Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications

John P. McGregor and Ruby B. Lee
Department of Electrical Engineering
Princeton University
{mcgregor, rblee}@ee.princeton.edu

Abstract

We propose two new instructions, `swperm` *and* `sieve`, *that can be used to efficiently complete an arbitrary bit-level permutation of an n-bit word with or without repetitions. Permutations with repetitions are rearrangements of an ordered set in which elements may replace other elements in the set; such permutations are useful in cryptographic algorithms. On a 4-way superscalar processor, an arbitrary 64-bit permutation with repetitions of 1-bit subwords can be completed in 11 instructions and only 4 cycles using the two proposed instructions. For subwords of size 4 bits or greater, an arbitrary permutation with repetitions of a 64-bit register can be completed in a single cycle using a single* `swperm` *instruction. This improves upon previous permutation instruction proposals that require log(r) sequential instructions to permute r subwords of a 64-bit word without repetitions. Our method requires fewer instructions to permute 4-bit or larger subwords packed in a 64-bit register and fewer execution cycles for 1-bit subwords on wide superscalar processors.*

1. Introduction

As the popularity of security applications grows, the underlying cryptographic algorithms consume an increasingly large percentage of processor workloads. These applications often include several operations involving 1-bit or multiple-bit register subwords. Many microprocessor instruction set architectures have been extended to include subword arithmetic instructions that improve performance by executing several operations on low-precision data in parallel. These extensions include MAX [5] and MAX-2 [8] for HP PA-RISC, VIS [16] for Sun SPARC, AltiVec [3] for PowerPC, MMX [11] for Intel IA-32, and IA-64 multimedia instructions [4].

Before performing subword arithmetic operations, it may be necessary to rearrange the subwords within a single register or between multiple registers to obtain the desired result. In addition, subword permutations can be employed to efficiently perform transformations such as matrix transposition in multimedia applications [6]. Permutations are also used to achieve diffusion, a critical characteristic of a secure cipher [15], in symmetric-key encryption algorithms such as DES [10], Twofish [13] and Serpent [2]. Some permutations in cryptographic algorithms are not bijections. For instance, the Expansion Permutation in DES maps some bits in the source datum to multiple destinations in the result datum. We define such rearrangements of an ordered set in which elements can replace other elements in the set to be permutations with repetitions. If no information is lost in a permutation with repetitions, the permutation is invertible and therefore can be used in any cryptographic algorithm. In addition, one-way hash functions and encryption algorithms based upon Feistel networks can employ permutations with repetitions that lose information [12].

1.1. Past Work

Several methods exist for performing permutations in software. In one method, individual bits of the source datum are selected and shifted to their destination locations using a series of logical AND, logical OR, and shift instructions [18]. For an arbitrary permutation of the bits in an n-bit word, this procedure requires as many as $4n$ instructions. If the architecture includes instructions such as `extract` and `deposit` [7], one can reduce the instruction count of this procedure to $2n$, yet this method is still unacceptably slow.

Lookup tables can also be employed to perform permutations with repetitions in software [18]. First, the n-bit source datum is divided into x groups of bits; each group is used to index a unique lookup table. The output of a lookup table represents the input group of bits permuted per the desired permutation. The bits of the table output that do not represent any of the input bits are set to zeroes. Therefore, the outputs of the x lookup tables can be combined using (x-1) bitwise OR or bitwise XOR operations to generate the desired permuted n-bit result.

In general, assuming the extract instruction is available, the number of instructions required to complete an n-bit permutation using x lookup tables is $(3x-1)$. Each of the x lookup tables consists of $2^{(n/x)}$ entries, and each entry is n bits in size, so the total size of the tables is $(nx)\cdot 2^{(n/x)}$ bits. This technique is commonly used but is unattractive because the permutations must be statically encoded in the tables at compile-time. Furthermore, the space required to store the lookup tables is extremely large for acceptably small permutation instruction sequences. For example, 2 megabytes of storage are required to permute a 64-bit datum in 11 instructions using 4 lookup tables. With 8 lookup tables, 16 kilobytes of storage and 23 instructions are needed to permute a 64-bit value.

Multiple instruction set architectures have been amended to include instructions for permutations of 8-bit or larger subwords. The permute instruction in the MAX-2 extension to PA-RISC supports permutations with and without repetitions of 16-bit subwords in a 64-bit word by statically encoding the permutation function in the instruction [8]. In IA-64, the mux instruction supports a small set of permutations of 8-bit subwords in a 64-bit word and supports all permutations of 16-bit subwords in a 64-bit word [4]. Similar to permute in MAX-2, the permutation function is statically encoded in the mux instruction at compile-time. The vperm instruction in the AltiVec extension to the PowerPC instruction set architecture permutes the 8-bit subwords of a 128-bit vector register [3]. This instruction requires three 128-bit register reads and one 128-bit register write, and the permutation function is encoded in one of the vector source registers. None of the permutation instructions in popular ISAs efficiently support arbitrary permutations of 4-bit or smaller subwords.

Recently, several instructions for dynamically specified arbitrary permutations of 1-bit or larger subwords have been proposed. The pperm3r instruction can complete an arbitrary permutation of n bits with or without repetitions in $O(\log n)$ instructions using expensive hardware [14]. This instruction essentially dynamically configures and invokes an n x n crossbar without requiring the processor to maintain any additional state information. Amending an ISA by requiring additional state variables is undesirable: such changes require explicit OS support and increase the complexity of context switches and interrupts. In addition, the pperm3r instruction requires 3 register reads and 1 register write, and the number of pperm3r instructions required to complete an arbitrary permutation does not decrease as subword size increases.

The grp instruction can complete an arbitrary permutation without repetitions of n bits in $\log_2 n$ instructions [14]. The hardware needed to support the grp instruction is expensive, however. The cross instruction employs a Benes network to complete an arbitrary permutation without repetitions of b-bit subwords in an n-bit word using $\log_2(n/b)$ instructions [19]. The omflip instruction improves upon the cross instruction by completing arbitrary permutations without repetitions of b-bit subwords in an n-bit word with $\log_2(n/b)$ instructions using more efficient hardware [18]. Although cross and omflip can complete an arbitrary permutation without repetitions of 1-bit subwords quickly, these instructions cannot efficiently perform permutations with repetitions.

1.2. Outline

In this paper, we describe two instructions that accelerate the performance of subword permutations with repetitions. Since the development of DES, cryptographers have often avoided permutations of 1-bit subwords because general-purpose microprocessors cannot complete these operations quickly. By adding our proposed instructions to general-purpose ISAs, cryptographers can employ bit permutations with and without repetitions to rapidly achieve a desired level of diffusion in future ciphers. As a result, the proposed instructions could greatly improve the overall throughput of cryptographic algorithms.

In Section 2, we discuss the mathematics of permutations and define two new instructions. We demonstrate how to apply these instructions to achieve arbitrary subword permutations with repetitions in Section 3. In Section 4, we present the hardware required to implement the two new instructions, and we analyze the performance of permutations for different-sized subwords in Section 5. We summarize in Section 6.

2. Permutation Instructions

We propose two new instructions to efficiently support permutations with repetitions of 1-bit or multiple-bit subwords: swperm and sieve. These instructions allow permutations with repetitions to be dynamically specified during program execution rather than force the permutations to be statically encoded at compile-time.

2.1. Permutations with Repetitions

A permutation is a rearrangement of the elements in an ordered set, *i.e.*, a bijective map from a set S to itself [1]. We define a surjective map from a set S to another set D (where the cardinality of S equals that of D) to be a *permutation with repetitions*. In other words, a permutation with repetitions can map an element in the source set S to multiple elements in the destination set D, whereas a permutation without repetitions cannot map an element in S to more than one element in D. For example, if $S = \{a,b\}$, there exist 2 possible permutations of S, $\{a,b\}$ and $\{b,a\}$, but there exist 4 possible permutations with repetitions of S: $\{a,b\}$, $\{b,a\}$, $\{a,a\}$, $\{b,b\}$. We can encode a permutation with repetitions by specifying the source element in S that is mapped to a particular destination element in D for all the elements in D. If the permutation is arbitrary, the following expression describes the minimum number of bits needed to encode a permutation with repetitions:

$$\sum_{i=1}^{\|D\|} \log_2 \|S\|$$

We are concerned with permutations with repetitions of b-bit subwords from an n-bit source register to b-bit subwords of an n-bit destination register. Hence, $\|S\|$ is equivalent to the number of bits in the source register, n, divided by the subword size, b, and $\|D\|$ is the number of bits in the destination register, n, divided by the subword size, b. We can rewrite the expression as follows:

$$\sum_{i=1}^{\|D\|} \log_2 \|S\| = \sum_{i=1}^{n/b} \log_2 (n/b) = \frac{n}{b} \log_2 \left(\frac{n}{b} \right)$$

In this paper, we assume that all registers are 64 bits wide. Table 1 summarizes the minimum number of bits needed to specify an arbitrary 64-bit permutation with repetitions when using subword sizes ranging from 1 bit to 32 bits.

Table 1. Minimum number of bits needed to specify an arbitrary permutation with repetitions

Subword size	Number of subwords per 64-bit register	Number of bits to encode a 64-bit permutation with repetitions
32 bits	2 subwords	2 bits
16 bits	4 subwords	8 bits
8 bits	8 subwords	24 bits
4 bits	16 subwords	64 bits
2 bits	32 subwords	160 bits
1 bit	64 subwords	384 bits

RISC instructions typically allow two register reads and one register write per instruction. We wish to design instructions that allow permutations to be dynamically specified at run-time, so we use one of the 64-bit source registers, `rs`, to store the information to be permuted, and we use the other 64-bit source register, `rp`, to store information concerning the permutation function. For subwords of size greater than or equal to 4 bits, we require at most 64 bits of information to specify the entire permutation. Hence, we can specify the entire permutation in a single instruction. Since 64 more configuration bits can be specified with each additional permutation instruction, permutations of 32 2-bit subwords require at least 3 RISC instructions, and permutations of 64 1-bit subwords require at least 6 RISC instructions.

In the rest of this paper, we use the term "permutation" to mean a permutation with repetitions.

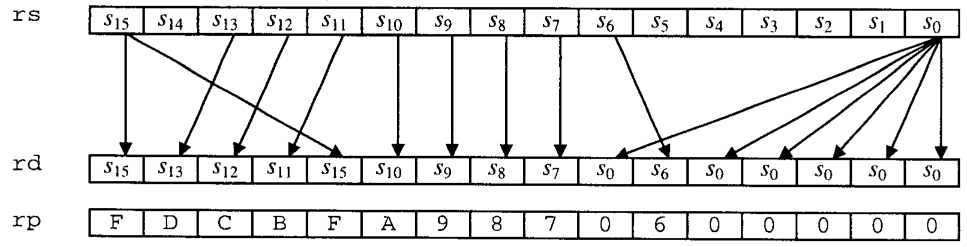
2.2. The `swperm` Instruction

The `swperm` instruction permutes the sixteen 4-bit subwords of a 64-bit source register `rs` according to information stored in a 64-bit source register `rp`. The permuted result is written to the 64-bit destination register `rd`. The permutation is entirely described with the information stored in `rp`, so the permutation function can be specified dynamically. The instruction format of `swperm` is:

```
swperm rd,rs,rp
```

This instruction was designed to permute subwords of size 4 bits or greater in a single cycle and to expedite permutations of 1-bit and 2-bit subwords. Figure 1 illustrates an example operation of `swperm`.

In Figure 1, s_i is the ith 4-bit aligned subword of the source register `rs`. The contents of `rp` necessary to complete the example permutation are listed in hexadecimal. The value of the ith 4-bit subword in `rp` indicates which aligned 4-bit subword in the source register should be mapped to the ith 4-bit subword in the destination register. The `swperm` instruction is similar to the MAX-2 `permute` instruction [8], but the configuration bits for `swperm` are specified in a register rather than statically in the instruction.



rs
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
rd
s_{15} s_{13} s_{12} s_{11} s_{15} s_{10} s_9 s_8 s_7 s_0 s_6 s_0 s_0 s_0 s_0 s_0
rp
F D C B F A 9 8 7 0 6 0 0 0 0 0

Figure 1. Example operation of the `swperm` instruction

2.3. The `sieve` Instruction

The `sieve` instruction is used to "filter" bits from `rs` and then direct the resulting bits into particular destinations in `rd`. More specifically, 1 (or 2 bits) from each 4-bit subword of `rs` are directed to 4 (or 2) possible locations in the corresponding 4-bit subword of `rd`. A third register, `rp`, is used to configure the bit filter. Whereas the `swperm` instruction operates globally over the 4-bit subwords of `rs`, the `sieve` instruction operates locally within the 4-bit subwords of `rs`. In combination with the `swperm` instruction, the `sieve` instruction can be used to implement arbitrary permutations of 1-bit or 2-bit subwords. The instruction format for `sieve` is:

```
sieve,h,f rd,rs,rp
```

The 4-bit function code of `sieve` consists of a 1-bit value, h (h), and a 3-bit value, f ($f_2f_1f_0$).

Figures 2 and 3 illustrate two example operations of the `sieve` instruction on the ith 4-bit subword of `rs` in 1-bit and 2-bit mode, respectively. In both figures, $s_{i,j}$ represents the jth bit of the ith subword of `rs`, and $d_{i,j}$ represents the jth bit of the ith 4-bit subword of `rd`. In 1-bit mode, one of the 4 bits in the ith subword of `rs` is directed to one of the 4 bits of the ith subword of `rd`; the remaining 3 bits in the ith subword of `rd` are set to 0. Similarly, in 2-bit mode, either the leftmost two bits or the rightmost 2 bits of the ith 4-bit subword of `rs` are directed to either the leftmost two bits or the rightmost two bits of the ith 4-bit subword of `rd`. The remaining two bits of the ith subword of `rd` are set to 0.

Bits from `rp` and the function code bit h specify which 1-bit or 2-bit subword is selected from the ith subword of `rs`. In 1-bit mode, 1 bit from every 4-bit subword of `rs` is selected and passed to `rd`. Hence, there exist 4 possible selection operations per `rs` subword, so 2 bits are needed to encode the selection operation for each subword. Since there are 16 4-bit subwords in a 64-bit register, a total of 32 bits are needed to encode the selection operations for all 16 subwords. These 32 bits are stored in the register `rp`. To minimize the number of instructions required to load the bit selection information into registers, one 64-bit register is used to store the 32 bits of selection information for two `sieve` instructions. The function code bit h indicates whether to use the leftmost or rightmost 2-bit half of each 4-bit `rp` subword to perform the `rs` bit selection.

In 2-bit mode, one of two 2-bit blocks from every 4-bit subword of `rs` is selected and passed to the corresponding subword of `rd`, so there exist two possible selection operations per subword. Hence, only 1 bit of information is needed to encode the selection operation for each subword, so a total of 16 bits are needed to encode the selection operations for a 64-bit register. These bits are stored in `rp`, and h indicates whether to use the left or right 2-bit halves of the 16 4-bit subwords in `rp`. The odd bits of the 2-bit halves store the configuration information; the 32 even bits of `rp` are ignored. To avoid wasting bits of `rp`, one could employ an additional function code bit to select one of four 16-bit groups of `rp` rather than select one of two 16-bit groups. This would not improve performance, however. Using the 2-bit mode of the `sieve` instruction, no more than two `sieve` instructions are required to complete a permutation of 2-bit subwords. Storing the bit filter configuration information in a single register for four rather than two `sieve` instructions therefore does not reduce the number of instructions required to load the configuration information into registers or reduce the total number of registers needed.

Figures 4 and 5 illustrate how h and the bits of `rp` are used to select bits from 4-bit subwords of `rs` in 1-bit and 2-bit mode, respectively. p_k is the kth bit of `rp`. If h is 1, the 2-bit value $p_{k+1}p_k$ that corresponds to the ith subword

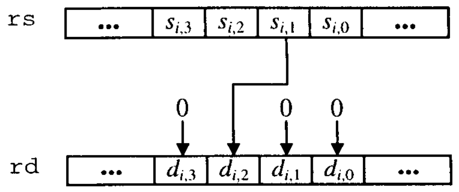


Figure 2. Example operation of the `sieve` instruction in 1-bit mode

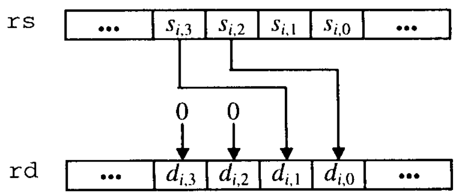


Figure 3. Example operation of the `sieve` instruction in 2-bit mode

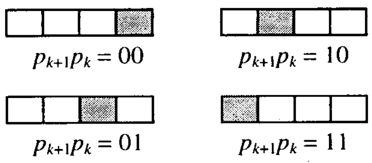


Figure 4. Bit selected from the ith 4-bit subword of `rs` by `sieve` in 1-bit mode

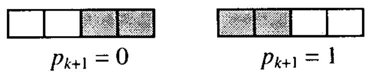


Figure 5. Bits selected from the ith 4-bit subword of `rs` by `sieve` in 2-bit mode

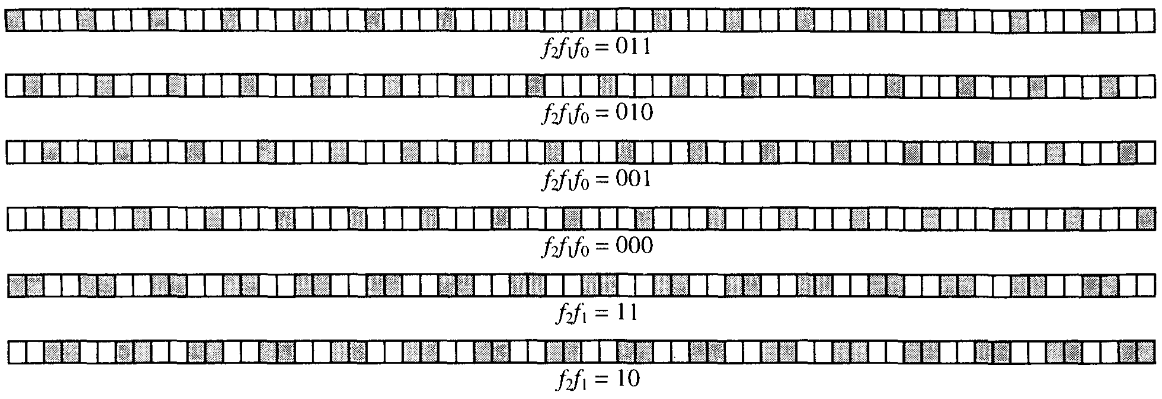


Figure 6. Effect of `sieve` function code bits on `rd`

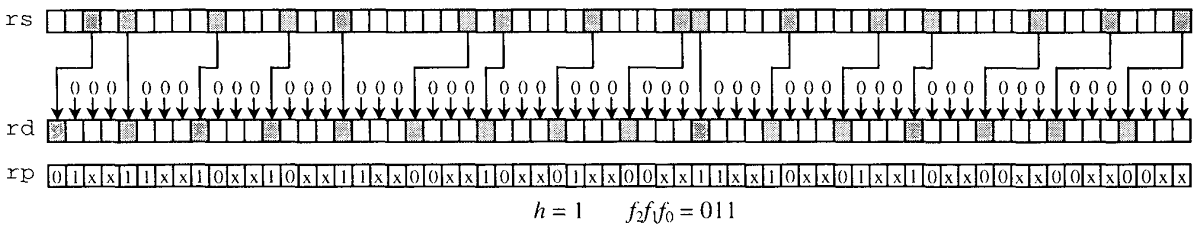


Figure 7. Complete example operation of `sieve`

of `rs` is selected from the leftmost 2-bit half of the ith subword of `rp`. Otherwise, $p_{k+1}p_k$ is selected from the rightmost 2-bit half of the ith subword of `rp`. Hence, the value of k is a function of h and the subword index i: $k = 2 \cdot h + 4 \cdot i$. The gray blocks indicate which bit or bits of the ith subword of `rs` are selected, and the white boxes indicate which bits of the ith 4-bit `rs` subword are discarded.

We now discuss how the bits $f_2f_1f_0$ of the function code are used to choose which bits in `rd` receive the selected bits from `rs` and which bits of `rd` are set to 0. f_2 indicates whether to use 1-bit or 2-bit mode. If 1-bit mode is employed, 3 bits out of every 4-bit subword of `rd` are set to 0, so a total of 48 bits of `rd` are set to 0. If 2-bit mode is used, 2 bits out of every 4-bit subword of `rd` are set to 0, so a total of 32 bits of `rd` receive zeroes. Bits f_1f_0 of the function code indicate which bit of each `rd` subword receives a selected bit from `rs` in 1-bit mode. In 2-bit mode, f_0 is ignored, and f_1 indicates which 2-bit half of each 4-bit `rd` subword receives selected bits from `rs`.

Figure 6 illustrates which bits of `rd` receive bits of `rs` given different values of the function code bits $f_2f_1f_0$. In the figure, the boxes containing 64 blocks represent the 64-bit register `rd`. The gray blocks represent bits that receive bits from `rs`; the white blocks represent the bits of `rd` that are set to zeroes. Counting from zero, the most significant bit (third bit) of each 4-bit subword is located on the left end of the subword, and the most significant 4-bit subword (fifteenth subword) of the 64-bit register is located at the left end of the register.

To summarize, the `sieve` instruction allows a single bit or an aligned pair of bits to be selected from each of the 16 4-bit subwords of the source register `rs`, but `sieve` only allows these selected bits to be mapped to the destination register `rd` in 1 of 6 possible ways, as shown in Figure 6. Figure 7 illustrates a complete example operation of the `sieve` instruction. For each of the registers, the least significant bit is located on the right end of the box representing the register. The gray blocks in the `rs` and `rd` boxes indicate which bits are selected and the locations where the selected bits are placed, respectively. The 64 bit values in the `rp` box specify the contents of the configuration register `rp` required to complete the example `sieve` operation. The right 2-bit halves of each 4-bit subword of `rp` have values of xx, *i.e.*, "don't care", because the value of h is 1.

3. Applying `swperm` and `sieve`

3.1. Permuting 1-bit and 2-bit Subwords

Using `swperm` and `sieve`, we can complete an arbitrary permutation of 64 1-bit subwords with 11 instructions as shown on the left side of Figure 8. We can perform an arbitrary permutation of 32 2-bit subwords with 5 instructions as shown on the right side of Figure 8.

In both cases, the 64-bit value to be permuted is initially stored in r1; upon completion, r1 will contain the desired permuted result. For 1-bit subwords, r5 through r10 store configuration information for the swperm and sieve instructions, and r1 through r4 are used to store intermediate values. For 2-bit subwords, r1 and r2 store intermediate values, and r3 through r5 store configuration information.

We assume the registers used to store configuration information are loaded with the appropriate data prior to the execution of these code segments. This pre-loading could require 6 or 3 memory load instructions for permutations of 1-bit or 2-bit subwords, respectively. In cryptographic algorithms, the same fixed permutation is often employed in every encryption or hash round. A round can usually be performed without spilling any registers to memory, so the 6 or 3 permutation configuration values could be loaded into general-purpose registers once before the execution of the thousands or millions of rounds required to encipher or hash kilobytes or megabytes of data. As a result, the cost of the loads would be negligible. Alternatively, these configuration registers may be intermediate encryption or hash results; therefore zero memory loads would be required.

To complete a permutation of 1-bit subwords, we first perform 4 permutations of 4-bit subwords using swperm. Upon completion of these 4 instructions, the subwords in registers r1, r2, r3, and r4 will contain the zeroth, first, second, and third bits of the corresponding subwords of the desired permuted result, respectively. For example, after execution of the first swperm instruction, 1 of the 4 bits contained in the *i*th subword of r2 will ultimately be placed in bit position 1 of the *i*th subword of the desired permuted result. Likewise, following the execution of the second swperm instruction, 1 of the 4 bits stored in the *i*th subword of r3 will eventually be placed in bit position 2 of the *i*th subword of the desired permuted result.

The four sieve instructions (in 1-bit mode) move 1 bit from every 4-bit subword of r1 through r4 to either the zeroth, first, second or third bit positions of the subwords in the destination registers. Upon completion of the sieve instructions, the desired permuted result is distributed across four 64-bit registers. The 16 bits in the zeroth position of each 4-bit subword in r1 are the bits that belong in the zeroth position of each subword in the desired result. The remaining 48 bits of r1 are set to zeroes by the first sieve instruction. Similarly, the bits located in the first positions of the 4-bit r2 subwords, the second positions of the 4-bit r3 subwords, and the third positions of the 4-bit r4 subwords belong in the first, second, and third positions of the corresponding subwords of the desired permuted result, respectively. The last 3 sieve instructions set the 144 bits in r2, r3 and r4 that do not correspond to bits of the desired result to zeroes.

```
      1-bit subwords                 2-bit subwords
swperm         r2,r1,r5        swperm         r2,r1,r3
swperm         r3,r1,r6        swperm         r1,r1,r4
swperm         r4,r1,r7        sieve,0,100    r1,r1,r5
swperm         r1,r1,r8        sieve,1,110    r2,r2,r5
sieve,0,000    r1,r1,r9        xor            r1,r1,r2
sieve,1,001    r2,r2,r9
sieve,0,010    r3,r3,r10
sieve,1,011    r4,r4,r10
xor            r1,r1,r2
xor            r3,r3,r4
xor            r1,r1,r3
```

Figure 8. Assembly code for performing permutations of 1-bit and 2-bit subwords

The top four 64-block boxes in Figure 6 illustrate this distribution of bits in r4, r3, r2, and r1. We collect the results of the 4 sieve instructions into a single register by performing 3 bitwise XOR (or bitwise OR) operations. Following the completion of the xor instructions, r1 will contain the 64-bit permuted result.

To permute 32 2-bit subwords packed into a 64-bit register, we use the same method but fewer instructions. The last two rows in Figure 6 show how the 64-bits of the desired permuted result are distributed over the two registers r2 and r1 after the sieve instructions complete. We can combine these two registers into the final 64-bit permuted result by performing a single xor (or a single or) instruction.

We developed an algorithm that generates the configuration registers for the swperm and sieve instructions given a list that represents a mapping from subwords in the source value to subwords in the permuted value. The algorithm runs in $O(n)$ time, where n is the number of bits in a register.

3.2. Permuting 4-bit or Larger Subwords

A permutation of 4-bit or larger subwords can be performed using a single swperm instruction. Given a register r1 that stores a 64-bit value to be permuted and a 64-bit register r2 that contains the configuration information necessary to conduct the permutation, the execution the following instruction completes a permutation of 4-bit subwords in a single cycle:

```
swperm r1,r1,r2
```

The swperm instruction stores the desired permuted result in r1. One can also complete 64-bit permutations of 8-bit, 16-bit, and 32-bit subwords by executing a single swperm instruction. 8-bit and larger subwords can be divided into 4-bit subwords, and it is trivial to translate a permutation encoding for 8-bit or larger subwords into a permutation encoding usable by swperm for 4-bit subwords.

4. Hardware Implementation

We now describe the CMOS hardware implementation for the `swperm` and `sieve` instructions. The Selection Unit enables the execution of the `swperm` instruction. We can implement the Selection Unit by building a 4-bit 16-to-1 multiplexer for every 4-bit subword in `rd`. Such a design is extremely expensive in hardware, however. Using a reduced crossbar, we can greatly decrease the transistor and wire cost. The reduced crossbar only requires 1 decoder for every 16 intersections between `rs` and `rd` tracks as opposed to 1 decoder for each intersection in a full crossbar.

We present an example cell of the reduced crossbar in Figure 9. Each cell consists of a 4-input AND gate, 4 n-type transistors, and 0, 1, 2, 3 or 4 inverters. s_i is the *i*th 4-bit subword of `rs`, d_j is the *j*th 4-bit subword of `rd`, and p_j is the *j*th 4-bit subword of `rp`. Recall that the `swperm` instruction directs the s_i to d_j if and only if p_j equals i. In the example cell, the leftmost and bottommost wires are the most significant bits of the subwords. From inspecting the negation bubbles on the inputs to the AND gate, we know that $i = 5$ in Figure 9. Hence, only the fifth 4-bit subword s_5 is enabled onto d_j. The other fifteen 4-bit subwords from `rs` similarly connected to d_j are not enabled onto d_j.

We now discuss the hardware cost of this implementation. The example cell is replicated 16 times for each of the 16 4-bit subwords of `rd`, so the total number of cells in the reduced crossbar is $16 \cdot 16 = 256$. On average, there are two negation bubbles on the inputs to the AND gate per cell, so the average number of transistors per cell is 16. Since there are 256 cells in the crossbar, the total transistor count is 4096. Furthermore, the number of vertical tracks is roughly the number of bits in `rs`, 64, and the number of horizontal tracks is the number of bits in `rd` plus the number of bits in `rp`, 128. The critical path of this circuit is at most the sum of the propagation delays of a 4-input AND gate, an inverter, an n-type transistor, a horizontal track, and a vertical track. Assuming the delays through the wires are not extremely high, we contend that the Selection Unit can complete an `swperm` instruction in a single cycle. In a deeply pipelined processor, however, the propagation delays through wires could force multiple-cycle execution of `swperm` instructions.

Figure 10 shows a 4-bit slice of the Filter Unit, which supports the `sieve` instruction. We can implement each 4-bit slice of the Filter Unit using four 1-bit 5-to-1 multiplexers. Using the implementation illustrated in Figure 10, however, we can reduce the transistor count without increasing the critical path length. The slice in the figure is replicated 16 times, once for each 4-bit subword in `rd`. The variable $s_{i,j}$ represents the *j*th bit of the *i*th subword of `rs`; the variable $d_{i,j}$ represents the *j*th bit of the *i*th subword of `rd`. Each 4-bit slice requires two 1-bit 2-to-1 multiplexers and four 1-bit 4-to-1 multiplexers. In addition, the *i*th subword slice includes a set of signals to control these multiplexers: A_i, B_i, C_i, D_i, E_i, and F_i. These signals are defined in Figure 11, where p_k is the *k*th bit of `rp`, and h, f_2, f_1, and f_0 are function code bits.

We can implement a 2-to-1 multiplexer using 4 transistors, and we can implement a 4-to-1 multiplexer using only 7 transistors each since the two lowest inputs are hard-wired to zeroes. Using buffers to reduce the fan-out of the function code bits and logic optimization techniques to reduce the transistor count, each 4-bit subword slice requires 116 transistors. The total number of transistors required for the 16 4-bit subword slices of the Filter Unit is 1856. All the data and control for each 4-bit subword slice in the Filter Unit is local, so no long

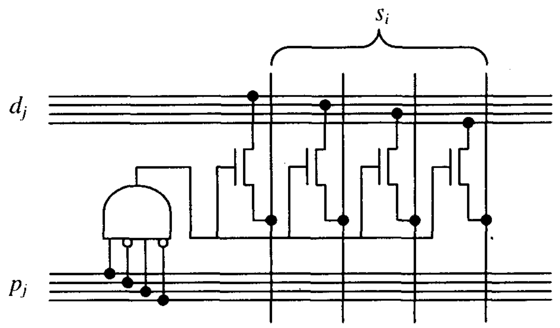


Figure 9. Selection Unit cell

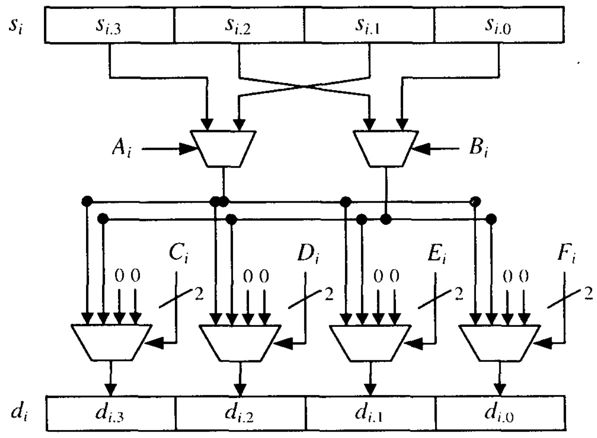


Figure 10. 4-bit slice of the Filter Unit

$$A_i = B_i = (h \cdot p_{4i+3}) + (\neg h \cdot p_{4i+1})$$
$$C_{i,1} = f_1 \cdot (f_2 + f_0) \qquad D_{i,1} = f_1 \cdot (f_2 + \neg f_0)$$
$$E_{i,1} = \neg f_1 \cdot (f_2 + f_0) \qquad F_{i,1} = \neg f_1 \cdot (f_2 + \neg f_0)$$
$$C_{i,0} = E_{i,0} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i}))) + f_2$$
$$D_{i,0} = F_{i,0} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i})))$$

Figure 11. Control signals in the Filter Unit

vertical or horizontal tracks are required. The critical path in the Filter Unit is the sum of the propagation delays through a 2-to-1 multiplexer, a 4-to-1 multiplexer, and the logic required to compute A_i. Therefore, it is highly likely that the Filter Unit can complete the execution of a `sieve` instruction in a single cycle.

The total number of transistors needed to implement a Permutation Unit, which consists of a Selection Unit and a Filter Unit, is 5952. This transistor count is of the same order of magnitude as that required to construct a simple 64-bit CMOS ripple-carry adder [17]. The total numbers of long horizontal and vertical tracks are 128 and 64, respectively. We compare the hardware cost of the Permutation Unit to past work in Table 2. Due to the imprecision of the track metric, we compare numbers of tracks using $O(\cdot)$ notation in terms of the number of bits in a register, n. When considering both transistor count and wire area, we argue that the Permutation Unit is as efficient as a VLSI implementation of the `omflip` instruction. The Permutation Unit requires nearly twice as many transistors as an `omflip` implementation, but it potentially consumes much less wire area due to constants hidden by the $O(\cdot)$ notation. The Permutation Unit also requires significantly fewer transistors and tracks than a crossbar network.

Table 2. Hardware cost comparison

Implementation	Horizontal Tracks	Vertical Tracks	Transistor Count
Permutation Unit (`swperm/sieve`)	$O(n)$	$O(n)$	5952
Omega-flip Network (`omflip`) [18]	$O(n)$	$O(n)$	3072
Crossbar Network [18]	$O(n)$	$O(n \log n)$	> 73,728

5. Permutation Performance

Table 3 summarizes the number of instructions, cycles and registers required to complete arbitrary permutations of different-sized subwords packed into a 64-bit register. For subword sizes of 4 bits or larger, only one `swperm` instruction and two registers are needed to complete an arbitrary 64-bit permutation with repetitions. Using both `sieve` and `swperm`, arbitrary 64-bit permutations with repetitions of 2-bit and 1-bit subwords require 5 and 11 instructions, respectively. In past work, Yang and Lee demonstrated that the `omflip` instruction could be used to complete 64-bit permutations without repetitions using 5 and 6 instructions, respectively [18]. These `omflip` instruction sequences must be executed serially, however. Therefore, even on an ultra-wide superscalar processor, a 64-bit permutation of 1-bit subwords without repetitions requires 6 cycles using the `omflip` instruction.

In cryptographic algorithms, operations performed on intermediate values are highly serialized. Therefore, the superscalar execution of the instructions involved in a permutation plays a major role in performance. The instruction sequences presented in this paper that employ `sieve` and `swperm` are highly parallelizable. True data dependencies do not exist between any of the `swperm` instructions or between any of the `sieve` instructions listed in Figure 8. Hence, the performance of an arbitrary 64-bit permutation with repetitions using these instructions may be limited by the issue width of the processor. On a 4-way superscalar processor, permutations of 1-bit and 2-bit subwords can be completed in as few as 4 and 3 cycles, respectively. Note that the performance improvement provided by `sieve` and `swperm` over existing methods on 2-way and 4-way superscalar processors requires 2 or 4 Permutation Units.

Table 3. Performance of 64-bit permutations using `sieve` and `swperm`

Subword size	Maximum # of instructions required	Minimum # of cycles for single-issue	Minimum # of cycles for 2-way superscalar	Minimum # of cycles for 4-way superscalar	Maximum # of registers required
32 bits	1	1	1	1	2
16 bits	1	1	1	1	2
8 bits	1	1	1	1	2
4 bits	1	1	1	1	2
2 bits	5	5	3	3	5
1 bit	11	11	6	4	10

Table 4. Permutation performance comparison

Instruction(s) used to perform a 64-bit permutation		Maximum # of instructions						# of cycles for 4-way superscalar					
	Subword Size	32 bits	16 bits	8 bits	4 bits	2 bits	1 bit	32 bits	16 bits	8 bits	4 bits	2 bits	1 bit
`sieve/swperm`		1	1	1	1	5	11	1	1	1	1	3	4
`pperm3r` [14]		8	8	8	8	8	8	4	4	4	4	4	4
`omflip` [18], `cross` [19], and `grp` [14]		1	2	3	4	5	6	1	2	3	4	5	6
Existing ISAs		1	1	1	23	23	23	1	1	1	10	10	10

Methods that employ cross, grp, and omflip only require 1 unit to achieve the cycle counts listed in Table 3.

We compare the performance of sieve and swperm to past work in Table 4. The table lists the number of instructions and cycles required by the different methods to complete a 64-bit permutation (such as the Initial Permutation in DES [10]). The bit values in the heading of the table indicate the size of the subwords to be permuted within a 64-bit word. We determine the cycle counts using a simulation of a 4-way superscalar processor with 4 integer execution units and a single load/store unit. The *Existing ISAs* row indicates the minimum number of instructions in conventional ISAs required to perform a 64-bit permutation using 8 lookup tables or existing permutation instructions.

If the pperm3r instruction is restricted to reading 2 registers rather than 3, 64-bit permutations require 15 instructions rather than 8 [9]. Furthermore, if 3 register reads per instruction are permitted, sieve can be trivially modified to reduce the total number of instructions required to perform permutations of 1-bit and 2-bit subwords by nearly a factor of 2. Other than sieve/swperm, only the pperm3r instruction is capable of completing permutations with repetitions; the omflip, cross and grp instructions only perform permutations without repetitions. Also, sieve and swperm do not scale as efficiently as grp in permuting values larger than 64 bits that are stored in multiple 64-bit registers. We observe that sieve and swperm perform as well as or better than all previously proposed permutation instructions and existing ISAs with the exception of the number of instructions required to complete a 64-bit permutation using 1-bit subwords.

6. Conclusion

In this paper, we proposed two 64-bit instructions for accelerating the performance of subword permutations with repetitions: swperm and sieve. Using these two instructions, we can complete 64-bit permutations with repetitions of 4-bit or larger subwords in 1 instruction. In addition, we can achieve permutations with repetitions of 1-bit and 2-bit subwords using 11 instructions and 5 instructions, respectively. These instructions are highly parallelizable, and a 4-way superscalar processor can execute these two instruction sequences in 4 cycles and 3 cycles, respectively. We also described hardware that efficiently implements swperm and sieve and can execute both instructions in a single cycle.

Using these instructions, cryptographers can design ciphers and hash algorithms that obtain a desirable level of diffusion more rapidly. As a result, less encryption rounds may be required to achieve adequate security, and the throughput of encryption algorithms could be significantly improved. Future work includes investigating the degree to which permutations with and without repetitions contribute to the security of a cipher.

References

[1] M. Artin, *Algebra*, Upper Saddle River, NJ: Prentice-Hall, Inc., 1991.

[2] E. Biham, R. Anderson and L. Knudsen, "Serpent: A New Block Cipher Proposal," *Proceedings of the 5th International Workshop on Fast Software Encryption*, Springer-Verlag, pp. 260-271, 1998.

[3] K. Diefendorff, et al., "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, pp. 85-95, March/April 2000.

[4] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, Intel Corporation, 1999.

[5] R. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22-32, April 1995.

[6] R. Lee, "Multimedia Extensions for General-purpose Processors," *Proceedings of the IEEE Workshop on Signal Processing Systems: Design and Implementation*, pp. 9-23, November 1997.

[7] R. Lee, "Precision Architecture," *IEEE Computer*, vol. 22, no. 1, pp. 78-91, January 1989.

[8] R. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16, no. 4, pp. 51-59, August 1996.

[9] R. Lee, Z. Shi and X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography," Princeton University Department of Electrical Engineering Technical Report no. CE-L01-001, 2001.

[10] National Bureau of Standards, "Data Encryption Standard," NBS FIPS Publication 46, January 1977.

[11] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42-50, August 1996.

[12] B. Schneier, *Applied Cryptography*, 2nd ed., New York, NY: John Wiley & Sons, Inc., 1996.

[13] B. Schneier, et al., *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

[14] Z. Shi and R. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography," *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 80-86, July 2000.

[15] D. Stinson, *Cryptography: Theory and Practice*, Boca Raton, FL: CRC Press, LLC, 1995.

[16] M. Tremblay, et al., "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, pp. 10-20, August 1996.

[17] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed., Reading, Massachusetts: Addison-Wesley, 1993.

[18] X. Yang and R. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages," *Proceedings of the International Conference on Computer Design*, pp. 15-22, September 2000.

[19] X. Yang, M. Vachharajani and R. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks," *Proceedings of SPIE: Media Processors 2000*, vol. 3970, pp. 80-86, January 2000.