Efficient Permutation Instructions for Fast Software Cryptography

Ruby Lee, Zhijie Shi and Xiao Yang
Department of Electrical Engineering
Princeton University

Abstract

To achieve pervasive secure information processing over the public wired and wireless Internet, it is desirable to be able to perform cryptographic transformations rapidly and conveniently. The performance of software-implemented cryptographic functions is hampered by certain operations which have not been optimized in the Instruction Set Architecture of processors, due to their infrequency in earlier programming workloads. One such operation is the permutation of bits within a block to be encrypted, which is particularly difficult in word-oriented processors. This paper introduces four novel permutation instructions and the underlying methodology for performing arbitrary *n*-bit permutations efficiently in programmable processors. While targeted at solving the more difficult problem of permuting *n* 1-bit elements, we also address the issue of permuting a smaller number of multi-bit subwords packed into an n-bit word, a feature needed to accelerate multimedia processing in software. By providing the ability to do fast permutations in software, we open the field for new cryptography and multimedia algorithms using these powerful yet simple permutation primitives. This results in much faster cryptography and multimedia processing, while retaining the flexibility of software implementations, for secure multimedia information appliances and servers.

1. Introduction

The need for secure information processing increases with the increasing use of the public internet and wireless communications in e-commerce, e-business and many other aspects of our daily lives. The internet today is not secure. Secure information processing includes authentication of users and host machines, confidentiality of messages sent over public networks, and assurances that messages, programs and data have not been maliciously changed. These security functions can be achieved by using different security protocols employing different cryptographic algorithms, such as public key, symmetric key and hash algorithms. To facilitate pervasive secure information processing, it is important that such security functions be implemented in a fast and painless way, otherwise users will not use them. One way to achieve this is to perform the cryptographic functions very rapidly with software, rather than requiring additional special-purpose hardware chips and boards. For the purposes of this paper, we assume that some programmable processor is available in an information appliance or server machine, in order to perform regular (non-secure) transactions over the Internet. What general-purpose operations can be added to this programmable processor, or as a cryptography coprocessor, so that it can execute cryptographic functions at network link speeds, or at least with insignificant performance degradation? These are operations frequently used in encrypting and decrypting information, which were not frequent in general-purpose programming workloads before the need for pervasive secure information processing. In this paper, we focus on one set of operations, permutations, used in encrypting large amounts of data for confidentiality or privacy. Bit-level permutations are particularly difficult for processors, so they tend to be avoided in the design of new cryptography algorithms, for example the Advanced Encryption Standard [1], where it is desired to have fast software implementations. In this paper, we show how very efficient bit permutations can be achieved in programmable processors, thus enabling future cryptography algorithms to be designed using the superior diffusion [2] capabilities of permutations if desired. In addition, we also show how these same permutation instructions can also be used to accelerate subword parallel processing in multimedia applications [3,4].

For encrypting large amounts of data, symmetric key cryptography algorithms are used [2]. These algorithms use the same secret key to encrypt and decrypt a given message, and encryption and decryption have the same computational complexity. In symmetric key algorithms, the cryptographic techniques of confusion and diffusion are synergistically employed. *Confusion* obscures the relationship between the

plaintext (original message) and the ciphertext (encrypted message), for example, through substitution of arbitrary bits for bits in the plaintext. *Diffusion* spreads the redundancy of the plaintext over the ciphertext, for example through permutation of the bits of the plaintext block. Such bit-level permutations are very slow when implemented with the current instructions available in microprocessors and other programmable processors.

Permutations are also an important new requirement for fast processing of digital multimedia information, using subword-parallel instructions [3,4], more commonly known as multimedia instructions. Every major microprocessor Instruction Set Architecture (ISA) now has these subword parallel instructions [3-10] for fast multimedia information processing. With subwords packed into 64-bit words, it is often necessary to rearrange the subwords within the word. However, such subword permutation instructions are not provided by many of these multimedia ISA extensions, except for a few initial attempts in MAX-2 [4], IA-64 [7] and Altivec[8].

In this paper, we propose innovative permutation instructions for solving the permutation problems in both cryptography and multimedia. For fast cryptography, bit-level permutations are required, whereas for multimedia, permutations on subwords of typically 8 bits or 16 bits are required. Since the complexity of permutations increases with the number of elements (subwords) being permutated, we solve the most complex problem of allowing any arbitrary permutation of sixty-four 1-bit subwords in a 64-bit processor, i.e., a processor with 64-bit words, registers and datapaths, for fast cryptography. We then show how these instructions can also be used for permuting subwords greater than 1 bit in size, for fast multimedia processing. For example, in addition to being able to permute sixty-four 1-bit subwords in a register, the permutation instructions and underlying functional unit should also be able to permute thirty-two 2-bit subwords, sixteen 4-bit subwords, eight 8-bit subwords, four 16-bit subwords, or two 32-bit subwords. These powerful and flexible permutation instructions may be added as new instructions to the Instruction Set Architecture of an existing microprocessor, or they may be used in new processors or coprocessors that are designed from scratch to be efficient for both cryptography and multimedia software.

1.1 Past work

Since current microprocessors are word-oriented, performing bit-level permutations is very painful. Every bit has to be extracted from the source register, moved to its new location in the destination register, and combined with the bits that have already been moved. This requires 4 instructions per bit (mask generation, AND, SHIFT, OR), and 4*n* instructions to perform an arbitrary permutation of *n* bits. With certain microprocessors like PA-RISC [11], more powerful bit-manipulation instructions exist, such as the EXTRACT and DEPOSIT instructions, which can essentially perform the four operations required for each bit in 2 instructions (EXTRACT, DEPOSIT), resulting in 2*n* instructions for any arbitrary permutation of *n* bits. Pre-defined permutations with some regular patterns can of course be done in fewer instructions, for example, the permutations in DES [2]. But, in general, an arbitrary 64-bit permutation could take 128 or 256 instructions on current microprocessors.

Since this is unacceptably slow, table lookup methods have been used to implement fixed permutations. To achieve a fixed permutation of *n* input bits with one table lookup, we would need a table with 2^n entries, each entry being *n* bits. For a 64-bit permutation, this would be 2^{67} bytes, which is clearly infeasible. The table would have to be broken up into smaller tables, and several table lookup operations would be needed. For example, a 64-bit permutation could be implemented by permuting 8 consecutive bits at a time, then combining these 8 intermediate permutations into the final permutation. This requires 8 tables, each with 256 entries, each entry being 64 bits. Each entry would have zeros in all positions, except the 8 bit positions to which the selected 8 bits in the source are permuted. After the eight table lookups done by 8 LOAD instructions, we have to combine the results with 7 OR instructions to get the final permutation. In addition, 8 instructions are needed to extract the index for the LOAD instruction, for a total of 23 instructions. The memory requirement is 8*256*8=16 kilobytes for eight tables. Although 23 instructions sounds considerably better than the 128 or 256 instructions in the previous method, the actual execution time could be much longer due to cache miss penalties or memory access latencies. Suppose half of the 8 Load instructions miss in the cache, and each cache miss takes 50 cycles to fetch the missing cache line

from main memory. Then, the actual execution time is more than 4*50=200 cycles, which is actually longer than the previous 128 cycles using EXTRACT and DEPOSIT. Also, the memory requirement of 16 kilobytes for just the tables is not insignificant.

For subword permutation instructions, MAX-2 has a general-purpose PERMUTE instruction which can do any permutation, with and without repetitions, of the subwords packed in a register. However, it is only defined for 16-bit subwords. IA-64 also has the MUX instruction, which is a fully general permute instruction for 16-bit subwords, with five new permute byte variants. Altivec has the VPERM instruction, which extends the general permutation capabilities of MAX-2's PERMUTE instruction to 8-bit subwords selected from two 128-bit source registers, into a single 128-bit destination register. Since there are 32 such subwords from which 16 are selected, this requires 16*lg32=80 bits for specifying the desired permutation. This means that VPERM has to use another 128-bit register to hold the permutation control bits, making it a very expensive instruction with three source registers and one destination register, all 128 bits wide. In [18], Lee proposes a small set of subword permutation primitives useful for two-dimensional multimedia processing. None of the subword permutation instructions defined so far can perform arbitrary bit-level permutations efficiently.

In this paper, we present significantly faster and more economical ways to perform arbitrary permutations of n bits, without any need for table storage. In searching the mathematical literature on permutations, we could not find any literature on the ***efficiency*** of achieving arbitrary permutations. The problem of performing any arbitrary n-bit permutation in the minimum number of steps using permutation primitives (encoded in instructions) is an unsolved problem so far. Hence, in Section 2, we investigate the minimal number of steps needed to generate any arbitrary permutation of n bits. But what are the permutation primitives needed for these efficient permutations? In Section 3, we propose four alternative methods for permuting n bits, each based on a new permutation instruction that we define. In Sections 4 and 5, we compare the hardware complexity and performance of these permutation methods with each other and with current microprocessors. In section 6, we consider how arbitrary n-bit permutations may be specified in a high-level language. In section 7, we summarize the paper.

2. Minimal number of steps for arbitrary n-bit permutations

The number of n-bit permutations without repetitions is $n!$. According to Stirling's approximation [12,13], we can prove that

$$n! = o(n^n) \tag{1}$$

and

$$\lg(n!) = \Theta(n \lg(n)) \tag{2}$$

which means that the number of bits needed to specify one permutation is on the order of $n\lg n$. We also need to do permutations 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

$$\lg(n^n) = n \lg(n) \tag{3}$$

Assume we conform to the standard datapath arrangements of current microprocessors with general-purpose registers, where each instruction has two source registers and one destination register (see figure 1), and no extra state is stored between the permutation instructions. Then, one source register must be used for the n bits to be permuted, and the destination register gives an intermediate result leading to the final desired permutation. Since the second source register can hold only n bits of permutation specification, and $n\lg n$ bits are needed to specify an n-bit permutation, $\lg n$ permutation instructions are needed. Hence, the minimum number of instructions needed for performing any arbitrary n-bit permutation

is lg$n$ instructions, if each instruction has only two source registers and one destination register, and no extra state is stored. Similarly, the minimum number of instructions needed for performing any 2n-bit permutation, using n-bit registers, is (2n*lg(2n)/n)=2lg$n$+2.

3. Alternative permutation methodologies

We start with bit-level permutations of all n bits in a register, for n=64, since microprocessors today have registers and datapaths that are either 32 or 64 bits wide, and common symmetric key block ciphers like DES and Triple DES [2] encrypt 64-bit blocks. We describe four methods:

- PPERM, a new incarnation of the PERMUTE instruction, first introduced in MAX-2 [4],
- GRP, a new approach that separates bits into left and right parts,
- CROSS, a new approach using Benes interconnection network theory, and
- OMFLIP, a new approach that uses Omega-Flip interconnection network theory.

In each case, the same solution can be applied to different subword sizes of 2^i bits, for i=0, 1, 2, …, m, where n=2^m bits. For a fixed word size of n bits, clearly the hardest permutation problem is for 1-bit subwords, where there are n subwords to be permuted. For n=64 bits, we show how the solution for such bit-level permutations can also be used for subword permutations of fewer multi-bit subwords, e.g., eight 8-bit subwords, commonly used in multimedia programs. We also discuss how the solution can scale up from n=64 to n=128 bits, since future cryptography algorithms are being designed to encrypt 128-bit blocks [1]. We also consider if the permutation methodology easily allows permutations where bits can be repeated or omitted. All the four permutation instructions fall into the conventional 2-source, 1-result execution paradigm. However, new permutation functional units are needed to support these new instructions. These permutation functional units fit into a microprocessor's datapath the same way as an ALU does. This is shown in Figure 1.

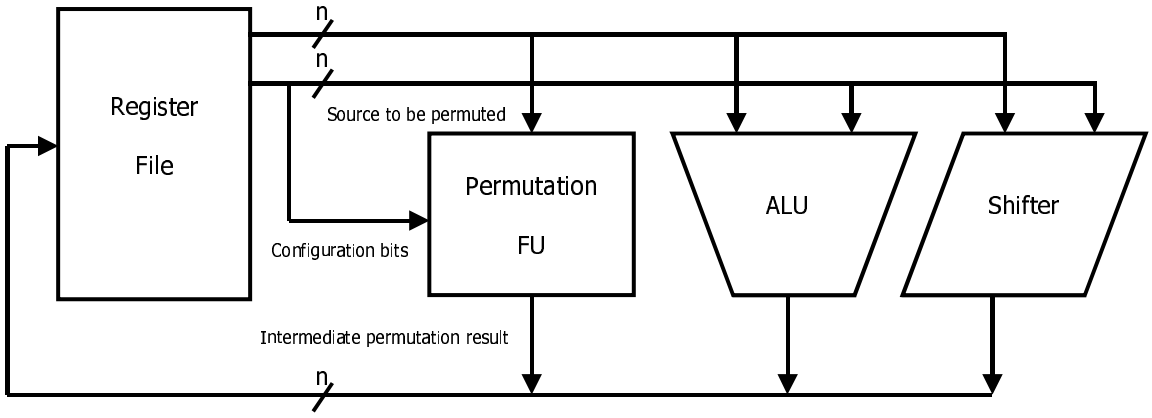


Figure 1: Standard Microprocessor Datapath with a new Permutation Functional Unit

3.1 PPERM permutation 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 [4], 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 eight bits in the instruction. To perform bit-level permutation, we need to generalize the PERMUTE instruction to support the subword size of one. In this case, each bit is a subword and the number of subwords becomes n. Therefore, $n$lg$n$ bits are required to specify the position information for all these n bits. Obviously, they cannot be specified in one instruction. We have to use more than one instruction to do a single permutation. 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 PPERM instructions as follows

PPERM,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 configuration bits. x specifies which *k* bits in R3 will change. In R3, only *k* bits specified by x are updated, the other bits are set to zero (see Figure 2). 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

$$k \lg n \le n \tag{4}$$

Therefore,

$$k \le \frac{n}{\lg n} \tag{5}$$

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. When *n*=64, we can choose *k*=8. Therefore, we need eight PPERM instructions for a 64-bit permutation, and seven OR instructions to merge these results to get the desired permutation. The pseudo code for the operation performed by PPERM instruction on 64-bit architecture is shown in Table 2.

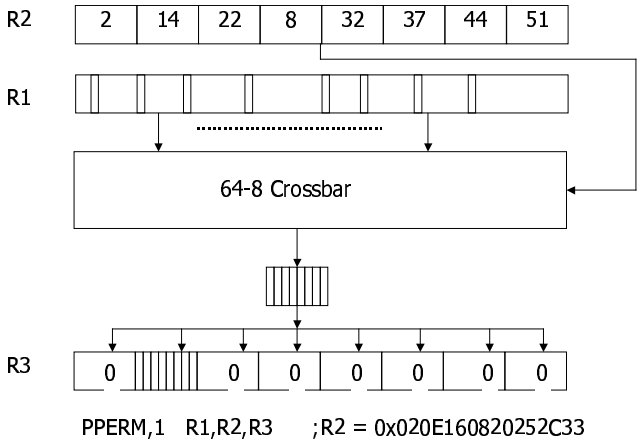


Figure 2: Diagram of PPERM instruction

Specifying the configuration bits for the PPERM instruction is straightforward. For every one of the *k* bits to be changed in the final permutation, we use $\lg n$ bits to specify which bit in the source register should go there.

To allow PPERM to permute bits from two or more source registers, an extra bit (denoted "otherreg") is used to select each bit in the source register. For n=64 bits, each index into the source register is now ($\lg n$ +1) = 7 bits. If this "otherreg" bit = 0, then the remaining 6-bit index selects a bit in the source register to place in the destination register, as described above. However, if this "otherreg" bit = 1, then the corresponding bit in the destination register is forced to zero. This allows two or more registers to be used as source registers, in different PPERM instructions.

3.2 GRP permutation instruction

The GRP instruction also looks like any two-operand, one-result instruction on typical microprocessors (see Figure 1):

GRP R1, R2, R3

R1 and R2 are the source registers, and R3 is the destination register. The operation of a GRP instruction is shown as pseudo code in Table 2.

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 from R1 into the first group. Otherwise we put this bit into the second group. During this process we do not change the relative positions of bits in the same group. Finally, by putting the first group to the left of the second group, we get the result value in R3. Figure 3 shows how the GRP instruction works on 8-bit architectures.

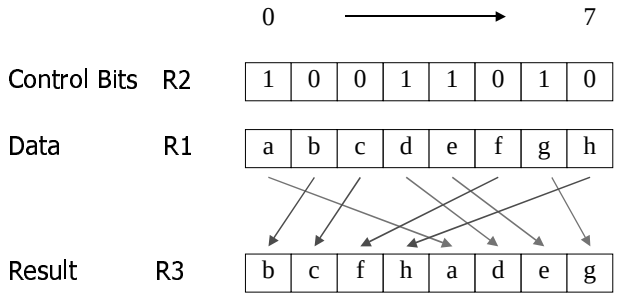


Figure 3:The GRP instruction on 8-bit architectures

On an *n*-bit architecture, we can do any *n*-bit permutations with no more than lg*n* GRP instructions. This can be proved by construction [14]. Rather than giving this proof in this paper, we show an example illustrating how the GRP instruction achieves 8-bit permutations.

Table 1: The GRP instruction sequence for an 8-bit permutation

	Iteration 1	Iteration 2
P	(5, 0, 1, 2, 4, 3, 7, 6)	(3, 5, 7, 0, 1, 2, 4, 6)
MISes in P	(5)(0, 1, 2, 4)(3, 7)(6)	(3, 5, 7), (0, 1, 2, 4, 6)
Combining MISes	(5, 3, 7), (0, 1, 2, 4, 6)	(3, 5, 7, 0, 1, 2, 4, 6)
Sorting	(3, 5, 7), (0, 1, 2, 4, 6)	(0, 1, 2, 3, 4, 5, 6, 7)
New Permutation	(3, 5, 7, 0, 1, 2, 4, 6)	(0, 1, 2, 3, 4, 5, 6, 7)
Control bits for GRP instruction	(1, 0, 1, 0, 0, 0, 0, 1)	(1, 1, 1, 0, 1, 0, 1, 0)

Table 1 shows the procedure of finding the control bits for GRP instructions performing an 8-bit permutation (0, 1, 2, 3, 4, 5, 6, 7)→(5, 0, 1, 2, 4, 3, 7, 6). This means that bit 5 in R1 becomes bit 0 in R3, bit 0 in R1 becomes bit 1 of R3, and so forth. At the beginning of iteration 1 (see the second column), P is the permutation we need to do. We divide the integer sequence into monotonically increasing sequences (MIS), as shown in the third row. Underlined MISes are in the right half. Then we combine the MISes in the left and right halves. The first in the left is merged with the first in the right, the second in the left with the second in the right, and so on. Then we sort the merged groups. Results of sorting are shown in the fifth row. This is an intermediate permutation, which is "closer" to the desired final permutation. Examining the numbers in this intermediate permutation, we get control bits for one GRP instruction. If the number in the new permutation is in the second half (underlined in the third row), the corresponding control bit is 1, and 0 otherwise. Next, we do the iteration again with the new intermediate permutation. The process terminates when the newly generated permutation is a single monotonically increasing sequence (0, 1, 2, …, *n*-2, *n*-1), which represents the original input. Using the control bits generated in this process, we can perform the permutation with the following sequence of GRP instructions

```
GRP     R1, R2, R1          ; R2 = 0b11101010
GRP     R1, R3, R1          ; R3 = 0b10100001
```

R2 and R3 contain the control bits we generated in iteration 2 and iteration1, respectively.

Table 2: Pseudo code for the new permutation instructions

Instruction		Pseudo code

PPERM,x	R1, R2, R3	`R3[0..n-1] = 0;`
		`for (i = 0; i < k; i ++)`
		`  otherreg = R2[i*(lg(n)+1)];`
		`  j = R2[i*(lg(n)+1)+1 .. ((i+1)*(lg(n)+1)-1];`
		`  if (otherreg == 0)`
		`  R3[x*k+i]= R1[j];`

GRP	R1, R2, R3	`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];`

CROSS,m1,m2	R1, R2, R3	`R3 = R1;`
		`j = 0;`
		`dist = 1 << m1;`
		`for (s = 0; s < n; s += (dist * 2))`
		`  for (i = 0; i < dist; i ++)`
		`  if (R2[j++] == 1)`
		`  swap(R3[s+j], R3[s+j+dist]);`
		`dist = 1 << m2;`
		`for (s = 0; s < n; s += (dist * 2))`
		`  for (i = 0; i < dist; i ++)`
		`  if (R2[j++] == 1)`
		`  swap(R3[s+j], R3[s+j+dist]);`

OMFLIP,c	R1, R2, R3	`for (i = 0; i < 2; i++)`
		`  if (c[i] == 0)`
		`  for (j = 0; j < n/2; j++)`
		`  R3[2 * j] = R1[j];`
		`  R3[2 * j + 1] = R1[j + n/2];`
		`  if (R2[j] == 1)`
		`  swap(R3[2 * j], R3[2 * j + 1]);`
		`  else`
		`  for (j = 0; j < n/2; j++)`
		`  R3[j] = R1[2 * j];`
		`  R3[j + n / 2] = R1[2 * j + 1];`
		`  if (R2[j + n/2] == 1)`
		`  swap(R3[j], R3[j + n / 2]);`

Note: n is the number of bits in registers. i, j, and s are temporary variables. k in the codes for PPERM is the number of bits we can permute with one instruction. (In this paper, k=8). "swap" is a function that exchanges the value of its two parameters. R1[1] means the bit 1 in R1. R1[0..7] means bit 0 to bit 7 in R1.

3.3 CROSS permutation instruction

In interconnection networks such as the butterfly network, data at any input can be directed to any output by setting up the proper connections in the network. We observe that a permutation of *n* elements (without

repetitions) is like *n* pieces of data being routed simultaneously to *n* different destinations. We want to look for networks that can achieve this task using edge-disjoint paths and can also be easily implemented as a simple functional unit in a processor. Furthermore, there must exist an algorithm to specify how the *n* data are to be routed.

The CROSS instruction is based on the Benes network, which is formed by connecting two butterfly networks of the same size back-to-back. An 8-input Benes network is shown in Figure 4. An *n*-input Benes network can be used to perform any permutation of its *n* inputs with edge-disjoint paths [15]. In addition, Benes network has the following properties

1. An n-input Benes network can be broken into 2lg*n* stages, lg*n* of them are distinct.
2. In each stage of a Benes network, every input has two outputs to the next stage. For each input, there is another input that shares the same two outputs with it. We call these pairs of inputs *conflict pairs.* The connections for a conflict pair can be configured with one bit. The distances between conflict pairs in one stage are the same. They are different in different stages.

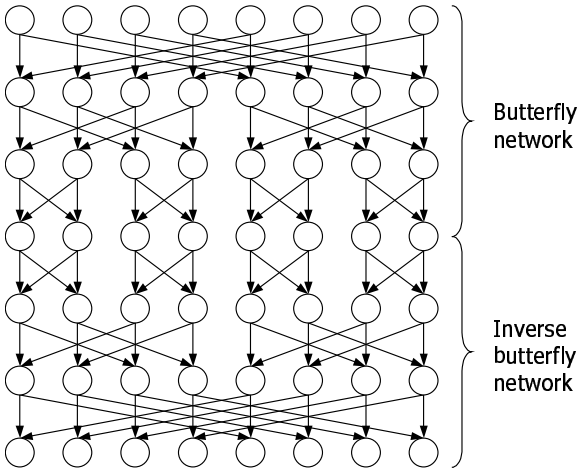


Figure 4: An 8-input Benes network

Due to the existence of conflict pairs, each stage of an *n*-input Benes network can be configured with *n*/2 bits. If the configuration bit for a conflict pair is 0, the two inputs take straight paths to the next stage. If it is 1, the two inputs cross paths to the next stage. We can define a set of basic operations for a Benes network. Each basic operation corresponds to one distinct stage in a Benes network. A basic operation is specified by a parameter *m*, where 2^m is the distance between conflict pairs for the corresponding stage. A basic operation uses *n*/2 configuration bits to set up the connections in the corresponding stage and move the *n* input bits to the output. Since an *n*-input Benes network has lg*n* distinct stages, we have lg*n* different basic operations. With these basic operations, the CROSS instruction is defined as follows

CROSS,m1,m2 R1, R2, R3

R1 is the source register, which contains the bits to be permuted, R3 is the destination register for the permuted bits, and R2 is the configuration register. One CROSS instruction performs two basic operations on the source according to the contents of the configuration register and the values of m1 and m2. m1 and m2 are the parameters that specify the two basic operations to be used. The operation of a CROSS instruction is shown in Figure 5. The pseudo code is shown in Table 2.

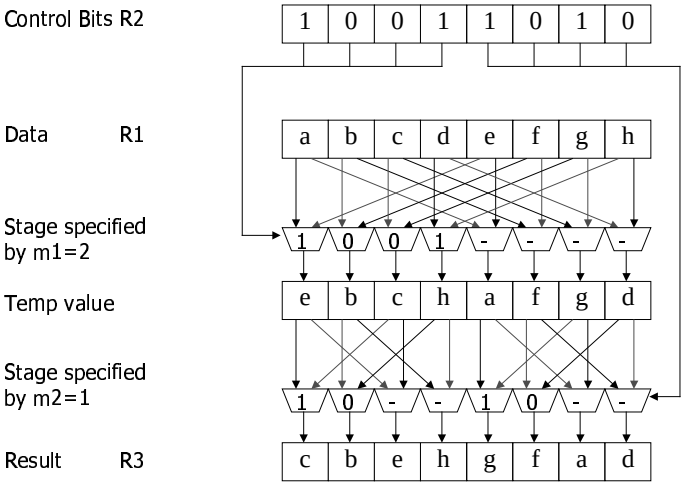


Figure 5: Operation of CROSS,2,1 R1,R2, R3

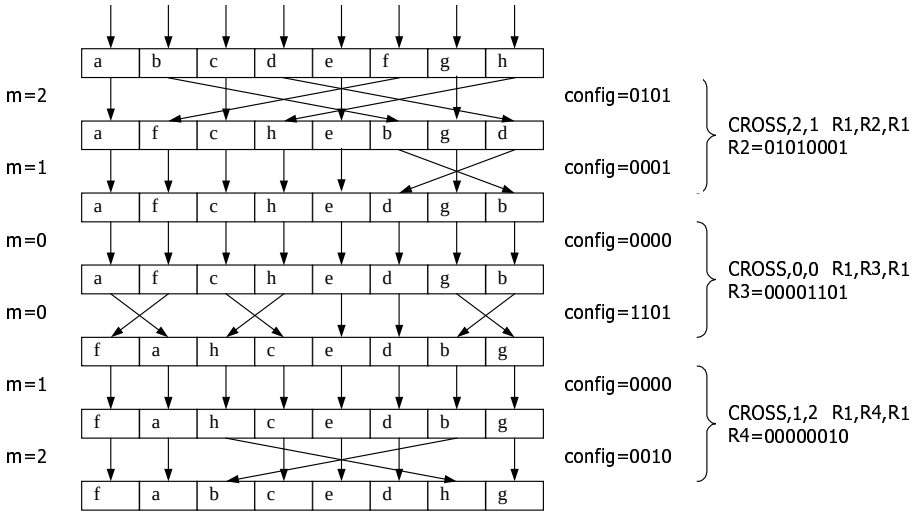


Figure 6: An example showing permutation (abcdefgh)→(fabcedhg) on an 8-input Benes network

In order to generate the CROSS instruction sequence for a specific permutation, we perform the following steps

1. Obtain a valid configuration for the desired permutation on a Benes network, the method is discussed in [16];
2. Break the configured Benes network into pairs of stages;
3. For each pair of stages, assign a CROSS instruction. Each CROSS instruction uses the output from the previous CROSS instruction and generates input for the next CROSS instruction. The first CROSS instruction takes the original input and the last CROSS instruction generates the final result.

Because there are 2lg*n* stages in an *n*-input Benes network, we need at most lg*n* CROSS instructions for any *n*-bit permutation. An example is shown in Figure 6.

3.4 OMFLIP permutation instruction

The CROSS instruction is very efficient because any permutation of n bits can be achieved with at most lg$n$ CROSS instructions. However, because an n-input Benes network has lg$n$ distinct stages and these stages are in a specific order, a permutation FU must have a whole Benes network in hardware to implement all CROSS instructions. This is not efficient in area for machines with wider words, for example, 64 bits. In this case, each CROSS instruction only uses two stages out of twelve stages needed for the whole network.

We explore alternative possibilities which can achieve the same level of performance as the CROSS instructions, but yield more efficient hardware implementations. The disadvantage of the CROSS implementation stems from the fact that a Benes network has many distinct stages. We search for a network that has uniform stages, which can also achieve any arbitrary permutation in 2lg$n$ stages.

One solution is the omega-flip network, which consists of an omega network followed by a flip network. An n-input omega network has lg$n$ identical omega stages. An n-input flip network is the exact mirror image of an n-input omega network. It has lg$n$ identical flip stages. An 8-input omega-flip network is shown in Figure 7. An n-input omega-flip network is isomorphic to an n-input Benes network. As a result, they share common properties. An n-input omega-flip network can be broken into 2lg$n$ stages. However, it only has two distinct stages, one omega stage and one flip stage. In each stage of an n-input omega network or flip network, we also have n/2 conflict pairs and the connections for each conflict pair can be configured with one bit.

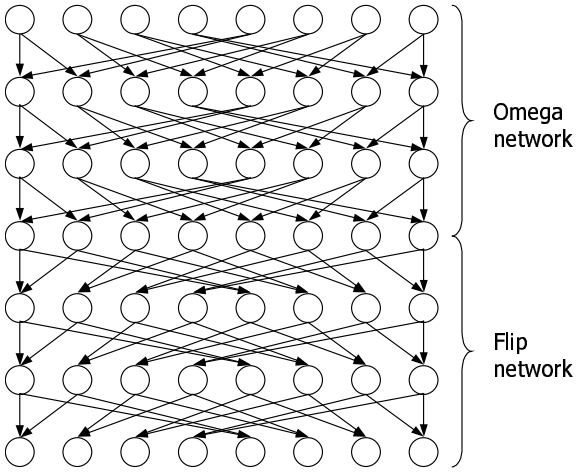


Figure 7: An 8-input omega-flip network

For an omega-flip network, we can define two basic operations as opposed to lg$n$ basic operations for a Benes network. One basic operation is the omega operation, which accomplishes the operation of an omega stage. It uses n/2 configuration bits to set up an omega stage and move the input bits to their corresponding outputs. Another basic operation is the flip operation, which performs the operations of a flip stage. It also uses n/2 configuration bits to set up a flip stage and move input bits to the output. The definition of the configuration bits follows the same convention as that for the Benes network. An OMFLIP instruction carries out two basic operations on the source word. It is defined as

```
OMFLIP,c    R1, R2, R3
```

R1, R2 and R3 hold source, configuration bits and result, respectively. c is a two-bit sub-opcode defining which two basic operations to use. For each bit in c, 0 indicates the omega operation and 1 indicates the flip operation. The operation of an OMFLIP instruction is shown in Figure 8. The corresponding pseudo code is given in Table 2.

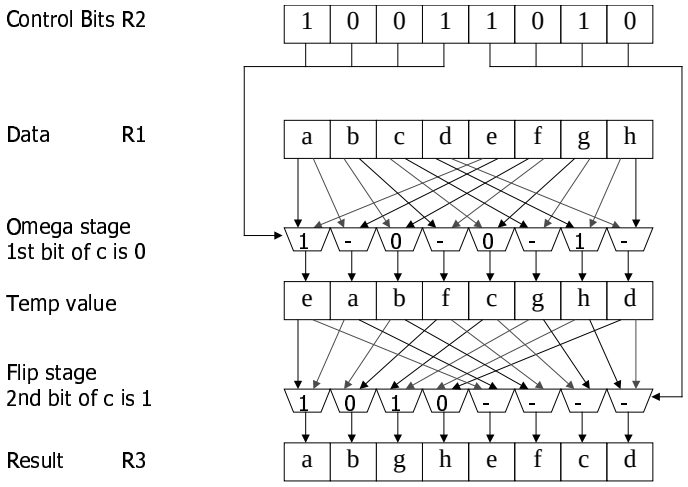


Figure 8: Operation of OMFLIP,01 R1,R2, R3

Since the omega-flip network is isomorphic to the Benes network, we can use the following steps to generate the OMFLIP instruction sequence for a specific permutation

1. Obtain a valid configuration for the desired permutation on a Benes network;
2. Map the configuration of the Benes network to the omega-flip network based on the isomorphism between them;
3. Break the configured omega-flip network into pairs of stages;
4. For each pair of stages, assign an OMFLIP instruction. Each OMFLIP instruction uses the output from the previous OMFLIP instruction and generates input for the next OMFLIP instruction. The first OMFLIP instruction takes the original input and the last OMFLIP instruction generates the final result.

Because there are 2lg*n* stages in an *n*-input omega-flip network, we need at most lg*n* OMFLIP instructions for any *n*-bit permutation. An example is shown in Figure 9.

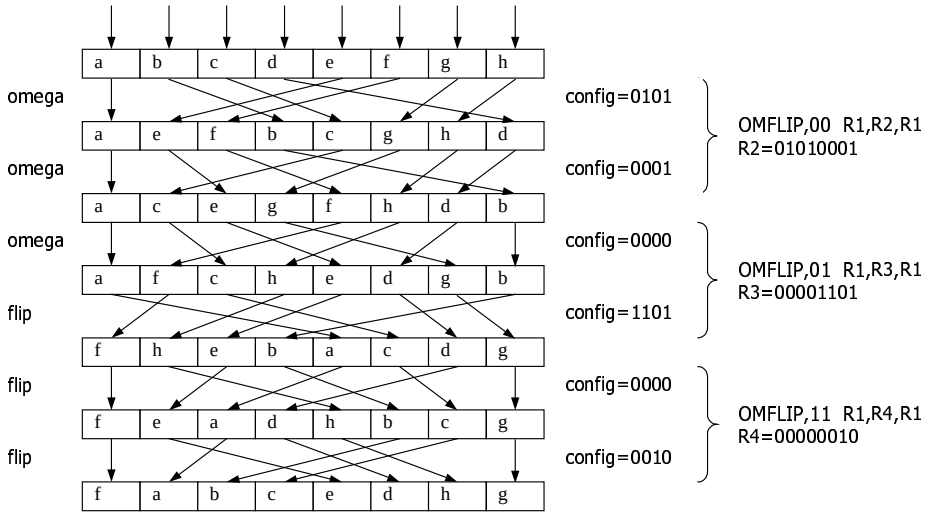


Figure 9: An example showing permutation (abcdefgh)→(fabcedhg) on an 8-input omega-flip network

A permutation unit implementing the OMFLIP instruction is only four stages deep, two omega stages followed by two flip stages [17]. This is sufficient to implement the four combinations of the two basic operations, omega and flip. This number of stages does not change with n, unlike the Benes network which requires the implementation of $2\lg n$ stages.

3.5 Subwords, scalability and repetition

In this section, we consider the extensibility of the permutation instructions defined above for efficient permutation of multi-bit subwords, scalability to $2n$ bits, and whether permutations with repetition of bits can be supported.

3.5.1 Subwords

If we have subwords greater than one bit packed into an n-bit register, we will have fewer subwords to permute. For example, with 8-bit subwords in a 64-bit register, there are only eight elements to permute, rather than 64 elements with 1-bit subwords. Fewer permutation instructions should be needed, lg8=3 instructions rather than lg64=6 instructions. Can the permutation instructions defined above for bit-level permutations also allow efficient permutation of multi-bit subwords?

The PPERM instruction, as currently defined, is not efficient for multi-bit subword permutations. It can perform permutations of larger subwords packed in a register, but cannot take advantage of the smaller number of elements which have to be permuted, to reduce the number of permutation instructions required.

The GRP instruction is as efficient for permuting multi-bit subwords as for 1-bit subwords. The number of GRP instructions required depends only on the number of elements to be permuted, and hence fewer GRP instructions are required to permute larger subwords. For example, on 64-bit processors, if the subword size is eight, there are only eight elements. Therefore, at most lg8=3 instructions are enough to do any permutation of the eight 8-bit subwords, whereas at most lg64=6 GRP instructions are required for any permutation of the 64 bits. In the previous example, if the elements in the permutation (5, 0, 1, 2, 4, 3, 7, 6) are 8-bit subwords in a 64-bit processor, rather than 1-bit subwords in an 8-bit processor, we also need only two instructions to do it. By substituting each 0 or 1 with eight consecutive 0's or 1's, respectively, we get the control bits for n=64 with 8-bit subwords.

The CROSS and OMFLIP instructions are also as efficient for multi-bit subwords as for 1-bit subwords. One important property of a Benes network is that when it is configured for k-bit subword permutations, the middle $2\lg k$ stages are configured as pass-throughs [16]. The same is true of the omega-flip network [17]. These bypassing stages can be eliminated before assigning CROSS or OMFLIP instructions. Therefore, when permuting k-bit subwords in an n-bit word, the maximum number of instructions needed is $\lg n - \lg k = \lg(n/k) = \lg r$, where r is the number of subwords.

3.5.2 Scalability to $2n$ bits

We now consider if the permutation instructions can permute subwords packed into more than one register. In particular, if a register is n bits, we consider the scalability to permutations of $2n$ bits. Here, we are doubling the number of elements to be permuted, since each element is one bit. As discussed earlier in Section 2, this should require at least $2\lg n + 2$ instructions to achieve any arbitrary permutation of $2n$ bits. We wish to see how close to this optimal number of instructions we can achieve with the alternative permutation instructions defined.

The PPERM instruction allows permuting $2n$ bits stored in two n-bit registers, by the use of the extra "otherreg" bit in each index that selects a bit of the source register. This allows different PPERM instructions to pick bits from more than one source register. To permute $2n$ bits, two source registers must be used, and two destination registers are produced. For each destination register, 8 PPERM instructions are used on each source register, requiring a total of 16 PPERM instructions and 15 OR instructions to

combine the results into one destination register. The same must be repeated to produce the other destination register. Hence, a total of 2(16+15) = 62 instructions are needed to permute 2*n* bits. This is not efficient compared to other methods described below.

The GRP instruction scales very efficiently to *2n*-bit permutations with the help of an instruction like the Shift Pair instruction in PA-RISC [11] and IA-64 [7]. The Shift Pair instruction concatenates two source registers to form a double-word value, and then extracts any contiguous single-word value. Suppose R1 and R2 store the bits to be permuted, and the results will be placed in R3 and R4. Figure 10 shows how two GRP instructions followed by two Shift Pair instructions can be used to separate the 2*n* bits into the appropriate left and right registers (R3 and R4) of the result. Finally, two separate *n*-bit permutations are performed on R3 and R4 to get the desired *2n*-bit permutation.

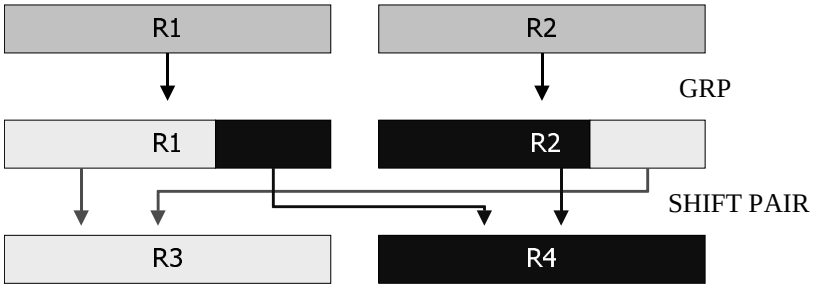


Figure 10: Use the SHIFT PAIR and GRP instructions to do 2n-bit permutations

In total, excluding the instructions needed for loading control bits, we need 4+2lg*n* instructions to do a *2n*-bit permutation. These are only 2 instructions more than the minimum number of instructions required. With 64-bit registers, a 128-bit permutation can be done with 16 instructions.

CROSS or OMFLIP instructions can also be used to permute 2*n* bits stored in two registers. We use a similar approach to that used with GRP instructions. We first permute each of the two *n*-bit words using CROSS or OMFLIP instructions so that the bits going to the left *n*-bit word and the right *n*-bit word in the result are separated. Then we use two Shift Pair instructions to move the bits belonging to the left word and right word into two registers. Finally, we permute the bits in these left and right registers to get the final result. The first step uses a maximum of 2lg*n* CROSS or OMFLIP instructions, the last step also uses at most 2lg*n* instructions. Therefore, it takes at most 4lg*n*+2 instructions to permute 2*n* bits using *n*-bit words. Compared to the ideal instruction count of 2lg*n*+2, this method uses 2lg*n* more instructions. Hence, CROSS and OMFLIP instructions are not as efficient as the GRP instruction when scaling up to 2*n*-bit permutations.

3.5.3 Permutations with Repetitions

So far, we have discussed permutations where each bit in the source goes to a different bit in the destination. These are permutations where no bit in the source is repeated or omitted in the destination. We now consider the case where we allow bits in the source to be repeated.

PPERM can easily handle permutations with repetitions, since any of the *n* bits can be selected any number of times. This is useful for some cryptography algorithms. For example, DES has an expansion permutation in which some bits are replicated.

Currently, none of the other permutation instructions defined (GRP, CROSS and OMFLIP) support permutations with repetitions.

To summarize, GRP, CROSS and OMFLIP instructions are all efficient for permuting multi-bit subwords. GRP is the most efficient for scaling up to bit permutations of 2*n* bits. CROSS and OMFLIP can perform 2*n* bit permutations, but require (2lg*n*-2) more instructions than GRP. PPERM is not efficient for multi-bit

subwords and can scale to permuting 2*n* bits, but not efficiently. However, it is the only permutation alternative defined above that can perform permutations with repetitions.

4. Hardware requirements for new permutation instructions

We now consider the hardware complexity of the permutation functional unit corresponding to each of the new subword permutation instructions we have proposed. On a 64-bit processor architecture, the PPERM instruction requires a 64-by-8 crossbar network and a specialized shifter in hardware. This is because each PPERM instruction generates eight consecutive bits in the result. These eight bits can come from any of the 64 bits in the source and can be deposited into different contiguous eight bits in the destination.

The GRP instruction requires a hierarchical lg*n*-stage bit gathering network in hardware. This is due to the fact that a GRP instruction needs to separate bits in the source into two groups based on their corresponding configuration bits. The separation process has to be done in a recursive manner.

To implement the CROSS instruction, a full Benes network is needed in hardware. The hardware required for the OMFLIP instruction is much simpler. Because there are only two distinct stages in an omega-flip network, and each OMFLIP instruction can perform the operation of two stages, the hardware only needs to implement four stages, two omega stages and two flip stages. For the execution of each OMFLIP instruction, we use two of the four stages and bypass the other two. The rough transistor counts for the hardware required for these four permutation instructions are given in Table 3.

Table 3: Transistor counts for the hardware required for the four permutation instructions

	PPERM	GRP	CROSS	OMFLIP
Number of transistors	7k	68k	4.6k	3k

The hardware cost for the GRP instruction for 64-bit permutations is an order of magnitude larger than the other three instructions. OMFLIP needs the fewest transistors, and potentially the smallest area.

5. Performance

Figure 11a shows the minimum instruction sequence for permuting 64 bits using either GRP, CROSS or OMFLIP instructions, assuming we need to load the six configuration registers. This takes lg*n*+2=8 steps, with a 2-way superscalar machine, assuming the base register for the address of the LOAD instructions can be set with one load offset (LDO) instruction. Figure 11b shows that PPERM would take *n*/8 +6=14 steps, with the same assumptions. Without any new permutation instruction, the table lookup method (see Section 1.1) used in current microprocessors would take 16 steps on a 2-way superscalar machine (Figure 11c). We assume that instructions like EXTR in PA-RISC [11] are available to extract the index for loading a table entry, and the base register for each of the eight tables can be set with one LDO instruction. In Figure 11d, we show the instructions required for one table lookup. We do not consider higher degrees of superscalar execution for this paper since the additional hardware cost of multiple LOAD pipes and permutation functional units may be too high for internet information appliances.

The number of instructions and cycles required for 64 bit permutations with the different methods is summarized in Table 4. The cycle counts assume no LOAD delays and no cache misses. GRP, CROSS and OMFLIP have the same performance in all cases. Although PPERM and table lookup have similar numbers, the configuration bits to be loaded for PPERM are only eight words (see Table 6), which typically fit in one or two cache lines. Hence loading configuration bits for PPERM instructions is much less likely to cause cache misses than loading random entries from eight different tables occupying 16 Kbytes in the table lookup method.

In Table 5, we compare the number of instructions and cycles for permuting eight 8-bit subwords in a 64-bit register. GRP, CROSS and OMFLIP reduce the number of instructions by half. PPERM and table lookup cannot take advantage of fewer elements with the larger subword size. They do the subword

permutations in the same way as doing bit permutations. Since the number of elements in these permutations is only eight, it is reasonably efficient to do 8-bit subword permutations using existing instructions (see last column). Each subword is taken from the source register with an EXTRACT instruction as in PA-RISC [11], and put in the destination register with the DEPOSIT instruction. Only 16 instructions are required for eight subwords.

```
      a. GRP, CROSS, OMFLIP      b.  PPERM          c.  Table Lookup
         (2=way ss)              (2-way ss)         (2-way ss)

 1    LDO                        LDO                LDO   EXTR
 2    LD                         LD                 LDO   EXTR
 3    LD   GRP                   LD    PPERM        LD    LDO
 4    LD   GRP                   LD    PPERM        LD    EXTR
 5    LD   GRP                   LD    PPERM        LDO   EXTR
 6    LD   GRP                   LD    PPERM        LD    LDO
 7    LD   GRP                   LD    PPERM        LD    EXTR
 8         GRP                   LD    PPERM        LDO   EXTR
 9                               LD    PPERM        LD    LDO
10                               OR    PPERM        LD    EXTR
11                               OR    OR           LDO   EXTR
12                               OR    OR           LD    OR
13                               OR                 LD    OR
14                               OR                 OR    OR
15                                                  OR    OR
16                                                  OR
```

```
      d. One table lookup

 1    LDO   TABLE(R27), R2    ; Load the address of the table into R2
 2    EXTR  R9, 7, 8, R3      ; Extract the index into R3
 3    LD    (R2, R3), R4      ; Load a 64-bit entry into R4
```

Figure 11: Scheduling instruction sequences for Permutations

Table 4: Number of instructions for a 64-bit permutation

	PPERM	GRP	CROSS/OMFLIP	Table Lookup
# of instructions without LDs	15	6	6	31
# of instructions with LDs	24	13	13	
# of cycles on 2-way ss with LDs	14	8	8	16

Table 5: Number of instructions for an 8-bit subword permutation

	PPERM	GRP	CROSS/OMFLIP	Table Lookup	EXTR/DEP
# of instructions w/o LDs	15	3	3	31	16
# of instructions with LDs	24	7	7		
# of cycles on 2-way ss with LDs	14	5	5	16	9

Table 6: Storage requirement on 64-bit processors

	PPERM	GRP	CROSS/OMFLIP	Table lookup
Storage requirement (bytes)	64	48	48	16K

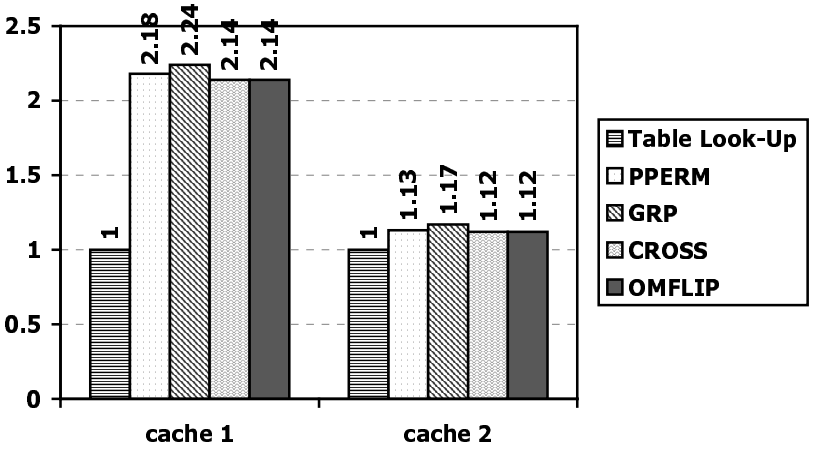
GRP, CROSS and OMFLIP only need 6*8=48 bytes of memory for configuration information, PPERM needs 8*8=64 bytes of memory for configurations, whereas the table lookup method in current

microprocessors needs 16 Kilobytes of memory for the eight tables describing one 64-bit permutation. Not only is this three orders of magnitude more storage, but these LD instructions for the table lookup method are much more likely to miss in the cache. If a cache miss costs 50 cycles, then even if only two out of the eight LD instructions miss for the table lookup method, this can cause an extra 100 cycles of execution time, resulting in two orders of magnitude more in execution time to perform a 64-bit permutation.

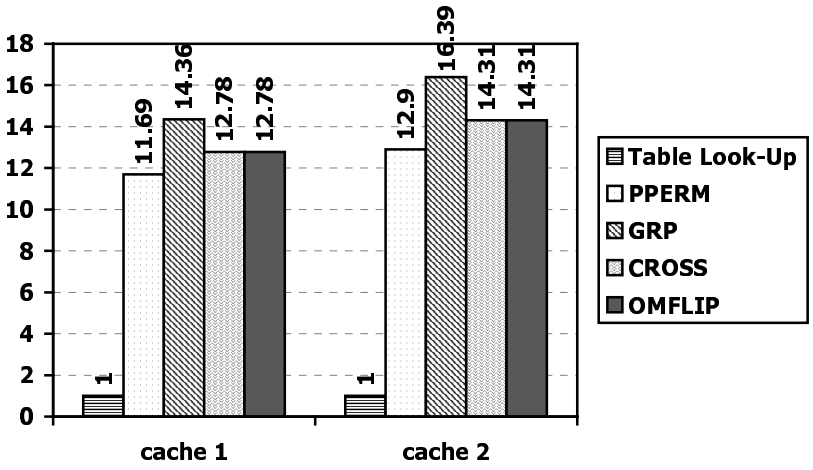
Figure 12 shows the speedup of DES using the new permutation instructions over traditional software (without new ISA). We assume a 64-bit processor with 2-way superscalar issue and one load/store unit. We consider two cache configurations: Cache 1 may represent an internet information appliance, while Cache 2 may represent a web-server. Cache 1 is a one-level split cache system, with 16 Kbytes of data cache memory, and a cache miss penalty of 50 cycles. Cache 2 is a two-level cache system: L1 cache is on the processor chip with a miss penalty of 10 cycles; L2 is off-chip with a miss penalty of 50 cycles. Cache 2 has 16K bytes L1 data cache and 256K bytes unified L2 cache. We use table lookup as the baseline for comparison because we find that table lookup is almost as fast as, and sometimes faster than, the most optimized DES routine we can find in libdes for 64-bit processors.

Figure 22a shows that the speedup of encryption more than doubles for Cache 1, and improves by 12% to 17% for Cache 2. The new permutation instructions perform very well on systems with a small cache, even though we have optimized the baseline program to eliminate all permutations in the encryption kernel, except the initial and final permutations. In the table lookup method, the P-box permutation in the DES encryption kernel is eliminated by combining it with the S-box substitution, which is more efficient when permutation instructions are not available in the processor. When permutation instructions are available, the S-box tables are much smaller, since the P-box permutation can be achieved efficiently with these new permutation instructions. The much larger tables in the table lookup method results in more cache misses, which degrade performance.

Figure 22b shows that the speedup of key generation increases tremendously from 11 to 16 times, because this routine is dominated by the time taken for performing permutations.



a) Speedup of DES encryption/decryption.



b) Speedup of DES key scheduling

Figure 12: Speedup of DES

6. High-Level Language Specification

We need to provide programmers with a high-level language interface so that they can specify permutations in a convenient manner without worrying about how the permutations are actually performed by the processors. We also need to add support in the compiler or macro assembler so that it can translate the programmers' high-level permutation specifications into the corresponding permutation instruction sequence. For example, a permutation may be specified explicitly by either a new high-level language statement, or a macro call, of the form

Permute(m, r, int spec[]),

where *m* is the number of subwords to permute, *r* is the number of bits in a subword, and *spec* is an array of integers that specifies the desired order of the subwords from the source register in the permuted result.

During macro expansion, this is translated to the available permutation instructions, as described in Section 3. Based on the type of permutation instructions supported by the target processor, the compiler or macro assembler invokes the corresponding instruction generation algorithm to produce the instruction sequence for the desired permutation. In case no permutation instructions are supported by the target processor, the compiler can always fall back to the traditional AND/SHIFT/OR or EXTRACT/DEPOSIT methods, or generate tables for permutation by the table lookup method.

7. Conclusions

We defined four permutation methodologies and created innovative permutation instructions for performing arbitrary *n*-bit permutations efficiently with software running on programmable processors. We characterized the performance of these different permutation methodologies, and the tradeoffs between them. We also described how arbitrary permutations could be specified in high-level software by means of an array of integers, in a processor-independent way. This can be used by compilers or macro assemblers to generate efficient sequences of permutation instructions to achieve any arbitrary permutation, using the permutation instructions available in the target processor.

Table 7: Summary

	Table Lookup	PPERM	GRP	CROSS	OMFLIP
performance for arbitrary n-bit permutation	slowest	medium (O(lgn))	**fast (lg n)**	**fast (lg n)**	**fast (lg n)**
permutations of fewer number (r<n) of multi-bit subwords	inefficient	inefficient (O(lgn))	**efficient (lgr)**	**efficient (lg r)**	**efficient (lg r)**
scalability to 2n bits	inefficient	inefficient	**efficient**	less efficient	less efficient
permutations with bit repetitions	**efficient**	**efficient**	not possible	not possible	not possible
area for permutation functional unit	system memory	medium	largest	small	**smallest**
memory requirements (for n=64 bits)	large (16 Kbytes)	**small (64 bytes)**	**very small (48 bytes)**	**very small (48 bytes)**	**very small (48 bytes)**
single-cycle permutation primitive	not applicable	**yes**	maybe (complex circuitry)	yes (with caveats)	**yes**

Table 7 summarizes the relative advantages and disadvantages of our permutation methodologies and the table-lookup method, for achieving any arbitrary permutation of *n* bits. All four permutation instructions, PPERM, GRP, CROSS and OMFLIP, are standard 2-source 1-destination instructions. GRP, CROSS and OMFLIP can perform any *n*-bit permutation in lg*n* instructions, not counting loading of the configuration registers. If we count the loading of the configuration registers, then it takes 2lg*n* instructions, but only lg*n*+2 cycles, if the processor is two-way superscalar. PPERM is less efficient in terms of instructions and cycles needed. Without at least one of these new permutation instructions, the fastest way to perform 64-bit permutations is with Table Lookup methods, which can be one to two orders of magnitude larger, depending on the cache configuration.

All four methods can also perform permutations of any subword size that is a power of two, with the same permutation functional unit. GRP, CROSS and OMFLIP require only lg*r* instructions, where *r* is the number of subwords to be permuted. Hence, they are efficient in permuting a smaller number of multi-bit subwords in the *n*-bit register. PPERM and the table lookup method cannot reduce the number of instructions needed to permute fewer elements.

The GRP instruction is the most efficient in terms of scaling up to perform a 2*n*-bit permutation, using *n*-bit registers. CROSS and OMFLIP require more instructions, and PPERM is even more inefficient. The PPERM instruction is advantageous in that permutations with repetitions are handled with the same ease as permutations without repetitions of elements. None of GRP, CROSS and OMFLIP is defined to handle permutations with repetitions.

The OMFLIP permutation functional unit is the smallest in terms of the number of transistors, followed by CROSS, PPERM, with GRP having the most complex hardware implementation. All four instructions need three orders of magnitude less storage than the table lookup method for performing permutations in today's microprocessors. OMFLIP and PPERM can be implemented in a single cycle, assuming a cycle is typically one or two times the time taken for an addition of two *n*-bit integers. CROSS and GRP may be implementable in one cycle, but GRP will require complex circuitry to achieve single-cycle execution.

Only one of the four permutation instructions, PPERM, GRP, CROSS or OMFLIP needs to be implemented in a processor, since they overlap in functionality. We have shown the advantages and disadvantages of each.

By providing the ability to do fast permutations in software, we not only speed up current cryptography and multimedia programs, but also open the field for new algorithms using these powerful yet simple permutation primitives. This facilitates much faster software cryptography and multimedia processing, combining high performance with flexibility in secure multimedia information appliances and servers.

8. References

1. NIST, "Announcing request for candidate algorithm nominations for the advanced encryption standard (AES)", http://csrc.nist.gov/encryption/aes/pre-round1/aes_9709.htm
2. Bruce Schneier, "Applied Cryptography", 2nd Ed., *John wiley & Sons*, Inc.,1996
3. Ruby Lee,"Accelerating Multimedia with Enhanced Microprocessors", *IEEE Micro*, Vol. 15, No. 2, 1995, pp.22-32
4. Ruby Lee, "Subword Parallelism in MAX-2", *IEEE Micro*, Vol. 16, No. 4, 1996, pp.51-59
5. 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
6. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture", *IEEE Micro*, Vol. 16, No. 4, 1996, pp. 10-20
7. Intel Corporation, "IA-64 Application Developers' Architecture Guide", *Intel Corporation*, May 1996
8. Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing Across the Spectrum", http://www.motorola.com/AltiVec
9. S. Oberman and F. Weber and N. Juffa and G. Favor, "AMD 3Dnow! Technology and the K6-2 Microprocessor", *HotChips 10*, August 16-18, 1998
10. Intel Corp., "Intel Architecture Software Developer's Manual", http://developer.intel.com/design/PentiumIII
11. Ruby Lee, "Precision Architecture", *IEEE Computer*, Vol. 22, No. 1, Jan 1989, pp.78-91
12. Milton Abramowitz, Irene A. Stegun, "Handbook of Mathematical Functions", 9th printing, *US Department of Commerce, National Bureau of Stanards*, November 1970
13. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", *The MIT Press*, 1994
14. Zhijie Shi and Ruby 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 2000.
15. F. Thomson Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes", *Morgan-Kaufmann Publishers, Inc.*, 1992
16. Xiao Yang, Manish Vachharajani, Ruby Lee, "Fast Subword Permutation Instructions Based on Butterfly networks", *Proceedings of Media Processors 2000*, SPIE, pp. 80-86, Jan 22-28, 2000.
17. Xiao Yang and Ruby Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages", *Proceedings of the International Conference on Computer Design, pp. 15-22*, September 2000.
18. Ruby 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, pp. 3-14.* July 2000.