

# Bit Permutation Instructions for Accelerating Software Cryptography

Zhijie Shi, Ruby B. Lee

*Department of Electrical Engineering, Princeton University*  
*{zshi, rblee}@ee.princeton.edu*

## **Abstract**

*Permutation is widely used in cryptographic algorithms. However, it is not well-supported in existing instruction sets. In this paper, two instructions, PPERM3R and GRP, are proposed for efficient software implementation of arbitrary permutations. The PPERM3R instruction can be used for dynamically specified permutations; the GRP instruction can be used to do arbitrary  $n$ -bit permutations with up to  $\lg(n)$  instructions. In addition, a systematic method for determining the instruction sequence for performing an arbitrary permutation is described.*

## **1. Introduction**

Secure information processing using cryptography is becoming increasingly important. Confusion and diffusion are two basic techniques used in cryptography algorithms [1]. Diffusion spreads the redundant information in the plain text over the cipher text. As a primary method to achieve diffusion, permutation is widely used in cryptographic algorithms [1]. For example, there are six different permutations in DES [1], two permutations in Twofish [8], and two permutations in Serpent [11]. Although it is not a problem for hardware to implement a pre-defined permutation [3], data-dependent permutations are not easy. Furthermore, there is no efficient way to do arbitrary permutations in software on most existing processor architectures.

Currently, several methods are used to perform permutation operations in software. The most straightforward method moves bits one by one [1]. We fetch one bit from the source, and put it into the correct position in the destination. We need  $4n$  operations for an  $n$ -bit permutation with this method on most existing microprocessors. The first two operations generate the mask and extract a bit with an AND instruction, the third moves the bit to the correct position with a SHIFT instruction, and the fourth sets the correct bit in the destination register with an OR instruction. This process can be accelerated on systems having instructions like EXTRACT and DEPOSIT [5][6], but still require  $2n$  operations. This method does not require an extra functional unit, and uses a small amount of memory, but it is expensive in execution time.

Another widely used technique is table lookup [4]. In this method, the source is divided into several sections. Then, the bits in each section are permuted simultaneously by looking up a table. Finally, we combine the result of each section to obtain the result of the permutation. The number of instructions in this method is dependent on how many sections we divide the source into. Fewer sections require fewer instructions. However, fewer sections also lead to more memory. For example, suppose we want to do a 64-bit permutation, we could do it with

one table lookup by putting all possible results in a large table of  $2^{64} \times 8$  bytes, which is too large to be feasible. Alternatively, we can divide the source into 8 sections, and build 8 tables. Each table has 2K bytes (256 entries and 8 bytes in each entry). Only 8 bits are permuted in one table lookup. Each table entry has 64 bits, and bits not permuted with that table are set to 0. Including the instructions for extracting indexes, we need 8 EXTRACTs [5][6], 8 table lookups and 7 ORs for one 64-bit permutation. This method is faster but consumes too much memory. The comparison of these two methods is shown in Table 1.

**Table 1: Traditional methods to do a 64-bit permutation on 64-bit systems**

	Number of instructions	Memory needed
AND,SHIFT,OR	256 ( $4n$ )	0
Table Lookup	23 (8 EXTRs, 8 LOADs and 7 ORs,)	16Kbytes (8 2K-byte tables)

Smart methods exist for some regular permutations. For example, the initial permutation in DES is similar to a transpose of bit matrices. It can be done in an efficient way with 30 operations on a 32-bit platform [1], rather than  $4n = 256$  instructions as in the first method described above. However, not all permutations are regular. Even for regular permutations, we do not have a systematic strategy to find the shortest sequence of instructions for an arbitrary permutation, which implies that we have to attack each permutation separately and can not guarantee finding the fastest solution.

Due to the difficulty in the software implementation of permutations, complex permutation operations are avoided when designing new algorithms. In this paper, we propose two novel permutation instructions that can perform arbitrary permutations of  $n$  bits efficiently. There has been limited past work on general-purpose permutation instructions. In [7], Lee proposed the first instruction-level support for general-purpose permutations in microprocessors: the MIX and PERMUTE instructions. One PERMUTE instruction can generate in a single cycle any permutation of four 16-bit subwords, including permutations with repetition of elements. The MIX instruction can combine the contents of two registers. If we extend it to handle 1-bit subwords, it can perform many regular permutations efficiently, such as the initial permutation in DES. In [2], Hansen proposed the group-8-mux and group-transpose-8-mux instructions. However, their instructions do 64-bit permutations on 128-bit systems, and require more than two operands. By proposing two general-purpose permutation instructions in this paper for arbitrary bit-level permutations, we hope to give cryptography algorithm designers more flexibility in designing faster and more secure encryption algorithms. Our permutation techniques can be implemented in any programmable processor, whether they are general-purpose microprocessor or application-specific cryptography processors. This paper is organized as follows: Section 2 provides some mathematical properties of permutations. Section 3 and section 4 introduce two new permutation instructions, PPERM3R and GRP, respectively. Section 5 compares the performance of these two permutation methods with the table lookup method. Section 6 concludes the paper.

## 2. Mathematical properties of permutations

Before we get into the details of the instructions, let us take a look at the mathematical properties of permutations. The number of  $n$ -bit permutations without repetitions is  $n!$ . According to Stirling's approximation [9][10],

$$n! = \sqrt{2\pi n} n^{n+\frac{1}{2}} e^{-n+\frac{q}{12n}} \quad (n > 0, 0 < q < 1) \quad (1)$$

When doing an  $n$ -bit permutation, we need to choose one result out of  $n!$  possibilities. At least, we should be able to distinguish all the  $n!$  possibilities. Thus, at least  $\lg(n!)$  bits need to be specified in a permutation operation. According to [10], we can prove that

$$n! = o(n^n) \quad (2)$$

$$\lg(n!) = \Theta(n \lg(n)) \quad (3)$$

which means that the number of bits needed to specify one permutation is on the order of  $n \lg(n)$ . Besides performing normal permutations without repeating any bit, we also need to do permutations, such as the expansion permutation in DES, in which some bits are replicated. The repetition of bits increases the number of possible permutations to  $n^n$ . Therefore, the number of bits required to specify a permutation of  $n$  bits with repetition is

$$\lg(n^n) = n \lg(n) \quad (4)$$

Most instruction-set architectures today have instructions with only two operands. If one of them specifies the  $n$  bits to be permuted, the other can only specify  $n$  bits of control information towards determining the required permutation. Consequently, the number of instructions required to specify any arbitrary permutation of  $n$  bits is  $\lg(n)$  instructions.

### 3. The PPERM3R instruction

An intuitive way to do permutations is to explicitly specify the position in the source for each bit in the destination. This is done by the Permute instruction in MAX-2 [7], where any subword in the source register can go to any subword position in the destination register. The Permute instruction currently supports only 16-bit subwords, so all position information can be encoded in 8 bits in the instruction. To perform bit-level permutation, we need to generalize the Permute instruction to support the subword size of 1. In this case, each bit is a subword and the number of subwords becomes  $n$ . Because there are  $n$  positions in the source, we need  $\lg(n)$  bits to represent each of them.  $n \lg(n)$  bits are required to specify the position information for all of the  $n$  bits. Obviously, all of these bits can not be specified in one instruction. So we have to use more than one instruction to do a single permutation, and with each instruction, we specify the permutation of a subset of bits. Suppose the source positions for  $k$  bits can be specified with one instruction, we define PPERM3R instructions as follows:

PPERM3R,  $x$  R1, R2, R3

R1 and R2 are the source registers, and R3 is the destination register. R1 contains the bits to be permuted; R2 contains the control bits.  $x$  specifies which  $k$  bits in R3 will change. In R3, only  $k$  bits specified by  $x$  are updated; other bits are copied from R1 unchanged. We use  $k \lg(n)$  bits in R2 to specify where to extract the  $k$  consecutive bits. In order to store the position information in one register, the following inequality should hold:

$$k \lg n \leq n \quad (5)$$

Therefore,

$$k \leq \frac{n}{\lg(n)} \quad (6)$$

Approximately  $n / \lg n$  bits can be specified with one instruction. In total, we need around  $n / k \approx \lg n$  instructions for an  $n$ -bit permutation.

Table 2 shows the number of instructions to perform one permutation on systems with different word sizes and PPERM3R instructions permuting different number of bits per instruction.

**Table 2: Number of the PPERM3R instructions for one permutation on different systems**

Register width $n$	$\lg(n)$	Maximum $k$ $\frac{n}{\lg n}$	Least number of instructions required	Power of 2 less than $k$	Typical number of instructions
32	5	6	6	4	8
64	6	10	7	8	8
128	7	18	8	16	8

As shown in Table 2, although the least number of instructions is around  $\lg(n)$ , the actual number may be greater than  $\lg(n)$  if we permute a subset of bits,  $k$ , that is rounded to the nearest power of 2.

The following codes give an example of using the PPERM3R instruction to do the initial permutation in DES. Suppose all registers are 64 bits in width. R1 is the source and R2 is the target. R10 through R17 are registers containing permutation control bits. We use 8 bits to represent the position of each bit in the source register. Thus, each instruction permutes a subset of 8 bits and 8 instructions are required to permute all 64 bits.

```

PPERM3R,0 R1, R10, R2 ; R10 = 0x3931292119110901
PPERM3R,1 R1, R11, R2 ; R11 = 0x3B332B231B130B03
PPERM3R,2 R1, R12, R2 ; R12 = 0x3D352D251D150D05
PPERM3R,3 R1, R13, R2 ; R13 = 0x3F372F271F170F07
PPERM3R,4 R1, R14, R2 ; R14 = 0x3830282018100800
PPERM3R,5 R1, R15, R2 ; R15 = 0x3A322A221A120A02
PPERM3R,6 R1, R16, R2 ; R16 = 0x3C342C241C140C04
PPERM3R,7 R1, R17, R2 ; R17 = 0x3E362E261E160E06

```

The first byte in R10, 0x39, indicates that the first bit in the target register R2 is bit 57 in the source register R1.

The advantages of PPERM3R are:

1. Besides normal permutations without any repetition of elements, permutations with repetitions can also easily be done with PPERM3R. For instance, the expansion permutation in DES is a transformation from 32 bits to 48 bits in which some bits are replicated.
2. Since the control bits for each PPERM3R instruction are very easy to generate, we can handle dynamically specified permutations. This is a feature of software-implemented permutations that is not easily achieved by reordering wires in hardware-implemented permutations.
3. If the PPERM3R functional unit allows storage of intermediate permutation results, then fewer instructions are needed as both source registers can carry control bits. A new PPERM3R\_ctl instruction is needed.

The PPERM3R instruction has the following disadvantages:

1. Because only part of the destination register is updated, PPERM3R requires that the register file either has a partial write capability, or has three read ports. In the latter case,

we read out the destination register, combine the updated bits with the unchanged bits, and write back a whole word. Either alternative is expensive.

2. Usually, we need the same number of instructions,  $\lg(n)$ , for all permutations of  $n$  bits, no matter how simple a particular permutation might be.
3. It is hard to perform a  $2n$ -bit permutation on  $n$ -bit systems. Two registers are needed to store  $2n$  source bits, and one more is needed for control bits. We need three source registers. Moreover, because the bits in the target register might come from any position in the source registers, we need more control bits to specify positions.
4. When the size of  $\lg(n)$  does not match  $n$  well, some bits in control register R2 are wasted. More than  $\lg(n)$  PPERM3R instructions are required for a permutation.

Because of the disadvantages of PPERM3R, we propose the GRP instruction.

#### 4. The GRP instruction

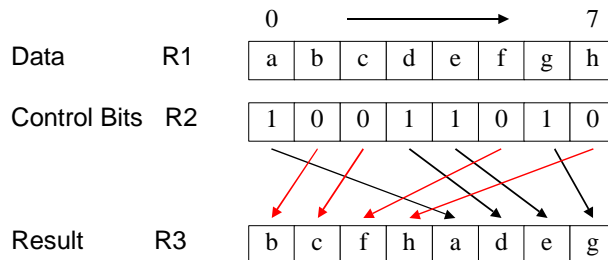
The GRP instruction looks like any two-operand, one-result instruction on typical microprocessors:

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

R1 and R2 are the source registers, and R3 is the destination register. If we use  $R1[i]$  to represent the  $i$ (th) bit of R1, the function of the GRP instruction can be described with pseudo codes as follows:

```
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]
```

The basic idea of the GRP instruction is to divide 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, we put this bit in R1 into the first group. Otherwise we put this bit in R1 into the second group. During this process we do not change the relative positions of bits in the same group. Finally, putting the first group to the left of the second group, we get the result value in R3. From the position of two groups, we call the first the left group, and the second the right group. Figure 1 shows how the GRP instruction works on 8-bit systems.



**Figure 1: The GRP instruction on 8-bit systems**

On  $n$ -bit systems, we can do any  $n$ -bit permutations with no more than  $\lg(n)$  GRP instructions. This can be proved by construction: we show the sequence of GRP instructions to

perform any arbitrary permutation, and we show how to determine the control bits for the GRP instructions in the sequence. But first, we introduce some definitions used to describe the algorithm.

To avoid confusion, we use *permutation* to represent the transformation we want to perform on a sequence of bits, and *arrangement* to represent the order of bits in a sequence. So, each permutation operation takes an arrangement as input, and produces another arrangement. The permutation determines the relationship between the initial arrangement and the final arrangement. We call the input the initial arrangement, and the output the final arrangement.

We use an integer sequence to represent an arrangement. An integer in the sequence denotes the position of the corresponding bit in the initial arrangement. For example, in an 8-bit permutation, the integer sequence (1, 2, 5, 7, 0, 3, 6, 4), which represents an arrangement, indicates that bit 0 in this arrangement is bit 1 in the initial arrangement. Similarly, bit 1 in this arrangement is bit 2 initially; bit 2 is bit 5 initially, and so forth. In an identity permutation, the final arrangement can be represented with a sequence (0, 1, 2, 3, 4, 5, 6, 7), which indicates that the bits in this arrangement come from the same positions in the initial arrangement. Since the  $n$ -bit initial arrangement can always be represented as (0, 1, 2, ...,  $n-1, n$ ), we use the integer sequence for the final arrangement to represent a permutation.

We define monotonically increasing sequences as follows:

**Definition 1:** Given an integer sequence  $b_1, b_2, \dots, b_i, \dots, b_j, \dots, b_n$ , its subsequence  $b_i, \dots, b_j$  is a *monotonically increasing sequence* (MIS) if  $b_i < b_{i+1} < b_{i+2} < \dots < b_j$ ,  $b_{i-1} > b_i$  or  $i = 1$ ,  $b_j > b_{j+1}$  or  $j = n$ .

For example, given a sequence (1, 2, 5, 7, 0, 3, 6, 4), we can find the following MISes in it: (1, 2, 5, 7), (0, 3, 6), (4). It is easy to see that any integer sequence can be considered a concatenation of a number of non-overlapping MISes. We define a function numMIS to get the number of MISes in an integer sequence.

**Definition 2:** *numMIS* is a function that takes an integer sequence as input, and returns the number of MISes in it.

For example,  $\text{numMIS}(1, 2, 5, 7, 0, 3, 6, 4) = 3$ ;  $\text{numMIS}(1,2,3,4) = 1$ ;  $\text{numMIS}(1,3,2,4) = 2$ ;  $\text{numMIS}(4,3,2,1) = 4$ .

Because each MIS has at least one number, it is easy to notice Fact 1.

**Fact 1:** The maximum value of numMIS is the length of the input sequence.

Because we can use an integer sequence to represent an arrangement, we also define the numMIS value of arrangements for convenience.

**Definition 3:** The *numMIS value of an arrangement* is the value of the numMIS function when it takes as input the integer sequence representation of the arrangement. We use  $\text{numMIS}(P)$  to represent the numMIS value of arrangement P.

For an  $n$ -bit permutation, the numMIS value of the final arrangement can not exceed  $n$ . The numMIS value of the initial arrangement is always 1.

**Fact 2:** Given an arrangement P, if  $\text{numMIS}(P) = k$  ( $k > 1$ ), there exists an arrangement Q, such that  $\text{numMIS}(Q) = \lceil k/2 \rceil$ , and P can be generated from Q with one GRP instruction.  $\lceil k/2 \rceil$  denotes the least integer not less than  $k/2$ .

Fact 2 inspires us to perform any arbitrary permutations of  $n$  bits with the GRP instruction. For example, let us consider a case where  $n = 8$ . We have an arrangement of 8 bits, which is the final arrangement of an 8-bit permutation:  $P_d = (7,6,5,4,3,2,0,1)$ .  $\text{numMIS}(P_d) = 7$ .

According to Fact 2, we can find an arrangement  $P_c$  and an instruction  $I_c$ , such that  $\text{numMIS}(P_c) = 4$  and  $I_c$  generates  $P_d$  from  $P_c$ . Similarly, we can find arrangements  $P_b$  and  $P_a$ , instructions  $I_b$  and  $I_a$ , such that  $I_b$  generates  $P_c$  from  $P_b$  and  $\text{numMIS}(P_b) = 2$ . Similarly,  $I_a$  generates  $P_b$  from  $P_a$  and  $\text{numMIS}(P_a) = 1$ . Since  $\text{numMIS}(P_a) = 1$ ,  $P_a$  is the initial arrangement. Hence, we can use  $I_a$ ,  $I_b$ , and  $I_c$  to perform the permutation corresponding to  $P_d$ . We shall show that Fact 2 holds by giving a method, which is described in Algorithm 1, to find  $Q$  and the GRP instruction that generates  $P$  from  $Q$ .

**Algorithm 1:** To generate one GRP instruction and the arrangement  $Q$

INPUT: Arrangement  $P$

OUTPUT: Arrangement  $Q$  and control bits  $c$  for GRP instruction

Let  $P_i$  represent the  $i$ (th) MIS in  $P$ .  $(x, y)$  denotes the operations that combine integer sequence  $x$  and  $y$  into a longer sequence.  $\text{Sort}(x)$  is a function that sorts elements in sequence  $x$  in increasing order.  $P$  can be represented by  $k$  MISes as follows:

$$P = (P_1, P_2, P_3, \dots, P_m, P_{m+1}, P_{m+2}, \dots, P_{k-1}, P_k)$$

Note that  $m = \lceil k/2 \rceil$ , and  $P_1, P_2, P_3, \dots, P_m$  is the first half MISes.

1. Generate temporary sequences  $T_1, T_2, \dots, T_m$ :

For  $i = 1, 2, \dots, m-1$

$$T_i = (P_i, P_{i+m})$$

If ( $k$  is odd) then

$$T_m = P_m$$

else

$$T_m = (P_m, P_k)$$

2. Generate  $Q$ :

For  $i = 1, 2, \dots, m$

$$Q_i = \text{Sort}(T_i)$$

Let  $Q = (Q_1, Q_2, Q_3, \dots, Q_m)$ .

3. Generate control bits  $c$ :

$Q$  can also be considered as a bit string:

$$Q = (Q_1, Q_2, Q_3, \dots, Q_m) = (b_0, b_1, b_2, \dots, b_{n-1})$$

For  $j = 0, 1, \dots, n-1$

if ( $b_j$  is in  $P_1, P_2, P_3, \dots$ , or  $P_m$ )

$$c_j = 0$$

else

$$c_j = 1$$

If  $R1$  and  $R2$  contain  $Q$  and  $c$ , respectively,  $P$  will be in  $R3$  after executing the following instruction:

$$\text{GRP } R1, R2, R3$$

Finally, we give Algorithm 2, which generates the GRP instruction sequence for a permutation. In Algorithm 2, starting from the final arrangement, we keep invoking Algorithm 1 to search some temporary arrangements that have a lower  $\text{numMIS}$  value until we get an arrangement whose  $\text{numMIS}$  value is 1. At the same time, we keep the control bits generated in Algorithm 1 in an array  $aC$ . Every time we call Algorithm 1, we reduce arrangement  $P$  to

an *easier* arrangement Q. When the process terminates, we get an arrangement whose numMIS value is 1, which means it is the initial arrangement.

**Algorithm 2:** To generate sequence of GRP instructions for a permutation

INPUT: The final arrangement of a permutation P

OUTPUT: *aC* : an array that stores the control bits for the GRP instructions that perform the permutation

*num\_instr* : the number of valid entries in *aC*

1. *num\_instr* = 0;
2. If (numMIS(P) == 1) return;
3. Call Algorithm 1 with P, and put the results of Algorithm 1 in Q and *tmp\_c*.
4. *aC[num\_instr ++]* = *tmp\_c*;
5. P = Q;
6. Goto step 2.

We know from Fact 2 that every time we generate a new arrangement, its numMIS value decreases by half. We also know that the maximum numMIS value of the final arrangement is its length *n*. It implies that we need at most  $\lg(n)$  steps to reduce the numMIS value of arrangements from *n* to 1. Thus, the number of iterations in Algorithm 2 does not exceed  $\lg(n)$  and the number of valid entries in the returned array *aC* does not exceed  $\lg(n)$ . For each entry in *aC*, we generate one GRP instruction. The total number of GRP instructions does not exceed  $\lg(n)$ .

When performing the permutation P, we start from the initial arrangement whose numMIS value is 1. Using the control bits kept in array *aC* from entry (*num\_instr* - 1) to entry 0, we generate the temporary arrangements in the reverse order, and achieve the final arrangement after the last GRP instruction.

Table 3 shows what happens on the example we mentioned above, an 8-bit permutation: (7,6,5,4,3,2,0,1).

**Table 3: Find out the GRP instruction sequence for an 8-bit permutation**

Iteration	1	2	3
<b>P</b>	(7,6,5,4,3,2,0,1)	(3, 7, 2, 6, 0, 1, 5, 4)	(0, 1, 3, 5, 7, 2, 4, 6)
<b>MISes in P</b>	(7)(6)(5)(4)(3)(2)(0,1)	(3,7)(2,6)(0,1,5)(4)	(0, 1, 3, 5, 7)(2, 4, 6)
<b>After Alg. 1, step 1</b>	(7,3)(6,2)(5,0,1)(4)	(3,7, 0,1,5)(2,6,4)	(0, 1, 3, 5, 7, 2, 4, 6)
<b>After Alg. 1, step 2</b>	Q = (3,7)(2,6)(0,1,5)(4)	Q=(0, 1, 3, 5, 7)(2, 4, 6)	(0, 1, 2, 3, 4, 5, 6, 7)
<b>After Alg. 1, step 3</b>	<i>c</i> = 10101100	<i>c</i> = 11010010	<i>c</i> = 00101010

The permutation can be performed with the following instruction sequence:

```
GRP  R1, R2, R1      ; R2 = 00101010 kept in aC[2]
GRP  R1, R3, R1      ; R3 = 11010010 kept in aC[1]
GRP  R1, R4, R1      ; R4 = 10101100 kept in aC[0]
```

Hence,  $\lg(8) = 3$  GRP instructions are sufficient to permute 8 bits.

Not only can the GRP instruction do an *n*-bit permutation efficiently, but it can also do *2n*-bit permutations with the help of an instruction like the SHIFT PAIR instruction in PA-RISC [5][6]. The SHIFT PAIR instruction can process operands that cross word boundaries. It concatenates two source registers to form a double-word value, then extracts any contiguous

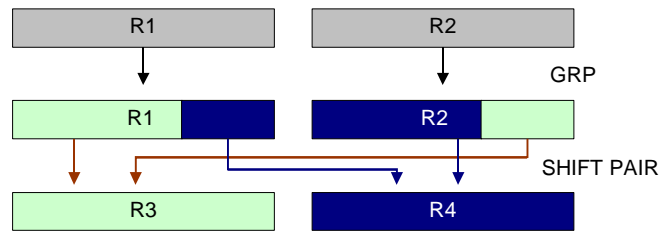


single-word value. Suppose R1 and R2 store the bits to be permuted, and the results will be put in R3 and R4. The following steps perform the  $2n$ -bit permutations:

1. Use GRP instructions to divide bits R1 and R2 into two groups. In R1, put the bits going to R3 into the left group, and the bits going to R4 into the right group. In R2, put the bits going to R4 into the left group, and the bits going to R3 into the right group. This can be done with two GRP instructions, one for R1 and one for R2.
2. Using two SHIFT PAIR instructions, we put all bits going to R3 into R3, and all bits going to R4 into R4. Two instructions are enough.
3. Considering R3 and R4 as separate  $n$ -bit words, do  $n$ -bit permutations on them. Each of R3 and R4 may need up to  $\lg(n)$  instructions.

Figure 2 shows what happens when we perform  $2n$ -bit permutations.

In total, excluding the instructions needed for loading control bits, we need  $(2\lg(n)+4)$  instructions to do a  $2n$ -bit permutation, which is only two instructions more than  $2\lg(2n)$ , the minimum number of instructions required for any  $2n$ -bit permutation on  $n$ -bit systems.



**Figure 2: Use the SHIFT PAIR and GRP instructions to do  $2n$ -bit permutations**

The GRP instruction has overcome all shortcomings of the PPERM3R instruction. It requires only two operands. It can do any permutations with  $\lg(n)$  instructions and do some permutations with fewer. Also, it can be used to do  $2n$ -bit permutations. The shortcomings of this instruction are that the hardware implementation is either slow or expensive, and currently, it does not support bit repetition.

## 5. Performance comparison

Table 4 shows the comparison of methods for performing permutations on 64-bit systems. PPERM3R and GRP have their own advantages. GRP is faster than PPERM3R, but currently, its hardware implementation is either expensive or has a long latency. The current ISA method and table lookup methods were described in section 1. Note that the 23 instructions in the table lookup method include 8 LOAD instructions, each of which can cause a cache miss, or other memory delay, resulting in an execution time that is much longer than that for 23 instructions.

**Table 4: Compare the PPERM3R and GRP instruction for permuting  $n=64$  bits**

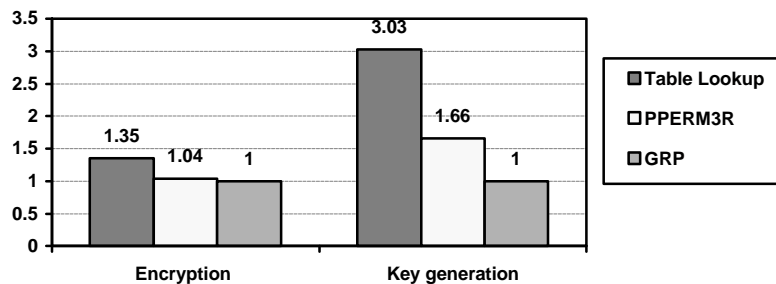
Instruction	PPERM3R	GRP	Current ISA	Table Lookup
Number of operands	3	2	2	2
Max number of instructions	8	6	256 ( $4n$ )	23(typically)
Do 128-bit permutations?	no	yes	yes	yes
Do permutations with bit repetition ?	yes	no	yes	yes
Hardware cost	mid	high	low	low

Table 5 shows the number of instructions needed to do permutations in DES with three different methods. The first four permutations are used in encryption and decryption. The expansion permutation and P-Box are in the loop, and need to be performed 16 times for each block. The other two permutations are used in key generation, and the compression permutation needs to be done 16 times. For the PPERM3R and GRP methods, the instructions needed for loading control bits are included. If the same permutation is repeated on many 64-bit data, then the control bits are loaded only once, and the cost of loading can be amortized over all these permutations. For the table lookup, we divide the source into sections of 8 bits each, and include the instructions that load the pointers to the permutation tables. We also assume that the instructions equivalent to EXTRACT and Load Indexed in PA-RISC are available [5][6]. As shown in Table 5, while the GRP instruction needs the fewest instructions, the table lookup needs the most.

**Table 5: Number of instructions required for permutations in DES**

	Table Lookup	Load control bits		Control bits already loaded	
		PPERM3R	GRP	PPERM3R	GRP
<b>Initial permutation</b>	31	16	12	8	6
<b>Final permutation</b>	31	16	12	8	6
<b>Expansion permutation</b>	15	12	9	6	5
<b>P-Box</b>	15	8	8	4	4
<b>Key permutation</b>	31	14	12	7	6
<b>Compression permutation</b>	27	12	10	6	5

The relative number of instructions required for the encryption and key generation in DES is shown in Figure 3. Smaller numbers are better. We do not give the number of cycles required by each method because this depends on the processor's microarchitecture. The left three bars are for the programs explicitly doing all 6 permutations using the specified permutation methodology. The table lookup method used by current microprocessors needs 35% more instructions for DES than the GRP method. Different optimized versions of DES may eliminate some of these permutations, resulting in different instruction counts. The right three bars are for key generation, where the computation is dominated by the two key permutations. Here, the table lookup method needs more than 3 times the number of instructions than when the GRP instruction is present.



**Figure 3: Number of instructions for DES encryption and key generation**

Table 6 shows the number of instructions needed to do permutations in Serpent with three different methods. We use the same assumption as in Table 5. Since the permutations are performed on 128-bit data, we need two LOADs and two ORs for each section of the table lookup.

**Table 6: Number of instructions required for permutations in Serpent**

	Table Lookup	GRP Load control bits	GRP Control bits already loaded
<b>Initial permutation</b>	110	26	14
<b>Final permutation</b>	110	14	8

The two 8-bit permutations in Twofish are simple. They can be done with one table lookup or one PPERM3R instruction, which is faster than the GRP instruction. Also, due to the regular structure of the function in each round of Twofish, permutation operations can be avoided in its software implementation.

## 6. Conclusion

We know that  $n \lg(n)$  bits are necessary and sufficient to specify an arbitrary  $n$ -bit permutation with bit repetition. Different methods can be used to specify these bits. Two instructions are proposed in this paper. The GRP instruction needs only two operands, and requires fewer instructions than the PPERM3R instruction to perform an arbitrary permutation. But it does not support bit repetition. The PPERM3R instruction can perform permutations with repetition of bits and can handle dynamically specified permutations, but it may need more instructions. Both of them can lead to a significant speedup on permutations in DES and in new symmetric key cryptographic algorithms using permutations.

## References:

1. Bruce Schneier, "Applied Cryptography", 2nd Ed., *John wiley & Sons, Inc.*, 1996
2. Craig Hansen, "Microunity's Media Processor Architecture", *IEEE Micro*, Vol. 16, No. 4, Aug 1996, pp.34-41
3. Frank Hoornaert, Jo Goubert, Yvo Desmedt, "Efficient Hardware Implementation of the DES", *Advances in Cryptology, CRYPTO '84 Proceedings*, Springer-Verlag, 1985, pp.147-173
4. Matt Bishop, "An Application of a Fast Data Encryption Standard Implementation", *Computing Systems*, Vol. 1, No. 3, 1988, pp.221-254
5. Ruby Lee, "Precision Architecture", *IEEE Computer*, Vol. 22, No. 1, Jan 1989, pp.78-91
6. Ruby Lee, Michael Mahon, Dale Morris, "Pathlength Reduction Features in the PA-RISC Architecture", *Proceedings of IEEE Comcon*, Feb 24-28, 1992, San Francisco, California, pp.129-135
7. Ruby Lee, "Subword Parallelism in MAX-2", *IEEE Micro*, Vol. 16, No. 4, 1996, pp.51-59
8. Bruce Schneier, John Kelsey, "Twofish: A 128-bit block cipher", <http://www.counterpane.com/twofish-paper.html>
9. Milton Abramowitz, Irene A. Stegun, "Handbook of Mathematical Functions", 9th printing, *US Department of Commerce, National Bureau of Standards*, November 1970
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", *The MIT Press*, 1994
11. Ross Anderson, Eli Biham and Lars Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", <http://www.cl.cam.ac.uk/~rja14/serpent.html>