 8, L" indicates MIX operation on 8-bit subwords, selecting the left subwords of each pair. Let R1, R2, R3 and R4 be 64-bit registers, and each a0, a1, …, a7, b0, b1, …, b7 be an 8-bit subword. "MIX, 8, L" puts all left subwords into R3, and "MIX, 8, R" puts all right subwords into R4.

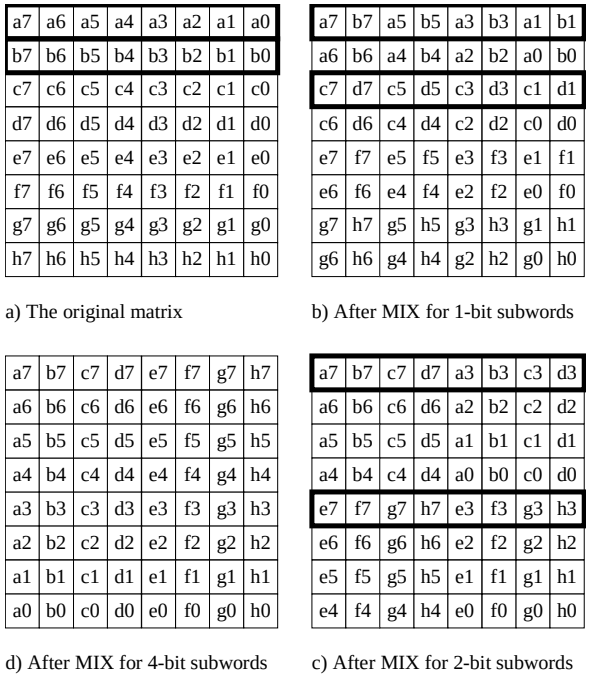
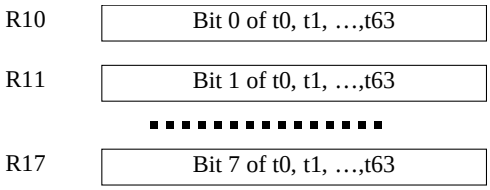


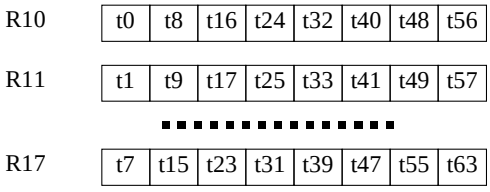
Figure 8: 8x8 matrix transpose with MIX

As proposed in [7], the MIX instruction is extended to support subwords that are powers of 2, including 1-bit subwords. We use Lee's method [2] to do bit matrix transpose in the pre-transpose step. Figure 8 shows how the pre-transpose of an 8x8 bit matrix can be done with 24 MIX instructions. The other seven matrices are transposed

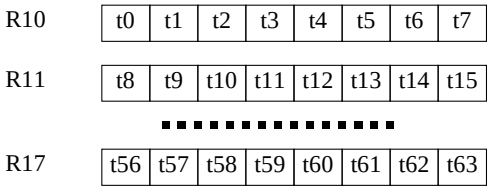
in parallel at the same time, courtesy of subword parallelism. Figure 8a is the original matrix. All bits in a subword are in the same register. Using "MIX, 1, L" and "MIX, 1, R" on pairs of registers, we can get the matrix shown in Figure 8b. Then, we generate the Figure 8c with "MIX, 2". Then, Figure 8d with "MIX, 4". In each step, the MIX instructions work on different pairings of registers. The first row of the matrix in Figure 8b to Figure 8d is generated by the pairs highlighted in Figure 8a to Figure 8c, respectively. The total number of MIX instructions used in Figure 8 is 24 because each row in Figure 8b, Figure 8c and Figure 8d is generated by one MIX instruction. Generally, the number of MIX instructions to do a k x k matrix transpose (singly or in parallel) is $k\lg(k)$. When eight 8x8 bit matrices packed into 64-bit registers are transposed in parallel, the number of MIX instructions required remains the same.



a) After sorting, where
(t0, t1, …, t63) = sort(w0, w1, w2,…,w63)



b) After bit matrix transpose



c) After subword matrix transpose

Figure 9: Post-transpose after sorting 8-bit subwords

After the pre-transpose step, sorting is done as shown in Figure 6. After sorting, a *post-transpose* step is needed to convert the sorted subwords back into the original format. Again, we use the MIX instruction. However, we need more instructions here than in the pre-transpose step. A subword matrix transpose needs to be done after the local bit matrix transpose (performed on all eight 8x8 bit matrices in parallel). Figure 9 gives an example of the post-transpose process for sorting 8-bit subwords. Figure 9a shows the register contents right after the sorting. R10 has bit 0 of all sorted subwords, R11 has bit 1 of all sorted subwords, and so on. Figure 9c is the format we need to store sorted subwords into memory. After performing eight 8x8 bit matrix transpositions in parallel on the eight registers in Figure 9a, all bits in each subword gather together as shown in Figure 9b. (This is identical to the pre-transpose step, using MIX instructions on 1, 2 and 4-bit subwords.) Figure 9b can now be considered a single 8x8 matrix where each element is an 8-bit subword. Figure 9b has to be converted to Figure 9c with an additional subword matrix transpose, which can be done again with the MIX instructions, this time on 8, 16 and 32 bit subwords, to get the final sorted list in the desired format for storing to memory.

Table 2 summarizes the number of instructions required for sorting subwords of different sizes using 64-bit registers. Basically we need to do pre-transpose, followed by sorting and post-transpose. If each subword consists of k bits and each register has r subwords, k^2 GRP instructions are required for sorting; $k\lg(k)$ MIX instructions for the pre-transpose; $k\lg(k)$ MIX instructions for the bit matrix transpose in the post-transpose; and $k\lg(k)$ (or $k\lg(r)$ when $k > r$) MIX instructions for the subword matrix transpose in the post-transpose.

The GRP instruction is useful for bit-level permutations, as well as for sorting subwords. Although it has not been shown, it appears that other permutation instructions like OMFLIP, PPERM and SIEVE are not as amenable for sorting since they do not incorporate the intrinsic sorting capability of GRP. However, the GRP instruction may be conceptually harder to implement than these other bit permutation instructions. In the next section we describe how it may be implemented, and then estimate its latency.

Subword size, k		**4**	**8**	**16**
# of subwords per register, r		16	8	4
# of registers for storing 64 subwords = k		4	8	16
# of instructions for pre-transpose		8	24	64
# of instructions for sorting		16	64	256
# of instructions for post-transpose	bit matrices	8	24	64
	subword matrices	8	24	32
Total # of instructions	**Total**	**40**	**136**	**416**
	with load/ store (+2k)	48	152	448

Table 2: Number of instructions for sorting 64 subwords using GRP and MIX on 64-bit registers

3. Hardware design of the GRP operation

A GRP operation is composed of three conceptual steps. Step 1 grabs input bits whose corresponding control

bits are 0. These bits are referred to as *z bits*. Step 2 grabs input bits whose corresponding control bits are 1. These bits are referred to as *w bits*. In Step 3, the results of the previous two steps are merged to get the result of the GRP instruction. The circuit that grabs z bits puts all z bits at the left end, and pads out the word with zeros. The same circuit can grab w bits with inverted control bits, and produce a result with w bits and padded zeros. Hence, the control bits are inverted when fed into the circuit for grabbing w bits. To easily combine the z bits and the w bits, we want the w bits at the right end with the padded zeros at the left. This allows a simple OR operation to combine the z bits from Step 1 with the w bits from Step 2, since all other bits have been set to 0. This can be achieved by either using the mirror image of the circuit in Step 1 for Step 2, or using the same circuit for both steps but providing data and control bits in the reverse order in Step 2. Also, the control bits are inverted in Step 2, as described earlier.

We use the divide-and-conquer strategy to grab z bits from *n* bits, as shown in Figure 10. First, the *n* input bits are divided into two halves of *n*/2 bits each. After putting z bits at the left end in each half, we combine the z bits in both halves, putting all z bits at the left end and setting the rest of the bits to 0. For each half of *n*/2 bits, we can apply the same method by dividing *n*/2 bits into two halves of *n*/4 bits. Each group of *n*/4 bits can be further divided until a 1-bit group has been reached. For groups of 1 bit, the z bit is already at the left end if the only bit is a z bit. Otherwise, we set it to 0. We call this a GRP1Z circuit.

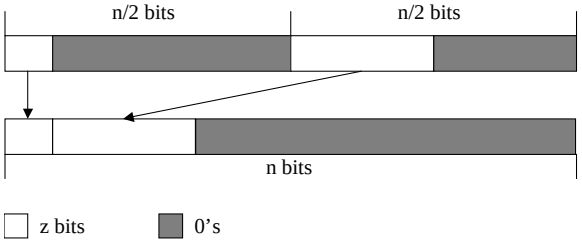


Figure 10: Grab z bits recursively

A GRP1Z circuit grabs z bits from 1-bit groups. A GRP2Z circuit consists of two GRP1Zs, and combines their outputs; this circuit is called a GRP2ZD circuit. GRP4Z consists of two GRP2Zs, and combines their results in a GRP4ZD circuit, and so on. Besides the data bits (z bits and padded 0s) generated in the previous stages, the combining circuit also needs to know the number of padded 0s in each set of data bits. We call the circuit that generates this information GRP1ZS, GRP2ZS, GRP4ZS, and so on, since they are similar to the GRP1ZD, GRP2ZD, …, circuits.

Figure 11 shows a diagram of GRP8ZD. The small boxes are the basic cell shown in Figure 12. It has a data input *i*, a data output *o*, and a select signal *s*. The output *o* is connected with the input *i* only when *s* = 1. A P-type transistor can be put in the basic cell for better signal at the output. In GRP8ZD, (I_0, I_1, I_2, I_3) and (I_4, I_5, I_6, I_7) are the outputs of two GRP4Z circuits. Both of them have the z bits at the left end and padded 0s at the right end. (S_4, S_3, S_2, S_1, S_0) is the one-hot encoded number of padded 0s in (I_0, I_1, I_2, I_3). Depending how many padded 0s are in (I_0, I_1, I_2, I_3), one of (S_4, S_3, S_2, S_1, S_0) is set to 1. That bit determines at which row the outputs are connected to the inputs. Padded 0s in (I_0, I_1, I_2, I_3) are replaced with bits shifting in from (I_4, I_5, I_6, I_7). For example, when (I_0, I_1, I_2) are z bits and I_3 is padded with 0, only S_1 is set to 1. The inputs and outputs are connected at the second row. The output (O_0,…,O_7) = (I_0, I_1, I_2, I_4, I_5, I_6, I_7, 0).

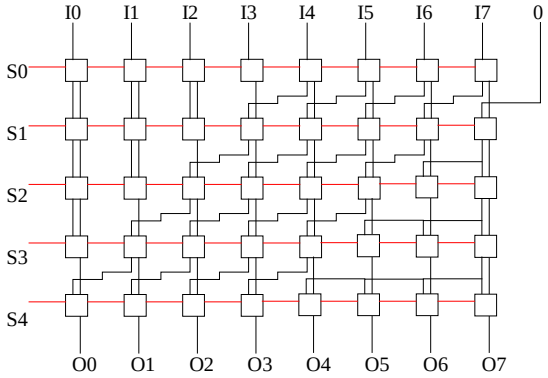


Figure 11: Diagram of GRP8ZD

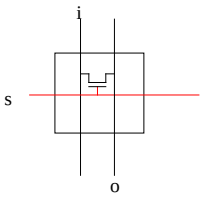


Figure 12: Basic cell

Figure 13 shows the block diagram of the datapath of GRP64, a GRP functional unit for 64 bits. We first use GRP1Z to generate z bits and w bits for 1-bit groups. Then, we keep combining the output of smaller groups to generate z bits and w bits for a larger group until we get all the z bits and w bits for the full 64 bits. Then, the z bits and w bits are combined with OR gates to get the result of the 64-bit GRP operation.

We use logical effort to estimate the latency of GRP64, in a technology independent manner. Logical effort is a method to estimate the delay in MOS circuits [20]. It can quickly determine a circuit's maximum possible speed and how to achieve it. The time unit used in logical effort is τ, the delay of an inverter driving an identical inverter with no parasitic capacitance.

Our calculations show that with aggressive design, the delay of GRP32 can reach $15\tau_4$, and the delay of GRP64

can reach $19\tau_4$, where $\tau 4$ is the delay of a fanout-of-4 (FO4) inverter, an inverter driving 4 identical inverters. A typical processor's cycle time is estimated at $16\tau_4$ [21]. Even if GRP64 can only finish in two cycles, the throughput can still be one operation per cycle if the circuit is pipelined. For example, a row of pipeline latches can be added before GRP32ZD or GRP64ZD in Figure 13 to divide the circuit into 2 execution stages, each taking one cycle.

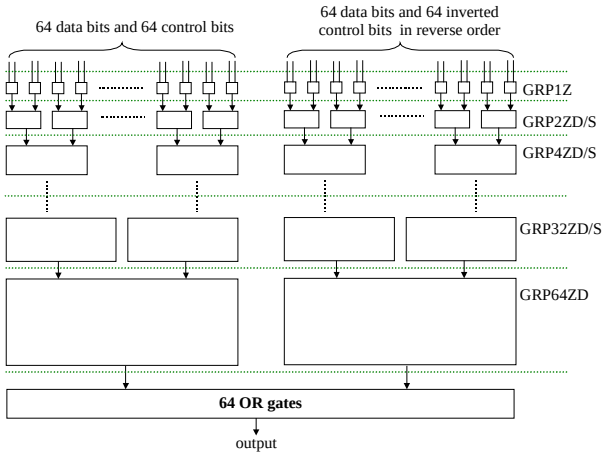


Figure 13: Hierachical structure of GRP64

4. Performance

We now discuss the performance of our sorting method versus popular sorting algorithms like bubble sort, selection sort and quicksort [18,19]. These algorithms are implemented assuming each element to be sorted occupies one memory word. Comparison is made to a basic RISC-like ISA, without subword parallelism.

Bubble sort is a common easily implemented sorting algorithm. In each iteration of bubble sort, we compare two neighboring elements in the unsorted list, and swap them if necessary, so that the largest element is moved to the bottom of the unsorted list at the end of each iteration. This largest element is not included in the unsorted list in the next iteration. In Selection sort, we maintain two lists, an unsorted list and a sorted list, and repeat the following until the unsorted list is empty: scan the unsorted list, pick up the largest element, and move it into the sorted list. The time complexity of both bubble sort and selection sort is $O(n^2)$.

Quicksort is a recursive algorithm with two phases: the partition phase and the sort phase. The partition phase divides the elements to be sorted into two parts, a lower part and an upper part, such that all elements in the lower part are smaller than those in the upper part; the sort phase sorts each part recursively. In the partition phase, we choose the element in the middle of the list as pivot and keep two pointers: one moving in form the left and the other from the right. They are moved towards the center until the left pointer finds an element greater than the pivot and the right one finds an element less than the pivot. These two elements are then swapped. The pointers are then moved inward again until they cross over. The average time of quicksort is $O(n \lg n)$.

We assume that the GRP functional unit is pipelined with a latency of two cycles. One GRP instruction can be issued, or completed, in each cycle. Tables 3 and 4 show the speedup of our sorting algorithms over traditional sorting algorithms described above. Our speedup estimates are conservative since we do not consider superscalar processors and cache misses. The conditional branch instructions incurred by the three methods above can degrade performance relative to our methods on a superscalar processor. Similarly, our methods require much fewer memory accesses and hence will perform better if cache miss cycles are counted.

Although bubble sort and selection sort are slow when sorting a large number of elements, their performance is comparable with quicksort when only a small number of items are sorted. Table 3 shows that quicksort is the slowest when we sort only 4 or 8 items, since our method has the largest speedup for quicksort. When sorting 8 bytes, GRP and BroadcastBit is 13 times faster. The speedup of GRP and BroadcastBit decreases when the subword size increases, since there are fewer subwords to sort, and increases as the subword size decreases, for more subwords are packed in a single 64-bit register.

# of subwords	16 4-bit	8 8-bit	4 16-bit
over bubble sort	89.9	12.2	1.7
over selection sort	65.5	9.8	1.4
over quicksort	62.1	13.3	3.0

Table 3: Speedup of GRP and BroadcastBit

Table 4 shows the speedup of GRP and MIX for sorting 64 values. When sorting 64 16-bit integers, our method achieves 10x speedup even when compared with the fastest quicksort. We achieve a 30x speedup when sorting 64 bytes, and 94x speedup when sorting 64 nibbles. The number of cycles taken in our method depends on the subword size. Compared to bubble sort of 64 bytes or nibbles, we are two orders of magnitude faster.

Subword size	4 bits	8 bits	16 bits
over bubble sort	408.3	128.9	43.7
over selection sort	272.7	86.1	29.2
over quicksort	94.4	29.8	10.1

Table 4: Speedup of GRP and MIX for 64 values

When sorting a larger number of elements, we can use our methods in recursive algorithms like quicksort or

merge sort in which the elements are divided and sorted recursively. For example, when sorting 128 elements in a 64-bit processor using merge sort, we sort two lists of 64 elements by our method. We then merge the two sorted lists, as in normal merge sort, by comparing the first elements of two lists and moving the smaller one to the new list.

For simplicity, we have only discussed sorting unsigned subwords. Sorting signed subwords needs only one more instruction: a NOT instruction to invert the control bits when subwords are sorted by the MSB. This has insignificant impact on the speedup.

5. Conclusions

This paper introduced a new method for sorting a small number of positive integers using the GRP bit permutation instructions, with support from either the MIX or the BroadcastBit instruction. Our methods achieve large speedups up to 408x compared with bubble sort, and from 10x to 94x compared with quicksort when sorting 64 integers. We show that it is possible to accelerate control intensive algorithms like sorting with subword parallelism, while eliminating conditional branches and conditional execution. Our methods are very fast for sorting up to *n* integers, where *n* is the word size of the processor. For sorting more elements, our method can be incorporated into recursive sorting algorithms, to speedup the sorting of small sets of *n* or fewer elements.

We have demonstrated that the GRP instruction is useful not only for fast permutations of *n* bits, but also for sorting *n* subwords. This versatility is important if GRP is to be included in a general-purpose processor. Our proposed implementation of GRP indicates that it can be done with a latency of 2 cycles, with 1 GRP instruction completed per cycle using a pipelined functional unit. Future work will investigate whether GRP can be implemented even more efficiently and its uses in other applications.

6. References

[1] Ruby Lee, "Accelerating Multimedia with Enhanced Microprocessors", *IEEE Micro*, Vol. 15, No. 2, 1995, pp.22-32

[2] Ruby Lee, "Subword Parallelism in MAX-2", *IEEE Micro*, Vol. 16, No. 4, 1996, pp.51-59

[3] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, Vol. 16, No. 4, 1996, pp. 10-20

[4] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing", *IEEE Micro*, Vol. 20, No. 2, 2000, pp. 85-95

[5] Stuart Obeman, Greg Favor, Fred Weber, "AMD 3Dnow! Technology: Architecture and Implementations", *IEEE Micro*, Vol. 19, No. 2, 1999, pp. 37-48

[6] Marc Tremblay and J. Michael O'Connor Venkatesh Narayanan and Liang He, "VIS Speeds New Media Processing", *IEEE Micro*, Vol. 16, No. 4, 1996, pp. 35-42

[7] Ruby B. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures", *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors* , July 10-12, 2000, pp. 3-14

[8] C. E. Shannon, "Communication Theory of Secrecy Systems", *Bell System Tech. Journal*, Vol. 28, Oct., 1949, pp. 656-715

[9] Bruce Schneier, *Applied Cryptography*, 2nd Ed., John Wiley & Sons, Inc.,1996

[10] National Bureau of Standards (NBS), "DATA ENCRYPTION STANDARD (DES)", *Federal Information Processing Standards Publication 46-2*, Dec 1993

[11] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128-Bit Block Cipher", June 1998, http://www.counterpane.com/twofish-paper.html

[12] Ross Anderson, Eli Biham and Lars Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", http://www.cl.cam.ac.uk/~rja14/serpent.html

[13] Zhijie Shi and Ruby B. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography", *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 138-148, July 10-12, 2000.

[14] Xiao Yang and Ruby B. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages", *Proceedings of the International Conference on Computer Design* , pp. 15-22, September 17-20, 2000.

[15] Ruby B. Lee, Zhijie Shi and Xiao Yang, "Efficient Permutation Instructions for Fast Software Cryptography", *IEEE Micro* , Vol. 21, No. 6, pp. 56-69, December 2001

[16] John P. McGregor and Ruby B. Lee, "Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications", *Proceedings of ICCD 2001 International Conference on Computer Design* , September 23-26, 2001, pp. 453-461

[17] Intel Corporation, *IA-64 Application Developers Architecture Guide*, Intel Corporation, May 1996

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1994

[19] C. A. R. Hoare, "Quicksort", *Computer Journal*, Vol. 5, NO. 1, 1962, pp.10-15

[20] Ivan Sutherland, Bob Sproull, David Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann Publishers, 1999

[21] Vikas Agarwal, M. S. Hrishikesh, S. W. Keckler, D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures", *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 248-259

[22] Priyadarshan Kolte, Roger Smith, Wen Su, "A Fast Median Filter Using AltiVec", *Proceedings of ICCD 1999 International Conference on Computer Design*, Oct 10-13, 1999, pp. 384-391