Fast Subword Permutation Instructions Based on Butterfly Networks

Xiao Yang, Manish Vachharajani and Ruby B. Lee
Department of Electrical Engineering
Princeton University
Princeton, NJ 08540
{xiaoyang, manishv, rblee}@ee.princeton.edu

ABSTRACT

Many contemporary microprocessor architectures incorporate multimedia extensions to accelerate media-rich applications using subword arithmetic. While these extensions significantly improve the performance of most multimedia applications, the lack of subword rearrangement support potentially limits performance gain. Several means of adding architectural support for subword rearrangement were proposed and implemented but none of them provide a fully general solution. In this paper, a new class of permutation instructions based on the butterfly interconnection network is proposed to address the general subword rearrangement problem. It can be used to perform arbitrary permutation(without repetition) of n subwords within $\log n$ cycles regardless of the subword size. The instruction coding and the low-level implementation for the instructions are quite simple. An algorithm is also given to derive an instruction sequence for any arbitrary permutation.

1. INTRODUCTION

The rapid growth of multimedia applications places huge demands for computing power on microprocessors. Several multimedia extensions were introduced into microprocessor architectures to accelerate multimedia processing. Examples are HP's MAX-1[1] and MAX-2,[2] Sun's VIS,[3] Intel's MMX,[4] AMD's 3DNow[5] and recently the AltiVec[6] extension to PowerPC. All these extensions take advantage of the fact that most multimedia applications have a high level of data parallelism and use low-precision data. The idea is to pack several pieces of low-precision data, called subwords,[1,2] into a single machine word so that they can be processed by one instruction in parallel. The number of instructions for data processing can be greatly reduced. Although the performance gain from these extensions is significant, the extensions contain mostly parallel versions of conventional instructions. Support for subword rearrangement is quite limited, which may limit the performance gain achieved by using subword arithmetic.

1.1. Previous Work

A few microprocessor architectures have subword rearrangement instructions. First, HP implemented the `mix` and `permute` instructions in their MAX-2 extension to PA-RISC.[2] Later Intel and HP put `mix` and `mux` instructions into the IA-64 architecture,[7] which are extensions to the `mix` and `permute` instructions of MAX-2. Motorola, IBM and Apple also have the `vperm` instruction in their AltiVec extension to PowerPC.[6]

Two approaches are used in the above implementations. One is to design primitives that can do permutations of fixed patterns. Since the patterns are independent of subword size, the same instructions can be used to do permutations for all subword sizes. The other approach is to design a single instruction such that it can do all possible permutations for subwords greater than a certain size. Since larger subwords result in a smaller number of subwords per register, fewer configuration bits are needed to specify all possible permutations, in an immediate or in a register. The `mix` instructions belong to the first category, `permute` and `vperm` belong to the second category, and `mux` belongs to both categories for different subword sizes. `permute` and `mux` use an immediate in the instruction to specify the permutation to be performed. `vperm` uses a separate register to specify the permutation, therefore a load to the configuration register is needed before doing the permutation. In addition, `vperm` needs three source registers, two for the data to be permuted and one for the permutation configuration. For architectures that don't have support for subword rearrangement, solutions using conventional instructions must be used, which are usually much slower.

Because instructions based on the `mix` approach can only do fixed permutations, it is not efficient to do all possible permutations using these instructions alone. Instructions based on the second approach can not be used to handle

Proceedings of SPIE, *Media Processor 2000*, January 27-28, 2000, San Jose, California, pp. 80-86

subword sizes smaller than a fixed size because there are not enough bits to hold the parameters in the instruction or in a register, which is especially the case for bit permutations. Furthermore, the second approach normally requires crossbar networks at circuit level, which can be quite costly. Neither of the current approaches can be used to do all permutations for all subword sizes with reasonable speed and hardware efficiency.

1.2. Our Objective

Our goal is to provide a cost-effective solution that can handle all permutations for all subword sizes fast. For n elements, there are $n!$ different permutations without repetition. Specifying a particular permutation requires approximately $n \log n$ bits. Suppose n bits can be specified in one instruction, where n is the number of bits in a register, at most $\log n$ instructions are needed. Based on these observations we desire the following properties for our instructions.

- The same set of instructions can be used to permute subwords of all possible sizes using the same hardware. The hardware should be easy to design.
- The number of instructions for any permutation should be at most $\log n$ where n is the number of bits in a register.
- There is a simple algorithm to derive an instruction sequence for any arbitrary permutation.
- No processor state other than register values need to be maintained during the process of permutation.

2. BACKGROUND

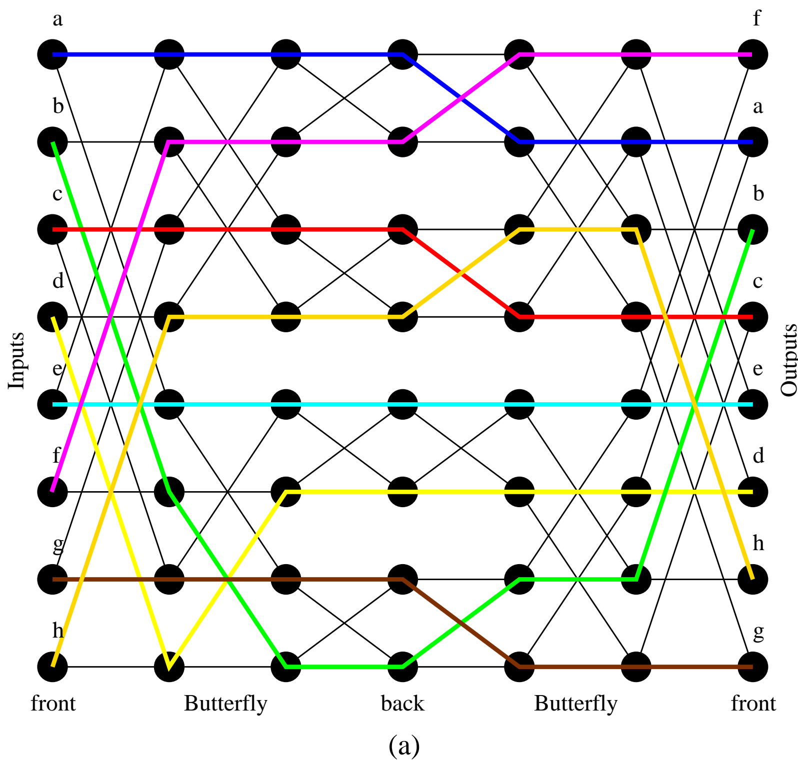
Various interconnection networks have been used to solve permutation problems in many areas. These networks have different properties that fit different applications. We wish to find an appropriate type of network for use in a microprocessor that is more area-efficient than a full $n \times n$ crossbar switch. For instructions based on crossbar networks, since there are not enough bits in a register to hold the configuration of the network when the subword size is small and the network does not produce useful results when partially configured, we may have to introduce additional processor state such as partial crossbar configuration or partial register writeback to permute small subwords. To avoid these undesirable artifacts of crossbar networks, we turn to more efficient interconnection networks.

2.1. Butterfly Network and Its Properties

Butterfly network and its isomorphic networks have been widely used in communications to solve the non-blocking switching problem, which is similar in nature to the permutation problem. Detailed discussion of butterfly networks can be found in the literature,[8] and thus is not repeated here. A 3-dimensional(8-input) butterfly network is shown in Figure 1. Butterfly networks have some nice properties that make them attractive for a solution to our subword permutation problem.

- The network formed by two $\log n$-dimensional butterfly networks connected back-to-back, called a Benes network, can be used to perform any permutation of its n inputs with *edge disjoint paths*, i.e. no two paths share the same crosspoint. One example is given in Figure 1(a).
- It can be seen that the butterfly network can be broken into stages. Therefore we can use the network stage by stage.
- The total number of stages in an n-input butterfly network is $\log n$ and the total number of crosspoints is $n \log n$, which is significantly fewer than n^2 in an n-input crossbar network, thus its circuit implementation should be more area-efficient.
- In each stage of a butterfly network, for every input, there is another input that shares the same two outputs with it. We call these pairs of inputs *conflict inputs*, their corresponding pairs of outputs *conflict outputs*. These *conflict pairs* can be configured using one bit.
- The butterfly network has a recursive structure. By removing the top stage of an n-input butterfly network, we get two identical butterfly networks each having $n/2$ inputs.

Proceedings of SPIE, *Media Processor 2000*, January 27-28, 2000, San Jose, California, pp. 80-86



(a)

```
Before optimization:

cross,2,1  r1,r1,r2  ;r2=10001010b
;  before, r1=hgfedcba
;  after,  r1=bgdehcfa
cross,0,0  r1,r1,r3  ;r3=10110000b
;  before, r1=bgdehcfa
;  after,  r1=gbdechaf
cross,1,2  r1,r1,r4  ;r4=01000000b
;  before, r1=gbdechaf
;  after,  r1=ghdecbaf


After optimization:

cross,2,1  r1,r1,r2  ;r2=10001010b
;  before, r1=hgfedcba
;  after,  r1=bgdehcfa
cross,0,2  r1,r1,r3  ;r3=01001011b
;  before, r1=bgdehcfa
;  after,  r1=ghdecbaf
```

(b)

Figure 1. (a) An n-input Benes network is formed by connecting two n-input butterfly networks back-to-back. It can be used to do arbitrary permutations of its n inputs. The example shows how to do the permutation $(hgfedcba) \rightarrow (ghdecbaf)$. (b) The instruction sequence for this permutation using our `cross` instructions is shown. An optimized instruction sequence is also shown.

2.2. Configuring a Benes Network Given a Permutation

Due to the recursive structure of a butterfly network, Benes network also has a recursive structure. Removing the first and last stages of an n-input Benes network, we get two identical $n/2$-input Benes networks. This procedure can be repeated until we have only two stages left. Therefore, if we can consistently direct the input bits and output bits of an n-input Benes network to the appropriate one of the two $n/2$-input sub-networks, we can reduce the size of the configuration problem by half in each step. The Benes network configuration problem can be solved by repeating this procedure until we reach $n = 2$, where the configuration is trivial. The key problem then becomes how to find a consistent mapping from inputs and outputs to the appropriate sub-networks, while maintaining the edge disjoint property mentioned above. We use the following algorithm to solve the problem.

We see that conflict inputs cannot be sent to the same sub-network. Similarly, conflict outputs cannot come from the same sub-network. These two constraints must be satisfied. Our algorithm works as follows. "Inputs" and "outputs" refer to the inputs and outputs of the sub-networks, $sub1$ and $sub2$.

1. Starting from the first input that is not configured, call it *current_input*, set *end_input* to be the conflict input of *current_input*. If all inputs have already been configured, go to Step 4.

2. a. Connect *current_input* to the sub-network $sub1$ that is on the same side as *current_input*. Connect the output that has the same value as *current_input*, call it *output*(*current_input*), to $sub1$ too. Set *current_output* to the conflict output of *output*(*current_input*) and go to Step 3.

   b. Connect *current_input* to the sub-network $sub1$ such that $sub1$ is not $sub2$. Connect *output*(*current_input*) to $sub1$ too. Set *current_output* to the conflict output of *output*(*current_input*).

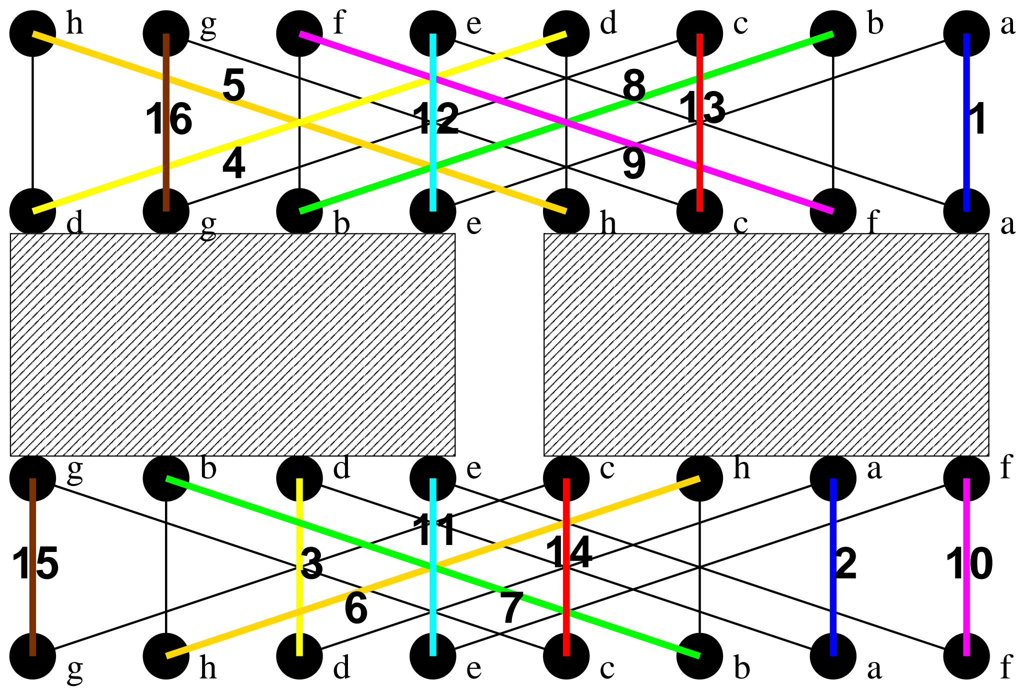


Figure 2. This example demonstrates how the algorithm is used to configure a Benes network. The permutation from Figure 1 is used and only the configuration for the first and last stage is shown. The number next to each thick line indicates the order in which these connections are configured.

3. Connect *current_output* to sub-network $sub2$ such that $sub2$ is not $sub1$. Also connect the input that has the same value as *current_output*, call it *input*(*current_output*), to $sub2$. If *input*(*current_output*) is the same as *end_input*, go back to Step 1. Otherwise set *current_input* to the conflict input of *input*(*current_output*) and go to Step 2b.

4. At this point, all the inputs and outputs have been connected to the two sub-networks. If the configuration of the two sub-networks is trivial, i.e., $n = 2$, the configuration is done. Otherwise for each sub-network, we treat it as a full Benes network and repeat the above procedure from Step 1.

To show that the above algorithm works correctly, we first notice that the algorithm always configures inputs and outputs in pairs, so no inputs or outputs would be missed. We also notice that we always go along the path input $\rightarrow$ output $\rightarrow$ output $\rightarrow$ input $\rightarrow$ ..., A problem that can occur at the output side is that the second output in the above chain is already configured. This situation cannot happen, however, because it means that an output has two different conflict outputs in one stage, which is impossible. The same argument applies to the input side. No other problem can arise in this algorithm, so it works. A detailed verification is beyond the scope of this paper. An example demonstrating the application of the algorithm is shown in Figure 2 instead. If the Benes network to be configured has n stages, we first configure the first stage(input) and the n-th stage(output), then the second stage(input) and the $(n-1)$-th stage(output), and so forth until all stages have been configured.

3. ARCHITECTURAL IMPLEMENTATION BASED ON BUTTERFLY NETWORK

3.1. Basic Operation

The basic operation for our proposed permutation methodology corresponds to the single stage operation in a butterfly network, which is to conditionally exchange pairs of conflict bits based on the given configuration. Because the distances between bits in a conflict pair are different in different stages, a parameter m is given to our basic operation to specify the distances between bits in a conflict pair. The distance specified by parameter m is 2^m. Since the decision to exchange a pair of conflict bits can be configured using one bit, the maximum number of

configuration bits for the basic operation is $n/2$ for permutations of n bits. A pair of conflict bits are exchanged when their configuration bit is 1.

3.2. Instruction

The instruction we propose takes a source register, permutes the subwords in it based on the contents of a configuration register, and puts the result into a destination register. From the above discussion, when we permute subwords in a register, the number of configuration bits for a basic operation can fill at most half of a register. Therefore, we can pack two basic operations into an instruction to make better use of the configuration register. The instruction format of our permutation instruction is

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

`rs` denotes the source register which contains the subwords to be permuted, `rd` denotes the destination register where the permuted subwords will be stored, and `rc` is the configuration register that holds the configuration bits. `m1` and `m2` are the parameters for the distances between a pair of bits for the two basic operations performed. The `cross` instruction performs two basic operations on the source register based on the contents of a configuration register and the values of `m1` and `m2`. For the first operation, `m1` is used to specify the distance for a pair of conflict bits so that the correct butterfly stage for this operation is chosen, and the right half of `rc` contains the configuration bits. This operation permutes the source register and produces a temporary result. The second operation is determined by `m2` and the left half of `rc`. It uses the temporary result of the first stage and produces the result for the destination register. For each operation, the bit pairs in a register are ordered starting from the right side of the register, and each pair of bits are exchanged if their corresponding configuration bit is 1 and not exchanged otherwise.

An example is shown in Figure 1(b). In the unoptimized instruction sequence, the first instruction, `cross,2,1 r1,r1,r2` performs two basic operations. The first basic operation uses the butterfly stage where the distance between a pair of conflict bits is $2^2 = 4$. The configuration bits come from the right side of `r2`, and are `1010b`. Starting from the right side of `r1`, `a` and `e` are not exchanged, `b` and `f` are exchanged, `c` and `g` are not exchanged, `d` and `h` are exchanged. The intermediate result after the first basic operation is `dgbehcfa`. The second basic operation uses the butterfly stage where the distance between a pair of conflict bits is $2^1 = 2$. The configuration bits come from the left side of `r2`, and are `1000b`. Only `d` and `b` are exchanged and the other pairs of conflict bits are not. Therefore, the result in `r1` after the first instruction is `bgdehcfa`. The second and the third `cross` instructions work the same way. To derive the optimized instruction sequence from the unoptimized sequence, we only need to notice that for the second and the third instructions in the unoptimized sequence, the configuration bits for one of the two basic operations in each instruction are all `0`. Since those two basic operations only serve to copy the inputs to outputs, they can be eliminated.

3.3. Using `cross` Instruction

We know that all permutations of an n-bit word, including subword permutations for all subword sizes, can be achieved using an n-input Benes network, which contains two n-input butterfly networks and $2\log n$ stages. Therefore, for each permutation, we can use the algorithm described in Section 2.2 to obtain the configuration of a Benes network for the desired permutation. Then we can break the network into pairs of stages, and for each pair of stages, we assign a `cross` instruction. Each `cross` instruction uses the result of the previous `cross` instruction. The first `cross` instruction in the sequence takes the original source and the last `cross` instruction produces the final result. In this way, we build a virtual n-input Benes network using our `cross` instruction. Therefore, we are able to perform any permutations for an n-bit word with at most $\log n$ instructions.

3.4. Permutation for Subword Sizes Greater than a Bit

One important property of a Benes network is that when it is configured for r-bit subword permutations, the middle $2\log r$ stages are configured as pass-throughs if we constrain our configurations to have the property that a pair of adjacent inputs to a stage in the same subword stay adjacent and in the same order during permutation. This happens because the middle $2\log r$ stages of a Benes network are used to permute adjacent r bits. One example is shown in Figure 3. For bypassing connections, we do not need to assign instructions, because all these stages do is to copy inputs to outputs. Therefore, when permuting r-bit subwords in an n-bit word, the maximum number of instructions needed is $\log n - \log r = \log(n/r) = \log n'$, where n' is the number of subwords.

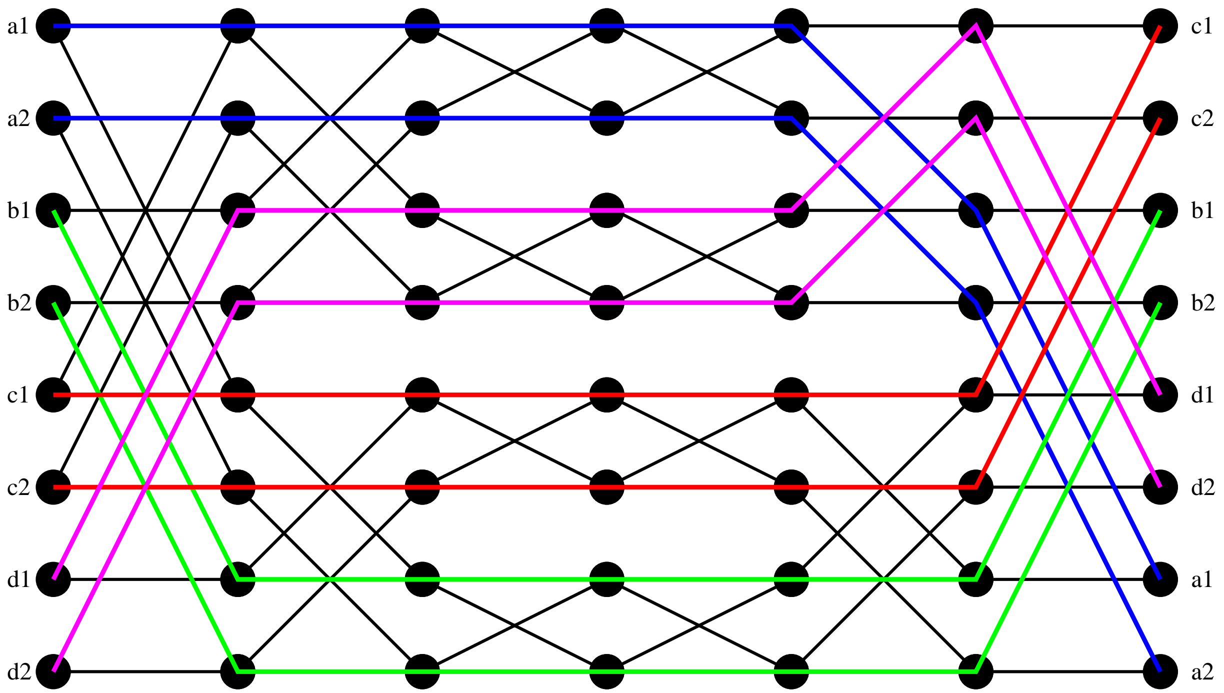


Figure 3. This example demonstrates that when a Benes network is used to perform an r-bit subword permutation, the middle $2 \log r$ stages become pass-through connections. Permutation $(d_2 d_1 c_2 c_1 b_2 b_1 a_2 a_1) \rightarrow (a_2 a_1 d_2 d_1 b_2 b_1 c_2 c_1)$ is used.

3.5. Basic Optimization

For many permutations, we will have some stages of the Benes network configured as pass-throughs even if they are not subword permutations. Because these bypassing connections only serve to copy the inputs to the outputs, we can remove these stages before we do the instruction assignment. If we can remove $2k$ stages, we will have k fewer instructions. An example is shown in Figure 1(b).

4. LOW LEVEL IMPLEMENTATION

In order to implement the `cross` instruction in hardware, we need to implement a Benes network at the circuit level. An n-input Benes network has $2n \log n$ switch points, which is far less than the n^2 switch points for an n-input crossbar network. It is both easier to control and more area-efficient than its crossbar counterpart. When executing a `cross,m1,m2 rd,rs,rc` instruction, the control logic selects the two stages for the two basic operations based on the value of `m1` and `m2`. Because a Benes network has two of each butterfly stage, stages can always be selected for all possible `m1` and `m2`. We then use the right and left half of `rc` to configure the two stages selected and configure all the other stages as pass-throughs. Next `rs` is put through the configured network, and we get the result `rd`.

5. PERFORMANCE COMPARISON WITH CURRENT METHODS

The number of instructions needed for permutations of a 64-bit word with different subword sizes are measured for our method and the current methods to compare their performance.

5.1. Permutations of 1-bit, 2-bit and 4-bit Subwords

Our method can do arbitrary bit permutations of a 64-bit word with a maximum of $\log 64 = 6$ `cross` instructions. For 2-bit subwords, at most $\log(64/2) = 5$ instructions are needed and for 4-bit subwords, at most $\log(64/4) = 4$ instructions are needed. The current implemented subword permutation instructions like `mux`, `permute` and `vperm` apply to 8-bit and larger subwords. Hence, they are not able to perform permutations on smaller subwords efficiently, even with the use of conventional `shift` and `rotate` instructions.

5.2. Permutations of 8-bit, 16-bit and 32-bit Subwords

Our method can do arbitrary 8-bit subword permutations of a 64-bit word with at most $\log(64/8) = 3$ cross instructions. The average number of instructions needed is 2.37, which is obtained by applying the basic optimization mentioned in Section 3.5 and averaging over all 8-bit subword permutations. vperm is able to perform these permutations using one instruction. mux can do five common fixed permutations for 8-bit subwords using one instruction, but needs more than one instruction for arbitrary permutations. However, cross and vperm require an instruction to load the configuration into a register, but mux does not.

Our method can do arbitrary 16-bit subword permutations of a 64-bit word with at most $\log(64/16) = 2$ cross instructions. The average number of instructions needed is 1.21. Both permute, mux and vperm are able to do these permutations in one instruction.

Permuting 32-bit subwords is the trivial case of subword permutation that all methods can handle in one instruction.

Our use of the cross instruction provides a fast and scalable solution for the subword permutation problem, while all of the current methods can only deal with a subset of the problem. We notice that for large subwords, our new method does not perform as well as the current methods. However, this can be improved by using the optimization that will be mentioned in Section 6.

6. FUTURE RESEARCH

There are several directions for future research. First, our solution only addresses the problem of subword permutation without repetitions. We are looking to extend this method to handle permutations with repetitions. Second, our solution requires a full Benes network at the circuit level. We are considering using other more area-efficient interconnection networks in a separate paper. For permutations of large subwords, we can compress the configuration bits and put them into one register because the configuration bits are the same for bits in the same subword. With this in mind, we can perform these permutations using fewer instructions, but with the expense of additional variants of the cross instructions. We also did not consider the instructions needed to load the configuration bits into registers, based on the assumption that multiple executions of the same permutation would amortize the cost.

7. CONCLUSIONS

The proposed subword permutation instructions based on butterfly networks offer a satisfactory solution to the general subword rearrangement problem that has not been solved. It provides a generalized and systematic method for performing arbitrary subword permutations(without repetition) for all subword sizes. The maximum number of instructions used in our method for permuting n subwords is $\log n$, or $2\log n$ if we consider loads for the configuration registers. For existing solutions, although the minimum number of instructions needed for some permutations is 1, the maximum number for an arbitrary permutation is not bounded by $\log n$. Furthermore, these instructions cannot be used to perform arbitrary permutations easily. Finally, our cross permutation instructions result in a more area-efficient implementation than instructions requiring a full $n \times n$ crossbar implementation.

REFERENCES

1. R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro* **15**, pp. 22–32, April 1995.
2. R. B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro* **16**, pp. 51–59, August 1996.
3. M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, "VIS Speeds New Media Processing," *IEEE Micro* **16**, pp. 10–20, August 1996.
4. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro* **16**, pp. 10–20, August 1996.
5. S. Oberman, F. Weber, N. Juffa, and G. Favor, "AMD 3Dnow! Technology and the K6-2 Microprocessor." Presentation. From California Microprocessor Division, Advanced Micro Devices.
6. ""AltiVec extension to PowerPC" Instruction Set Architecture Specification," May 1998. http://www.motorola.com/AltiVec.
7. "IA-64 Application Developer's Architecture Guide," May 1999. http://developer.intel.com/design/ia64.
8. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures Arrays, Trees, Hypercubes*, Morgan-Kaufmann Publishers, Inc., San Mateo, Calfornia, 1992.