J Sign Process Syst (2008) 53:145–169
DOI 10.1007/s11265-008-0212-8

Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors

Yedidya Hilewitz · Ruby B. Lee

Received: 10 January 2007 / Revised: 25 February 2008 / Accepted: 12 April 2008 / Published online: 4 June 2008

Abstract Advanced bit manipulation operations are not efficiently supported by commodity word-oriented microprocessors. Programming tricks are typically devised to shorten the long sequence of instructions needed to emulate these complicated bit operations. As these bit manipulation operations are relevant to applications that are becoming increasingly important, we propose direct support for them in microprocessors. In particular, we propose fast bit gather (or parallel extract), bit scatter (or parallel deposit) and bit permutation instructions (including group, butterfly and inverse butterfly). We show that all these instructions can be implemented efficiently using both the fast butterfly and inverse butterfly network datapaths. Specifically, we show that parallel deposit can be mapped onto a butterfly circuit and parallel extract can be mapped onto an inverse butterfly circuit. We define static, dynamic and loop invariant versions of the instructions, with static versions utilizing a much simpler functional unit. We show how a hardware decoder can be implemented for the dynamic and loop-invariant versions to generate, dynamically, the control signals for the butterfly and inverse butterfly datapaths. The simplest functional unit we propose is smaller and faster than an ALU. We also show that these instructions yield significant speedups over a basic RISC architecture for a variety of different application kernels taken from applications domains including bioinformatics, steganography, coding, compression and random number generation.

Y. Hilewitz · R. B. Lee (✉)
Princeton Architecture Laboratory for Multimedia and Security (PALMS), Department of Electrical Engineering, Princeton University,
Princeton, NJ 08544, USA
e-mail: rblee@princeton.edu

Y. Hilewitz
e-mail: hilewitz@princeton.edu

Keywords Bit manipulations · Permutations · Bit scatter · Bit gather · Parallel extract · Parallel deposit · Pack · Unpack · Microprocessors · Instruction set architecture · ISA · Algorithm acceleration · Bioinformatics · Pattern matching · Compression · Steganography · Cryptology

1 Introduction

Bit manipulation operations are typically not well supported by microprocessors. However, these types of bit operations are relevant for applications that are increasing in importance. Previously, accelerating bit manipulations has mostly been associated with developing clever programming tricks that use existing microprocessor features in non-intuitive ways to speedup bit-string processing. Many of these tricks have been collected and published, e.g., in *Hacker's Delight* [1]. In this paper, we show how a few advanced bit manipulation *instructions* may be defined and efficiently implemented in a commodity microprocessor, to accelerate numerous applications.

For example, pattern matching and searching play a central role in data mining applications. The data in question may be genetic patterns in bioinformatics, iris or fingerprint information in biometrics, keywords in communication surveillance, and so on. The data and search terms may be represented by vectors of bit-strings. We do a comparison to see if some set of properties or features are present in a database record and then gather the result bits from the comparison for further processing. We call this a *bit gather* instruction.

Consider the bioinformatics alignment program BLASTZ [2]. An alignment program takes two DNA strings and tries to align them on the best match (where the best match allows substitution, insertions and deletions). These programs typically compare a seed, a short substring of DNA data,

and if the match is good, the seed is extended. The BLASTZ program allows a seed in which certain substring positions are specified and others are left as wildcards. The program selects and compresses the data in the specified substring positions and uses the result as an index into a hash-table to find where in the second string this seed is found. These steps involve several bit manipulation operations on bit-strings.

Rather than allowing the acceleration of bit manipulations to be relegated to esoteric "programming tricks" [1, 3], we want to accelerate bit manipulations by directly building support for these operations into commodity microprocessors. Supercomputers often have direct support for advanced bit operations (see, for example, the Cray bit matrix multiply instruction [4]). We will show that we can add a low cost functional unit to a commodity microprocessor that supports a useful set of bit operations that accelerate many different applications.

In particular, the operations that we propose to support are:

- the *bit gather* operation, which we also call *parallel extract* [5, 6];
- the *bit scatter* operation, which we also call *parallel deposit* [6]; and
- bit permutations, which can arbitrarily rearrange the bits in a processor word. Specifically, we focus on the *butterfly* and *inverse butterfly* permutation instructions [7–9] and the *group* permutation instruction [10, 11].

We give the instruction set architecture (ISA) definitions for these instructions, consider the usage patterns of these operations, and show how to design an efficient functional unit that supports them using a few simple circuit building blocks. We show that the bit gather operation can be mapped to the inverse butterfly network and that the bit scatter operation can be mapped to the butterfly network. The simplest functional unit we propose is smaller and faster than an arithmetic logic unit (ALU).

We also show that these instructions improve the performance of a number of applications including bioinformatics, image processing, steganography, compression and coding. Our performance results indicate that a processor enhanced with parallel deposit and parallel extract instructions achieves a 10.04× maximum speedup, 2.29× on average, over a basic RISC architecture.

The paper is organized as follows: Section 2 describes our advanced bit manipulation operations. Section 3 discusses the datapaths required to perform the operations and culminates with an overview of the advanced bit manipulation functional unit. Section 4 summarizes the ISA definition of the new instructions. Section 5 details the applications that benefit from bit manipulation instructions and Section 6 summarizes benchmark results. Section 7 provides the detailed implementation of the functional unit and gives synthesis results. Section 8 gives related work. Section 9 concludes the paper.

2 Advanced Bit Manipulation Operations

Simple bit manipulation operations found in microprocessors include and, or, xor and not. These are bit-parallel operations that are easily accomplished in word-oriented processors. Other common bit manipulation instructions found in microprocessors that are not bit-parallel are shift and rotate instructions. In these instructions, all the bits in a word move by the same amount. In this section, we introduce more advanced bit manipulation instructions that are not currently found in commodity microprocessors. Each of these operations can take tens to hundreds of cycles to emulate in current microprocessors. However, by defining new instructions for them, each can be implemented in one, or a few, cycles.

2.1 Bit Gather or Parallel Extract: pex

A bit gather instruction collects bits dispersed throughout a register, and places them contiguously in the result. Such selection of non-contiguous bits from data is often necessary. For example, in pattern matching, many pairs of features may be compared. Then, a subset of these comparison result bits are selected, compressed and used as an index to look up a table. This selection and compression of bits is a bit gather instruction.

A bit gather instruction can also be thought of as a parallel extract, or pex instruction [5, 6]. This is so named because it is like a parallel version of the extract (extr) instruction [12, 13]. Figure 1 compares extr and pex. The extr instruction extracts a single field of bits from any position in the source register and right justifies it in the destination register. The pex instruction extracts multiple bit fields from the source register, compresses and right justifies them in the destination register. The selected bits (in r_2) are specified by a bit mask (in r_3) and placed contiguously and right-aligned in the result register (r_1).

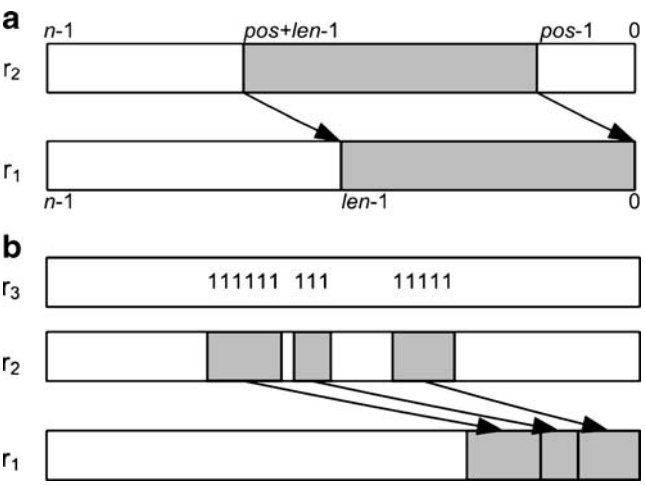

Figure 1 **a** extr r_1=r_2, pos, len **b** pex r_1=r_2, r_3.

2.2 Bit Scatter or Parallel Deposit: pdep

Bit scatter takes the right-aligned, contiguous bits in a register and scatters them in the result register according to a mask in a second input register. This is the reverse operation to bit gather. We also call bit scatter a parallel deposit instruction, or pdep, because it is like a parallel version of the deposit (dep) instruction found in processors like PA-RISC [12, 13] and IA-64 [14]. Figure 2 compares dep and pdep. The deposit (dep) instruction takes a right justified field of bits from the source register and deposits it at any single position in the destination register. The parallel deposit (pdep) instruction takes a right justified field of bits from the source register and deposits the bits in different non-contiguous positions indicated by a bit mask.

2.3 Bit Permutation Primitive: grp

In [10, 11] Shi and Lee defined a bit permutation instruction, grp, and showed that arbitrary bit permutations can be accomplished by a sequence of at most lg(n) of these grp instructions, where n is the word-size of the processor. grp is a permutation primitive that gathers to the right the data bits selected by "1"s in the mask, and to the left those selected by "0"s in the mask (see Fig. 3). This can be considered two parallel operations: grp_right (grpr) and grp_left (grpl) [15]. The pex instruction is the grp_right half of a grp operation. It conserves some of the most useful properties of grp, while being easier to implement—the bits that would have been gathered to the left are instead zeroed out.

2.4 Butterfly and Inverse Butterfly Operations: bfly and ibfly

While the grp operation can be used to achieve any arbitrary permutation of n bits in lg(n) instructions, Lee et al. [7–9] also found bit permutation primitives that can do this in only two instructions. These are the butterfly (bfly) and inverse butterfly (ibfly) instructions which permute their inputs using the butterfly and inverse butterfly networks, respectively (Fig. 4). The concatenation of the butterfly and inverse butterfly networks forms a Beneš network, a general permutation network [16]. Consequently, only a single execution of bfly followed by ibfly is required to compute any of the n! permutations of n bits.

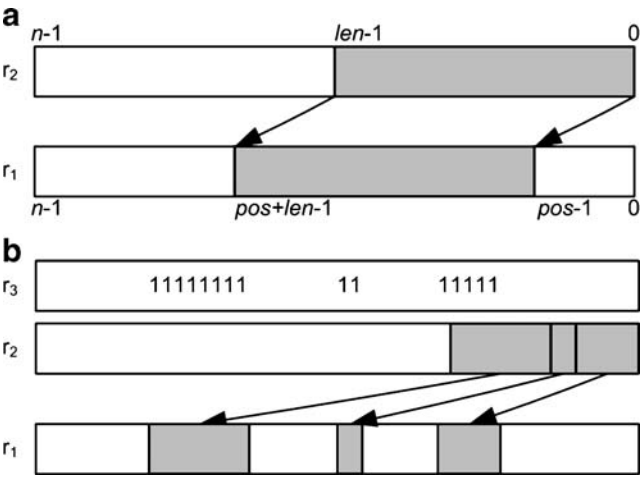

Figure 2 **a** dep r_1=r_2, pos, len **b** pdep r_1=r_2, r_3.

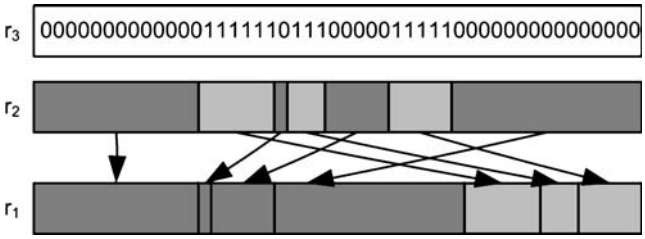

Figure 3 grp r_1=r_2, r_3.

The structure of the networks is shown in Fig. 4, where n=8. The n-bit networks consist of lg(n) stages. Each stage is composed of n/2 two-input switches, each of which is constructed using two 2:1 multiplexers. These networks are faster than an ALU of the same width, since an ALU also has lg(n) stages which are more complicated than those of the butterfly and inverse butterfly circuits. Assuming a processor cycle to be long enough to cover the ALU latency, each bfly and ibfly operation will have single cycle latency, since these circuits are simpler than an ALU's circuit. Furthermore, as each network has only $n\times$lg(n) multiplexers, the overall circuit area is small. (See Section 7 for circuit evaluation details.)

In the ith stage (i starting from 1), the input bits are $n/2^i$ positions apart for the butterfly network and 2^{i-1} positions apart for the inverse network. A switch either passes through or swaps its inputs based on the value of a configuration bit. Thus, the bfly, or ibfly, operation requires $n/2\times$lg(n) configuration bits. For n=64, three 64-bit registers are needed to hold the configuration bits in addition to the one register for the input. So, while a bfly followed by an ibfly instruction can accomplish arbitrary n-bit permutations in at most two instructions as opposed to

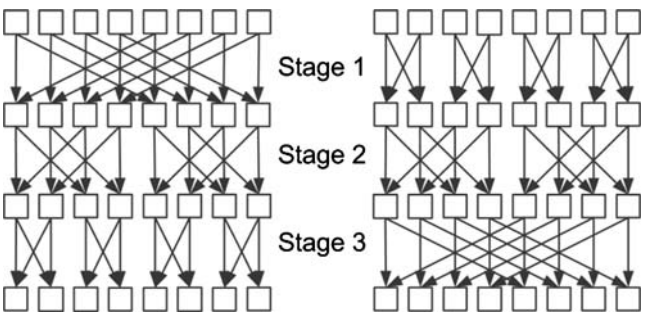

Figure 4 **a** Eight-bit butterfly network and **b** 8-bit inverse butterfly network.

needing up to lg(n) grp instructions, grp only requires two n-bit operands per instruction while each of the bfly or ibfly instructions requires (1+lg(n)/2) n-bit operands per instruction. This presents a challenge, as typical ISAs and processor datapaths support only two operands per instruction. We discuss this in more detail in Section 3.4.

3 Datapaths for pex and pdep

Our prior work on accelerating the grp permutation instruction shows that it can also be implemented by two inverse butterfly networks operating in parallel, one implementing grpr and one implementing grpl [15]. Since the pex operation is like the grpr operation, it can be implemented by one inverse butterfly network. Since pdep is the inverse of pex, we will show that it can be implemented by the butterfly circuit, which reverses the stages of the inverse butterfly circuit.

It is not immediately clear that parallel deposit can be mapped to the butterfly network and that parallel extract can be mapped to the inverse butterfly network. We will show that these mappings are indeed correct and also that the reverse is not true—parallel deposit cannot be mapped to the inverse butterfly network and parallel extract cannot be mapped to the butterfly network. Thus in order to design a single functional unit that supports both pex and pdep, both butterfly and inverse butterfly datapaths are required.

3.1 Parallel Deposit on the Butterfly Network

We first show an example parallel deposit operation on the butterfly network. Then, we show that any parallel deposit operation can be performed using a butterfly network.

Figure 5 shows our labeling of the left (L) and right (R) halves of successive stages of a butterfly network. Since we often appeal to induction for successive stages, we usually omit the subscripts for these left and right halves.

Figure 6a shows an example pdep operation with mask= 10101101. Figure 6b shows this pdep operation broken down into steps on the butterfly network. In the first stage, we transfer from the right (R) to the left half (L) the bits whose destination is in L, namely bits d and e. Prior to stage 2, we right rotate e00d by 3, the number of bits that stayed in R, to right justify the bits in their original order, 00de, in the L half. Note that bits that stay in the right half, R, are already right-justified.

In each half of stage 2 we transfer from the local R to the local L the bits whose final destination is in the local L. So in R, we transfer g to R_L and in L we transfer d to L_L. Prior to stage 3, we right rotate the bits to right justify them in their original order in their new subnetworks. So d0 is right rotated by 1, the number of bits that stayed in L_R, to yield

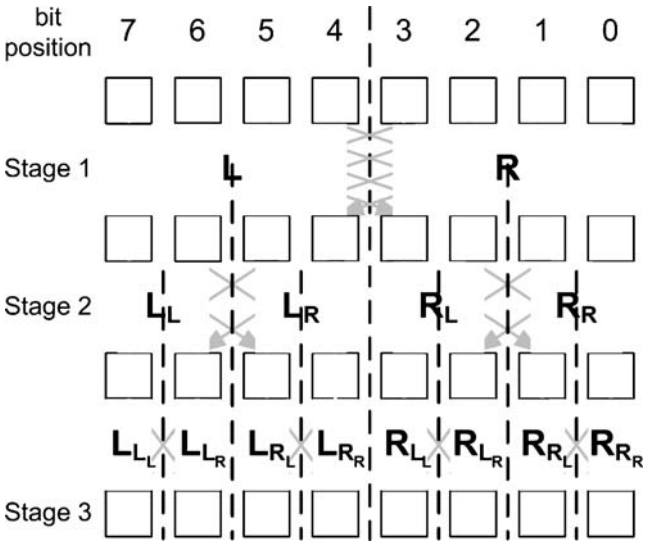

Figure 5 Labeling of butterfly network.

0d, and gf is right rotated by 1, the number of bits that stayed in R_R, to yield fg.

In each subnetwork of stage 3 we again transfer from the local R to the local L the bits whose final destination is in the local L. So in L_L we transfer d and in L_R we transfer e. After stage 3 we have transferred each bit to its correct final destination: d0e0fg0h. Note that we use a control bit of "0" to indicate a swap, and a control bit of "1" to indicate a pass through operation.

Rather than explicitly right rotating the data bits in the L half after each stage, we can compensate by modifying the control bits. This is shown in Fig. 6c. How the control bits are derived will be explained later in Section 7.1.

We now show that any pdep operation can be mapped to the butterfly network.

Fact 1 Any single data bit can be moved to any result position by moving it to the correct half of the intermediate result at every stage of the butterfly network.

This can be proved by induction on the number of stages. At stage 1, the data bit is moved within $n/2$ positions of its final position. At stage 2, it is moved within $n/4$ positions of its final result, and so on. At stage lg(n), it is moved within $n/2^{\lg(n)}=1$ position of its final result, which is its final result position. Referring back to Fig. 6b, we utilized Fact 1 to decide which bits to keep in R and which to transfer from R to L at each stage.

Fact 2 If the mask has k "1"s in it, the k rightmost data bits are selected and moved, i.e., the selected data bits are contiguous. They never cross each other in the final result.

This fact is by definition of the pdep instruction. See the example of Fig. 6a where there are five "1"s in the mask and the selected data bits are the five rightmost bits, defgh;

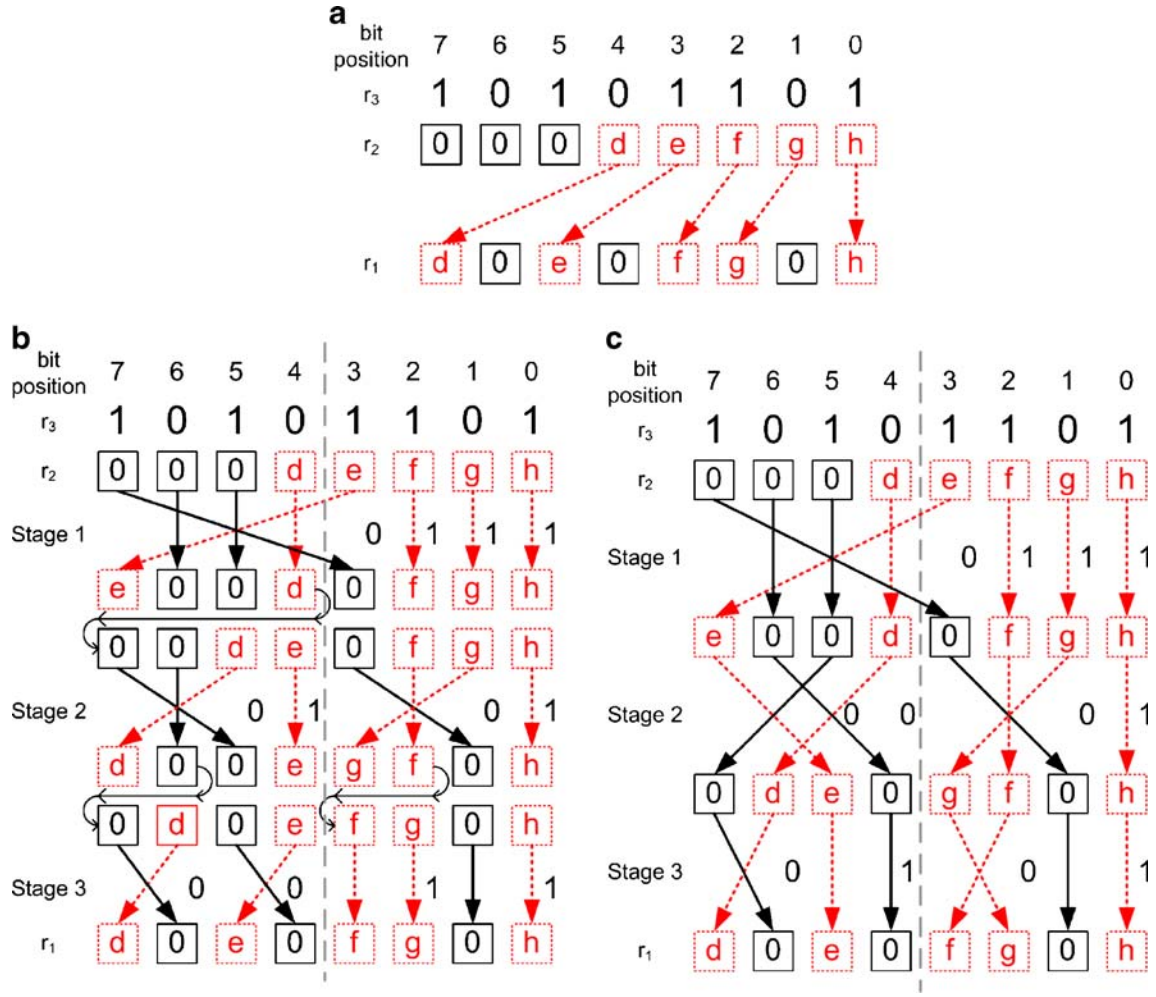

Figure 6 **a** 8-bit pdep operation **b** mapped onto butterfly network with explicit right rotations of data bits between stages and **c** without explicit rotations of data bits by modifying the control bits.

these bits are spread out to the left maintaining their original order, and thus never crossing each other in the result.

Fact 3 If a data bit in the right half (R) is swapped with its paired bit in the left half (L), then all selected data bits to the left of it will also be swapped to L (if they are in R) or stay in L (if they are in L).

Since the selected data bits never cross each other in the final result (Fact 2), once a bit swaps to L, the selected bits to the left of it must also go to L. Hence, if there is one "1" in the mask, the one selected data bit, d_0, can go to R or L. If there are two "1"s in the mask, the two selected data bits, d_1d_0, can go to RR or LR or LL. That is, if the data bit on the right stays in R, then the next data bit can go to R or L, but if the data bit on the right goes to L, the next data bit must also go to L. If there are three "1"s, the three selected data bits, $d_2d_1d_0$, can go to RRR, LRR, LLR or LLL. For example, in Fig. 6b stage 1, the 5 bits have the pattern LLRRR as e is transferred to L and d must then stay in L.

Fact 4 The selected data bits that have been swapped from R to L, or stayed in L, are all contiguous mod $n/2$ in L.

From Fact 3, the destinations of the k selected data bits $d_{k-1}\ldots d_0$ must be of the form L…LR…R, a string of zero or more L's followed by zero or more R's (see Fig. 7). Define X as the bits staying in R, Y as the bits going to L that start in R and Z as the bits going to L that start in L. It is possible that:

1. X alone exists—when there are no selected data bits that go to L,
2. Y alone exists—when all bits that start in R go to L and there are no selected data bits that start in L and,
3. X and Y exist—when some bits that start in R stay in R and some go to L and there are no selected data bits that start in L and,
4. X and Z exist—when all the bits in R are going to R, and all bits going to L start in L, or
5. X, Y and Z exist.

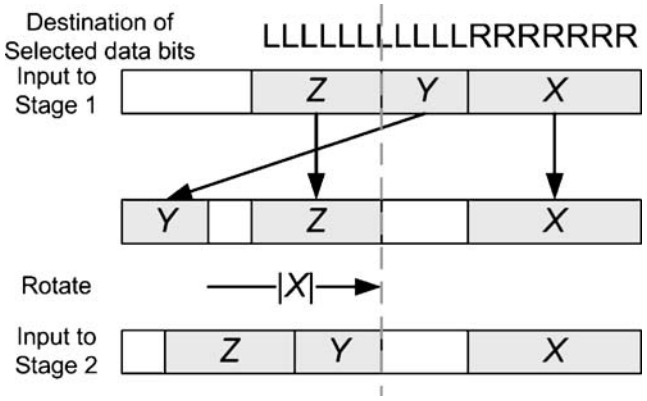

Figure 7 At the output of a butterfly stage, Y and Z are contiguous mod $n/2$ in L and can be rotated to be the rightmost bits of L.

When X alone exists (1), there are no bits that go to L, so Fact 4 is irrelevant.

The structure of the butterfly network requires that when bits are moved in a stage, they all move by the same amount. Fact 2 states that the selected data bits are contiguous. Together these imply that when Y alone exists or X and Y exist (2 and 3), Y is moved as a contiguous block from R to L and Fact 4 is trivially true.

When X and Z exist (4), Z is a contiguous block of bits that does not move so again Fact 4 is trivially true.

When X, Y and Z exist (5), Y comprises the leftmost bits of R, and Z the rightmost bits in L since they are contiguous across the midpoint of the stage (Fact 2). When Y is swapped to L, since the butterfly network moves the bits by an amount equal to the size of L or R in a given stage, Y becomes the leftmost bits of L. Thus Y and Z are now contiguous mod $n/2$, i.e., wrapped around, in L (Fig. 7).

Thus Fact 4 is true in all cases.

For example, in Fig. 6b at the input to stage 1, X is bits fgh, Y is bit e and Z is bit d. Y is the leftmost bit in R and Z is the rightmost bit in L. After stage 1, Y is the leftmost bit in L and is contiguous with Z mod 4, within L, i.e., de is contiguous mod 4 in e00d.

Fact 5 The selected data bits in L can be rotated so that they are the rightmost bits of L, and in their original order.

From Fact 4, the selected data bits are contiguous mod $n/2$ in L. At the output of stage 1 in Fig. 7, these bits are offset to the left by the size of X (the number of bits that stayed in R), denoted by $|X|$. Thus if we explicitly rotate right the bits by $|X|$, the selected data bits in L are now the rightmost bits of L in their original order (Fig. 7). In Fig. 6b, Fact 5 was utilized prior to stages 2 and 3.

At the end of this step, we have two half-sized butterfly networks, L and R, with the selected data bits right-aligned and in order in each of L and R (last row of Fig. 7). The above can now be repeated recursively for the half-sized butterfly networks, L and R, until each L and R is a single bit. This is achieved after lg(n) stages of the butterfly network. (See the final output in Fig. 6b.)

The selected data bits emerge from stage 1 in Fig. 7 rotated to the left by $|X|$. In Fact 5, the selected data bits are explicitly rotated back to the right by $|X|$. Instead we can compensate for the rotation by modifying the control bits of the subsequent stages to limit the rotation within each subnetwork. For example, if the n-bit input to stage 1 is rotated by k positions, the two $n/2$-bit inputs to the L and R subnetworks are rotated by k (mod $n/2$) within each subnetwork. At the output of stage lg(n), the subnetworks are 1-bit wide so the rotations are absorbed.

Fact 6 If the data bits are rotated by x positions left (or right) at the input to a stage of a butterfly network, then at the output of that stage we can obtain a rotation left (or right) by x positions of each half of the output bits by rotating left (or right) the control bits by x positions and complementing upon wrap around.

Consider again the example of Fig. 6b. The selected data bits emerge from stage 1 left rotated by 3 bits, i.e., L is e00d, left rotated by three positions from 00de. In Fig. 6b, we explicitly rotated the data bits back to the right by 3. Instead, we can compensate for this left rotation by left rotating and complementing upon wrap around by three positions the control bits of the subsequent stages. This is shown in Fig. 8. For stage 2, the control bit pattern of L,

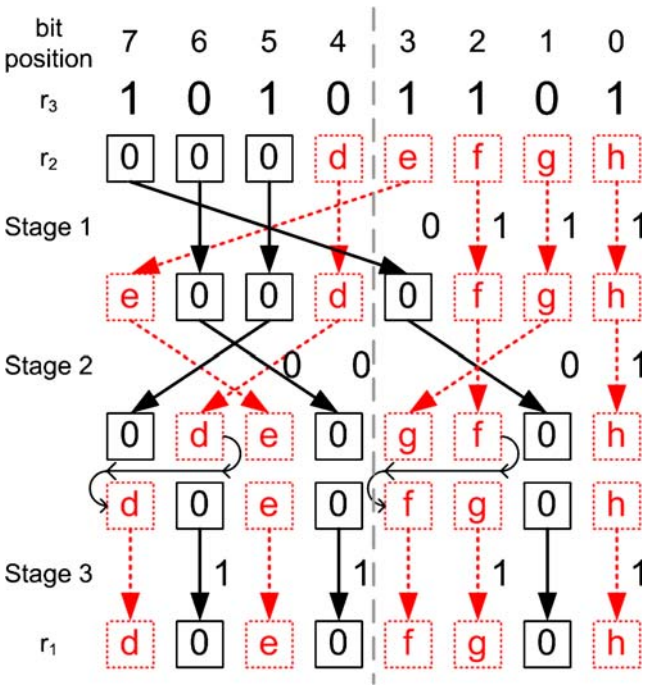

Figure 8 The elimination of explicit right rotations after stage 1 in Fig. 6b, prior to stage 2.

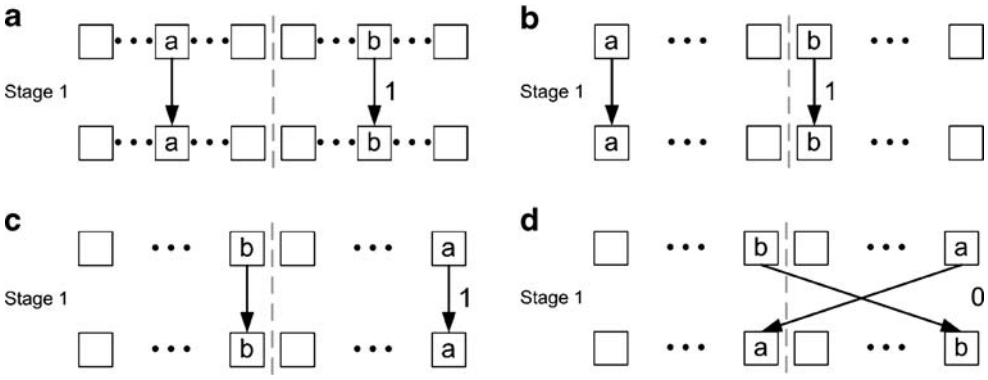

Figure 9 **a** A pair of data bits initially passed through; **b** rotation of the paired data bits and control bit; **c** wrapping of the data bits and control bit; **d** complementation of the control bit.

after left rotate and complement by 3, becomes 01 → 11 → 10 → 00. The rotation by 3 is limited to a rotation by 3 mod 2=1 within each half of the output of L of stage 2 as the output is transformed from d0,0e in Fig. 6b to 0d,e0 in Fig. 8. For stage 3, the rotation and complement by three of the two single control bits in L become three successive complements: 0 → 1 → 0 → 1, and the left rotation of L is absorbed as the overall output is still d0e0. Hence, Fig. 8 shows how the control bits in stages 2 and 3 compensate for the left rotate by 3 bits at the output of stage 1 (cf. Fig. 6b.)

Figure 6c shows the control bits after also compensating for the left rotate by 1 bit of R_L and L_L, prior to stage 3 in Fig. 8. The explicit right rotation prior to stage 3 is eliminated. Instead, the two control bits in R_L and L_L transform from 1 → 0 to absorb the rotation. By doing the same control bit transformations for stage 2, the overall result in Fig. 6c remains the same as in Fig. 6b. The explicit data rotations in Fig. 6b are replaced with rotations of the control bits instead, complementing them on wraparound.

We now explain why the control bits are complemented when they wrap around. The goal is to keep the data bits in the half they were originally routed to at each stage of the butterfly network, in spite of the rotation of the input. Figure 9a shows a pair of bits, a and b, that were originally passed through. So we wish to route a to L and b to R in spite of any rotation. As the bits are rotated (Fig. 9b), the control bit is rotated with them, keeping a in L and b in R, as desired. When the bits wrap around, (Fig. 9c), a wraps to R and b crosses the midpoint to L. If the control bit is simply rotated with the paired bits, then a is now passed through to R and b is passed through to L, which is contrary to the originally desired behavior. If instead the control bit is complemented when it wraps around (Fig. 9d), then a is swapped back to L and b is swapped back to R, as is desired.

Similarly, if a and b were originally swapped (Fig. 10a), a should be routed to R and b to L. As the bits rotate (Fig. 10b), we simply rotate the control bit with them. When the bits wrap around (Fig. 10c), input a wraps to R and b crosses to L. When they are swapped, a is routed to L and b to R, contrary to their original destinations. If the control bit is complemented on wraparound, a is passed through to R and b is passed through to L, conforming to the originally desired behavior.

Thus complementing the control bit when it wraps causes each of the pair of bits to stay in the half it was originally routed to despite the rotation of the input pushing each bit to the other half. This limits the rotation of the input to be within each half of the output and not across the entire output.

We now give a theorem to formalize the overall result:

Theorem 1 *Any parallel deposit instruction on n bits can be implemented with one pass through a butterfly network*

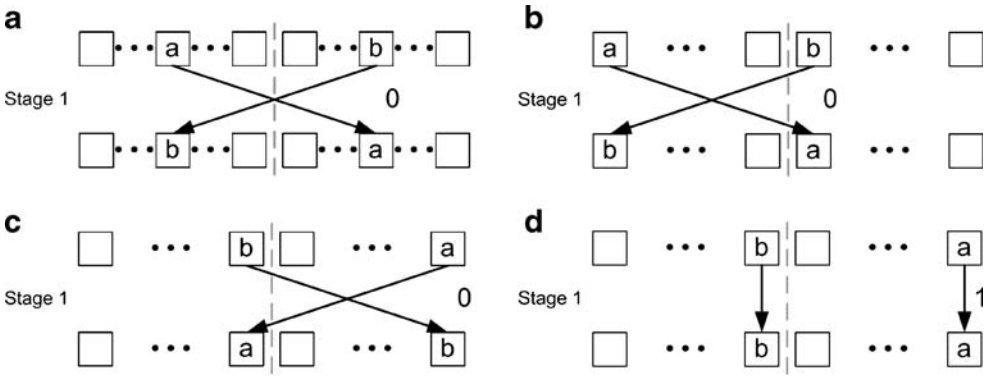

Figure 10 **a** A pair of data bits initially swapped; **b** rotation of the paired data bits and control bit; **c** wrapping of the data bits and control bit; **d** complementation of the control bit.

of lg(n) stages without path conflicts (with the bits that are not selected zeroed out externally).

Proof We give a proof by construction. Assume there are k "1"s in the right half of the bit mask. Then, based on Fact 1, the k rightmost data bits (block X) must be kept in the right half (R) of the butterfly network and the remaining contiguous selected data bits must be swapped (block Y) or passed through (block Z) to the left half (L). This can be accomplished in stage 1 of the butterfly network by setting the k rightmost configuration bits to "1" (to pass through X), and the remaining configuration bits to "0" (to swap Y).

At this point, the selected data bits in the right subnetwork (R) are right-aligned but those in the left subnetwork (L) are contiguous mod $n/2$, but not right aligned (Fact 4, Fig. 7); they are rotated left by the size of block X or the number of bits kept in R. We can compensate for the left rotation of the bits in L and determine the control bits for subsequent stages as if the bits in L were right aligned. This is accomplished by left rotating and complementing upon wraparound the control bits in the subsequent stages of L by the number of bits kept in R (once these control bits are determined pretending that the data bits in L are right aligned). Modifying the control bits in this manner will limit the rotation to be within each half of the output until the rotation is absorbed after the final stage (Fact 6).

Now the process above can be repeated on the left and right subnets, which are themselves butterfly networks: count the number of "1"s in the local right half of the mask and then keep that many bits in the right half of the subnetwork, and swap the remaining selected data bits to the left half. Account for the rotation of the left half by modifying subsequent control bits.

This can be repeated for each subnetwork in each subsequent stage until the final stage is reached, where the final parallel deposit result will have been achieved (e.g., Fig. 6c).

3.2 Parallel Extract on the Inverse Butterfly Network

We will now show that pex can be mapped onto the inverse butterfly network. The inverse butterfly network is decomposed into even and odd subnetworks, in contrast to the butterfly network which is decomposed into right and left subnetworks. See Fig. 11, where the even subnetworks are shown with dotted lines and the odd with solid lines. However, for simplicity of notation we refer to even as R and odd as L.

Fact IB1 Any single data bit can be moved to any result position by just moving it to the correct R or L subnetwork

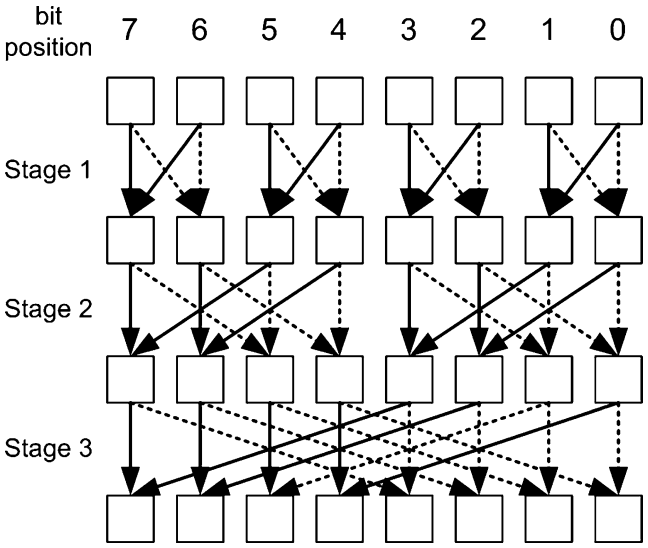

Figure 11 Even (or R, *dotted*) and odd (or L, *solid*) subnetworks of the inverse butterfly network.

of the intermediate result at every stage of the inverse butterfly network.

This can be proved by induction on the number of stages. At stage 1, the data bit is moved to its final position mod 2 (i.e., to R or L). At stage 2, it is moved to its final position mod 4 (i.e., to R_R, R_L, L_R or L_L), and so on. At stage lg(n), it is moved to its final position mod $2^{\lg(n)}=n$, which is its final result position.

Fact IB2 A permutation is routable on an inverse butterfly network if the destinations of the bits constitute a complete set of residues mod m (i.e., the destinations equal 0, 1,…, m-1 mod m) for each subnetwork of width m.

Based on Fact IB1, bits are routed on the inverse butterfly network by moving them to the correct position mod 2 after the first stage, mod 4 after the second stage, etc. Consequently, if the 2 bits entering stage 1, (with 2-bit wide inverse butterfly networks), have destinations equal to 0 and 1 mod 2 (i.e., one is going to R and one to L), Fact IB2 can be satisfied for both bits and they are routable through stage 1 without conflict. Subsequently, the 4 bits entering stage 2 (with the 4-bit wide networks) must have destinations equal to 0, 1, 2 and 3 mod 4 to satisfy Fact IB2 and be routable through stage 2 without conflict. A similar constraint exists for each stage.

Theorem 2 *Any Parallel Extract (pex) instruction on n bits can be implemented with one pass through an inverse butterfly network of lg(n) stages without path conflicts (with the unselected bits on the left zeroed out).*

Proof The pex operation compresses bits in their original order into adjacent bits in the result. Consequently, two adjacent selected data bits that enter the same stage 1

subnetwork must be adjacent in the output—1 bit has a destination equal to 0 mod 2 and the other has a destination equal to 1 mod 2. Thus the destinations constitute a complete set of residues mod 2 and are routable through stage 1. The selected data bits that enter the same stage 2 subnetwork must be adjacent in the output and thus form a set of residues mod 4 and are routable through stage 2. A similar situation exists for the subsequent stages, up to the final n-bit wide stage. No matter what the bit mask of the overall pex operation is, the selected data bits will be adjacent in the final result. Thus the destination of the selected data bits will form a set of residues mod n and the bits will be routable through all lg(n) stages of the inverse butterfly network.

3.3 Need for Two Datapaths

It would be convenient if both pex and pdep can be implemented using the same datapath circuit. Unfortunately, this is not possible.

We first consider trying to implement pex using a butterfly circuit. From Fact 1 we see that parallel extract cannot be mapped to the butterfly network. Parallel extract compresses bits and thus it is easy to encounter scenarios where 2 bits entering the same switch would both require the same output in order to be moved to the correct half (L or R) corresponding to their final destinations. Consider the parallel extract operation shown in Fig. 12. In order to move both bits d and h to the correct half of their final positions, both must be output in the right half after stage 1. This clearly is a conflict and thus parallel extract cannot be mapped to butterfly.

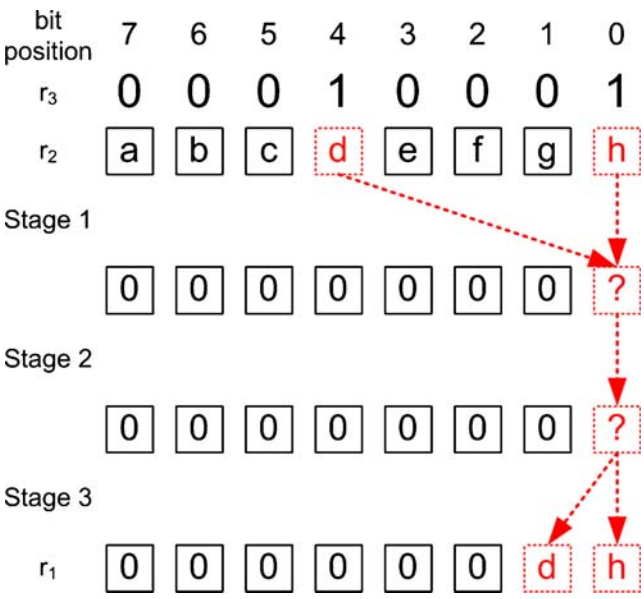

Figure 12 Conflicting paths when trying to map pex to butterfly.

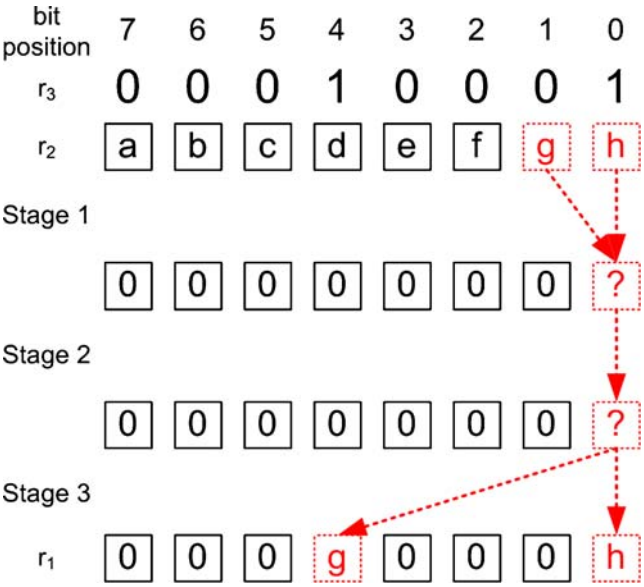

Figure 13 Conflicting paths when trying to map pdep to inverse butterfly.

We now consider implementing pdep using an inverse butterfly circuit. From Fact IB1 we see that parallel deposit cannot be mapped to the inverse butterfly network. Parallel deposit scatters right-aligned bits to the left, and thus it is easy to encounter scenarios where 2 bits entering the same switch would both require moving to the same even, or odd, subnetwork. Consider the parallel deposit operation shown in Fig. 13. In order to move both bits g and h to their final positions mod 2, both must be output in the right subnet (i.e., even network: 0 mod 2) after stage 1. This clearly is a conflict and thus parallel deposit cannot be mapped to the inverse butterfly datapath.

3.4 Towards an Advanced Bit Manipulation Functional Unit

Now that we have shown how the operations map to the datapaths, we can sketch the blocks of a functional unit that supports all four operations—parallel extract, parallel deposit, butterfly permutations and inverse butterfly permutations.

The first building block is a butterfly network followed by a masking stage and extra register storage to hold the configuration for the butterfly network (see Fig. 14, which shows a 64-bit functional unit with six stages). This supports the pdep and bfly instructions. The masking stage zeroes the bits not selected in the pdep operation. For bfly, we use a mask of all "1"s. The extra registers are associated with the functional unit and hold the control bits for the butterfly network. Some ISAs have pre-defined application registers that can be used as these extra registers: they are called *special function registers* in PA-RISC [12, 13] or

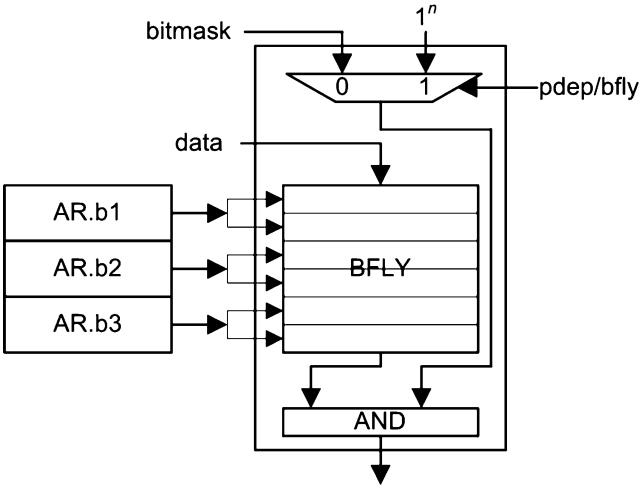

Figure 14 Functional unit supporting butterfly and parallel deposit operations.

application registers (AR's) in the Intel Itanium ISA [14]. In this paper, we will adopt the Itanium nomenclature and call them application registers.

Figure 14 supports butterfly permutations as the application registers can be explicitly loaded with the configuration for some permutation. Parallel deposit is also supported as we can go through the steps suggested by the proof to Theorem 1 to arrive at a configuration for the butterfly network that performs the desired parallel deposit operation. The ARs can be loaded with this configuration and then the input is permuted through the butterfly network; the bits that are not needed are then masked out using the bit mask that defines the desired parallel deposit operation.

Figure 15 shows the analog of this building block for implementing inverse butterfly permutations and the parallel extract instruction. This contains a masking stage followed by an inverse butterfly network and a set of application registers to hold the configuration. The ARs are loaded with the configuration for some inverse butterfly permutation or a parallel extract operation. For parallel extract we mask out the undesired bits with the bit mask and then we permute through the inverse butterfly network.

Figure 16 combines these two building blocks to form a single functional unit that supports all four operations.

Deriving a configuration for parallel deposit or parallel extract *a priori* (by the programmer or compiler) is not always possible. It is only possible when the pex or pdep operation is *static*, i.e., if the bit mask is known when the program is being written or compiled. However, the bit mask might be *variable* and only known at runtime and thus the configuration for the butterfly or inverse butterfly datapath must be produced on the fly. For the highest performance, we would like to have a hardware decoder circuit (Fig. 17) where the input is the n-bit mask and the output is the set of $n/2 \times \lg(n)$ control bits for pdep or pex. We will show how to design this decoder and show that the same decoder circuit can be used for both pex and pdep with the caveat that the ordering of the stages that control bits are routed to is reversed (the circular arrow in Fig. 17 indicates reversing the stages).

Another possible scenario is that the configuration is only known at runtime but it is unchanging across many iterations of a loop, i.e., it is *loop invariant*. In this case, we would like to use the hardware decoder circuit once to load a configuration for pex or pdep into the application registers (see the new multiplexers in front of the ARs in Fig. 18) and then use the application registers directly. This has the advantage of removing the decoder from the critical path, thus decreasing latency for the subsequent pex or pdep instructions, and also possibly conserving power by shutting down the decoder circuitry.

For completeness, we also consider a functional unit that supports the grp permutation instruction. The grp permutation instruction is equivalent to a standard parallel extract ORed with a "parallel extract to the left" of the bits selected with "0"s. Consequently, for the functional unit to support grp, we need to add a second decoder and a second inverse butterfly network to perform the "parallel extract to the left" (Fig. 19).

4 ISA Summary

Table 1 enumerates and defines all the advanced bit manipulation instructions we have discussed so far. We assume a 64-bit wordsize, without loss of generality.

bfly and ibfly For bfly and ibfly, the data bits in GR r_2 are permuted and placed in the destination register GR r_1. Application registers ar.b_i and ar.ib_i, i=1, 2, 3, are used to hold the configuration bits for the butterfly or inverse

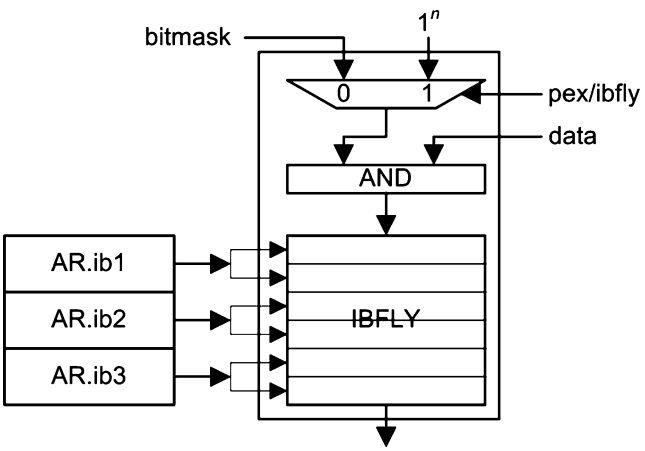

Figure 15 Functional unit building block supporting inverse butterfly and parallel extract operations.

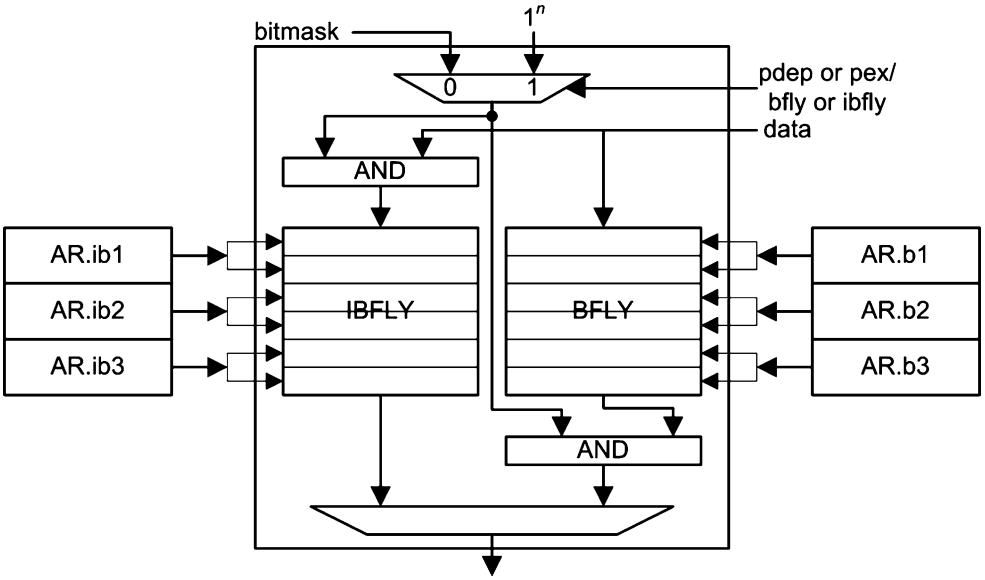

Figure 16 Functional unit supporting butterfly, inverse butterfly, parallel extract and parallel deposit.

butterfly datapath, respectively, and these registers must first be loaded by the mov_ar instruction. The mov ar instruction in Table 1 is used to move the contents of two general-purpose registers to the application registers. The sub-opcode, *x*, indicates which application register, or pair of application registers, are written.

Static pex and pdep Static versions of pex and pdep are used when desired mask patterns are known at compile time. In the static version of the pex instruction, GR r_2 is and'ed with mask GR r_3, then permuted using inverse butterfly application registers $ar.ib_{1-3}$, with the result placed in GR r_1. For static pdep, GR r_2 is permuted using butterfly application registers $ar.b_{1-3}$, then and'ed with mask GR r_3, with the result placed in GR r_1.

Dynamic pex.v and pdep.v Dynamic or variable versions of pex and pdep are used when desired mask patterns are only known at runtime. In the pex.v instruction, the data bits in GR r_2 selected by the "1" bits in the mask GR r_3 are placed, in the same order, in GR r_1. In the pdep.v instruction, the right justified bits in GR r_2 are placed in the same order in GR r_1,

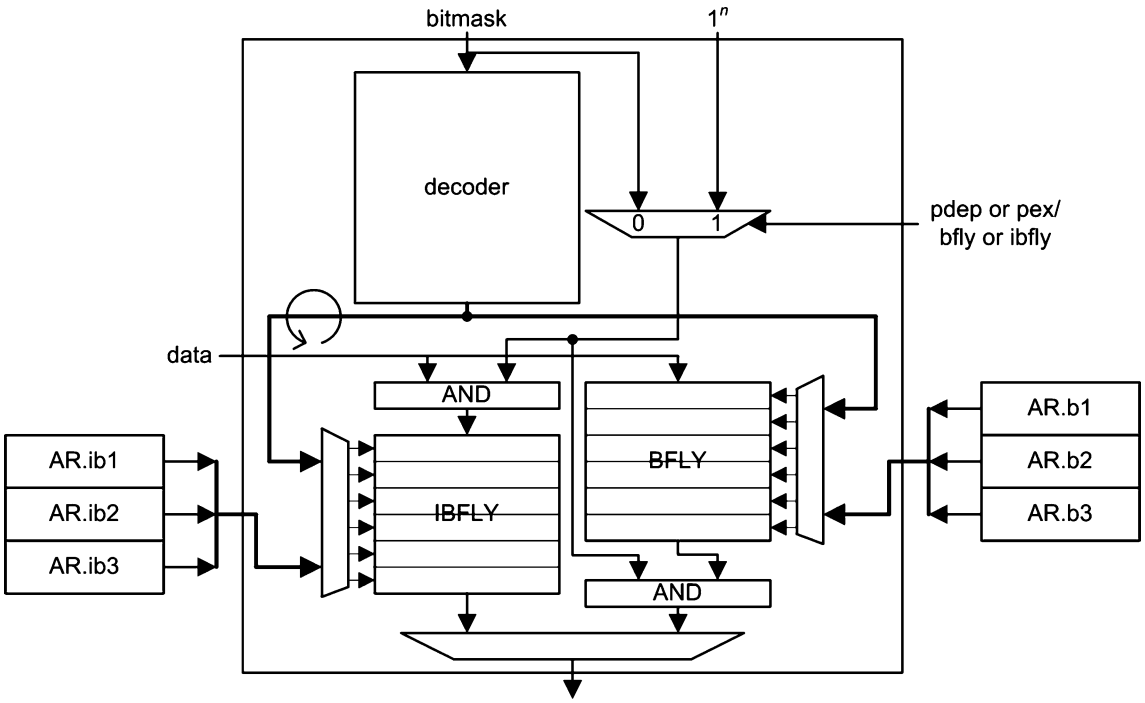

Figure 17 Functional unit supporting butterfly, inverse butterfly and static and variable parallel extract and parallel deposit instructions.

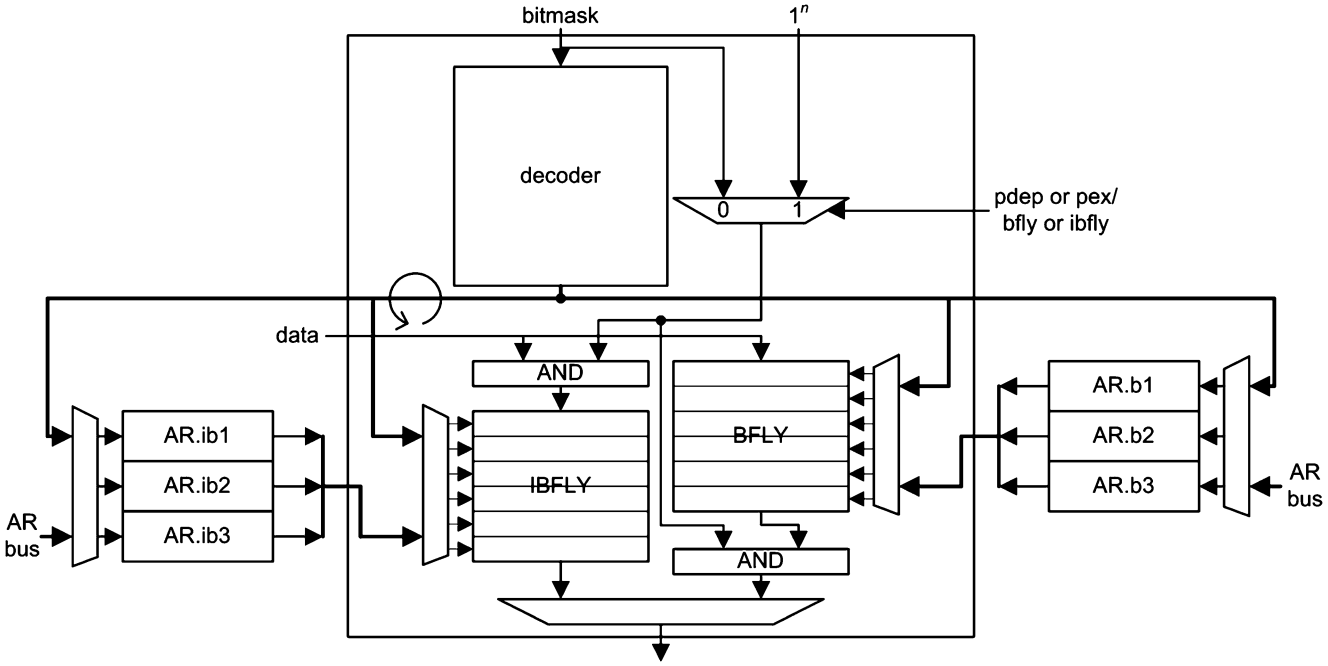

Figure 18 Functional unit supporting butterfly, inverse butterfly and static, variable and loop invariant pex and pdep.

in the positions selected by "1"s in mask GR r_3. For both instructions, the mask r_3 is translated dynamically by a decoder into control bits for an inverse butterfly or butterfly circuit.

Loop-invariant pex and pdep Suppose the particular pattern of bit scatter or gather is determined at execution time, but this pattern remains the same over many iterations of a loop. We call this a *loop-invariant* pex or pdep operation. The setib and setb instructions invoke a hardware decoder to dynamically translate the bitmask GR r_3 to control bits for the datapath stages; these control bits are written to the inverse butterfly or butterfly application registers, respectively, for later use in static pex and pdep instructions.

Table 1 also shows the grp instruction which can perform arbitrary *n*-bit permutations. Due to the complexity

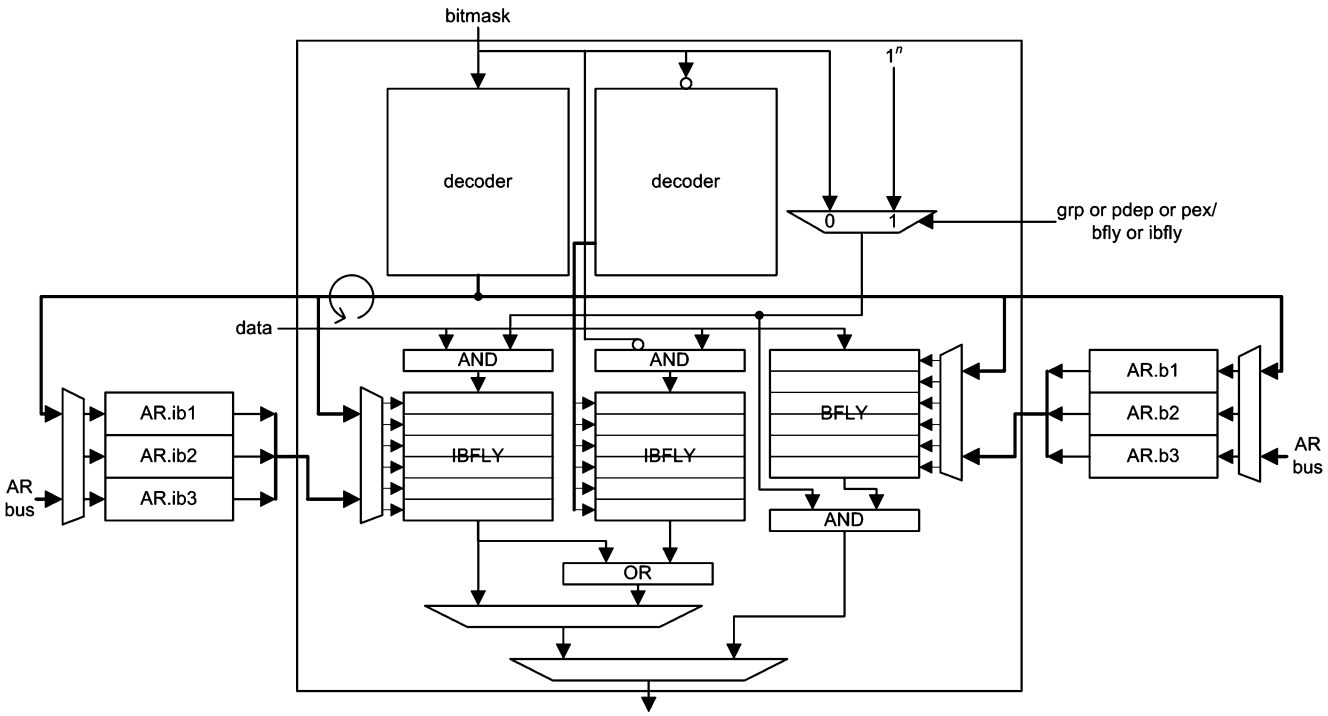

Figure 19 Functional unit supporting butterfly; inverse butterfly; static, variable and loop invariant pex and pdep; and grp.

Table 1 New advanced bit manipulation instructions.

Instruction	Description	Cycles
bfly r_1=r_2, *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$	Perform *Butterfly permutation* of data bits using controls in associated ARs	1
ibfly r_1=r_2, *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$	Perform *Inverse Butterfly permutation* of data bits using controls in associated ARs	1
pex r_1=r_2, r_3, *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$	*Parallel extract, static*: Data bits in r_2 selected by a mask r_3 are extracted, compressed and right-aligned in the result r_1, using datapath controls which have been pre-decoded from the mask and placed in the associated ARs	1
pdep r_1=r_2, r_3, *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$	*Parallel deposit, static*: Right-aligned data bits in r_2 are deposited, in order, in result r_1 in bit positions marked with a "1" in the mask r_3, which has been pre-decoded to give datapath controls placed in the associated ARs	1
mov *ar.x*=r_2, r_3	*Move values from GRs to ARs*, to set datapath controls (calculated by software) for pex, pdep, bfly or ibfly	1
pex.v r_1=r_2, r_3	*Parallel extract, variable*: Data bits in r_2 selected by a dynamically-decoded mask r_3 are extracted, compressed and right-aligned in the result r_1	3
pdep.v r_1=r_2, r_3	*Parallel deposit, variable*: Right-aligned data bits in r_2 are deposited, in order, in result r_1 in bit positions marked with a "1" in the dynamically-decoded mask r_3	3
setib *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$=r_3	*Set inverse butterfly datapath controls in the associated ARs*, using hardware decoder to translate the mask r_3 to inverse butterfly controls	2
setb *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$=r_3	*Set butterfly datapath controls in the associated ARs*, using hardware decoder to translate the mask r_3 to butterfly controls	2
grp r_1=r_2, r_3	*Perform Group permutation (variable)*: Data bits in r_2 corresponding to "1"s in r_3 are grouped to the right, while those corresponding to "0"s are grouped to the left	3

of implementing grp (see Fig. 19), and the fact that arbitrary permutations can be implemented in fewer instructions using bfly and ibfly instructions, it may not be necessary to implement grp. Furthermore, it can be emulated by a short sequence of pex and Boolean instructions.

The last column of Table 1 shows the expected number of cycles taken for the execution of the instruction. All static instructions (with pre-loaded ARs) take a single cycle, comparable to the time taken for an add instruction. The hardware decoder takes about two cycles, hence the setib and setb instructions take two cycles each. The variable pex.v and pdep.v and grp instructions each take up to three cycles each because they have to go through the hardware decoder first and then incur some additional datapath latency through the inverse butterfly or butterfly networks and the output multiplexers.

5 Applications

We now describe how bfly, ibfly, pex and pdep instructions can be used in existing applications, to give speedup estimates that are currently realizable. Use of these novel instructions in new algorithms and applications will likely produce even greater speedup.

Table 2 summarizes each of the applications described below in terms of the advanced bit manipulation instructions it uses. (Check marks indicate definite usage, check marks in parenthesis indicate usage in alternative algorithms and question marks indicate potential usage.) The use of the mov ar instruction is assumed (but not shown in Table 2) whenever the bfly, ibfly, or static pex and pdep instructions are used.

Table 2 shows that static pex and pdep are the most frequently used, the variable pdep.v is not used at all, and the variable pex.v is only used twice. The loop-invariant pex and pdep instructions, indicated by the use of setib and setb instructions, are only used for the LSB Steganography application. The grp instruction is only used in a block cipher proposed by Lee et al. [17].

5.1 Bit Compression and Decompression

The Itanium IA-64 [14] and IA-32 [18] parallel compare instructions produce subword masks—the subwords for which the relationship is false contain all zeros and for which the relationship is true contain all ones. This representation is convenient for subsequent subword masking or merging. The SPARC VIS [19] parallel compare instruction produces a bit mask of the results of the comparisons. This representation is convenient if some decision must be made based on the outcome of the multiple comparisons. Converting from the subword representation to the bitmask representation for k subwords requires k extract instructions to extract a bit from each subword and $k-1$ deposit instructions to concatenate the bits; a single static pex instruction accomplishes the same thing.

The SSE instruction pmovmskb [18] serves a similar purpose; it creates an 8- or 16-bit mask from the most

Table 2 Summary of bit manipulation instruction usage in various applications.

Application		Instruction							
		bfly and ibfly	pex	setib	pdep	setb	pex.v	pdep.v	grp
Binary compression			✓						
Binary decompression/expansion					✓				
LSB steganography	Encoding				✓	✓			
	Decoding		✓	✓					
Binary image morphology			✓				(✓)		
Transfer coding	Encoding				✓				
	Decoding		✓						
Bioinformatics	Compression		✓						
	Reversal	✓							
	Translation				✓				
	Alignment		✓						
Compressed integers	Encoding				✓				
	Decoding		✓						
Random number generation							✓		
Cryptology		✓	?	?	?	?	?	?	(✓)

significant bit from each byte of a MMX or SSE register and stores the result in a general purpose register. However, pex offers greater flexibility than the fixed pmovmskb, allowing the mask, for example, to be derived from larger subwords, or from subwords of different sizes packed in the same register.

Similarly, binary image compression performed by MATLAB's bwpack function [20] benefits from pex. Binary images in MATLAB are typically represented and processed as byte arrays—a byte represents a pixel and has permissible values 0x00 and 0x01. However, certain optimized algorithms are implemented for a bitmap representation, in which a single bit represents a pixel.

To produce one 64-bit output word requires only eight static pex instructions to extract 8 bits in parallel from 8 bytes and seven dep instructions to pack these eight 8-bit chunks into one output word (Fig. 20). For decompression, as with the bwunpack function, only seven extr instructions are required to pull out each byte and only eight pdep instructions to scatter the bits of each byte to byte boundaries.

5.2 Least Significant Bit Steganography

Steganography [21] refers to the hiding of a secret message by embedding it in a larger, innocuous cover message. A simple type of steganography is least significant bit (LSB) steganography in which the least significant bits of the color values of pixels in an image, or the intensity values of samples in a sound file, are replaced by secret message bits. LSB steganography encoding can use a pdep instruction to expand the secret message bits and place them at the least significant bit positions of every subword. Decoding uses a pex instruction to extract the least significant bits from each subword and reconstruct the secret message.

LSB steganography is an example of an application that utilizes the loop-invariant versions of the pex and pdep instructions. The sample size and the number of bits replaced are not known at compile time, but they are constant across a single message. Figure 21 depicts an example LSB steganography encoding operation in which the four least significant bits from each 16-bit sample of PCM encoded audio are replaced with secret message bits.

5.3 Binary Image Morphology

Binary image morphology is a collection of techniques for binary image processing such as erosion, dilation, opening, closing, thickening, thinning, etc. The bwmorph function in MATLAB [20] implements these techniques primarily

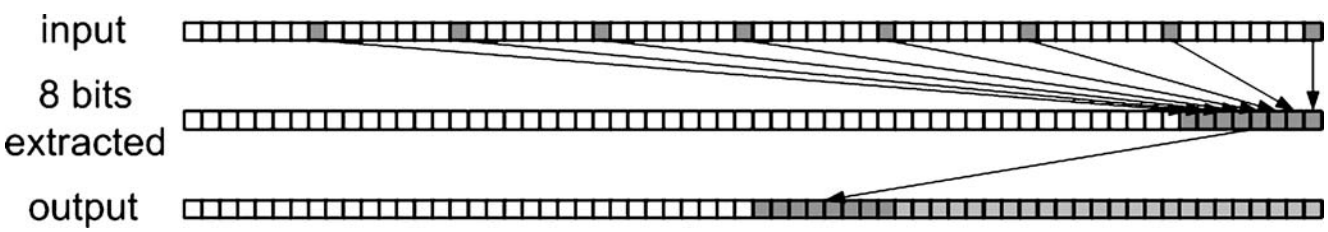

Figure 20 Compressing each input word requires one pex and one dep.

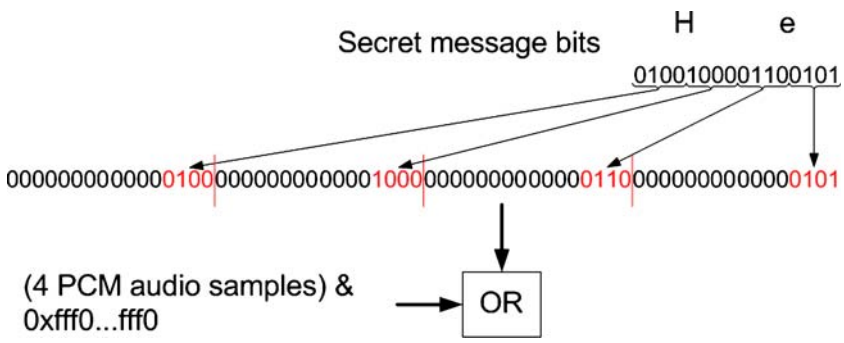

Figure 21 LSB steganography encoding (4 bits per 16-bit PCM encoded audio sample).

through one or more table lookups applied to the 3×3 neighborhood surrounding each pixel (i.e. the value of 9 pixels is used as index into a 512 entry table). In its current implementation, bwmorph processes byte array binary images, not bitmap images, possibly due to the difficulty in extracting the neighborhoods in the bitmap form.

If the images are processed in bitmap form, a single pex instruction extracts the entire index at once (assuming a 64-bit word contains an 8×8 block of 1-bit pixels, as in Fig. 22). As the algorithm steps through the pixels, the neighborhood moves. This means that the bitmask for extracting the neighborhood is shifted and a dynamic pex.v instruction may be needed. Alternatively, the data might be shifted, rather than the mask, such that the desired neighborhood always exists in a particular set of bit positions. In this case, the mask is fixed, and only a static pex is needed. Table 2 indicates the latter—with pex indicated by a check mark and pex.v in parenthesis for the alternative algorithm.

5.4 Transfer Coding

Transfer coding is the term applied when arbitrary binary data is transformed to a text string for safe transmission using a protocol that expects only text as its payload. Uuencoding [22] is one such encoding originally used for transferring binary data over email or usenet. In uuencoding, each set of 6 bits is aligned on a byte boundary and 32 is added to each value to ensure the result is in the range of the ASCII printable characters. Without pdep, each field is individually extracted and has the value 32 added to it. With pdep, eight fields are aligned at once and a parallel add instruction adds 32 to each byte simultaneously (Fig. 23 shows four parallel fields). Similarly, for decoding, a parallel subtract instruct deducts 32 from each byte and then a single pex compresses eight 6-bit fields.

5.5 Bioinformatics

Pattern matching and bit scatter/gather operations are also found in bioinformatics—the field of analysis of genetic information. DNA, the genetic code contained within the nucleus of each cell, is a strand of the nucleotide bases adenine, cytosine, guanine and thymine. These bases are typically represented by an ASCII string using the 8-bit characters A, C, G and T. However, a 2-bit encoding of the nucleotides is more efficient and can significantly increase performance of matching and searching operations on large genomic sequences (the human genome contains 3.2 billion nucleotides). The ASCII codes for the characters A, C, G and T differ in bit positions 2 and 1 (A: 00, C: 01, G: 11, T: 10) and these 2 bits can be used to encode each nucleotide [23]. Thus a fourfold compression of a genomic sequence simply requires a single pex instruction to select bits 2 and 1 of each byte of a word (Fig. 24).

A strand of DNA is a double helix—there are really two strands with the complementary nucleotides, A ↔ T and C ↔ G, aligned. When performing analysis on a DNA string, often the complementary string is analyzed as well. To obtain the complementary string, the bases are complemented and the entire string is reversed, as the complement string is read from the other end. The reversal of the DNA string amounts to a reversal of the ordering of the pairs of bits in a word. This is a straightforward bfly or ibfly permutation.

The DNA sequence is transcribed by the cell into a sequence of amino acids or a protein. Often the analysis of the genetic data is more accurate when performed on a protein basis, such as is done by the BLASTX program

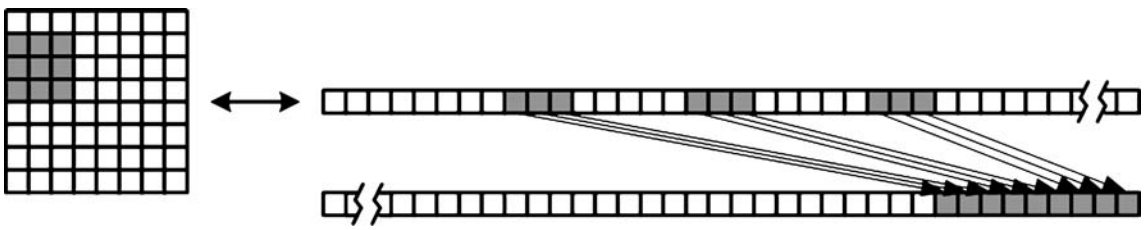

Figure 22 Using pex to extract a 3×3 neighborhood of pixels from register containing an 8×8 block of pixels.

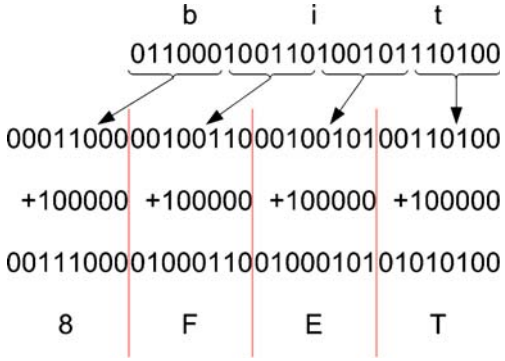

Figure 23 Uuencode using pdep.

[24]. A set of three bases, or 6 bits of data, corresponds to a protein codon. Translating the nucleotides to a codon requires a table lookup operation using each set of 6 bits as an index. An efficient algorithm can use pdep to distribute eight 6-bit fields on byte boundaries, and then use the result as a set of table indices for a parallel table lookup (ptlu) instruction [25–27] to translate the bytes (Fig. 25).

When aligning two DNA sequences, certain algorithms such as the BLASTZ program (mentioned in the introduction) use a "spaced seed" as the basis of the comparisons. This means that n out of m nucleotides are used to start a comparison rather than a string of n consecutive nucleotides. The remaining slots effectively function as wild cards, often causing the comparison to yield better results. For example, BLASTZ uses 12 of 19 (or 14 of 22 nucleotides) as the seed for comparison. The program compresses the 12 bases and uses the seed as an index into a hash table. This compression is a pex operation selecting 24 of 38 bits (Fig. 26).

5.6 Integer Compression

Internet search engine databases, such as Google's, consist of lists of integers describing frequency and positions of query terms. These databases are compressed to minimize storage space, memory bus traffic and cache usage. To support fast random access into these databases, integer compression—using less than 4 bytes to represent an integer—is utilized. One integer compression scheme is a variable byte encoding in which each byte contains seven data bits and a flag bit indicating whether the next byte is part of the current integer (flag="0") or starts the next integer (flag="1") [28]. pex can accelerate the decoding of

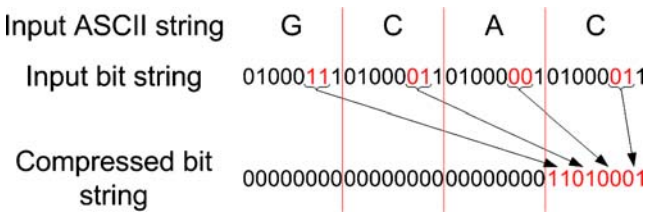

Figure 24 Compression of sequence GCAC using pex.

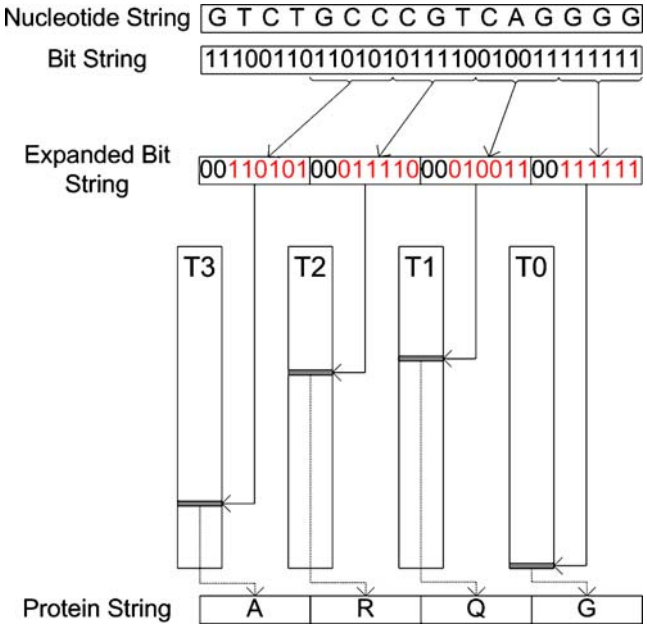

Figure 25 Translation of four sets of three nucleotides into four protein codons using pdep and ptlu.

integers by compacting 7-bit fields from each byte (Fig. 27), and pdep can speedup encoding by placing consecutive 7-bit chunks of an integer on byte boundaries.

5.7 Random Number Generation

Random numbers are very important in cryptographic computations for generating nonces, keys, random values, etc. Random number generators contain a source of randomness (such as a thermal noise generator) and a randomness extractor that transforms the randomness so that it has a uniform distribution. The Intel random number generator [29] uses a von Neumann extractor. This extractor breaks the input random bits, $X = x_1x_2x_3\ldots$, into a sequence of pairs. If the bits in the pair differ, the first bit is output. If the bits are the same, nothing is output. This operation is equivalent to using a pex.v instruction on each word X from the randomness pool with the mask:

$$\text{Mask} = x_1 \oplus x_2 \| 0 \| x_3 \oplus x_4 \| 0 \| \ldots$$

or equivalently,

$$\text{Mask} = (X \oplus (X << 1)) \& 0x\text{AAA}\ldots\text{A}.$$

5.8 Cryptology

A number of popular ciphers, such as DES, have permutations as primitive operations. In earlier work, we have shown that the inclusion of permutation instructions such as bfly, ibfly (or grp) can greatly improve the performance of the inner loop of these functions [9–11,

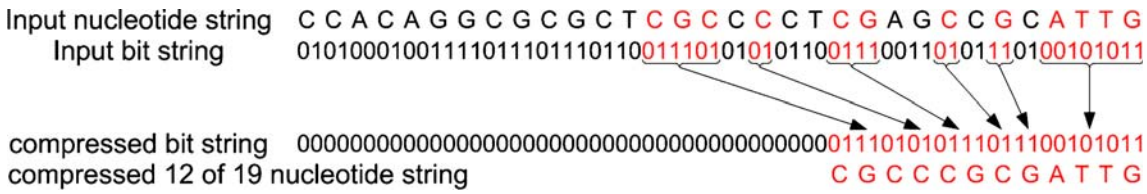

Figure 26 Compression of 12 of 19 bases (24 of 38 bits) in BLASTZ using pex.

30]. Also, these instructions can be used as powerful primitive operations in the design of the next generation of ciphers [17, 31] and hash functions (especially for the Cryptographic Hash Algorithm Competition (SHA-3) [32]).

Since all the pex and pdep instructions—static, loop—invariant and variable—are also very likely to be useful for cryptanalysis algorithms, we indicate them as "?" in Table 2.

6 Performance Results

We coded kernels for the above binary compression and decompression, steganography, transfer coding, bioinformatics translate, integer compression and random number generation applications and simulated them using the SimpleScalar Alpha simulator [33] enhanced to recognize our new instructions. The latencies of the instructions in the simulator are as given in Table 1. Figure 28 shows our performance results, normalized to the baseline Alpha ISA cycle counts. The processor with pex and pdep instructions exhibits speedups over the base ISA ranging from 1.13× to 10.04×, with an average of 2.29× (1.94× excluding the rng benchmark).

Random number generation exhibited the greatest speedup due to the fact that a variable pex.v operation is performed. A single pex.v instruction replaces a very long sequence of instructions that loops through the data and mask and conditionally shifts each bit of the input to the correct position of the output based on whether the corresponding bit of the mask is '0' or '1'. Of the benchmarks for static pex and pdep, the simple bit compression and decompression functions exhibited the greatest speedup as these operations combine many basic instructions into one pex or pdep. The speedup is lower in the steganography encoding case because there are only four fields per word, and also in the uudecode and BLASTX translate case because there are fewer fields overall. The lowest speedups were for integer compression cases as a smaller fraction of the runtime is spent on compression or decompressing bit fields.

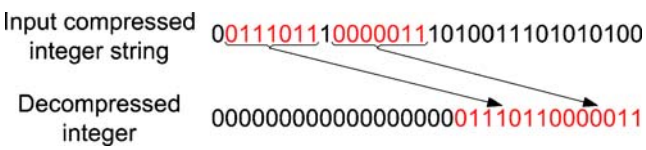

Figure 27 Decoding of 14-bit integer encoded in 2 bytes.

7 Detailed Implementation

To evaluate the cost of implementing the advanced bit manipulation functional units described in Section 3.4, Figs. 16, 17, 18 and 19, we first need to understand the implementation of the major components in such a functional unit. The most complex component in Figs. 17, 18 and 19 is the hardware decoder that takes a register value (representing a bitmask) and turns it into the control bits of the stages of the butterfly or inverse butterfly datapath. Figure 16 does not have such a hardware decoder, as it implements only the static versions of pex and pdep.

Hence, in order to evaluate the functional unit circuit latency and area, we must first define what is contained within the hardware decoder component. We will first develop an algorithm to obtain the $n/2 \times \lg(n)$ butterfly or inverse butterfly control bits from the n-bit pdep or pex bitmask. Then, we will show how to design a hardware decoder circuit that implements this algorithm. In the case of static pex or pdep instructions, the algorithm can be used by the compiler to generate the control bits for the inverse butterfly or

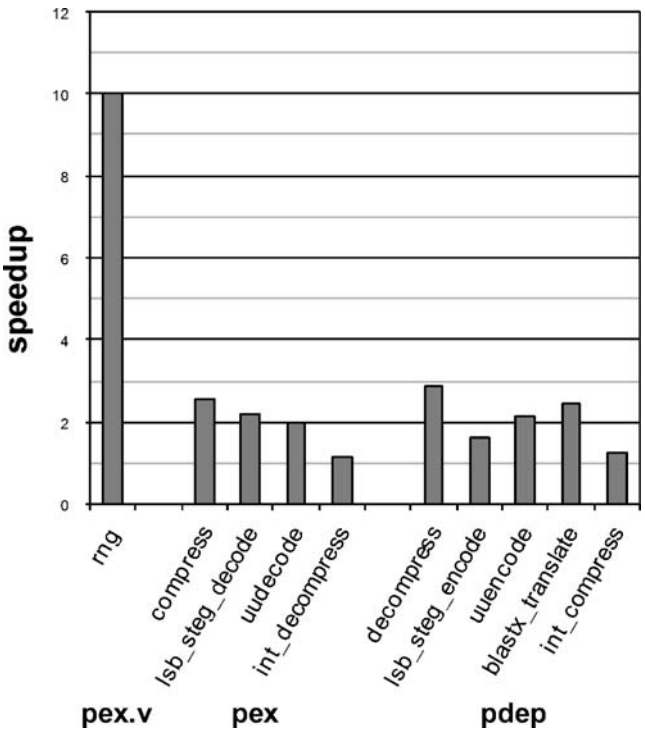

Figure 28 Speedup using pex.v, pex and pdep.

butterfly datapath. In the case of dynamic or loop-invariant pex and pdep instructions, the hardware decoder is used.

7.1 Decoding the Mask into Datapath Controls

The steps in the proof to Theorem 1 give an outline for how to decode the n-bit bitmask into controls for each stage of a butterfly datapath. For each right half of a stage of a subnetwork, we count the number of "1"s in that local right half of the mask, say k "1"s, and then set the k rightmost control bits to "1"s and the remaining bits to "0"s. This serves to keep block X in the local R half and export Y to the local L half (refer to Figs. 5 and 7 for nomenclature). We then assume that we explicitly rotate Y and Z to be the rightmost bits in order in the local L half. Then, we iterate through the stages and come up with an initial set of control bits. After this, we eliminate the need for explicit rotations of Y and Z by modifying the control bits instead. This is accomplished by a *left rotate and complement upon wrap around* (LROTC) operation, rotating the control bits by the same amount obtained when assuming explicit rotations.

We will now simplify this process considerably. First, note that when we modify control bits to compensate for a rotation in a given stage, we do so by propagating the rotation through all the subsequent stages. This means that when the control bits of a local L are modified, they are rotated and complemented upon wrap around by the number of "1"s in the local R, and by the number of "1"s in the local R of the preceding stage, and by the number of "1"s in all the local R's of all preceding stages up to the R in the first stage. In other words, the control bits of the local L are rotated by the total number of "1"s to its right in the bitmask.

Consider the example of Fig. 6b. The control bit in stage 3 in the L_L subnetwork is initially a "0" when we assumed explicit rotations. We first rotated and complemented this bit by 3, the number of "1"s in R of the bitmask: 0 → 1 → 0 → 1 (Fig. 8). We then rotated and complemented this bit by another 1 position, the number of "1"s in L_R of the bitmask: 1 → 0. This yielded the final control bit in Fig. 6c. Overall we rotated this bit by 4, the total number of "1"s to the right of L_L or to the right of bit position 6. This is a Population Count (POPCNT) of the bitstring from the rightmost bit to bit position 6.

Second, we need to produce a string of k "1"s from a count (in binary) of k, to derive the initial control bits assuming explicit rotations. This can also be done with a LROTC operation. We start with a zero string of the correct length and then for every position in the rotation, we wrap around a "0" from the left and complement it to get a "1" on the right. The end result, after a LROTC by k bits, is a string of the correct length with k rightmost bits set to "1" and the rest set to "0" (Fig. 29, where k=3).

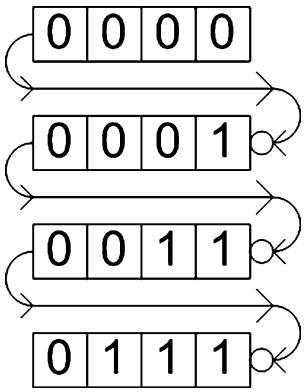

Figure 29 LROTC("0000", 3)= "0111".

We can now combine these two facts: the initial control bits are obtained by a LROTC of a zero string the length of the local R by the POPCNT of the bits in the bitmask in the local R and all bits to the right of it. We denote a string of k "0"s as 0^k. We specify a bitfield from bit h to bit v as {h:v}, where v is to the right of h. So,

- for stage 1, we calculate the control bits as LROTC($0^{n/2}$, POPCNT(mask{$n/2-1$:0})),
- for stage 2, we calculate the control bits as LROTC($0^{n/4}$, POPCNT(mask{$3n/4-1$:0})) for L and LROTC($0^{n/4}$, POPCNT(mask{$n/4-1$:0})) for R,
- for stage 3, we calculate the control bits as LROTC($0^{n/8}$, POPCNT(mask{$7n/8-1$:0})) for L_L, LROTC($0^{n/8}$, POPCNT(mask{$5n/8-1$:0})) for L_R, LROTC($0^{n/8}$, POPCNT(mask{$3n/8-1$:0})) for R_L and LROTC($0^{n/8}$, POPCNT(mask{$n/8-1$:0})) for R_R, and so on for the later stages.

Let us verify that this is correct using the example of Fig. 6c:

- for stage 1, the control bits are LROTC(0^4, POPCNT("1101"))=LROTC(0^4, 3)=0111,
- for stage 2, the control bits of L are LROTC(0^2, POPCNT ("101101"))=LROTC(0^2, 4)=00 and the control bits of R are LROTC(0^2, popcnt("01"))=LROTC(0^2, 1)=01,
- for stage 3, the control bit of L_L are LROTC(0^1, POPCNT("0101101"))=LROTC(0^1, 4)=0, the control bit of L_R is LROTC(0^1, POPCNT("01101"))=LROTC(0^1, 3)=1, the control bit of R_L is LROTC(0^1, POPCNT("101"))=LROTC(0^1, 2)=0 and the control bit of R_R is LROTC(0^1, POPCNT("1"))=LROTC(0^1, 1)=1.

This agrees with the result shown in Fig. 6c.

One interesting point is that for stage 1 we need the population count of the odd multiples of $n/2^1$ bits, for stage 2 we need the population counts of the odd multiples of $n/2^2$ bits, for stage 3 we need the population counts of the odd multiples of $n/2^3$ bits and so on. Overall we need the

counts of the k rightmost bits, for k=0 to $n-2$. We call this the set of *prefix population counts*.

Using the two new functions we have defined, LROTC and *prefix-population-counts*, we now present an algorithm (Fig. 30) to decode the n mask bits into the $n\lg(n)/2$ control bits for pdep (and for pex).

7.1.1 Decoding the pex Mask into Control Bits for the Inverse Butterfly Datapath

The control bits for the inverse butterfly for a pex operation can also be obtained using Algorithm 1, with the one caveat that the controls for stage i of the butterfly datapath are routed to stage $\lg(n)-i+1$ in the inverse butterfly datapath. This can be shown using an approach similar to that in the previous section, except for working backwards from the final stage.

7.2 Hardware Decoder

The execution time of Algorithm 1 in software is approximately 1,200 cycles on an Intel Pentium-D processor. This software routine is useful for static pex or pdep operations and perhaps for loop invariant pex or pdep if the amount of processing in the loop dwarfs the 1,200 cycle execution time. However, for dynamic pex.v and pdep.v we require a hardware decoder that implements Algorithm 1 in order to achieve a high performance. Fortunately, Algorithm 1 just contains two basic operations, population_count and LROTC, both of which have straightforward hardware implementations.

The first stage of the decoder is a *parallel prefix population counter*. This is a circuit that computes in parallel all the population counts of step 1 of Algorithm 1. The circuit is a parallel prefix network with each node performing carry-save addition (i.e. a set of full adders). The counters resemble carry shower counters [34] in which the inputs are grouped into sets of three lines which are input into full adders. The sum and carry outputs of the full adders are each grouped into sets of three lines which are input to another stage of full adders and so on. The parallel prefix architecture resembles radix-3 Han-Carlson [35], a parallel prefix look-ahead carry adder that has $\lg(n)+1$ stages with carries propagated to the odd positions in the extra final stage. The radix-3 nature stems from the carry shower counter design, as we group 3 lines to input to a full adder at each level. The similarity to Han-Carlson is due to the 1- and 2-bit counts (see next paragraph) being deferred to the end, similar to odd carries being deferred in the Han-Carlson adder. Thus, the counter has $\log_3(n)$+two stages. Figure 31 depicts a dot diagram of the parallel prefix network.

One simplification of the counter is based on the properties of rotations—that they are invariant when the rotation amount differs by the period of rotation. Thus, for the ith stage of the butterfly network, the POPCNTs are only computed mod $n/2^{i-1}$. For example, for the 64-bit hardware decoder, for the 32 butterfly stage 6 POPCNTs corresponding to the odd multiples of $n/64$, we need only compute the POPCNTs mod 2—only the least significant bit; for the 16 butterfly stage 5 POPCNTs, we need only compute the POPCNTs mod 4—the two least significant bits; and so on. Only the POPCNT of 32 bits for stage 1 requires the full $\lg(n)$-bit POPCNT.

The outputs from the population counter control the LROTC circuits, one for each local R of each stage. Each LROTC circuit is realized as a barrel rotator modified to complement the bits that wrap around (Fig. 32a). However, while a standard 2^m-bit rotator has m stages and control bits, this rotator has $m+1$ stages and control bits. The final stage selects between its input and the complement, as the

Algorithm 1: To generate the $n\lg(n)/2$ butterfly or inverse butterfly control bits from the n-bit mask.
Input: mask; the bitmask
Output: bcb; the $\lg(n) \times n/2$ matrix containing the butterfly control bits
ibcb; the $\lg(n) \times n/2$ matrix containing the inverse butterfly control bits

Let $x \| y$ indicate the concatenation of bit patterns x and y. POPCNT(a) is the population count of "1"s in bitfield a. mask{h:v} is the bitfield from bit h to bit v of the mask. 0^k indicates a bit-string of k zeros. LROTC(a, rot) is a "left rotate and complement on wrap" operation, where a is the input and rot is the rotation amount.

1. Calculate the prefix population counts:
    For i = 1, 2, …, n–2
        pc[i] = POPCNT(mask{i:0})

2. Calculate the butterfly (and inverse butterfly) control bits for each stage by performing LROTC(0^k, pc[m]), where k is the size of the local R and m is the set of the leftmost bit positions of the local R's:
    bcb = ibcb = {}
    For i = 1, …, lg(n) //for each stage
        $k = n/2^i$ //number of bits in local R
        For j = 1, 3, 5, … 2^i-1 //for each local R
            $m = j*k$-1 //the leftmost bit position of the local R
            temp = LROTC(0^k, pc[m])
            bcb[i] = temp ‖ bcb[i] //for butterfly datapath
            ibcb[lg(n)–i+1] = temp ‖ ibcb[lg(n)–i+1] *//for inverse butterfly datapath*

Figure 30 Algorithm 1 for decoding a bitmask into controls for a butterfly (or inverse butterfly) datapath.

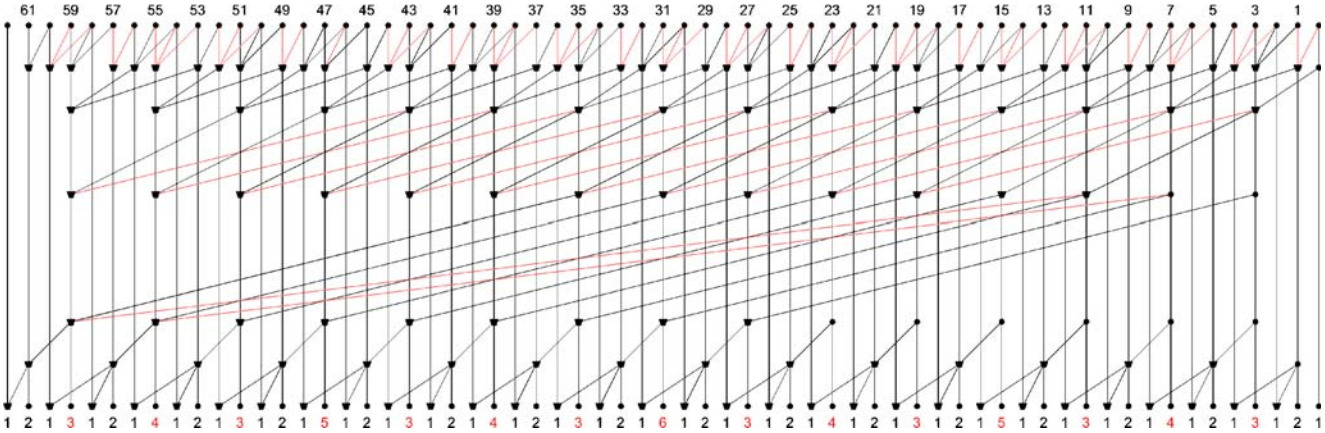

Figure 31 Parallel prefix circuit. The *numbers at the top* are bit position. The *numbers at the bottom* indicate the number of bits in the sum. The *trapezoids* indicate adder blocks.

bits wrap 2^m positions. Propagating the zeros at the input greatly simplifies the circuit (see Algorithm 1—we always calculate LROTC(0^k, count); Fig. 32b). Furthermore, for the counts of the last butterfly stage, the LROTC circuits can be eliminated as LROTC("0", b)=b for single bit b. An overall diagram of the decoder is shown in Fig. 33.

The outputs from the decoder are routed to the butterfly network and its application registers to be used for the pdep instruction. Additionally, the outputs are routed to the inverse butterfly network and its application registers, reversing the order of the stages, to be used for the pex instruction. Unfortunately, there is no overlap between the decoder and the routing of the data through the butterfly network for the pdep instruction since the control bits for the first stage of the butterfly network depend on the widest population count (see Algorithm 1 and Fig. 33), which takes the longest to generate.

7.3 Circuit Evaluation Results

We evaluated the functional units of Figs. 16, 18 and 19 for timing and area. These support static pex and pdep only (Fig. 16), loop invariant and dynamic pex.v and pdep.v as well (Fig. 18), and grp as well (Fig. 19). All three also support bfly and ibfly permutation instructions. The circuits in Figs. 18 and 19 are implemented with a 3-stage pipeline. The hardware decoder occupies the first two pipeline stages due to its slow parallel prefix population counter. The butterfly (or inverse butterfly) network is in the third stage.

The various functional units were coded in Verilog and synthesized using Synopsys Design Compiler mapping to a TSMC 90 nm standard cell library [36]. The designs were compiled to optimize timing. The decoder circuit was initially compiled as one stage and then Design Compiler automatically pipelined the subcircuit. Timing and area figures are as reported by Design Compiler. We also synthesized a reference ALU using the same technology library as a reference for latency and area comparisons.

Table 3 summarizes the timing and area for the circuits. This shows that the 64-bit functional unit in Fig. 16 supporting static pex and pdep has a shorter latency than that of a 64-bit ALU and about 90% of its area (in NAND-gate equivalents). However, to support variable and loop-invariant pex and pdep (as in Fig. 18), which requires a

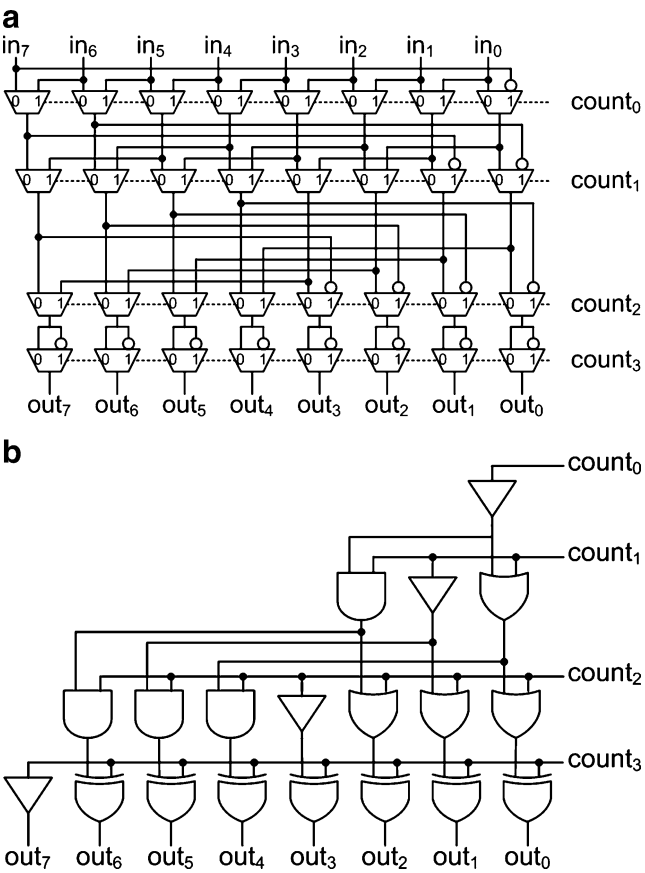

Figure 32 **a** Barrel rotator implementation of 8-bit LROTC circuit. **b** Simplified LROTC circuit obtained from propagating zeros at input.

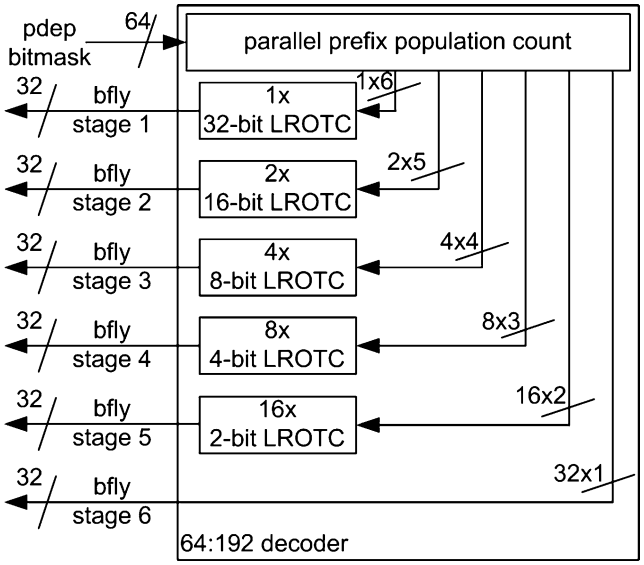

Figure 33 Hardware decoder for dynamic pdep and pex operation (for pex, the order of the outputs is reversed).

hardware decoder (as described in detail above), the advanced bit manipulation functional unit will be 16% slower than an ALU and 2.25×larger. To also support a fast implementation of the grp instruction, the proposed new unit will be 23% slower than an ALU and about 3.2×larger in area.

Table 4 shows the number of different circuit types, to give a sense for why the functional units supporting variable pex.v, pdep.v and grp are so much larger. It clearly shows that supporting variable operations comes at a high price. The added complexity is due to the complex decoder combinational logic and to the additional pipeline registers and multiplexer logic. This explains why in Table 3, the variable circuits (Figs. 18 and 19) have approximately 22–29% longer cycle time latencies compared to the static case (Fig. 16), due to the decoder complexity and pipeline overhead. They are also 2.5 to 3.6 times larger than the static case.

8 Related Work

Advanced bit manipulation operations have usually been supported only by supercomputers. The Soviet BESM-6 supercomputer had pack and unpack instructions which are similar to pex and pdep, respectively [37]. The use of these instructions appears to be for cryptanalytic purposes [1]. However, there is very little information available about this computer and its uses.

The Cray vector supercomputers contain a bit matrix multiply (bmm) instruction [4]. This instruction multiplies a $(1\times n)$ bit vector or an $(n\times n)$ bit matrix by an $(n\times n)$ bit matrix. The matrix–matrix multiplication takes approximately $n/2$ cycles (assuming two vector lanes), not counting loading of a 64×64 matrix into the bmm unit. This instruction can be used to emulate any of the advanced bit manipulation operations described in this paper. The cost of the instruction is a special vector register to hold the $(n\times n)$ multiplier matrix (or 4,096 bits of storage for $n=64$) and a large combinatorial circuit with 64 AND-XOR trees. Our proposed functional units are much smaller and are a better fit for a commodity microprocessor. We examine smaller bit matrix multiplication primitives in a separate paper [38].

Commodity ISAs also contain byte or subword permutation instructions such as the PowerPC AltiVec vperm instruction [39], the SPARC VIS bshuffle instruction [19], the PA-RISC permute and mix instructions [40], the IA-32 pshufb instruction [18] or the IA-64 mux instruction [14]. These permute 16-bit subwords or bytes, but not bits. For example, the vperm instruction takes three source operands—the first two are the data to be permuted and the third is a list of 16 indices that describe which source byte to write to each byte of the output. The vperm instruction can be used to permute bits in a routine that is more efficient than masking and shifting. However, our dedicated bit permutation instructions are still much faster.

In [41–43], Lee discussed the mix instruction she first proposed for PA-RISC MAX-2 [40], and also proposed new mixpair, check, excheck, exchange and permset instructions, for all powers of 2, down to a single bit. These subword and bit permutation instructions are proposed as the canonical primitives needed to accelerate rearrangements of 2-dimensional objects and bit-planes.

The MicroUnity MediaProcessor architecture [44] contains a number of instructions that manipulate the subwords (which are power of 2 bits wide) of a register. (Unfortunately, this processor was never implemented.) The operations supported include bit permutation; arbitrary byte permutation (similar to the instructions listed above for other commodity ISAs); shuffling and swizzling; and parallel shift, rotate, extract and deposit operations. These instructions are not like pex or pdep in which multiple

Table 3 Latency and area of proposed functional units.

Unit	Cycle time (ns)	Relative cycle time	Area (NAND gate equivalent)	Relative area
ALU	0.50	1	7.5K	1
Fig. 16: pex, pdep	0.48	0.95	6.8K	0.90
Fig. 18: pex.v, pdep.v	0.58	1.16	16.9K	2.25
Fig. 19: grp	0.62	1.23	24.2K	3.22

Table 4 Number of registers and logical blocks in functional units.

Unit	Pipeline Registers	Butterfly and Inverse Butterfly Networks	Decoders	MUXes
Fig. 16	0	2	0	1
Fig. 18	~9.25	2	1	13
Fig. 19	~14.5	3	2	14

arbitrary fields are compressed or expanded, but rather standard extract and deposit operations are performed on each subword in parallel. The bit permutation instruction operates on each byte of the input operand, performing any permutation with repetitions of the bits within a byte. The instruction also optionally transposes the input (interpreting the 128-bit input as two 8×8 bit matrices) before permuting the bits within each byte. A sequence of three such permutation instructions can perform any arbitrary permutation of the two 64-bit subwords of the 128-bit data input. Note that these instructions require four 128-bit operands (three inputs and one output).

Other permutation instructions have also been proposed in the research community for microprocessors. In [45], the xbox instruction was introduced to accelerate permutations in symmetric-key ciphers. This instruction takes two source operands—the data to be permuted and a list of eight indices (for 64-bit registers) that describe how to permute a byte of the first operand. Eight xbox instructions (and seven xor instructions) are required to permute a 64-bit word, considerably slower than the execution of our bfly and ibfly instructions.

In [10, 11, 30, 46–48], Lee et al. proposed the grp, pperm, cross, omflip, swperm and sieve bit permutations. The grp instruction [10, 11] was described in Section 2.3. The pperm instruction specifies the permuted ordering of bits, like the subword permute instruction in PA-RISC. The cross instruction [10, 46] permutes bits using two stages of the full Beneš network; one of the two input operands is used to hold the control bits for those two stages while the other input operand holds the data to be permuted. Thus lg (n) cross instructions are required for any arbitrary permutation of n bits. The omflip instruction [10, 47] permutes bits using two stages of omega and flip networks, which are isomorphic to butterfly and inverse butterfly networks, respectively. The advantage of omega and flip networks is that the stages are all identical. Similar to cross, the control bits for the two stages are supplied using an input operand and lg(n) omflip instructions are needed for arbitrary n-bit permutations. The swperm and sieve instructions [30, 48] can produce arbitrary bit permutations with repetitions. The swperm instruction is similar to the subword permutation instructions in commodity ISAs mentioned above except that it operates on 4-bit subwords rather than bytes. The sieve instruction permutes bits within each 4-bit subword. Using swperm and sieve, an arbitrary permutation of 64 1-bit subwords can be performed with 11 instructions and an arbitrary permutation of 32 2-bit subwords can be performed with five instructions.

In [7], Lee et al. first proposed the bfly and ibfly instructions using full butterfly and inverse butterfly datapaths, then discussed how to implement these instructions which require more than two n-bit operands in an application-specific instruction processor (ASIP) [8] and in a general-purpose processor with the restriction of two operands and one result per instruction [9]. In [5, 6], we discussed our early work on pex and pdep.

Recently, in [49], bit permutation instructions (bpi) for cryptography and general purpose applications are proposed. The cryptographic bpi is a 2-input operand instruction that routes the data input through a partial Beneš network that is configured by performing a bit expansion on the control input. The bpi is implemented using a full Beneš network configured using a special $n/2\times(2\times\lg(n)-1)$-bit register R. The R register is loaded n bits at a time from general purpose registers. This scheme for general permutations is similar to our proposed scheme, first described in [7, 10]. The difference is that we split the Beneš network into butterfly and inverse butterfly subnetworks. Our approach has a number of advantages. First, each network is faster than an ALU and thus is single cycle while the concatenation of the two networks may have greater latency than an ALU and thus the general purpose bpi might take two cycles. Additionally, there are a great many simple permutations that require only one of the bfly or ibfly subnetworks. Use of bpi for these permutations still requires routing the data bits through both subnetworks. One disadvantage of our approach is that two instructions must be issued for general permutations. However, given superscalar resources, this will likely have negligible effect on bit permutation workloads.

9 Conclusion

In this paper we showed that bit gather (pex), bit scatter (pdep) and bit permutation instructions (bfly and ibfly) can all be supported by a single functional unit. We showed that pex maps to the inverse butterfly datapath and pdep maps to the butterfly datapath and not vice versa. We showed how to implement various combinations of static, loop-invariant and variable versions of pex and pdep in functional units of increasing complexity.

We suggest that the simple functional unit in Fig. 16 suffices: it supports the static versions of pex and pdep, and also the bfly and ibfly permutation instructions. Since the applications studied showed that most of the time only static pex and pdep are needed, this simple functional unit

should suffice. This functional unit (Fig. 16) is smaller and faster than an ALU. If it is desired to support also dynamic and loop-invariant versions of pex.v and pdep.v, we recommend the full functional unit (Fig. 18), which is slightly more than twice the size of an ALU with only 16% longer latency. This allows a loop-invariant pex or pdep operation to calculate the control bits for the inverse butterfly or butterfly datapath, respectively, once, and load these into the Application Registers associated with this functional unit. Thereafter, the fast single-cycle static versions of the pex or pdep instructions can be used. Furthermore, fully dynamic pex.v or pdep.v operations can be accomplished in one instruction each, albeit each of these takes three pipelined cycles of latency rather than one cycle.

In order to configure the datapath for pdep and pex, we detail Algorithm 1 which shows how to decode the n-bit mask input in the pdep or pex instruction into the $n/2 \times \lg(n)$ butterfly or inverse butterfly control bits. Algorithm 1's long runtime in software motivated the design of a hardware decoder to support high performance dynamic pdep.v and pex.v. We describe optimized hardware implementations of the parallel prefix population count circuit and the LROTC circuit, the only two key components in the hardware decoder for generating datapath control bits.

We show how pex, pdep and bit permutation instructions improve the performance of a number of applications ranging from bioinformatics to compression to steganography. Benchmark results indicate that a processor enhanced with parallel deposit and parallel extract achieves a 10.04× maximum speedup, 2.29× on average, over a basic RISC architecture. These performance results are very promising, but future work should also study the performance of whole applications.

Overall, we have brought the acceleration of advanced bit manipulation operations out of the realm of "programming tricks." The most useful bit manipulation operations in the applications we examined are static versions of pex, pdep, bfly and ibfly, all of which can be executed in a single cycle. We have shown that these operations are needed by many applications and that direct support for them can be implemented in a commodity microprocessor at a reasonable cost.

Future work includes redesigning existing algorithms to make use of these advanced bit manipulation instructions, and designing new algorithms using them to advantage. Adding compiler support for advanced bit manipulation operations is also a fruitful area for future work; currently we rely upon compiler intrinsics. Improvements in circuit design can also be studied. Other bit manipulation operations such as bit matrix multiplication also need to be examined for usefulness in applications and cost-effective implementation in a commodity microprocessor. We hope to have laid the foundation for the ISA and implementation of powerful and cost-effective advanced bit manipulations in microprocessors.

Acknowledgements This work was supported in part by the Department of Defense and a research gift from Intel Corporation. Hilewitz is also supported by a Hertz Foundation Graduate Fellowship and an NSF Graduate Fellowship. The authors would also like to thank Roger Golliver of Intel Corporation for suggesting some applications that might benefit from bit manipulation instructions.

References

1. Warren Jr., S. (2002). *Hacker's delight*. Boston: Addison-Wesley Professional (revised online: http://www.hackersdelight.org/revisions.pdf).
2. Schwartz, S., Kent, W. J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R. C., et al. (2003). Human–mouse alignments with BLASTZ. *Genome Research*, *13*(1), 103–107, January.
3. Beeler, M., Gosper, B., & Schroeppel, R. (1972). "Hackmem," *Massachusetts Institute of technology-Artificial Intelligence Laboratory Memo 239*, available online: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-239.pdf.
4. Cray Corporation (2003). *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual*, version 1.2, October, available online: http://docs.cray.com/books/S-2314-51/S-2314-51-manual.pdf.
5. Lee, R. B., & Hilewitz, Y. (2005). Fast pattern matching with parallel extract instructions. *Princeton University Department of Electrical Engineering Technical Report CE-L2005-002*, February.
6. Hilewitz, Y., & Lee, R. B. (2006). Fast bit compression and expansion with parallel extract and parallel deposit instructions. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 65–72, September 11–13.
7. Lee, R. B., Shi, Z., & Yang, X. (2002). How a processor can permute n bits in O(1) cycles. *Proceedings of Hot Chips 14—A symposium on High Performance Chips*, August.
8. Shi, Z., Yang, X., & Lee, R. B. (2003). Arbitrary bit permutations in one or two cycles. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 237–247, June.
9. Lee, R. B., Yang, X., & Shi, Z. J. (2005). Single-cycle bit permutations with MOMR execution. *Journal of Computer Science and Technology*, *20*(5), 577–585 (September).
10. Lee, R. B., Shi, Z., & Yang, X. (2001). Efficient permutation instructions for fast software cryptography. *IEEE Micro*, *21*(6), 56–69 (December).
11. Shi, Z., & Lee, R. B. (2000). Bit permutation instructions for accelerating software cryptography. *Proceedings of the IEEE International Conf. on Application-Specific Systems, Architectures and Processors*, 138–148, July.
12. Lee, R. (1989). Precision architecture. *IEEE Computer*, *22*(1), 78–91 (Jan).
13. Lee, R., Mahon, M., & Morris, D. (1992). Pathlength reduction features in the PA-RISC architecture. *Proceedings of IEEE Compcon*, 129–135. San Francisco, California, Feb 24–28.
14. Intel Corporation (2002). *Intel® Itanium® Architecture Software Developer's Manual*, *1–3*, rev. 2.1, Oct.
15. Hilewitz, Y., Shi, Z. J., & Lee, R. B. (2004). Comparing fast implementations of bit permutation instructions. *Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers*, Nov.

16. Beneš, V. E. (1964). Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, *43*(4), 1641–1656 (July).
17. Lee, R. B., Rivest, R. L., Robshaw, M. J. B., Shi, Z. J., & Yin, Y. L. (2004). On permutation operations in Cipher design. *Proceedings of the International Conference on Information Technology (ITCC)*, *2*, 569–577 (April).
18. Intel Corporation (2007). *IA-32 Intel® Architecture Software Developer's Manual*, *1–2*.
19. Sun Microsystems (2002). *The VIS™ Instruction Set*, Version 1.0, June.
20. The Mathworks, Inc., *Image Processing Toolbox User's Guide*: http://www.mathworks.com/access/helpdesk/help/toolbox/images/images.html.
21. Franz, E., Jerichow, A., Möller, S., Pfitzmann, A., & Stierand, I. (1996). Computer based steganography. *Information Hiding, Springer Lecture Notes in Computer Science*, *1174*, 7–21.
22. "Uuencode," Wikipedia: The Free Encyclopedia, http://en.wikipedia.org/wiki/Uuencode.
23. Cray Corporation, Man Page Collection: Bioinformatics Library Procedures, 2004, available online: http://www.cray.com/craydoc/manuals/S-2397-21/S-2397-21.pdf.
24. National Center for Biotechnology Information, Translating Basic Local Alignment Search Tool (BLASTX), available online: http://www.ncbi.nlm.nih.gov/blast/.
25. Fiskiran, A. M., & Lee, R. B. (2005). Fast parallel table lookups to accelerate symmetric-key cryptography. *Proceedings of the International Conference on Information Technology Coding and Computing (ITCC)*, Embedded Cryptographic Systems Track, 526–531, April.
26. Fiskiran, A. M., & Lee, R. B. (2005). On-chip lookup tables for fast symmetric-key encryption. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 356–363, July.
27. Josephson, W., Lee, R. B., & Li, K. (2007). ISA support for fingerprinting and erasure codes. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July.
28. Scholer, F., Williams, H., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 222–229.
29. Jun, B., & Kocher, P. (1999). The Intel random number generator. Technical Report, Cryptography Research Inc.
30. McGregor, J. P., & Lee, R. B. (2001). Architectural enhancements for fast subword permutations with repetitions in cryptographic applications. *Proceedings of the International Conference on Computer Design (ICCD 2001)*, 453–461, September.
31. Moldovyan, N. A., Moldovyanu, P. A., & Summerville, D. H. (2007). On software implementation of fast DDP-based Ciphers. *International Journal of Network Security*, *4*(1), 81–89 (January).
32. NIST, Cryptographic Hash Function Competition, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.
33. Burger, D., & Austin, T. (1997). The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*.
34. Swartzlander, E. E., Jr. (2004). A review of large parallel counter designs. *IEEE Symposium on VLSI*, 89–98, February.
35. Han, T., & Carlson, D. A. (1987). Fast area-efficient VLSI adders. *Proceedings of the 8th Symposium on Computer Arithmetic*, 49–55, May.
36. Taiwan Semiconductor Manufacturing Corporation (2003). *TCBN90G: TSMC 90 nm Core Library Databook*, Oct.
37. Broukhis, L. A. "BESM-6 Instruction Set," available online: http://www.mailcom.com/besm6/instset.shtml.
38. Hilewitz, Y., & Lee, R. B. (2007). Achieving very fast bit matrix multiplication in commodity microprocessors. *Princeton University Department of Electrical Engineering Technical Report CE-L2007-4*, July.
39. IBM Corporation (2003). *PowerPC Microprocessor Family: AltiVec™ Technology Programming Environments Manual*, Version 2.0, July.
40. Lee, R. (1996). Subword parallelism with MAX-2. *IEEE Micro*, *16*(4), 51–59 (August).
41. Lee, R. (1997). Multimedia extensions for general-purpose processors. *Proceedings of the IEEE Signal Processing Systems Design and Implementation*, 9–23, November.
42. Lee, R. B. (1999). Efficiency of MicroSIMD architectures and index-mapped data for media processors. *Proceedings of Media Processors 1999 IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, 34–46, January.
43. Lee R. B. (2000). Subword permutation instructions for two-dimensional multimedia processing in MicroSIMD architectures. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2000)*, 3–14, July.
44. Hanson, C. (1996). MicroUnity's mediaprocessor architecture. *IEEE Micro*, *16*(4), 34–41 (August).
45. Burke, J., McDonald, J., & Austin, T. (2000). Architectural support for fast symmetric-key cryptography. *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), November.
46. Yang, X., Vachharajani, M., & Lee, R. B. (2000). Fast subword permutation instructions based on butterfly networks. *Proceedings of Media Processors IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, 80–86, January.
47. Yang, X., & Lee, R. B. (2000). Fast subword permutation instructions using omega and flip network stages. *Proceedings of the International Conference on Computer Design (ICCD 2000)*, 15–22, September.
48. McGregor, J. P., & Lee, R. B. (2003). Architectural techniques for accelerating subword permutations with repetitions. *IEEE Transactions on Very Large Scale Integration Systems*, *11*(3), 325–335 (June).
49. Moldovyan, A. A., Moldovyan, N. A., & Moldovyanu, P. A. (2007). Architecture types of the bit permutation instruction for general purpose processors. *Springer LNG&G*, *14*, 147–159.

Yedidya Hilewitz is currently pursuing a PhD degree in electrical engineering at Princeton University, Princeton, NJ. He is exploring the architecture and implementation of supercomputer-level bit manipulation instructions in commodity microprocessors. He is also interested in secure computer architectures.

Ruby B. Lee is the Forrest G. Hamrick Professor of Engineering and Professor of Electrical Engineering at Princeton University, with an affiliated appointment in the Computer Science department. She is the director of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Her current research is in designing security and new media support into core computer architecture, embedded systems and global networked systems. She is a Fellow of the Association for Computing Machinery (ACM) and a Fellow of the Institute of Electrical and Electronic Engineers (IEEE). She is Associate Editor-in-Chief of *IEEE Micro* and Editorial Board member of *IEEE Security and Privacy*. Prior to joining the Princeton faculty in 1998, Dr. Lee served as chief architect at Hewlett-Packard, responsible at different times for processor architecture, multimedia architecture and security architecture for e-commerce and extended enterprises. She was a key architect in the definition and evolution of the PA-RISC architecture used in HP servers and workstations, and also led the first CMOS PA-RISC single-chip microprocessor design. As chief architect for HP's multimedia architecture team, Dr. Lee led an inter-disciplinary team focused on architecture to facilitate pervasive multimedia information processing using general-purpose computers. This resulted in the first desktop computer family with integrated, software-based, high fidelity, real-time multimedia. Dr. Lee also co-led a multimedia architecture team for IA-64. Concurrent with full-time employment at HP, Dr. Lee also served as Consulting Professor of Electrical Engineering at Stanford University. She has a Ph.D. in Electrical Engineering and a M.S. in Computer Science, both from Stanford University, and an A.B. with distinction from Cornell University, where she was a College Scholar. She is an elected member of Phi Beta Kappa and Alpha Lambda Delta. She has been granted over 120 US and international patents, with several patents pending.