# Arbitrary Bit Permutations in One or Two Cycles

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

***Abstract***

*Symmetric-key block ciphers encrypt data, providing data confidentiality over the public Internet. For inter-operability reasons, it is desirable to support a variety of symmetric-key ciphers efficiently. We show the basic operations performed by a variety of symmetric-key cryptography algorithms. Of these basic operations, only bit permutation is very slow using existing processors, followed by integer multiplication. New instructions have been proposed recently to accelerate bit permutations in general-purpose processors, reducing the instructions needed to achieve an arbitrary n-bit permutation from O(n) to O(log(n)). However, the serial data-dependency between these log(n) permutation instructions prevents them from being executed in fewer than log(n) cycles, even on superscalar processors. Since application specific instruction processors (ASIPs) have fewer constraints on maintaining standard processor datapath and control conventions, can we achieve even faster permutations? In this paper, we propose six alternative ASIP approaches to achieve arbitrary 64-bit permutations in one or two cycles, using new BFLY and IBFLY instructions. This reduction to one or two cycles is achieved without increasing the cycle time. We compare the latencies of different permutation units in a technology independent way to estimate cycle time impact. We also compare the alternative ASIP architectures and their efficiency in performing arbitrary 64-bit permutations.*

## 1. Introduction

Symmetric-key block ciphers encrypt data and can be used to provide confidentiality for network transactions, stored data and programs. Such support for confidential information is important on all Internet-connected computers and computing devices due to the ease of eavesdropping attacks on the public Internet and wireless networks. Standard security protocols, including SSL and IPSEC, support a large variety of cryptographic algorithms. The cipher suite to be used is often negotiated at the beginning of a communication session. For inter-operability reasons, it is desirable to support a variety of ciphers efficiently. As secure computing paradigms become more pervasive, it is likely that such cryptographic computations will become a major component of every processor's workload. Understanding the new requirements of secure information processing is important for the design of all future programmable processors, whether general-purpose, application-specific, or embedded.

First, we investigate which operations are frequently used by symmetric-key ciphers but not efficiently supported by existing processors. Table 1 shows the basic operations used by a range

This work was supported in part by NSF under grant CCR-0105677, and in part by a research gift from Hewlett-Packard Laboratories.

*To be published in the Proceedings of the IEEE 14th International Conference on Application-Specific Systems, Architectures and Processors, June 2003*

of popular symmetric-key algorithms such as DES (Data Encryption Standard [1]) and AES (Advanced Encryption Standard [2]). All algorithms use ALU add, subtract or logical operations. Most use some form of table lookup, accomplished with memory load instructions. Three ciphers use integer multiplication while three others use bit permutations. Except for one cipher, all require some form of fixed rotation, variable (or data-dependent) rotation, or bit permutation.

**Table 1: Basic operations in symmetric-key block ciphers**

| | DES 3DES | AES | RC5 | IDEA | TWOFISH | SERPENT | RC6 | MARS | Kasumi |
|---|---|---|---|---|---|---|---|---|---|
| ALU (add,sub) | | | √ | √ | √ | | √ | √ | |
| ALU (logical) | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| Table Lookup | √ | √ | √ | | √ | √ | √ | √ | √ |
| Multiply | | | | √ | | | √ | √ | |
| Shift | | | | | | √ | | | |
| Fixed rotation | √ | √ | √ | | √ | √ | √ | √ | √ |
| Variable rotation | | | √ | | | | √ | √ | √ |
| Permutation | √ | | | | √ | √ | | | |

All current microprocessors, embedded processor cores, and digital signal processors support ALU instructions, memory access instructions, and shift instructions. Not all processors support integer multiply, and of those that do, often only 16-bit multiplies are supported. The ciphers typically use 32-bit multiplies [3, 4]. Not all processors support fixed or variable rotations, and none currently support bit permutations directly.

Next, we ask if a new application-specific instruction processor (ASIP) designed to support both current and future symmetric-key ciphers should support fast bit permutations (versus rotations or multiplications)? Symmetric-key ciphers are composed of operations that achieve "confusion" and "diffusion" in transforming the plaintext message into the encrypted ciphertext [5]. Permutation is very effective in achieving diffusion in symmetric-key algorithms. Bit permutation is potentially a much more powerful operation than data-dependent rotation or multiplication for cryptographic algorithms. Arbitrary bit permutations achieve any one of $n!$ outcomes rather than just one of $n$ outcomes achieved by rotations. A permutation functional unit can be implemented with less area than a multiplier. An $n$-bit multiplier is typically four times the area of an $n$-bit ALU, and takes 3-5 times the latency. The question we examine in this paper is whether $n$-bit permutations can be achieved in less time and cost than $n$-bit multiplications, and not much incremental time and cost compared to the much smaller set of data-dependent rotations? This implies that a permutation functional unit should be smaller than a multiplier, take less than three cycles of latency, and ideally take just one or two cycles of latency like a data-dependent rotation. Such fast permutation operations can accelerate many important symmetric-key ciphers and unleash the opportunity to design faster new ciphers.

In Section 2, we describe past work on permutation instructions, including how recent past work has reduced the time taken to achieve any $n$-bit permutation down from $O(n)$ to $O(\log(n))$ instructions and cycles. In Section 3, we investigate the latency through different permutation functional units, in order to determine which of these can be executed in one cycle on a typical processor. In Section 4, we describe alternatives for achieving arbitrary bit permutations in at most two cycles. In Section 5, we compare these alternatives, and conclude in Section 6.

## 2. Past work

With existing processor instruction set architectures (ISAs), arbitrary bit permutations can be done using logical operations or table lookups. While some simple $n$-bit permutations can be accomplished with a few instructions, a generic way to achieve any arbitrary $n$-bit permutation with current ISAs takes $O(n)$ instructions, which is unacceptably slow. For example, each of the $n$ bits has to be selected (with an AND instruction), shifted to its new position (with a SHIFT instruction), and then combined (with an OR instruction) with previously permuted bits. While the number of instructions can be reduced by table lookup methods, these can only achieve a small set of fixed permutations due to the memory space required for each table representing a fixed permutation. Also, the memory latency and cache misses caused by table lookup methods can degrade performance significantly in terms of the actual execution cycles taken.

Recently, we have proposed several different methods to perform arbitrary $n$-bit permutations, reducing the number of instructions needed from $O(n)$ down to $O(\log(n))$. The new permutation instructions proposed each require two operands and one result, following standard processor conventions. Table 2 shows the number of instructions and cycles needed to achieve an arbitrary $n$-bit permutation, for $n$=64 bits, for each of these methods.

**Table 2: Maximum number of instructions and cycles for any 64-bit permutation**

| | GRP | OMFLIP | CROSS | SWPERM & SIEVE | PPERM | This paper |
|---|---|---|---|---|---|---|
| Max # of instructions | 6 | 6 | 6 | 11 | 15 | <6 |
| Max # of cycles | 6 | 6 | 6 | 4* | 5* | <4 |

* on 4-way superscalar processors

The GRP [6] instruction is based on partitioning the $n$ bits to be permuted into two parts, based on whether the corresponding configuration bit is 0 or 1. A sequence of $\log(n)$ GRP instructions can perform any of the $n!$ permutations of $n$ bits. The CROSS [7] or OMFLIP [8] instructions build a virtual Benes network or omega-flip network, respectively, to permute the $n$ data bits. A Benes network for permuting $n$ bits consists of a butterfly network followed by an inverse butterfly network, each of which has $\log(n)$ stages. Figure 1 shows an 8-bit butterfly network and an 8-bit inverse butterfly network, which can be used to form a 6-stage (2log(8)) Benes network. An omega-flip network is isomorphic to a Benes network. Each CROSS or OMFLIP instruction executes the equivalent of two stages of the network. Hence, both can achieve any one of the $n!$ permutations in at most $\log(n)$ instructions. The $\log(n)$ instructions form a data-dependent sequence, where an intermediate permutation generated by one instruction is used in the next permutation instruction. Consequently, these $\log(n)$ instructions cannot be executed in fewer than $\log(n)$ cycles, even when the processor can execute more than one instruction per cycle.

The PPERM [9] instruction and the pair of instructions, SWPERM and SIEVE [10], are based on selecting a source bit by its numeric index. They both need more than $\log(n)$ instructions to do an arbitrary $n$-bit permutation. However, these instructions have less serial data-dependencies and can be executed in fewer than $\log(n)$ cycles on processors which can execute more than one instruction per cycle.

In this paper, we improve upon these previous permutation methods. Our new method achieves an arbitrary 64-bit permutation in one or two cycles, rather than log(64)=6 cycles. Specifically, our goal is to achieve arbitrary $n$-bit permutations with:

1. Fewer than log($n$) instructions and cycles;
2. No significant increase in cycle time;
3. Low cost (low datapath and control path overhead).

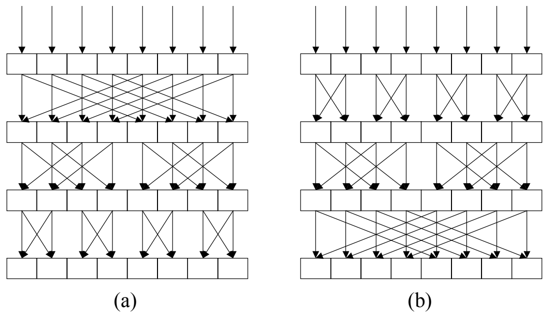


**Figure 1 (a) 8-input butterfly network (b) 8-input inverse butterfly network**

## 3. Cycle time and latency of permutation functional units

We now investigate the cycle time impact of different permutation functional units. Processor cycle time is often determined with respect to the ALU latency, because add, subtract, and logical instructions are usually the most frequently used instructions. Typically, an ALU instruction executes in a single cycle. We seek an $n$-bit permutation functional unit with a latency comparable to that of an $n$-bit ALU, so that the corresponding permutation instruction can achieve single-cycle execution in most processors.

Table 3 compares the latency of different permutation functional units in terms of the "logical effort" [11] required to implement them. Logical effort gives an estimate of the critical path of a circuit in a technology independent way. Latency is measured in terms of FO4 delays, where one FO4 is the delay of an inverter that drives four identical inverters. In Table 3, we compare the latencies of 64-bit permutation functional units implementing the GRP, OMFLIP, and CROSS instructions compared to a 64-bit ALU in terms of the number of FO4 delays.

**Table 3: Latency of different 64-bit functional units**

| | GRP | OMFLIP | CROSS | 6-stage butterfly (or inverse butterfly) | 6-stage omega (or flip) | ALU |
|---|---|---|---|---|---|---|
| Latency (in FO4) | 19.3 | 15.5 | 27.2 | 13.6 | 23.1 | 16-18 |

GRP uses a hierarchical network to partition data bits [12]. Its delay is slightly more than an ALU latency. Permuting $n$ bits with GRP instructions requires going through the GRP functional unit log($n$) times. An OMFLIP instruction can be implemented by a permutation functional unit with four stages, two omega stages and two flip stages, and some pass-through connections [8]. Only two of these four stages are used by one OMFLIP instruction. To permute 64 bits, a CROSS implementation needs a 12-stage Benes network, consisting of a 6-stage butterfly network followed by a 6-stage inverse butterfly network [9]. Although only two stages are used by a CROSS instruction, all 12 stages must be implemented because any two stages may be used. The 12-stage network yields a long latency of 27.2 FO4 units, much greater than an ALU latency.

We observe, however, that a 6-stage butterfly network has a delay shorter than one ALU latency. Both an *n*-input butterfly network and an *n*-bit ALU need almost the same number of log(*n*) stages; but the stages of the butterfly network are simpler since each stage is just a row of 2:1 multiplexors (see Figure 1). A 6-stage omega network has a longer latency because each stage has long wires, whereas some stages of a 6-stage butterfly network have very short wires.

This latency analysis suggests that an *n*-bit permutation can be achieved in two cycles, using a butterfly functional unit in the first cycle and an inverse butterfly functional unit in the second cycle. We propose two new instructions, BFLY and IBFLY, to achieve this.

We solve the remaining difficulty in the rest of this paper: how to get the many configuration bits to the butterfly network (or inverse butterfly network)? For *n*=64, a 6-stage butterfly network requires 3*n* configuration bits; *n*/2 = 32 bits for each stage. Together with the *n* data bits to be permuted, 4*n* bits, or four 64-bit source operands, need to be supplied to each BFLY or IBFLY instruction. In a typical microprocessor, only two source operands are supplied to each instruction.

## 4. Alternative ASIP architectures

We now examine how an operation with four source operands can be performed on a variety of ASIP architectures, where both the processor's instruction set architecture and micro-architecture have more flexibilities [13, 14].

We illustrate with the following example. Assume a 64-bit processor. The bits to be permuted are placed in R1. Both the butterfly functional unit and the inverse butterfly functional unit have log(64)=6 stages. The configuration bits for the 6-stage butterfly network are in R11, R12, and R13, and those for the 6-stage inverse butterfly network are in R14, R15, and R16.

Six CROSS instructions can perform any permutation of the 64 bits in R1, as shown below. The subop encodings, 5, 4,…, 0, indicate which of the six different stages of the butterfly or inverse butterfly network are used in a CROSS instruction. Because of data-dependencies between consecutive CROSS instructions, six cycles are needed for these six instructions.

```
I1: CROSS.5.4  R1, R1, R11        I4: CROSS.0.1  R1, R1, R14
I2: CROSS.3.2  R1, R1, R12        I5: CROSS.2.3  R1, R1, R15
I3: CROSS.1.0  R1, R1, R13        I6: CROSS.4.5  R1, R1, R16
```

We will show how this can be done in two cycles. We use BFLY to represent an instruction using a 6-stage butterfly network and IBFLY to represent an instruction using a 6-stage inverse butterfly network. We define these instructions, with minor variations, for each of six alternative solutions.

### 4.1 LdState

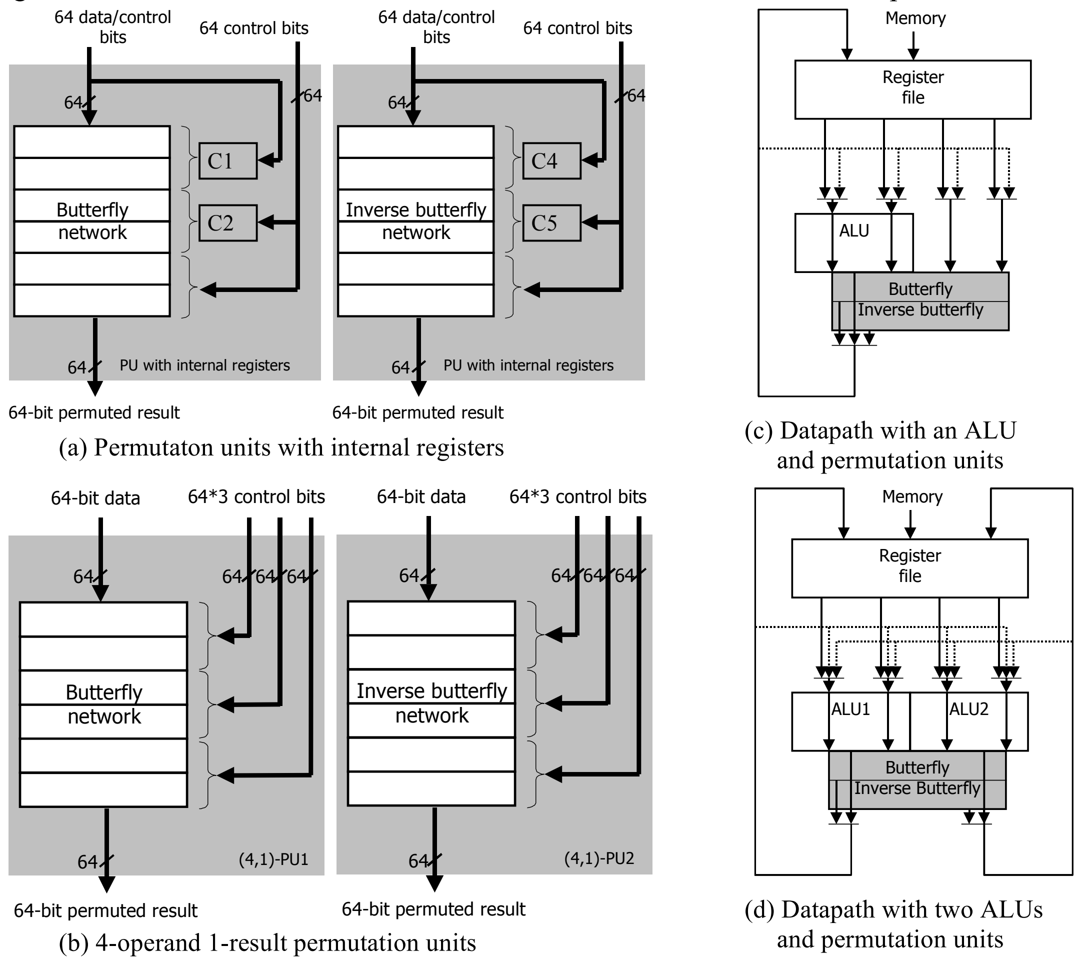
First, suppose the permutation functional units have internal storage to keep configuration bits, as shown in Figure 2a. We load configuration bits into the permutation units first, and then send the data bits (and another *n* control bits) to perform the permutation. Each network has two 64-bit internal registers to configure four stages.

The permutation instructions are defined in Figure 3a. LdState.bfly and LdState.ibfly load the value of two source registers into the internal registers, C1 and C2 in the butterfly network, and C4 and C5 in the inverse butterfly network, respectively. BFLY specifies the data to be permuted (Rs) and the configuration bits for the last two stages of the butterfly network (Rc3). The bits in Rs are permuted with the butterfly network configured by C1, C2, and Rc3 and then stored in the destination register Rd. Similarly, IBFLY permutes the bits in Rs with an inverse butterfly network configured with C4, C5, and Rc3, and saves the permuted bits in Rd. Four instructions can achieve any of the 64! permutations of the 64 bits in R1 in four cycles:

```
LdState.bfly   R11, R12
BFLY           R1, R1, R13
LdState.ibfly  R14, R15
IBFLY          R1, R1, R16
```

Since the configuration bits for the first four stages are still in the permutation units, if the same permutation needs to be performed again, only two instructions, BFLY and IBFLY, are needed, taking only two cycles. Since BFLY and IBFLY use different functional units, permutations on different data registers may be pipelined on multi-issue processors to achieve a throughput of one permutation per cycle.

When multiple processes share the same functional units, the internal registers have to be saved and restored during context switches, incurring operating system overhead. To save the internal registers, another instruction, MovePUtoGR, has to be defined to move the values from the permutation unit's internal registers to general registers. Four instructions are needed for this state saving, since four general registers have to be written, one per instruction. The general registers are then saved in the normal manner for context switches or interrupts.



(a) Permutaton units with internal registers

(c) Datapath with an ALU and permutation units

(b) 4-operand 1-result permutation units

(d) Datapath with two ALUs and permutation units

**Figure 2: Permutation units and processor datapaths**

## 4.2 Register pair

The LdState's internal registers increase the complexity of the permutation functional units, and cause context-switch overhead. Without the internal registers, four 64-bit values have to be provided to the butterfly network or the inverse butterfly network in one instruction. Normally, two source register specifiers are allowed in one instruction. In order to minimize the number of instructions, we can treat the four registers as two register pairs and define that each of the two source register specifiers in a permutation instruction specifies a register pair instead of a single register. The instructions are defined in Figure 3b. Although only two source registers, Rs1 and Rs2, are specified in the BFLY instruction, they actually refer to four registers: Rs1, R(s1+1), Rs2, and R(s2+1). The BFLY instruction permutes bits in Rs1 with configuration bits in Rs2, R(s2+1), and R(s1+1). Similarly, the IBFLY instruction permutes bits in Rs1 with configuration bits in Rs2, R(s2+1), and R(s1+1).

```
Ldstate.bfly   Rc1, Rc2
BFLY           Rd, Rs, Rc3
Ldstate.ibfly  Rc1, Rc2
IBFLY          Rd, Rs, Rc3
```

**(a)** LdState

```
BFLY      Rd, Rs1, Rs2
IBFLY     Rd, Rs1, Rs2
```

**(b)** Register pair

```
BFLY      Rd, Rs, Rc1, Rc2, Rc3
IBFLY     Rd, Rs, Rc1, Rc2, Rc3
```

**(c)** Two-length, v1

```
BFLY      Rd, Rc1, Rc2, Rc3
IBFLY     Rd, Rc1, Rc2, Rc3
```

**(d)** Two-length, v2

```
Cycle 1:  BFLY.ctl   Rc1, Rc2        BFLY    Rs, Rd, Rc3
Cycle 2:  IBFLY.ctl  Rc1, Rc2        IBFLY   Rs, Rd, Rc3
```

**(e)** Bundled, superscalar, and VLIW

**Figure 3: BFLY and IBFLY instruction variants**

The functional units shown in Figure 2b can perform BFLY and IBFLY defined here. They take four inputs and generate one result. We call this kind of permutation functional unit a (4,1)-PU. One of the four inputs is the data bits to be permuted, and the other three configure the butterfly or the inverse butterfly network. Four registers specified with two register pairs are read in one cycle, and all four values are fed into the (4,1)-PUs simultaneously. Figure 2c shows a processor datapath with (4,1)-PUs. The dashed lines are bypass paths.

This method requires a register file to have four read ports to read two register pairs in one cycle. Also, the decode logic has the overhead of checking the extra data availability for the BFLY and IBFLY instructions. In addition, this method causes compiler complexity due to the register pairing. Furthermore, one of the register pairs specifies both data bits and configuration bits, which may cause extra register move instructions.

## 4.3 Two-length ISA

Register pairing causes overhead to the compiler and decode/issue logic. To remedy this, we can explicitly specify four source registers in one instruction. The permutation instructions are defined in Figure 3c. The BFLY and IBFLY instructions permute bits in Rs with configuration bits in Rc1, Rc2, and Rc3 and put the permuted bits in Rd. It takes at most two instructions to achieve an arbitrary permutation of the *n* bits in R1 in our example.

The functional units and processor datapath are shown in Figure 2b and Figure 2c. Each BLFY and IBFLY takes only one cycle, and any permutation needs at most two cycles.

The problem with this method is that we may not have enough bits in the instruction word to specify all five registers. Suppose there are 32 registers, and an instruction length of 32 bits. We need five bits to specify each register and 25 bits for all five registers. Only seven bits are left in the instruction for specifying the opcode and other fields, which may not be enough. Longer instructions may have to be used. Hence, this method is called the two-length ISA method; each instruction is either a short instruction with two source operands or a longer instruction with four source operands. But variable length instructions introduce overhead to the fetch and decode logic. One alternative is to write the result into the original data register, as shown in Figure 3d. This can probably be encoded in 32 bits, allowing all instructions to have a fixed length of 32 bits. However, for ISAs with instructions shorter than 32 bits, permutation instructions may need to use a longer format, since they still have one extra register specifier.

Similar to the register pairing method, this method requires four register read ports. Because most other instructions use only two register read ports, it may be wasteful to add two register read ports just for the permutation instructions.

## 4.4 Bundled instruction

The two-length ISA complicates the fetch and decode logic. It may be desirable to use the same instruction format for permutation instructions as for other instructions. We achieve this by using a bundle of two instructions to deliver the four source registers: a BFLY.ctl instruction with a BFLY instruction, or an IBFLY.ctl instruction with an IBFLY instruction, as shown in Figure 2e. Each pair of instructions jointly specifies the data register Rs and the configuration registers: Rc1, Rc2, and Rc3. Instructions in a bundle must be executed together so that all four registers can be fed into the permutation units. Although we have four instructions, they can be executed in two cycles. The functional units and datapath are the same as shown in Figure 2b and Figure 2c.

This method incurs control overhead to detect bundles. For example, an instruction buffer is needed to store two instructions, and control overhead to detect a sequence of two bundled instructions. In addition, we still need four register read ports to accommodate a bundle of two instructions. The execution of a bundle will be fetch-bound in a single-issue processor where only one instruction can be fetched per cycle. This fetch bandwidth limitation can result in a 4-cycle latency for an *n*-bit permutation, even though the bundle can otherwise execute in 2 cycles.

## 4.5 Superscalar execution

The register pairing, two-length ISA, and bundled instruction methods use the datapath shown in Figure 2c. There are four register read ports and four source data buses with bypasses. But the ALU uses only two of them. The other two ports and buses are for permutation operations only. We may add another ALU into the datapath as shown in Figure 2d. This makes it possible to do 2-way superscalar execution; processors execute two ALU instructions simultaneously, utilizing all four input buses each cycle. For the permutation operations, we use the instructions defined in Figure 3e. In this case, the two instructions are just issued in the same cycle. Each instruction reads two registers. Four register values are fed into the (4,1)-PUs. Thus, an arbitrary *n*-bit permutation can be done in two cycles.

By increasing the issue width to 4-way superscalar execution, the butterfly functional unit and the inverse butterfly functional unit can perform different permutations in the same cycle. When performing multiple permutations, the process can be pipelined so that one permutation can complete every cycle. For example, when we perform the same permutation on R1, R2, R3, and R4, one permutation is completed every cycle starting from cycle 2.

```
Cycle 1:                                            BFLY.ctl R11,R12        BFLY R1,R1,R13
Cycle 2: IBFLY.ctl R14,R15      IBFLY R1,R1,R16     BFLY.ctl R11,R12        BFLY R2,R2,R13
Cycle 3: IBFLY.ctl R14,R15      IBFLY R2,R2,R16     BFLY.ctl R11,R12        BFLY R3,R3,R13
Cycle 4: IBFLY.ctl R14,R15      IBFLY R3,R3,R16     BFLY.ctl R11,R12        BFLY R4,R4,R13
Cycle 5: IBFLY.ctl R14,R15      IBFLY R4,R4,R16
```

This method incurs the control overhead for superscalar execution, which is significant if compared to single-issue execution. In addition, a pair of permutation instructions, e.g., BFLY.ctl and BFLY, needs to be detected and issued in the same cycle. However, compared to a conventional superscalar processor, the overhead of detecting and issuing a pair of instructions is relatively minor.

### 4.6 VLIW execution

Due to the complexity of control logic in superscalar execution, we can use VLIW (Very Long Instruction Word) to utilize the multiple ALUs in Figure 2d. The advantage of the VLIW execution is that the issue logic is much simpler than for the superscalar execution because the instructions to be executed together are already packed in one long instruction word during compile time. The issue logic does not need to perform complicated dependency checks. That complexity is done once at compile time. We use the same instructions defined in Figure 3e and require that an instruction bundle be put in the same VLIW instruction. Thus the proper instruction pair will always be executed simultaneously. To permute R1 in our example, four instructions are placed in two long instruction words, and they can be executed together with other instructions packed in the same long instruction words.

This method does not need to combine instructions or modify the issue logic. In addition, there are no internal registers in the permutation functional unit. VLIW execution achieves the same performance as superscalar execution with the same degree of instruction level parallelism, but with less control complexity.

## 5. Comparison

Table 4 compares the six alternative implementations of the BFLY and IBFLY permutation instructions. The last column shows the prior methods, GRP, OMFLIP, or CROSS permutation instructions, which all need log(*n*) instructions and cycles. The first two rows of Table 4 show that all the six new architectural alternatives for BFLY/IBFLY can do an arbitrary *n*-bit permutation in less than log(*n*) instructions, taking at most two cycles. This 2-cycle latency is achieved with simple single-issue processors.

With multi-issue processors which can issue two or four instructions each cycle, a maximum throughput of one permutation per cycle can be achieved, at the cost of more register ports and operand buses, and increased control complexity.

Only the superscalar and VLIW methods enable additional performance by allowing more than one ALU instruction to execute in the same cycle. This results in higher speedup for a symmetric-key cryptographic algorithm like DES, compared to the other 4 alternatives and the best prior work (last column).

**Table 4: Comparison of alternatives for 64-bit permutations**

| | **LdState** | **Reg-Pair** | **Two-length** | **Bundle** | **Super scalar** | **VLIW** | **log(n) methods** |
|---|---|---|---|---|---|---|---|
| # of instr. per permutation | 4/2* | 2 | 2 | 4 | 4 | 2 | **6** |
| # of cycles | 4/2* | 2 | 2 | 4/2** | 2 | 2 | **6** |
| Max throughput (# reg. Ports, # multi-issue) | 1 (4, 2) | 1 (8, 2) | 1 (8, 2) | 1 (8, 2) | 1 (8, 4) | 1 (8, 4) | **6** (2, 1) |
| Parallel ALU instructions | no | no | **no** | **no** | yes | yes | no |
| Speedup of DES | 1.09 | 1.10 | 1.10 | 1.10*** | 1.68 | 1.68 | **1** |
| OS complexity | **high** | none | none | none | none | none | none |
| Compiler complexity | low | **high** | low | low | low | low | none |
| ISA complexity | high | low | mid | low | low | **mid** | none |
| Datapath complexity | low | low | low | low | low | low | none |
| Control complexity | low | mid | **mid** | **mid** | **mid** | low | none |

* When the same permutation is repeated, only two instructions (and a latency of two cycles) are required.
** When the fetch/decode logic can process two instructions per cycle, the latency is two cycles.
*** Assumes 2 instructions fetched per cycle. If only 1 instruction fetched per cycle, speedup is only 1.03.

We also compare the implementation complexity in terms of operating system, compiler, ISA, datapath and control overhead. The entries in bold font indicate key reasons that make a method less desirable. The operating system overhead for context switches in the LdState method and the high compiler complexity in the Register Pair method are undesirable. The two-length ISA method and the bundle instruction method are less desirable than the 2-way superscalar method because the first two incur control complexity without the performance advantage of being able to execute ALU operations in parallel. Finally, the VLIW method has the same performance advantages as the superscalar method with significantly less control complexity, and hence may be the preferred method. This study allows the ASIP designer to choose the method that comes closest to meeting all his design goals.

## 6. Conclusions

We identify arbitrary bit permutation as a fundamental operation for the class of symmetric-key block ciphers that is not well supported in existing processor architectures. We identify two difficulties in implementing very fast arbitrary bit permutations: permutation functional units may have a latency longer than a typical cycle, and a large number of configuration bits are needed to specify an arbitrary permutation. This paper solves both problems.

To solve the latency problem, we compare the critical paths though different 64-bit permutation functional units with that of a typical 64-bit ALU. We find that the BFLY (or IBFLY) permutation functional unit has the shortest latency, shorter than an ALU. Hence, the cycle time of the processor would not be impacted; if an ALU instruction can be done in one cycle, so can a BFLY or IBFLY instruction. The BFLY permutation unit implements a log($n$)-stage butterfly network, while the IBFLY permutation unit implements a log($n$)-stage inverse butterfly network. By performing a BFLY followed by an IBFLY instruction, any arbitrary $n$-bit permutation is achievable in at most two cycles on a single-issue processor. In a multi-issue processor, a throughput of one permutation per cycle can be achieved.

To achieve this performance, we must be able to supply (log($n$)/2 +1) operands to a BFLY or

IBFLY permutation functional unit in a single cycle. For *n*=64, this is equal to four source operands per cycle. Illustrating with 64-bit processors, we present six solutions for achieving 4-source operations for ASIP architectures, which we call the LdState, Register-Pair, Two-length, Bundle, Superscalar, and VLIW methods. All these solutions take fewer than log(64)=6 instructions for an arbitrary 64-bit permutation. All require at most two cycles to achieve any arbitrary *n*-bit permutation, except LdState and Two-length which can take four cycles.

We compare the performance potential and implementation complexity of these solutions. It is interesting that even though ASIPs offer more flexibility in terms of ISA and micro-architecture choices when compared to general-purpose microprocessors, the highest performance solutions, Superscalar and VLIW, are in fact architectural techniques used by general-purpose processors. Hence, our BFLY and IBFLY permutation instructions can be integrated into either application-specific or general-purpose processors.

With these solutions, we can achieve arbitrary bit permutations at a performance level close to data-dependent rotations and an implementation complexity less than multiplication. Arbitrary bit permutation is potentially a much more powerful operation than data-dependent rotations or multiplications for symmetric-key cryptographic algorithms. By showing how arbitrary bit permutations can be performed by processors in one or two cycles with relatively simple hardware, we enable the acceleration of important existing ciphers and also provide new opportunities for the design of faster and more effective ciphers.

## 7. References

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

[2] NIST (National Institute of Standards and Technology),"Advanced Encryption Standard (AES) - FIPS Pub. 197", November 2001

[3] Carolynn Burwick, et. al., "MARS: a Candidate Cipher for AES", *NIST AES proposal*, June 1998

[4] Ronald Rivest, et. al., "The RC6 Block Cipher", *NIST AES proposal*, August 1998

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

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

[7] Xiao Yang, Manish Vachharajani and Ruby B. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks", *Proceedings of Media Processors 1999 IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, pp. 80-86, January 2000

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

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

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

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

[12] Zhijie Shi and Ruby B. Lee, "Subword Sorting with Versatile Permutation Instructions", *Proceedings of the International Conference on Computer Design (ICCD 2002)*, pp. 234-241, September 2002

[13] Ricardo Gonzalez, "XTENXA: a Configurable and Extensible Processor", *IEEE Micro,* Vol. 20, No. 2, pp.60-70, April 2000

[14] Atsushi Mizuno, Kazuyoshi Kohmo, et al, "Design Methodology and System for a Configurable Media Embedded Processor Extensible to VLIW Architecture", *Proceedings of ICCD 2002 IEEE Internaitonal Conference on Computer Design,* pp. 2-7, September 2003