Fast Subword Permutation Instructions Using Omega and Flip Network Stages

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

Abstract

This paper proposes a new way of efficiently doing arbitrary n-bit permutations in programmable processors modeled on the theory of omega and flip networks. The new `omflip` *instruction we introduce can perform any permutation of n subwords in $\log n$ instructions, with the subwords ranging from half-words down to single bits. Each* `omflip` *instruction can be done in a single cycle, with very efficient hardware implementation. The* `omflip` *instruction enhances a programmable processor's capability for handling multimedia and security applications which use subword permutations extensively.*

1 Introduction

Multimedia applications often deal with relatively low precision data and have high levels of data parallelism. Multimedia instructions using subword arithmetic are adopted into many modern microprocessor architectures to accelerate multimedia processing. Examples are MAX[2] and MAX-2[3] extensions to HP PA-RISC architecture, MMX[7], SSE[8] and SSE 2[9] extensions to Intel IA-32 architecture, IA-64[1], 3DNow![6] for AMD x86, VIS[13] for Sun SPARC and AltiVec[10] for PowerPC. The essence of these multimedia instructions is to pack several pieces of low precision data, called subwords, into a single machine word so that they can be processed in parallel using one instruction. These subword parallel instructions greatly improve the performance of many multimedia applications. However, they also raise some problems. An important one, the subword rearrangement problem, arises when the subwords in a word are not in the desired order. Although this situation does not exist in traditional architectures, it naturally occurs with the use of subword parallelism. We need to be able to rearrange subwords in an efficient and easy manner when necessary.

1.1 Past Work

Existing software methods. When instruction set architecture support for subword rearrangement is not available, table lookup methods and solutions based on conventional logical and shift, or extract and deposit, instructions are used to solve the subword rearrangement problem. These methods are quite slow for achieving arbitrary n-bit permutations. More recently, several subword rearrangement instructions were proposed and implemented in different architectures. Examples are the `mix` and `permute` instructions in MAX-2[3], `mix` and `mux` instructions in IA-64[1], and the `vperm` instruction in AltiVec[10]. All of these instructions can do certain types of subword rearrangements quite efficiently for subword sizes of 8 bits or larger. However, the concept of subword encompasses data ranging from full words to halfwords to single bits. Permutations of small subword sizes down to a bit are particularly important for accelerating cryptography, which is becoming increasingly important for secure communications and processing. None of these currently implemented subword rearrangement instructions provides a general solution for efficiently doing arbitrary permutations for all possible subword sizes.

`cross` instruction. In a previous paper[14], we proposed a new subword permutation instruction, the `cross` instruction. The `cross` instruction is able to do all possible permutations for all subword sizes efficiently. The idea is based on the fact that an n-input Benes network, which is formed by connecting two n-input butterfly networks back-to-back, can produce any permutation of its n inputs with edge-disjoint paths(see Figure 1(a)). Furthermore, Benes networks can be broken into separate butterfly network stages.

The `cross` instruction has the normal 3-operand format

```
cross,m1,m2  rd,rs,rc
```

`rs` and `rd` specifies the registers containing the bits to be permuted and the permuted bits, respectively. `rc` is the register holding the configuration bits. One `cross` instruction

0-7695-0801-4/00 $10.00 © 2000 IEEE

executes the operations of two butterfly stages. These two stages are specified by m1 and m2, where 2^{mi} represents the distance between paired inputs.

To use the cross instruction, we first obtain a valid configuration on the Benes network for the desired permutation using an algorithm presented earlier[14]. Then we break the configured network into pairs of stages. For each pair of stages, we assign a cross instruction. By chaining these cross instructions in sequence, we virtually construct a Benes network that is configured for the desired permutation. An example of a cross instruction sequence is given in Figure 1(c). Since there are $2 \log n$ stages in an n-input Benes network, we need at most $\log n$ cross instructions to perform any n-bit permutation.

Because a cross instruction may use any two stages in a Benes network in any order, we need to have a full Benes network in hardware to implement cross instructions. Alternatively, we can have two stages in hardware, but each stage needs to contain all the necessary connections for an n-input butterfly network. Neither of these two implementations is efficient in terms of hardware implementation.

pperm and grp instructions. Shi and Lee proposed two permutation instructions, pperm and grp[12]. They can do arbitrary n-bit permutations in $O(\log n)$ instructions. However, their circuit implementations are also not the most efficient. Lee also proposed subword permutation instructions for two-dimensional multimedia processing[4], but these have not been shown to be effective for arbitrary bit-level permutations.

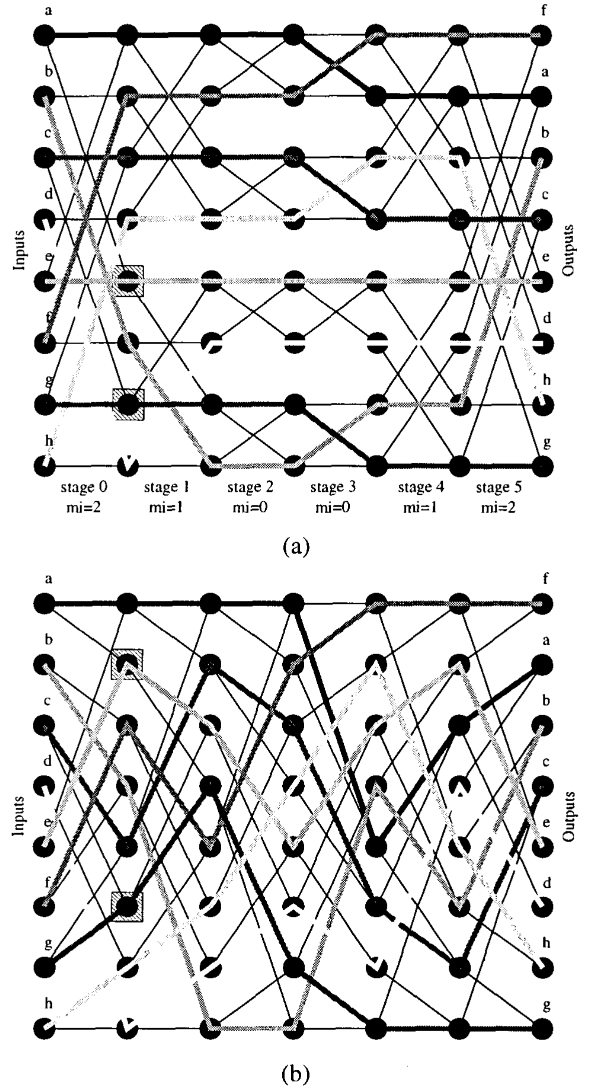
1.2 Objective of This Paper

Our objective is to explore alternative possibilities for solving the general subword permutation problem, for all subword sizes down to a single bit. We are trying to find an efficient approach that can achieve at least the same level of performance as the cross instruction, but can yield more efficient circuit implementations.

We chose the butterfly network as the basis for our cross instructions because it has certain desirable properties. It can be broken into stages that are simple to specify, and an n-input Benes network is able to do all permutations of its n inputs. However, one disadvantage of the butterfly network is that all its stages are distinct. From a circuit point of view, it is beneficial to use networks that have uniform stages, which can result in a smaller hardware implementation.

2 Omega and Flip Networks

The omega network has uniform stages, i.e., each stage is identical. This is also true for the flip network, which is a mirror image of the omega network[5]. Omega networks



```
cross,2,1  r1,r1,r2
  ;r2=10001010b

cross,0,0  r1,r1,r3
  ;r3=10110000b

cross,1,2  r1,r1,r4
  ;r4=01000000b
```

(c)

```
omflip,00  r1,r1,r2
  ;r2=10001010b

omflip,10  r1,r1,r3
  ;r3=10110000b

omflip,11  r1,r1,r4
  ;r4=01000000b
```

(d)

Figure 1. (a) Benes network configured for the permutation $(abcdefgh) \rightarrow (fabcedhg)$. (b) Omega-flip network configured for the same permutation. (c) cross **instruction sequence for (a). (d)** omflip **instruction sequence for (b).**

are isomorphic to butterfly networks and flip networks are isomorphic to inverse butterfly networks, as shown in Figure 2 for 8-input networks.

As a result of the isomorphism, the properties of omega and flip networks are similar to those of butterfly networks. The total number of stages in an n-input omega network or flip network is $\log n$ and the number of nodes in each stage is n. A *node* is where the path selection for an input takes place. In each stage of an omega network or flip network, for every input, there is another input that shares the same two outputs with it. We call these pairs of inputs *conflict inputs* and their corresponding outputs *conflict outputs.*

An *omega-flip network* is formed by connecting an n-input omega network and an n-input flip network. It can be used to perform any permutation of its n inputs with *edge disjoint paths*[5]. It is functionally equivalent to an n-input Benes network.

3 Architectural Implementation Based on Omega-flip Network

3.1 Basic Operations

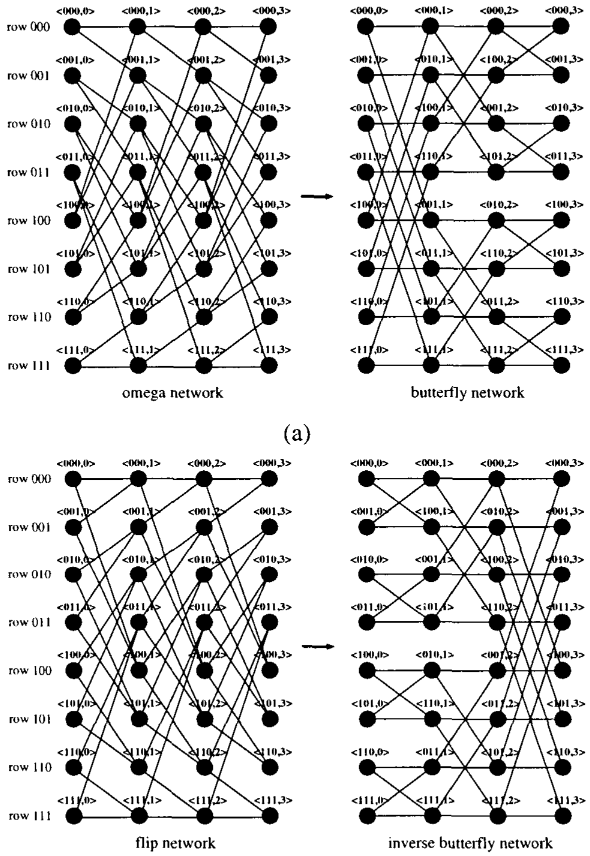
The basic operations for our proposed permutation instruction correspond to the single stage operations in an omega network or flip network. We call them the *omega operation* and the *flip operation*, respectively. Each of these two basic operations has two source operands: the bits to be permuted and the configuration specification. Bits from the source register are moved to the result register based on the configuration bits. If the configuration bit for a pair of conflict inputs is 0, the bits from these two inputs go through non-crossing paths to the outputs. Otherwise, the bits go through crossing paths to the outputs.

3.2 `omflip` Instruction

For each of the basic operations introduced above, we only need $n/2$ bits to specify the configuration for n input bits. Therefore, for permuting the contents in an n-bit register, we are able to pack the configuration bits for two basic operations into one configuration register and thus pack two basic operations into one single instruction. The instruction format for our permutation instruction is

```
omflip,c  rd,rs,rc
```

`rs` is the source register containing the subwords to be permuted, `rd` is the destination register where the permuted subwords are placed, and `rc` is the configuration register that holds the configuration bits for the two basic operations. `c` is a sub-opcode that indicates which two basic operations are used in this instruction. It contains two bits. For



(a)

(b)

Figure 2. Correspondence between nodes in (a) an omega network and a butterfly network. (b) a flip network and an inverse butterfly network.

each bit, 0 indicates that an omega operation is used and 1 indicates that a flip operation is used. There are four combinations of c: omega-omega, omega-flip, flip-omega and flip-flip. The first basic operation is determined by the right bit of c. It takes the source register rs and moves the bits in it based on the least significant half of the configuration register rc to an intermediate result. The second basic operation is determined by the left bit of c. It moves the bits in the intermediate result according to the most significant half of rc to the destination register rd.

An example instruction sequence is shown in Figure 1(d). For instance, the first omflip instruction, omflip,00 r1,r1,r2 performs two omega operations. The first omega operation uses the configuration bits from the least significant half of r2, 1010b. The configuration bit is 0 for nodes containing a and e, 1 for nodes containing b and f, etc. Therefore, in the first omega operation, a and e go through non-crossing paths, b and f go through crossing paths, and so on. The intermediate result after the first omega operation is $(aefbcghd)$. For the second omega operation, the configuration bits, 1000b, come from the most significant half of r2. Only the configuration bit for nodes holding b and d is 1. Therefore, only b and d go along crossing paths and all others go along non-crossing paths. The result after the first omflip instruction is $(acegfhdb)$, which corresponds to the result after the leftmost two stages of the omega-flip network shown in Figure 1(b).

3.3 Using omflip Instruction

In order to use omflip instructions to do arbitrary permutations, we first need to obtain the configuration of an omega-flip network for the desired permutation. Since there is a one-to-one correspondence between nodes in an n-input omega-flip network and an n-input Benes network, we can first obtain a valid configuration on a Benes network for the desired permutation, then translate the Benes network configuration to an omega-flip network configuration. For each node in an omega-flip network, we find its corresponding node in the Benes network and use this node's configuration for the corresponding node in the omega-flip network.

An example is given in Figure 1. We first configure the Benes network for the permutation $(abcdefgh) \rightarrow (fabcedhg)$ and obtain the configuration bits for each node. These configuration bits are shown in Figure 1(c) as the contents of the configuration registers r2, r3 and r4 for the cross instructions. The right half of the configuration register is for the even stage and the left half for the odd stage. The configuration bits are read from right to left as we go through nodes from top to bottom. For instance, the fifth and seventh nodes in the second stage of the Benes network are configured using one bit. This bit is the second bit from the left in r2, which is 0. This bit is also used to configure the second and the sixth nodes in the second stage of the omega-flip network in Figure 1(b), which correspond to the fifth and seventh nodes in the Benes network. This bit is the third bit from the left in r2 of the first omflip instruction.

After the omega-flip network is configured, we break it into pairs of stages. For each pair of stages, we assign an omflip instruction. This sequence of omflip instructions achieve the desired permutation.

Using this method, we can do all possible permutations(without repetition) of the n bits in an n-bit register using $\log n$ omflip instructions.

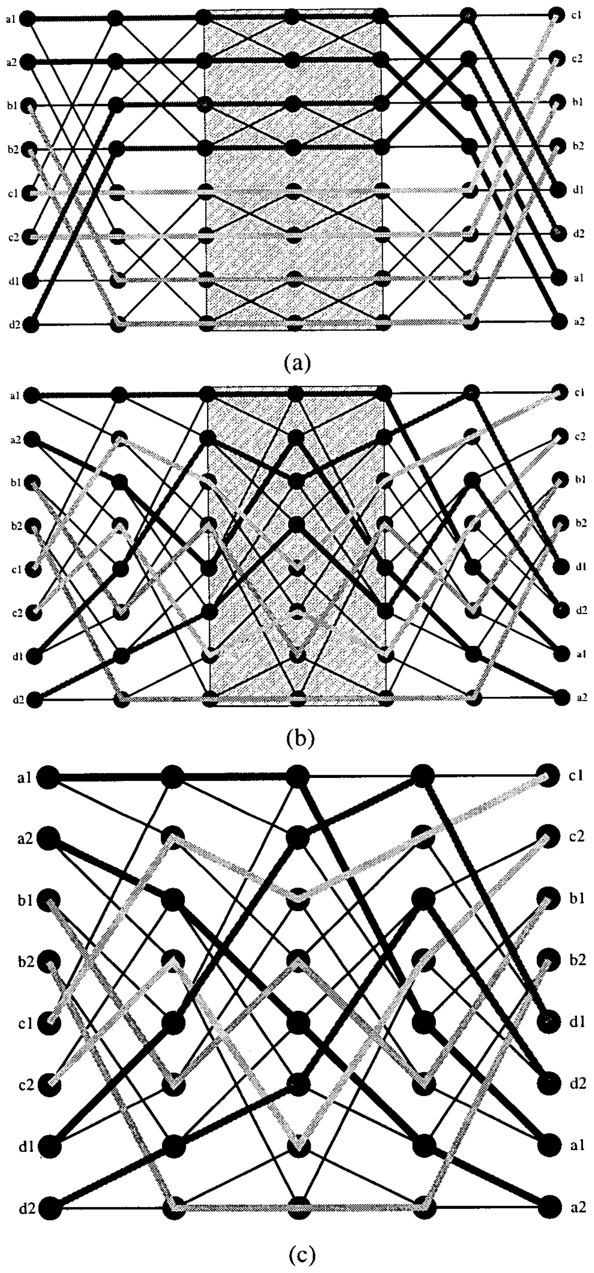
3.4 Permuting Multi-bit Subwords

When a Benes network is configured for r-bit subword permutations using our configuration algorithm, the middle $2\log r$ stages are configured as pass-throughs(see Figure 3(a)). Similarly, for r-bit subword permutation, the middle $2\log r$ stages of the omega-flip network also copy the input bits to their output without any change of order(see Figure 3(b)). Hence, we can eliminate them from the configured omega-flip network(see Figure 3(c)) and assign instructions to the remaining stages without affecting the result. Therefore, when permuting r-bit subwords in an n-bit word, the maximum number of instructions needed becomes $\log n - \log r = \log(n/r) = \log n'$, where n' is the number of subwords in a word.

4 Circuit Implementation

At the circuit level, to implement the four variants of omflip instructions, we only need to have two omega stages and two flip stages, rather than $2\log n$ stages for the entire omega-flip network. When executing an omflip instruction, the control logic selects the proper two stages for the two basic operations based on the sub-opcode c. It then configures these two selected stages according to the least significant half and most significant half of the configuration register rc. The stages that are not used are configured as pass-throughs. Notice that neither an omega stage nor a flip stage has pass-through connections except for the two nodes at the ends. We have to put bypassing connections on top of the two omega stages and the two flip stages so that all of the stages can be configured as pass-throughs. A conceptual diagram of the circuit implementation is shown in Figure 4 where the additional bypassing connections are shown using thick lines. A circuit diagram is shown in Figure 5.

For comparison, we show the circuit implementation for the cross instructions in Figure 6. We also show a circuit implementation for a crossbar network with the same number of inputs in Figure 7. We calculate their track counts



(a)

(b)

(c)

Figure 3. (a) 8-input Benes network configured for 2-bit subword permutation $(a_1a_2b_1b_2c_1c_2d_1d_2) \rightarrow (c_1c_2b_1b_2d_1d_2a_1a_2)$. (b) 8-bit omega-flip network configured for the same permutation. (c) Eliminating the two middle stages of the omega-flip network.

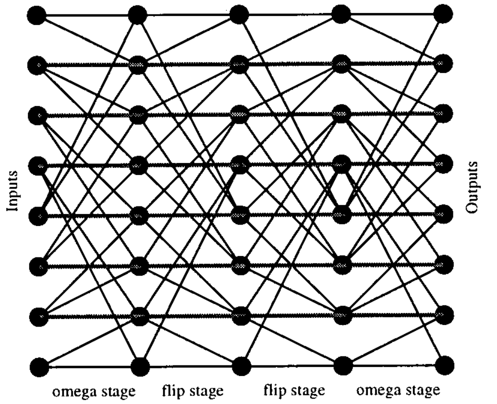


Figure 4. Conceptual diagram for the `omflip` circuit implementation.

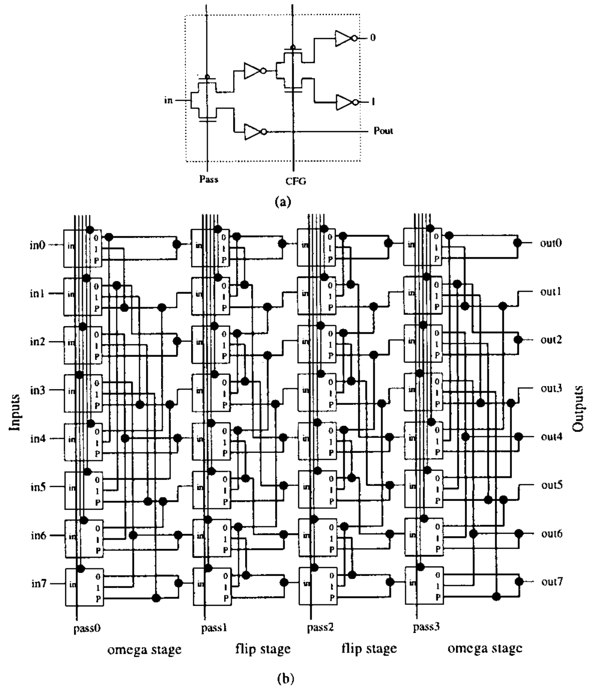


Figure 5. Schematic diagram for the circuit implementation of `omflip` instructions showing (a) an individual node. (b) an 8-bit implementation.

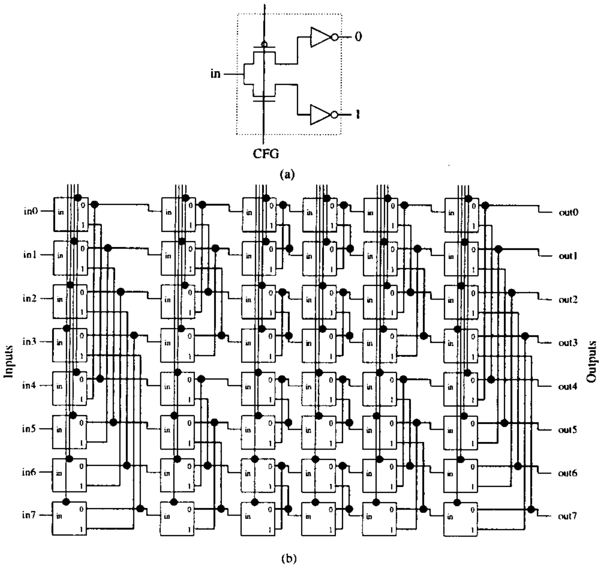


Figure 6. Schematic diagram for the circuit implementation of `cross` instructions showing (a) an individual node. (b) an 8-bit implementation.

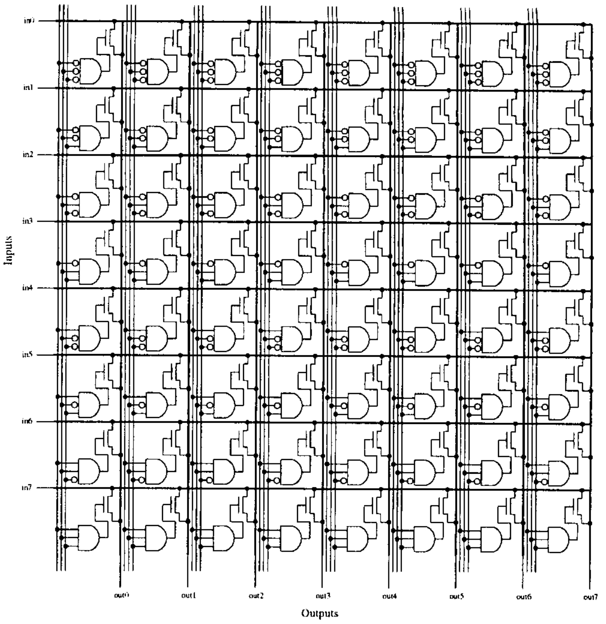


Figure 7. Schematic diagram for a circuit implementation of an 8-input crossbar network.

and transistor counts to give a rough idea of their sizes. The numbers are summarized in Table 1.

The numbers in Table 1 are computed as follows:

For the `omflip` implementation, we have

$$\begin{aligned} H\_Tracks &= 3n \\ V\_Tracks &= 4 \times (1 + \frac{n}{2}) + O(n) \\ &= 4 + 2n + O(n) \approx 4 + 6n \\ Transistors &= 4n \times 12 = 48n \end{aligned}$$

The $3n$ horizontal tracks come from the 3 output lines in each node. The number of vertical tracks is composed of three parts: 4 pass signals for the 4 stages, $n/2$ configuration lines per stage for the 4 stages, and the number of data tracks needed between adjacent stages, which is $O(n)$(about $4n$). The $48n$ transistors come from 12 transistors in each cell for $4n$ cells.

For the `cross`(Benes) implementation, we have

$$\begin{aligned} H\_Tracks &= 2n \\ V\_Tracks &= 2\log n \times \frac{n}{2} + 2 \times (2n - 2) \\ &= n \log n + 4n - 4 \\ Transistors &= 2n \log n \times 6 = 12n \log n \end{aligned}$$

The $2n$ horizontal tracks come from the 2 output lines in each node. The calculation of the vertical track number is similar to the calculation for `omflip` implementation minus the pass signals. The $12n \log n$ transistors are from 6 transistors in each cell for $2n \log n$ cells.

For the crossbar implementation we show in Figure 7,

$$\begin{aligned} H\_Tracks &= n \\ V\_Tracks &= n \times (1 + \log n) = n + n \log n \\ Transistors &= n \times (n + \sum_{i=0}^{\log n} \binom{\log n}{i} (2 \log n + 2i)) \\ &= O(n^2 \log n) > 3n^2 \log n \end{aligned}$$

The horizontal tracks consist of the n input data lines. The vertical tracks consist of the n output data lines and the $\log n$ configuration lines for each output data line. The number of transistors are for the AND gate and pass transistor at each cross point. An alternative implementation of crossbar is to provide a negated signal for each control signal so that no inverters before AND gates are needed. Then the vertical track count becomes $n + 2n \log n$ and the transistor count becomes $n^2(1 + 2\log n)$. This implementation may yield a larger size due to more vertical tracks used.

From these equations, we see that when n is large, the `omflip` implementation should yield the smallest size. As we see in table 1, the `omflip` circuit implementation yields the smallest transistor count and reasonable track counts for permutations of 64 bits. Therefore, it should yield the

		H tracks	V tracks	Transistors

(a)	Omega-flip
	(`omflip`)
		24
		24(data)
			50
			30(data)
			20(control)
				384

	Benes
	(`cross`)
		16
		16(data)
			52
			28(data)
			24(control)
				288

	Crossbar	8
		8(data)
			32
			8(data)
			24(control)
				640

(b)	Omega-flip (`omflip`)	192 192(data)	≈400 ≈250(data) 132(control)	3072

	Benes
	(`cross`)
		128
		128(data)
			636
			252(data)
			384(control)
				4608

	Crossbar	64
		64(data)
			448
			64(data)
			384(control)
				>73728

Table 1. Comparison of estimated horizontal and vertical track counts and transistor counts for circuit implementations of `omflip` instructions, `cross` instructions and a crossbar network for (a) 8-bit permutations and (b) 64-bit permutations.

most area-efficient implementation. Notice that we have not counted the control logic circuits for generating the configuration signals, which are more complex for the `cross` implementation and the crossbar than for `omflip`.

5 Performance

5.1 Arbitrary Permutations of a 64-bit Word

Table 2 shows the number of instructions needed for permutations of a 64-bit word with different subword sizes for different methods: using `omflip` instructions, using `cross` instructions, and the best method with existing instruction set architectures (ISAs).

The solution based on `omflip` instructions has similar performance to that based on `cross` instructions. The average performance, especially for large subword sizes, can be improved by instruction sequence optimizations. For example, we have not yet exploited the use of flip-omega operations provided by the `omflip` instruction.

Subword size in bits	Num of subwords in register	Max[a] num of `omflip`	Max[a] num of `cross`	existing ISAs
1	64	6	6	30[b]
2	32	5	5	30[b]
4	16	4	4	30[b]
8	8	3	3	1[c d]
16	4	2	2	1[c]
32	2	1	1	1[c]

[a] The maximum number here is $\log n$.

[b] Instruction counts using table lookup methods, actual cycle counts will be larger due to cache misses.

[c] Using subword permutation instructions.

[d] Only `vperm` in AltiVec is able to do this in one instruction.

Table 2. Comparing instruction counts for doing arbitrary permutations of subwords.

5.2 Permutations in DES

We measure the performance gain of our `omflip` instructions for a permutation in a real cryptography program, the Data Encryption Standard, DES[11].

The initial permutation in DES is a fixed 64-bit permutation done for each 64-bit data block at the beginning of encryption or decryption. We do this permutation in three different ways: table lookup, logical operations and `omflip` instructions. When using the table lookup method, the permutation can be achieved by referencing eight 256-entry tables:

$$\begin{aligned}
res \;\; = \;\; & IP\_tbl1[(src\&0x00000000000000ff)] \\
\mid \;\; & IP\_tbl2[(src\&0x000000000000ff00) >> 8] \\
\mid \;\; & IP\_tbl3[(src\&0x0000000000ff0000) >> 16] \\
\mid \;\; & IP\_tbl4[(src\&0x00000000ff000000) >> 24] \\
\mid \;\; & IP\_tbl5[(src\&0x000000ff00000000) >> 32] \\
\mid \;\; & IP\_tbl6[(src\&0x0000ff0000000000) >> 40] \\
\mid \;\; & IP\_tbl7[(src\&0x00ff000000000000) >> 48] \\
\mid \;\; & IP\_tbl8[(src\&0xff00000000000000) >> 56]
\end{aligned}$$

which is mapped to 30 instructions on a 64-bit machine. Using logical operations, the permutation can be done using 15 XOR's, 10 SHIFT's and 5 AND's on a 32-bit architecture, as in the libdes implementation[15]. This implementation can be mapped to 34 instructions on a 64-bit machine. With our `omflip` instructions, we only need 6 `omflip` instructions to do this permutation for a 5× speedup over the existing table lookup or logical operations approaches.

The entire DES program consists of two parts, encryption or decryption, and key scheduling. We compare the

	Encryption/decryption	Key scheduling
Table lookup	1	1
omflip	1.33	16.55

Table 3. Speedup of omflip over table lookup for DES.

performance of the implementation using omflip permutation instructions with that using the traditional table lookup method by simulation. Table 3 shows the speedup we achieve using omflip for a 2-way superscalar architecture with 1 load-store unit and a cache system similar to Pentium III processors. The huge speedup for the key scheduling is due to the many different permutations used and the cache misses generated by the table lookup method.

6 Conclusion

The omflip instructions for doing arbitrary subword permutations achieve good performance for permuting subwords of different sizes. The maximum number of omflip instructions needed for permuting n subwords is $\log n$. It is $2 \log n$ if we take into account the load instructions for the configuration registers. The performance of omflip instructions is comparable to the best method with existing architectural support when dealing with large subwords. When permuting small subwords, our method significantly outperforms the best existing method, because there is currently no architectural support for arbitrary permutations of small subwords and we have to fall back to the slow table lookup or logical operations approaches. In this aspect, our method is particularly advantageous because permuting subwords of one bit each is very important for fast cryptography. The circuit implementation for the omflip instruction is also efficient. We are able to reduce the number of stages required from $2 \log n$ for the cross instruction to only 4 stages for the omflip instruction. We also reduce the number of transistors needed from $O(n \log n)$ for cross instructions and $O(n^2 \log n)$ for crossbar implementations down to $O(n)$ for omflip instructions, which is great savings when n is large.

References

[1] IA-64 Application Developer's Architecture Guide. Technical report, Intel Corp., May 1999. http://developer.intel.com/design/ia64.

[2] R. B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

[3] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[4] R. B. Lee. Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures. In *Proceedings of the IEEE International Conference on Application -Specific Systems, Architectures, and Processors*, pages 3–14, July 2000.

[5] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan-Kaufmann Publishers, Inc., San Mateo, California, 1992.

[6] S. Oberman, F. Weber, N. Juffa, and G. Favor. AMD 3Dnow! Technology and the K6-2 Microprocessor. August 1998. Advanced Micro Devices, Inc., California Microprocessor Division.

[7] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[8] Intel Architecture Software Developer's Manual. Technical report, Intel Corp., 1999. http://developer.intel.com/design/PentiumIII.

[9] Willamette Processor Software Developer's Guide. Technical report, Intel Corp., 2000. http://developer.intel.com/design/processor/future.

[10] "AltiVec Extension to PowerPC" Instruction Set Architecture Specification. Technical report, Motorola, Inc., May 1998. http://www.motorola.com/AltiVec.

[11] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, New York, second edition, 1996.

[12] Z. Shi and R. B. Lee. Bit Permutation Instructions for Accelerating Software Cryptography. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 138–148, July 2000.

[13] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, August 1996.

[14] X. Yang, M. Vachharajani, and R. B. Lee. Fast Subword Permutation Instructions Based on Butterfly networks. In *Proceedings of SPIE, Media Processor 2000*, pages 80–86, January 2000.

[15] E. Young. libdes DES implementation, January 1997. ftp://ftp.psy.uq.oz.au/pub/Crypto/DES/.