A New Basis for Shifters in General-Purpose Processors for Existing and Advanced Bit Manipulations

Yedidya Hilewitz, *Member, IEEE*, and Ruby B. Lee, *Fellow, IEEE*

Abstract—This paper describes a new basis for the implementation of the shifter functional unit in microprocessors that can implement new advanced bit manipulations as well as standard shifter operations. Our design is based on the inverse butterfly and butterfly datapath circuits, rather than the barrel shifter or log-shifter designs currently used. We show how this new shifter can implement the standard shift and rotate operations, as well as more advanced extract, deposit and mix operations found in some processors. Furthermore, it can perform important new classes of even more advanced bit manipulation instructions like arbitrary bit permutations, bit gather (or parallel extract) and bit scatter (or parallel deposit) instructions. Thus, our new functional unit performs the functionality of three functional units – the basic shifter, the multimedia-mix unit and the advanced bit manipulation functional unit, while having a latency only slightly longer than that of the log-shifter. For performing only the existing functions of a shifter, it has significantly smaller area.

Index Terms—Shifter, rotation, shift, permutation, butterfly, inverse butterfly, bit manipulation, microprocessor, instruction set architecture, processor architecture, circuit design, extract, deposit, mix, multimedia, arithmetic, parallel operations.

1 Introduction

BIT manipulation operations for microprocessors have not been studied as thoroughly as integer and floating point arithmetic. Since a microprocessor is optimized around the processing of words, it is not surprising that bit-level operations are typically not as well supported by current word-oriented microprocessors. Simple *bit-parallel* operations such as AND, OR, XOR, and NOT are typically supported as the "logical" operations of the Arithmetic-Logic Unit (ALU), the most fundamental functional unit of a microprocessor. However, only very simple *non bit-parallel* operations are supported like shift and rotate operations, in which all bits of an operand move by the same amount. These non bit-parallel operations are typically supported by a separate *shifter* functional unit. In this paper, we propose a new basis for the shifter functional unit that can perform these basic shift and rotate operations together with more advanced non bit-parallel operations described below.

A few processor Instruction Set Architectures (ISAs) also have more advanced bit operations implemented in an enhanced shifter or another functional unit. These include subword extract and deposit operations (e.g., pextrw and pinsrw in IA-32 [1]), field extract and deposit operations (e.g., extr and dep in PA-RISC [2] or IA-64 [3]), or rotate and mask operations (e.g., rldimi in PowerPC [4]). These can be viewed as variants of the basic shift or rotate operation operations, with certain bits masked out and set to zeros, or sign bits replicated, or bits from a second operand merged into the result. Additionally, some instruction sets have multimedia permute operations that rearrange the subwords packed into one or more registers (e.g., mix in PA-RISC 2.0 [5], [6] and IA-64 [3]).

In addition, there are many emerging applications, such as cryptography, imaging and bioinformatics, where even more advanced bit manipulation operations are needed. While these can be built from the simpler logical and shift operations or performed using programming tricks (cf. *Hacker's Delight* [7]), the applications using these advanced bit manipulation operations are significantly sped up if the processor can support more powerful bit manipulation instructions. Such operations include arbitrary bit permutations, bit gather operations (performing multiple bit-field extract operations in parallel), and bit scatter operations (performing multiple bit-field deposit operations in parallel). In earlier work, we have shown that these advanced bit manipulation operations can be implemented in a single new permutation functional unit [8], [9], utilizing two simple datapaths – an inverse butterfly circuit and a butterfly circuit.

In this paper, we propose using this new permutation functional unit as a new basis for shifters, rather than simply adding it to a processor core, or enhancing the current shifter to also support the above advanced bit manipulation functions. We propose *replacing* two existing functional units (the shifter and the multimedia-mix functional units in the IA-32 and IA-64 processors) with the new permutation functional unit and performing all

- *Y. Hilewitz is with Intel Corporation, 77 Reed Rd., Hudson, MA 01749. Email: yedidya.hilewitz@intel.com. This work was done while Hilewitz was a Ph.D. student at Princeton University.*
- *R.B. Lee is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA. Email: rblee@princeton.edu.*

Manuscript received July 30, 2007. Revisions received January 8, 2008.

the operations previously done on these existing shifters as well as the advanced bit permutation, bit gather and bit scatter operations on the same datapath [10]. Our new design represents an evolution of the shifter from the classical barrel shifter and log shifter to a new Shift-Permute functional unit which can perform both basic shift operations and sophisticated bit manipulations.

The contributions of this paper are as follows:

First, we propose a new basis for the design of shifters, based on the inverse butterfly (or butterfly) circuit. This new basis implements a much more powerful set of shift and advanced bit manipulation instructions, previously requiring two existing functional units and a new permutation functional unit.

Second, we describe a recursive algorithm for determining the control bits for existing shift instructions like rotate, shift, extract, deposit and mix operations on the inverse butterfly (or butterfly) datapath circuits.

Third, we describe an algorithm to obtain the control bits for new advanced bit manipulation instructions like the bit gather (pex) and bit scatter (pdep) instructions.

Fourth, we demonstrate the implementation of this powerful shift-permute unit and compare its complexity to that of an ordinary log-shifter functional unit showing only a minimal increase in the latency (1.18×) and a reduction in area (0.69×) for the basic shift/rotate circuit. When advanced bit manipulation instructions are added, the increase in latency is still small (1.18×-1.20×) with a moderate increase in area (1.29×-1.87×).

In section 2, we describe the basic and advanced bit manipulation instructions. In section 3, we describe the new functional unit and show how to obtain the control bits for the inverse butterfly (or butterfly) datapath for both existing shifter instructions as well as new advanced bit manipulation instructions. In section 4, we compare the implementation to that of the barrel shifter and the log shifter. Section 5 concludes the paper.

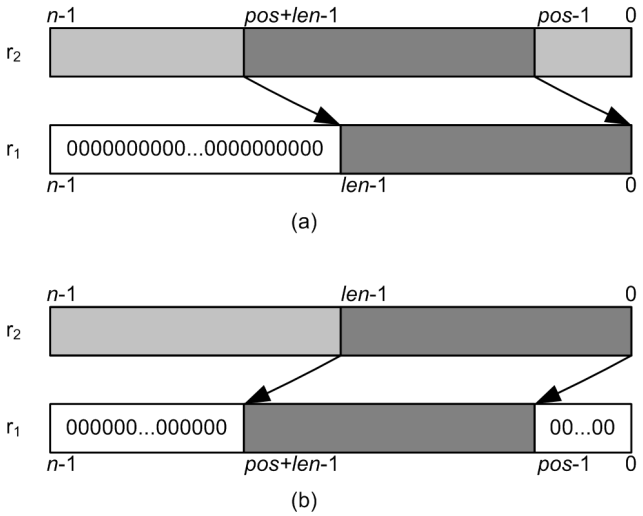

Fig. 1. (a) extr.u $r_1 = r_2$, pos, len; (b) dep.z $r_1 = r_2$, pos, len

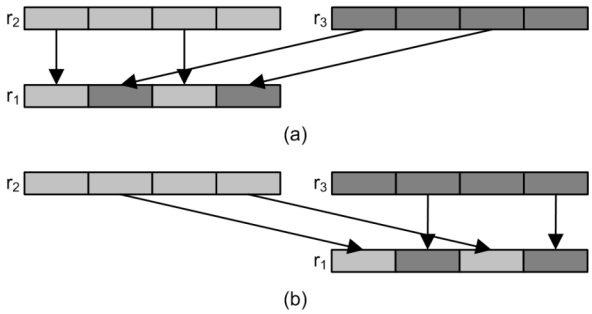

Fig. 2. (a) mix.left $r_1 = r_2, r_3$; (b) mix.right $r_1 = r_2, r_3$;

2 Basic and Advanced Bit Manipulation Operations

In this section, we define the set of basic shifter instructions and the set of advanced bit manipulation instructions considered in the rest of this paper. These are also summarized in Table 1.

2.1 Basic Shifter Instructions

The basic bit manipulation instructions consist of two groups. The first group consists of the *shift* and *rotate* instructions supported in essentially all microprocessors. These instructions include right or left shifts (with zero or sign propagation), and right or left rotates. While a few microprocessors support only shifts but not rotates, we will consider rotate as a basic supported operation in this paper.

The second group of instructions exists in a few Instruction Set Architectures (ISAs) such as PA-RISC [2], [5], [6], and IA-64 [3]. This group includes extract, deposit and mix instructions.

The *extract* operation selects a single field of bits of arbitrary length from any arbitrary position in the source register and right justifies that field in the result (Fig. 1(a)). Extract is equivalent to a shift right and mask operation. Extract has both unsigned and signed variants. In the latter, the sign bit of the extracted field is propagated to the most-significant bit of the destination register.

The *deposit* operation takes a single right justified field of arbitrary length from the source register and deposits it at any arbitrary position in the destination register (Fig. 1(b)). Deposit is equivalent to a left shift and mask operation. There are two variants of deposit: the remaining bits can be zeroed out (as shown in Fig. 1(b)), or they are supplied from a second register, in which case a masked merge operation is required.

The mix operation is a subword permutation operation, initially designed to accelerate multimedia applications operating on subwords of 8, 16 or 32 bits, packed into a word [5], [6], [3]. Mix_left selects the left subwords from each pair of subwords, alternating between the two source registers r_2 and r_3 (Fig. 2(a)). Mix_right does the same on the right subwords of the two source registers (Fig. 2(b)). The mix instruction was first introduced in PA-RISC for multimedia acceleration [6], and also appears in IA-64 (Itanium) [3], [11], where it is implemented in a separate multimedia functional unit. No ISA currently supports mix for subwords smaller than a byte, although this is very useful, e.g., for bit matrix transposition and

fast parallel sorting [12]. In our proposed new functional unit, mix for bits – and for all subword sizes that are powers of 2 – are supported. This includes 12 mix operations: mix_left and mix_right for each of 6 subword sizes of 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, for a 64-bit processor. Note that mix for 2^0, 2^1 and 2^2 are *new* operations for current processors, although we refer to all mix instructions as *basic or existing* operations in the rest of this paper.

2.2 Advanced Bit Manipulation Instructions

New classes of bit manipulation instructions have been proposed for accelerating various applications ranging from cryptography to bioinformatics. These include instructions for performing bit permutations, and for performing bit gather and bit scatter operations.

Bit Permutations. For bit permutations, several instructions have been proposed [13], [14], [15], [16], [17], [18], [19], most notably the group (grp) permutation instruction [13], [14], and the butterfly (bfly) and inverse butterfly (ibfly) permutation instructions [16], [17], [18]. An arbitrary permutation of the n bits within a register can be performed by a sequence of at most lg(n) grp instructions, or by a sequence of at most 2 instructions using bfly and ibfly instructions. Since the latter (bfly and ibfly) achieve arbitrary n-bit permutations in O(1) cycles rather than O(lg(n)) cycles, we focus on them in this paper.

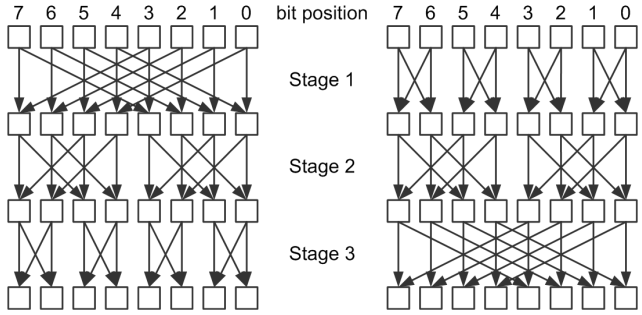

Fig. 3. 8-bit Butterfly (left) and Inverse Butterfly (right) Circuits

The *butterfly (bfly)* and *inverse butterfly (ibfly)* instructions route their inputs through butterfly and inverse butterfly circuits, respectively. The concatenation of these two circuits forms a Benes circuit, a general permutation network [20]. Thus a single execution of bfly followed by ibfly (or vice versa) can achieve any of the n! permutations of n bits in at most 2 cycles [16], [17], [18].

The structure of the circuits is shown in Fig. 3, which

TABLE 1
INSTRUCTIONS SUPPORTED BY NEW SHIFT-PERMUTE FUNCTIONAL UNIT

Instruction	Description
rotr $r_1 = r_2$, shamt rotr $r_1 = r_2, r_3$	Right rotate of r_2. The rotate amount is either an immediate in the opcode (shamt), or specified using a second source register r_3.
rotl $r_1 = r_2$, shamt rotl $r_1 = r_2, r_3$	Left rotate of r_2.
shr $r_1 = r_2$, shamt shr $r_1 = r_2, r_3$ shr.u $r_1 = r_2$, shamt shr.u $r_1 = r_2, r_3$	Right shift of r_2 with vacated positions on the left filled with the sign bit (most significant bit of r_2) or zero-filled. The shift amount is either an immediate in the opcode or specified using a second source register r_3.
shl $r_1 = r_2$, shamt shl $r_1 = r_2, r_3$	Left shift of r_2 with vacated positions on the right zero-filled.
extr $r_1 = r_2$, pos, len extr.u $r_1 = r_2$, pos, len	Extraction and right justification of single field from r_2 of length len from position pos. The high order bits are filled with the sign bit of the extracted field or zero-filled.
dep.z $r_1 = r_2$, pos, len dep $r_1 = r_2, r_3$, pos, len	Deposit at position pos of single right justified field from r_2 of length len. Remaining bits are zero-filled or merged from second source register r_3.
mix {r,l} {0,1,2,3,4,5} $r_1 = r_2, r_3$	Select right or left subword from a pair of subwords, alternating between source registers r_2 and r_3. Subword sizes are 2^i bits, for i = 0,1,2,…,5, for a 64-bit processor.
bfly $r_1 = r_2$, *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$	Perform *Butterfly permutation* of data bits in r_2
ibfly $r_1 = r_2$, *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$	Perform *Inverse Butterfly permutation* of data bits in r_2
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 pre-decoded mask r_3 are extracted, compressed and right-aligned in the result r_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 at bit positions marked with a "1" in the statically-decoded mask r_3
mov *ar.x* = r_2, r_3	*Move values from GRs to ARs*, to set controls (calculated by software) for pex, pdep, bfly or ibfly
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.
pdep.v $r_1 = r_2, r_3$	*Parallel deposit, variable*: Right-aligned data bits in r_2 are deposited, in order, in bit positions marked with a "1" in the dynamically-decoded mask r_3.
setib *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$ = r_3	*Set inverse butterfly circuit controls in associated ARs*, using hardware decoder to translate the mask r_3 to inverse butterfly controls.
setb *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$ = r_3	*Set butterfly circuit controls in associated ARs*, using hardware decoder to translate the mask r_3 to butterfly controls.

shows 8-bit circuits, requiring 3 stages. The n-bit circuits consist of lg(n) stages, each stage composed of $n/2$ 2-input switches. Each of these circuits takes at most one cycle, since they are less complicated than an ALU of the same width. (We normalize a processor cycle to the latency of an ALU.) Furthermore, each switch is composed of two 2:1 multiplexers, totaling $n \times \lg(n)$ multiplexers for each circuit, which results in small circuit area.

In the ith stage (i starting from 1), the paired bits are $n/2^i$ positions apart for the butterfly network and 2^{i-1} positions apart for the inverse butterfly network. A switch either passes through or swaps its inputs based on the value of a control bit. Thus, the operation requires $n/2 \times \lg(n)$ control bits. For $n = 64$, four 64-bit registers are required to hold the 64 data bits and the 32×6 control bits.

Our preferred implementation for bfly and ibfly instructions, in an architecture that has only 2 source operands per instruction, utilizes 3 Application Registers ($ar.b_1$, $ar.b_2$, $ar.b_3$) associated with the functional unit to supply the control bits during the execution of these instructions. Application registers are already available in some ISAs, e.g., IA-64 [3]. The control bits are determined either statically by the compiler, or dynamically by software. The need for additional application registers can be a disadvantage for processor ISAs that do not already have these. Alternative ISA solutions are discussed in [17], [18]. Other permutation primitives like grp discussed below do not need application registers.

The group (grp) instruction [13], [14], 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. 4). Arbitrary bit permutations can be accomplished by a sequence of at most lg(n) of these grp instructions. Its advantage over the faster sequence of 2 instructions (bfly followed by ibfly) for achieving arbitrary n-bit permutations is that it does not need additional (control or application) registers to configure its datapath. However, it is costly to implement, in terms of both area and latency. Instead, we implement "half" of a grp instruction efficiently in a parallel extract (pex) instruction.

Bit Gather and Bit Scatter. The *parallel extract (pex)* and *parallel deposit (pdep)* instructions [8], [9] can be viewed as generalizations of the extract and deposit instructions. Parallel extract performs a bit gather operation; it extracts and compacts bits from one source register from positions selected by "1"s in a second source register (see Fig. 5(a)). The rest of the bits in the result register are cleared to "0"s. Thus the parallel extract operation can also be thought of as the right half of the grp operation (grp_right) [21].

Parallel deposit performs a bit scatter operation; it deposits bits from one source register to positions selected by "1"s in a second source register (see Fig. 5(b)). The remaining bits in the result register are cleared to "0"s.

3 New Shift-Permute Functional Unit

Our new shift-permute functional unit consists of an inverse butterfly datapath (or a butterfly datapath) enhanced with an extra multiplexer stage. In section 3.1, we show that this functional unit can perform any of the basic shifter instructions listed in section 2.1 (and Table 1). The hard part is determining how the controls of the lg(n) stages of the circuit should be set, and we give definitive algorithms for this in sub-sections 3.1.1 and 3.1.2. Then in section 3.2, we show how the advanced bit manipulation instructions defined in section 2.2 (and Table 1) are performed on these datapaths. Again, since determining the control bits is non-obvious, we illustrate this with a specific algorithm for determining the control bits to implement parallel extract (pex) on an inverse butterfly datapath in section 3.2.1. In section 3.3, we discuss the alternative architectures of the entire shift-permute unit, and tradeoffs in terms of functionality and cost.

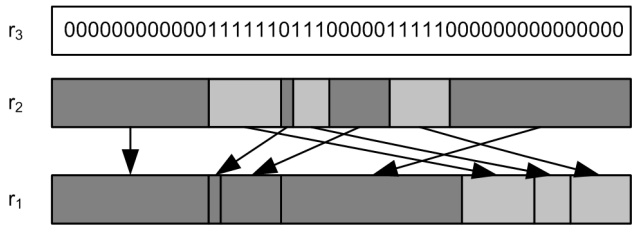

Fig. 4. grp $r_1 = r_2, r_3$

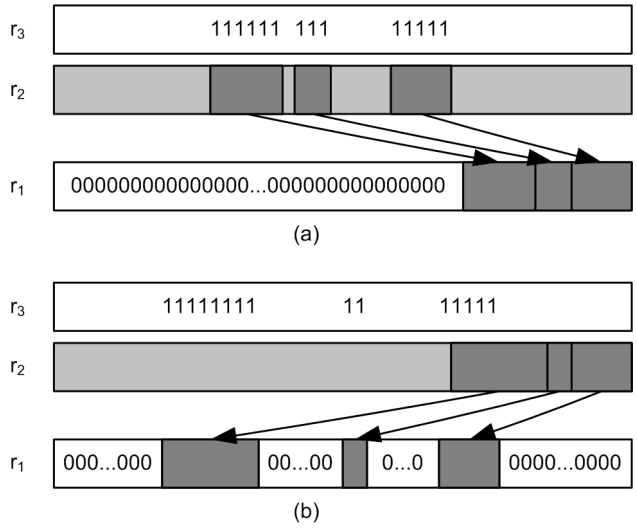

Fig. 5. (a) pex $r_1 = r_2, r_3$; (b) pdep $r_1 = r_2, r_3$

3.1 Basic Shifter Operations on the Inverse Butterfly Datapath

The key conceptual insight comes from recognizing that the set of basic shifter and mix operations in the top part of Table 1 is based on minor variations on a rotation operation, and that any rotation operation can be achieved on an inverse butterfly circuit (or on a butterfly circuit.)

Theorem 1. *An inverse butterfly circuit can achieve any rotation of its input.*

A proof of this can be found in [22], where rotations are called cyclic shifts.

Corollary 1.1. *An enhanced inverse butterfly circuit can perform on its input:*

a. Right and left shifts
b. Extract operations
c. Deposit operations
d. Mix operations

Proof. This follows from Theorem 1, with these operations modeled as a rotate with additional logic handling zeroing or sign extension from an arbitrary position, or merging bits from the second source operand (for deposit). Mix is modeled as a rotate of one operand by the subword size and then a merge of subwords alternating between the two operands.

As the inverse butterfly circuit only performs permutations without zeroing and without replication, the circuit must be enhanced with an extra 2:1 multiplexer stage at the end that either selects the rotated bits as is or other bits which are computed as either zero, or the sign bit (replicated), or the bits of the second source operand, depending on the operation.

Corollary 1.2. Theorem 1 and Corollary 1.1 are true for the butterfly network as well.

Proof. The butterfly and inverse butterfly networks exhibit a reverse symmetry of their stages from input to output. Thus a rotation on the inverse butterfly network is equivalent to a rotation in the opposite direction on the butterfly network when the flow through the network is reversed (see Fig. 6). Hence, a butterfly circuit can also achieve any rotation of its inputs. As in Corollary 1.1, a butterfly network enhanced with an extra multiplexer stage at the end is needed to handle zeroing or sign extension, or merging bits from the second source operand.

Next, we show how control bits are obtained for rotations on an inverse butterfly circuit in section 3.1.1, then for the other operations in section 3.1.2.

3.1.1 Determining the Control Bits for Rotations

To achieve a right (or left) rotation by s positions, for s = 0, 1, 2 … n-1, using the n-bit wide inverse butterfly circuit with lg(n) stages, the input must be right (or left) rotated by s mod 2^j within each 2^j-bit wide inverse butterfly circuit at each stage j. This is because from stage j+1 on, the inverse butterfly circuit can only move bits at granularities larger than 2^j positions (so the finer movements must have already been performed in the prior stages). We first give a conceptual explanation of this, then a formal constructive proof to obtain the actual control bits for a rotation.

An n-bit inverse butterfly circuit can be viewed as two (lg(n)-1)-stage circuits followed by a stage that swaps or passes through paired bits that are $n/2$ positions apart (see Fig. 3). To right_rotate the input {in_{n-1} … in_0} by s positions, the two (lg(n)-1)-stage circuits must have right_rotated their half inputs by $s' = s$ mod $n/2$ and the input to stage lg(n) must be of the form:

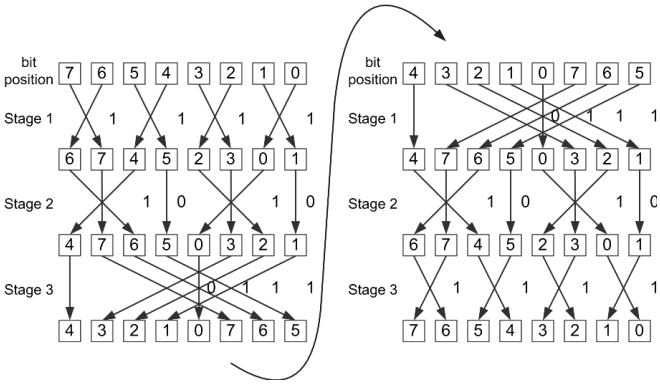

Fig. 6. Left rotate by three on inverse butterfly is equivalent to right rotate by three on butterfly

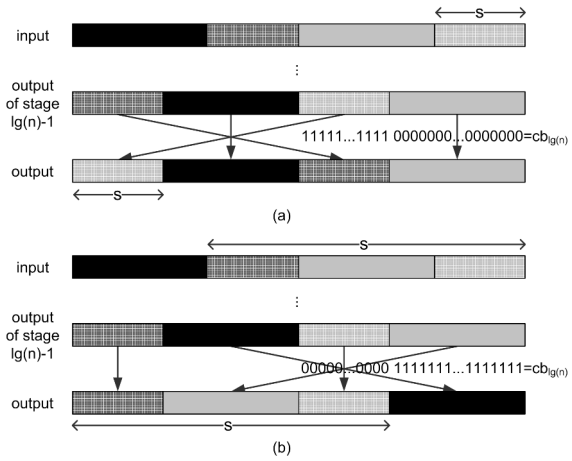

Fig. 7. (a) rotation by $s<n/2$ and (b) by $s \geq n/2$

$$\{in_{n/2+s'-1} \ldots in_{n/2}\, in_{n-1} \ldots in_{n/2+s'} \,||\, in_{s'-1} \ldots in_0\, in_{n/2-1} \ldots in_{s'}\} \quad (1)$$

We show this is true for s less than or greater than $n/2$.

When the rotation amount s is less than $n/2$ then the bits that *wrapped around* in the (lg(n)-1)-stage circuits (cross-hatched) must be swapped in the final stage to yield the input right_rotated by s (Fig. 7(a)):

$$\{in_{s'-1} \ldots in_0\, in_{n-1} \ldots in_{n/2+s'} \,||\, in_{n/2+s'-1} \ldots in_{n/2}\, in_{n/2-1} \ldots in_{s'}\}. \quad (2)$$

When the rotation amount is greater than or equal to $n/2$ then the bits that *do not wrap* in the (lg(n)-1)-stage circuits (solid) must be swapped in the final stage to yield the input right_rotated by s (Fig. 7(b)):

$$\{in_{n/2+s'-1} \ldots in_{n/2}\, in_{n/2-1} \ldots in_{s'} \,||\, in_{s'-1} \ldots in_0\, in_{n-1} \ldots in_{n/2+s'}\}. \quad (3)$$

For example, consider the 8-bit inverse butterfly network with right rotation amount s = 5, depicted in Fig. 8. As s=5 is greater than $n/2$ = 4, the bits that did not wrap in stage 2 are swapped in stage 3 to yield the final result.

As the rotation amount through stage 2, s mod 2^2 = 5 mod 4 = 1, is less than $n/4$ = 2, the bits that did wrap in stage 1 are swapped in stage 2 to yield the input to stage 3.

As the rotation amount through stage 1, $s \bmod 2^1 = 5 \bmod 2 = 1$, is equal to than $n/8 = 1$, the bits that did not wrap in the input, i.e., all the bits, are swapped in stage 1 to yield the input to stage 2.

We can mathematically derive recursive equations for the control bits, cb_j, $j = 1, 2, \ldots \lg(n)$, for achieving rotations on an inverse butterfly datapath. These equations yield a compact circuit (shown in Fig. 9) for the rotation control bit generator.

From (1)-(3) and Fig. 7, we observe that the pattern for the control bits for the final stage, which we call $cb_{\lg(n)}$, for a rotate of s bits, is:

$$cb_{\lg(n)} = \begin{cases} 1^s \,\|\, 0^{n/2-s}, & s < n/2 \\ 0^{s-n/2} \,\|\, 1^{n/2-(s-n/2)}, & s \geq n/2 \end{cases} \quad (4)$$

where a^k is a string of k "a"s, "1" means "swap" and "0" means "pass through." Note that $s = s \bmod n/2$ when $s < n/2$ and $s\text{-}n/2 = s \bmod n/2$ when $s \geq n/2$:

$$cb_{\lg(n)} = \begin{cases} 1^{s \bmod n/2} \,\|\, 0^{n/2-(s \bmod n/2)}, & s \bmod n < n/2 \\ 0^{s \bmod n/2} \,\|\, 1^{n/2-(s \bmod n/2)}, & s \bmod n \geq n/2 \end{cases}$$

$$cb_{\lg(n)} = \begin{cases} 1^{s \bmod n/2} \,\|\, 0^{n/2-(s \bmod n/2)}, & s \bmod n < n/2 \\ \sim \left(1^{s \bmod n/2} \,\|\, 0^{n/2-(s \bmod n/2)}\right), & s \bmod n \geq n/2 \end{cases} \quad (5)$$

where ~ indicates negation.

Furthermore, due to the recursive structure of the inverse butterfly circuit, we can generalize (5) by substituting j for $\lg(n)$, 2^j for n and 2^{j-1} for $n/2$:

$$cb_j = \begin{cases} 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-(s \bmod 2^{j-1})}, & s \bmod 2^j < 2^{j-1} \\ \sim \left(1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-(s \bmod 2^{j-1})}\right), & s \bmod 2^j \geq 2^{j-1} \end{cases} \quad (6)$$

There are j bits in $s \bmod 2^j$, with the most significant bit denoted s_{j-1}. The condition $s \bmod 2^j < 2^{j-1}$ is equivalent to s_{j-1} being equal to 0 and the condition $s \bmod 2^j \geq 2^{j-1}$ is equivalent to s_{j-1} being equal to 1:

$$cb_j = \begin{cases} 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-(s \bmod 2^{j-1})}, & s_{j-1} = 0 \\ \sim \left(1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-(s \bmod 2^{j-1})}\right), & s_{j-1} = 1 \end{cases} \quad (7)$$

Equation (7) can be rewritten as the pattern XORed with s_{j-1}:

$$cb_j = \left(1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-(s \bmod 2^{j-1})}\right) \oplus s_{j-1} \quad (8)$$

Since $s \bmod k \leq k\text{-}1$, $k - (s \bmod k) \geq 1$ and hence the length of the string of zeros in (8) is always ≥ 1 ($k=2^{j-1}$). Consequently, the least significant bit of the pattern (prior to XOR with s_{j-1}) is always "0":

$$cb_j = \left(1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-1-(s \bmod 2^{j-1})} \,\|\, 0\right) \oplus s_{j-1}$$

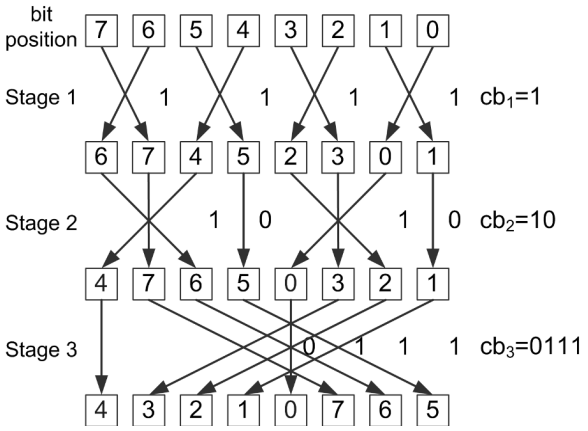

Fig. 8. Right rotate by 5 on 8-bit, 3-stage inverse butterfly network

$$cb_j = \left(\left(1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-1-(s \bmod 2^{j-1})}\right) \oplus s_{j-1}\right) \| \, s_{j-1} \quad (9)$$

We call the bit pattern inside the inner parenthesis of (9) $f(s, j)$, a string of 2^{j-1}-1 bits with the $s \bmod 2^{j-1}$ leftmost bits set to "1" and the remaining bits set to "0." This function is only defined for $j \geq 2$ and returns the empty string for $j = 1$:

$$f(s, j) = \begin{cases} 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-1-(s \bmod 2^{j-1})}, & j \geq 2 \\ \{\}, & j = 1 \end{cases} \quad (10)$$

$$cb_j = \left(f(s, j) \oplus s_{j-1}\right) \| \, s_{j-1} \quad (11)$$

Note that we can derive $f(s, j+1)$ from $f(s, j)$:

$$f(s, j+1) = 1^{s \bmod 2^j} \,\|\, 0^{2^j-1-(s \bmod 2^j)} \quad (12)$$

If bit $s_{j-1} = 0$ then $s \bmod 2^j = s \bmod 2^{j-1}$:

$$\begin{aligned} f(s, j+1) &= 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^j-1-(s \bmod 2^{j-1})} \\ &= 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-1-(s \bmod 2^{j-1})} \,\|\, 0^{2^{j-1}} \\ f(s, j+1) &= f(s, j) \,\|\, 0^{2^{j-1}} \end{aligned} \quad (13)$$

If bit $s_{j-1} = 1$ then $s \bmod 2^j = 2^{j-1} + s \bmod 2^{j-1}$:

$$\begin{aligned} f(s, j+1) &= 1^{2^{j-1}+s \bmod 2^{j-1}} \,\|\, 0^{2^j-1-\left(2^{j-1}+s \bmod 2^{j-1}\right)} \\ &= 1^{2^{j-1}} \,\|\, 1^{s \bmod 2^{j-1}} \,\|\, 0^{2^{j-1}-1-(s \bmod 2^{j-1})} \\ f(s, j+1) &= 1^{2^{j-1}} \,\|\, f(s, j) \end{aligned} \quad (14)$$

Combining (13) and (14), we get:

$$f(s, j+1) = \begin{cases} f(s, j) \,\|\, 0^{2^{j-1}}, & s_{j-1} = 0 \\ 1^{2^{j-1}} \,\|\, f(s, j), & s_{j-1} = 1 \end{cases}$$

$$f(s,j+1)=\begin{cases} f(s,j)\,\|\,0\,\|\,0^{2^{j-1}-1}, & s_{j-1}=0 \\ 1^{2^{j-1}-1}\,\|\,1\,\|\,f(s,j), & s_{j-1}=1 \end{cases} \tag{15}$$

Since $f(s, j)$ is a string of 2^{j-1}-1 bits, we can replace the string of ones and zeros in (15) by $f(s, j)$ ORed (+) with 1 and ANDed (•) with 0, respectively:

$$\begin{aligned} f(s,j+1)&=\begin{cases} f(s,j)+0\,\|\,0\,\|\,f(s,j)\bullet 0, & s_{j-1}=0 \\ f(s,j)+1\,\|\,1\,\|\,f(s,j)\bullet 1, & s_{j-1}=1 \end{cases} \\ f(s,j+1)&=\left(f(s,j)+s_{j-1}\right)\|\,s_{j-1}\,\|\left(f(s,j)\bullet s_{j-1}\right) \end{aligned} \tag{16}$$

From (10) and (16) we obtain a simple recursive expression for $f(s, j)$:

$$f(s,j)=\begin{cases} \left(f(s,j-1)+s_{j-2}\right)\|\,s_{j-2}\,\|\left(f(s,j-1)\bullet s_{j-2}\right), & j\geq 2 \\ \{\}, & j=1 \end{cases} \tag{17}$$

Fig. 9 depicts the hardware implementation of the control bit generator for rotations. Equation (17) is used to derive $f(s, 2)$, $f(s, 3)$, $f(s, 4)$ and $f(s, 5)$. Also, the control bits for rotations, cb_1, cb_2, cb_3, cb_4 and cb_5, are obtained using equation (11). This implementation is based on sharing of gates by reusing $f(s, j)$ for both cb_j and $f(s, j+1)$.

We now illustrate the use of these equations with the example of Fig. 8, the 8-bit inverse butterfly network with right rotation amount s = 5 ($s_2s_1s_0$ = 101). The first stage control bit, cb_1, replicated for the four 2-bit circuits, is given by equations (11) and (17):

$$cb_1=\left(f(5,1)\oplus s_0\right)\|\,s_0=\{\}\oplus s_0\,\|\,s_0=s_0=1.$$

The second stage control bits, cb_2, replicated for the two 4-bit circuits, are given by:

$$\begin{aligned} cb_2&=\left(f(5,2)\oplus s_1\right)\|\,s_1 \\ &=\left(\left(f(5,1)+s_0\,\|\,s_0\,\|\,f(5,1)\bullet s_0\right)\oplus s_1\right)\|\,s_1 \\ &=\left(\left(\{\}+s_0\,\|\,s_0\,\|\,\{\}\bullet s_0\right)\oplus s_1\right)\|\,s_1 \\ &=\left(s_0\oplus s_1\right)\|\,s_1 \\ &=\left(1\oplus 0\right)\|\,0 \\ &=10. \end{aligned}$$

Note that $f(5,2)$ = 1 in the above. The final stage control bits, cb_3, are given by:

$$\begin{aligned} cb_3&=\left(f(5,3)\oplus s_2\right)\|\,s_2 \\ &=\left(\left(f(5,2)+s_1\,\|\,s_1\,\|\,f(5,2)\bullet s_1\right)\oplus s_2\right)\|\,s_2 \\ &=\left(\left(1+s_1\,\|\,s_1\,\|\,1\bullet s_1\right)\oplus s_2\right)\|\,s_2 \\ &=\left(\left(1+0\,\|\,0\,\|\,1\bullet 0\right)\oplus 1\right)\|\,1 \\ &=\sim(100)\|\,1 \\ &=0111. \end{aligned}$$

Fig. 8 shows that this configuration of the inverse butterfly circuit does indeed right rotate the input by 5 mod 8 (and that the outputs of stage 2 are rotated by 5 mod 4 = 1 and that the outputs of stage 1 are rotated by 5 mod 2 = 1).

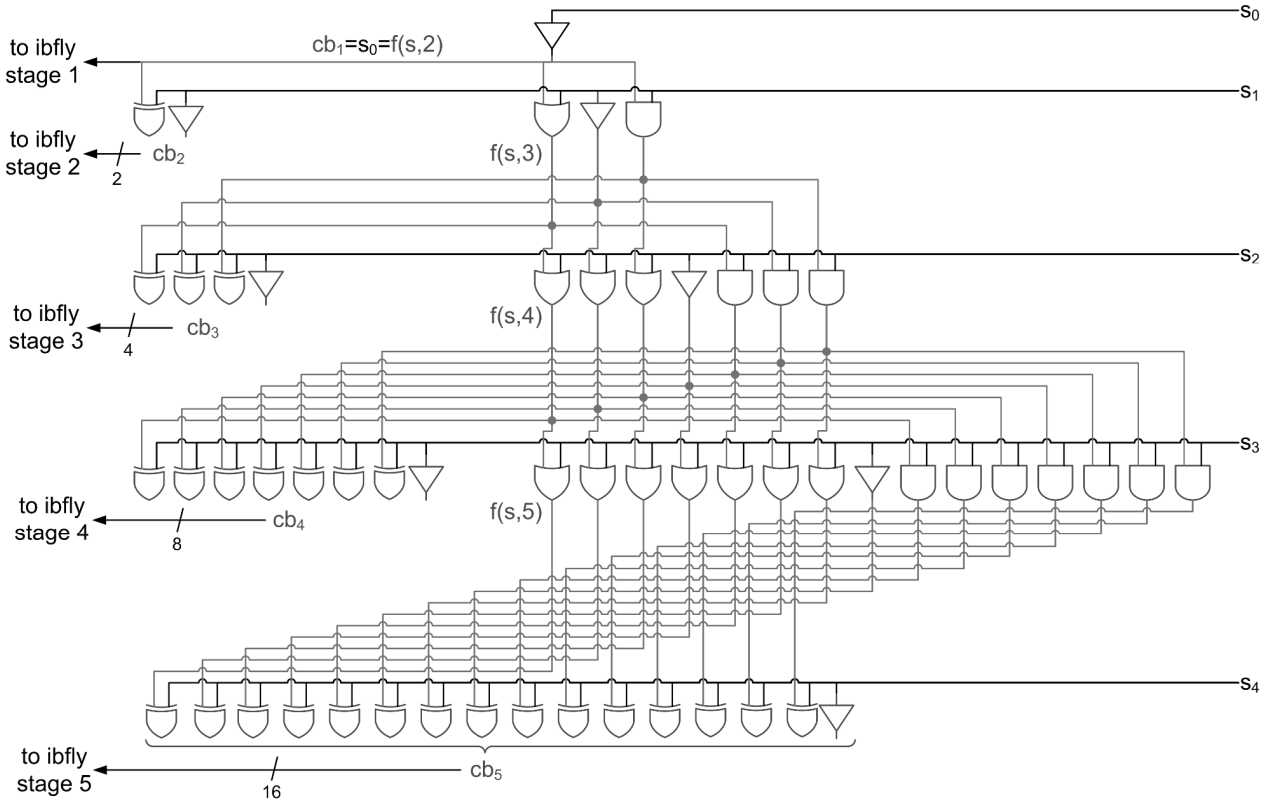

Fig. 9. Control bit generator circuit for rotations on inverse butterfly (first 5 stages)

3.1.2 Determining the Control Bits for Other Shift Operations

The other operations (shifts, extract, deposit and mix) are modeled as a rotation part plus a masked-merge part with zeroes, sign bits or second source operand bits. The rotation part can use the same rotation control bit generator described above to configure the inverse butterfly network datapath. We achieve the masked-merge part by using an *enhanced inverse butterfly datapath* with an extra multiplexer stage added as the final stage. The mask control bits are "0" when selecting the rotated bits and "1" when selecting the merge bits. We now describe how this is used to generate these other operations: shift, extract, deposit and mix.

For a right shift by s, the s sign or zero bits on the left are merged in. This requires a control string $1^s||0^{n-s}$ for the extra multiplexer stage. From the definition of $f(s, j)$, (10), we see that $f(s, \lg(n)+1)$ is the string $1^s||0^{n-1-s}$. Thus the desired control string is given by $f(s, \lg(n)+1)\ ||\ 0$. (Recall that $s < n$ therefore the least significant bit is always "0", i.e., the least significant bit is always selected from the inverse butterfly datapath.) $f(s, \lg(n)+1)$ can easily be produced by extending the rotation control bit generator by one extra stage. For left shift, which can be viewed as the left-to-right reversal of right shift, the control bits for the extra stage are obtained by reversing left-to-right the right shift control string to yield $0^{n-s}||1^s$.

For extract operations, which are like right shift operations with the left end replaced by the sign bit of the extracted field or zeros, our enhanced inverse butterfly network selects in its extra multiplexer stage the rotated bits or zeros or the sign bit of the extracted field i.e., the bit in position pos+len-1 in the source register (see Fig. 1(a)). The bit can be selected using an n:1 multiplexer. The control bit pattern for this stage is n-len "1"s followed by len "0"s ($1^{n\text{-len}}||0^{\text{len}}$) to propagate the sign bit of the extracted field in the output (which is in position len-1) to the high order bits. Note that $cb_{\lg(n)+1}(\text{len})$ is $1^{\text{len}}||0^{n\text{-len}}$ (as len ranges from 0 to n). So reversing left-to-right $cb_{\lg(n)+1}(\text{len})$ yields $0^{n\text{-len}}||1^{\text{len}}$ and then negating it produces $1^{n\text{-len}}||0^{\text{len}}$, the correct bit pattern for stage $\lg(n)+1$.

For deposit operations, which are like left shift operations with the right and left ends replaced by zeros or bits from the second operand, our enhanced inverse butterfly network selects in its extra multiplexer stage the rotated bits or zeros or bits from the second input operand. The correct pattern is a string of n-pos-len "1"s followed by len "0"s followed by s=pos "1"s ($1^{n\text{-pos-len}}||0^{\text{len}}||1^{\text{pos}}$) to merge in bits on the right and left around the deposited field. $cb_{\lg(n)+1}(\text{pos+len})$ is $1^{\text{pos+len}}||0^{n\text{-pos-len}}$. Reversing left-to-right this string yields $0^{n\text{-pos-len}}||1^{\text{pos+len}}$ and then negating it produces $1^{n\text{-pos-len}}||0^{\text{pos+len}}$. Bitwise ORing this with the left shift control string, $0^{n\text{-pos}}||1^{\text{pos}}$, yields $1^{n\text{-pos-len}}||0^{\text{len}}||1^{\text{pos}}$, the correct pattern for the masked-merge part of the deposit operation is produced.

For mix operations, the enhanced inverse butterfly network selects in its extra multiplexer stage the rotated bits or the bits from the second input operand. The control bit pattern is simply a pattern of alternating strings of "0"s and "1"s, the precise pattern depending on the subword size and whether mix left or mix right is executed. These patterns can be hard coded in the circuit for the 12 mix operations (6 operand sizes × 2 directions).

These mask-merged bit patterns are summarized in Fig. 16. Block diagrams of the functional unit are shown in Fig. 14 and Fig. 15, where only part of the non-dotted blocks are needed for implementing these basic shifter and mix operations. The other parts of the blocks are needed for the advanced bit manipulations (bottom half of Table 1). We describe how these are implemented on an inverse butterfly circuit (or butterfly circuit) below, before describing the whole functional unit.

3.2 Parallel Extract on the Inverse Butterfly Datapath

In this section we describe how an inverse butterfly datapath can perform the parallel extract (pex) operation, which is unpublished work [23]. We have described in earlier published work [8] how a parallel deposit (pdep) operation can be performed on a butterfly network.

Since it is not obvious that the pex instruction can be implemented by a butterfly or inverse butterfly datapath, we first give a conceptual explanation for mapping pex onto an inverse butterfly circuit. (In fact, pex cannot be implemented by a butterfly circuit [9], [23]).

First, we describe how an individual bit is routed on the inverse butterfly circuit based on its desired destination (Fact 1). Given this routing method, we then state how to determine whether an arbitrary permutation of a set of bits can be routed on the inverse butterfly circuit without path conflict (Fact 2). Finally, we show in Theorem 2 that the pex operation indeed satisfies this criterion and thus that pex can be mapped onto an inverse butterfly circuit.

The inverse butterfly circuit is decomposed into even and odd subcircuits. In Fig. 10, the even subcircuits are shown with dotted lines and the odd subcircuits with solid lines. These can also be called Right and Left subcircuits. For simplicity and clarity of notation we refer to even subcircuits as R (right) and odd as L (left).

Fact 1. *Any single data bit can be moved to any result position by just moving it to the correct R or L subcircuit of the intermediate result at every stage of the inverse butterfly circuit.*

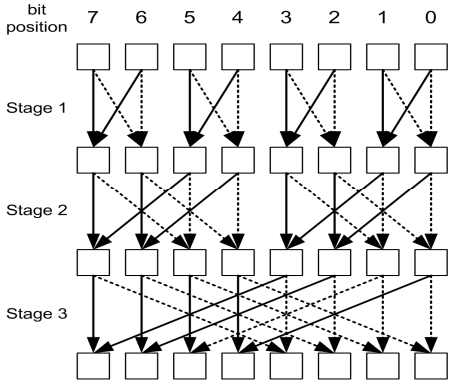

Fig. 10. Even or Right (dotted or R) and odd or Left (solid or L) subcircuits of the inverse butterfly circuit

Proof. 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, 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 2. *A permutation is routable on an inverse butterfly circuit 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 subcircuit of width m.*

Proof. Based on Fact 1, bits are routed on the inverse butterfly circuit by moving them to the correct position mod 2 after the first stage, mod 4 after the second stage, etc. Consequently, if the two bits entering stage 1, (with 2-bit wide inverse butterfly circuits as shown in Fig. 10), have destinations equal to 0 and 1 mod 2 (i.e., one is going to R and one to L), Fact 1 can be satisfied for both bits and they are routable through stage 1 without conflict. Subsequently, the four bits entering stage 2 (with the 4-bit wide butterfly circuits) must have destinations equal to 0, 1, 2 and 3 mod 4 to satisfy Fact 1 and be routable through stage 2 without conflict. A similar constraint exists for each stage.

Theorem 2. *Any Parallel Extract instruction on $n=2^{lg(n)}$ bits can be implemented with one pass through an inverse butterfly circuit of lg(n) stages without path conflicts (with the un-selected 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 subcircuit must be adjacent in the output. In other words, one bit has a destination equal to 0 mod 2 and the other has a destination equal to 1 mod 2 – the destinations constitute a complete set of residues mod 2 and thus are routable through stage 1. The selected data bits that enter the same stage 2 subcircuit 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 circuit. □

We do not formally prove here that parallel deposit can be performed using the butterfly datapath, rather we appeal to symmetry – the parallel deposit operation is the inverse of the parallel extract operation (see Fig. 5) and thus if parallel extract can be done on the inverse butterfly circuit, parallel deposit can be done on butterfly. For a full proof see [8], [9].

Also note that as parallel extract can be performed on the inverse butterfly circuit, the grp operation (Fig. 4) can be mapped to two parallel inverse butterfly circuits as grp is a combination of a grp_right (or pex) and a grp_left (a mirrored pex) [21].

3.2.1 Determining the Control Bits for Parallel Extract and Parallel Deposit

A decoder for the parallel extract (and parallel deposit) instruction takes as its input the n-bit mask (in the register operand, r3, in Fig. 5(a))and produces the $n/2 \times \lg(n)$ control bits for the inverse butterfly (or butterfly) circuit. The decoder can be designed to consist of only two types of operations that can be performed in software or implemented as circuits: a parallel prefix population counter, which counts the ones from position 0 (on the right) to every bit position from 0 to n–2, and a set of left rotators that complement bits upon wraparound (LROTC – Left ROTate and Complement).

To give some intuition on how the bits are computed, consider Fig. 11, which depicts the final stage of the pex operation. In the final stage, we call X the selected data bits that pass through in R, Y the selected data bits that are transferred from L to R, and Z the selected data bits that pass through in L. (Recall that we use R to denote the Right subnet and L to denote the Left subnet in an inverse butterfly circuit – see figure 10).

X is the rightmost bits of R, as the output of pex is the selected data bits compressed and right justified in the result. Y is rotated left from the midpoint by the size of X, $|X|$, at the input to the final stage so that when it is swapped into R in the final stage it is contiguous to X on its right. Z is the rightmost bits in L at the input to the final stage, so that when it is passed through in L, it is contiguous with the bits in R. Thus the control bit pattern for the final stage is $1^{n/2-|X|}||0^{|X|}$, where "1" denotes swap, and "0" denotes pass-through of the paired bits in L and R, in the last stage. $|X|$ is equal to the count of "1"s in the right half of the bit mask as all the selected data bits in the right half of the input have already been compressed and right justified prior to the input to the final stage.

Furthermore, we can view Y and Z as having been left rotated by $|X|$ from being the rightmost bits of L at the output of stage lg(n)-1 (see top half of Fig. 11). Consequently, we can iterate backwards through the stages, performing a pex operation within each half subnetwork and then explicitly left rotate each local L at the input of the next stage by the number of selected data bits in the local R at the input of the next stage prior to the swapping or passing through of bits in this next stage.

Rather than rotating the data bits explicitly, we can

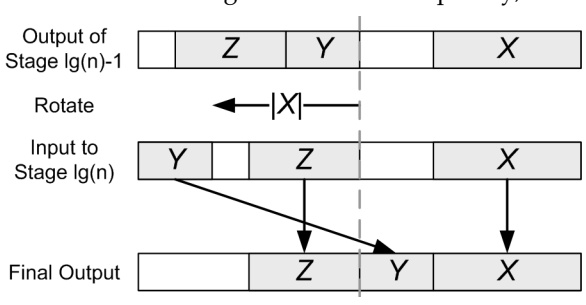

Fig. 11. Final stage of pex operation

compensate for the rotation by modifying the routing through the subsequent stages. This can be achieved by rotating the control bits by the same number of positions, complementing upon wraparound [9]. The counting of the number of "1"s in the right half of each subnet at each stage is done by the Parallel Prefix Population Count operation, while the rotation is done by the LROTC operation at each stage.

The full decoding algorithm is given in Fig. 12. A block diagram of the hardware decoder is shown in Fig. 13. (See [9] for a more thorough explanation of the decoder.) The same decoder can be used for pdep, except that the ordering of the resulting sets of $n/2$ control bits for the lg(n) stages is reversed.

3.2.2 Static, Dynamic and Loop-Invariant Instructions

As the hardware implementation of the decoder in Fig. 13 is costly we define three classes of parallel extract and parallel deposit instructions. The first consists of *static* versions of these instructions, where software or the compiler "pre-decodes" the mask in the second source register into control bits for the datapath, and moves the control bits into the Application Registers (ARs). This uses the mov instruction in Table 1, followed by the pex or pdep instructions.

The second class is *dynamic* mask decoding by a hardware implementation of the decoder of Fig. 13. This uses the pex.v or pdep.v instructions in Table 1. We have found dynamic pex.v and pdep.v unnecessary for most applications [8], [9].

The third class is *loop-invariant* pex and pdep. Here, the mask is known only at runtime but remains fixed for a long input stream. The hardware decoder can output the control bits to the application registers once, and subsequently the static versions of pex and pdep are used. This uses the setib and setb instructions in Table 1, followed by static pex or pdep instructions.

3.3 New Shift-Permute Functional Unit Implementation

The new shift-permute functional unit in Fig. 14 can support all the instructions listed in Table 1. For the grp instruction to also be supported, a second inverse butterfly network and a second decoder would be required at a severe hardware cost. As bfly and ibfly can be used to achieve permutations and pex can be used to emulate grp, albeit with a performance penalty, we choose not to include support for grp in the new functional unit. The last 4 instructions in Table 1 are needed only if dynamic and loop-invariant variants of the pex and pdep instructions are implemented.

The new functional unit consists of the two datapaths, the butterfly and inverse butterfly circuits, each enhanced with an extra 2:1 multiplexer stage (as well as premasking for the pex operation); the control bit generator for configuring the two datapaths; and the masked merge block which generates the merge bits and mask control for the extra multiplexer stage. Note that the control bits fed to the butterfly circuit are reversed left-to-right and also reversed top to bottom in terms of the order of the control bits for the stages. Also note that the butterfly cir-

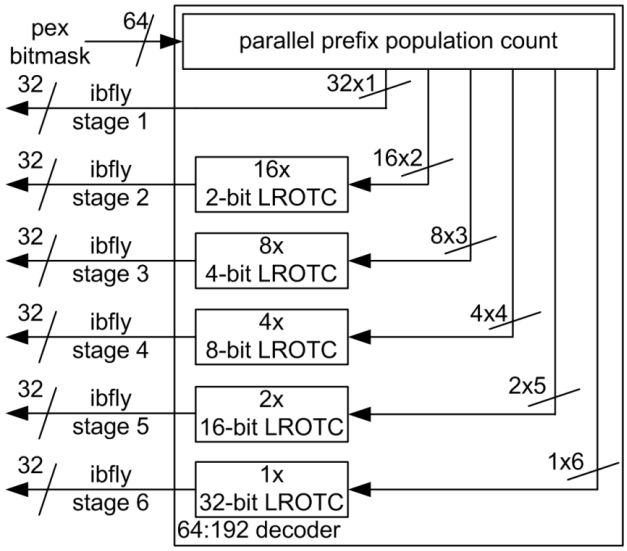

Fig. 13. Pex (and pdep and grp) bitmask decoder

Algorithm 1: To generate the lg(n) × $n/2$ inverse butterfly control bits from the n-bit mask.
Input: mask; the pex bitmask
Output: ibcb; the lg(n) × $n/2$ 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. 1^k indicates a bit-string of k ones. LROTC(a, rot) is a "left rotate and complement on wrap-around" 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 subnet and m is the leftmost bit position of each of the embedded local R subnets:

```
ibcb = {}
For i = 1, …, lg(n)                    //for each stage
   k = 2^(i-1)                         //number of bits in local R at the input to the ith stage
   For j = 1, 2, 3, …, 2^(lg(n)-i)     //for each local R
      m = k × (j × 2 – 1) – 1          //the left edge of the local R
      temp = LROTC(1^k, pc[m])
      ibcb[i] = temp ▯ ibcb[i]
```

Fig. 12. Algorithm for bitmask decoder

cuit is considered optional (and shown in dotted lines) as the functionality can be emulated using the inverse butterfly network at the cost of some performance. One pass of the butterfly network can be emulated with at most lg(n) passes of the inverse butterfly network. Also the latency of the control bit generation is serialized with respect to the latency of the butterfly circuit, since the control bits that take longest to generate are needed to control the first stage of the butterfly circuit, which is the last stage of the inverse butterfly circuit. Thus the inverse butterfly datapath is faster, since its control bit generation latency is overlapped with its datapath latency.

Fig. 15 shows the control bit generator block in more detail. The source of the control bits can be from the rotation control bit generator (Fig. 9), the pex/pdep decoder (Fig. 13), or the application registers. The pex/pdep decoder supplies the bits for the dynamic pex and pdep operations. This block is consider optional, as the pex/pdep decoder is large and has long latency (2 cycles), and is only needed for variable pex.v or pdep.v instructions which we found were rarely used [8], [9]. Hence, it is shown in dotted lines. The rotation control bit generator is extended to produce $cb_{\lg(n)}$. The shift amount is s for right rotates, right shifts, mix_left (the left subwords are shifted right) and extracts; it is n-s for left rotates, left shifts, mix_right and deposits. Also output is $f(s, \lg(n))$ to the masked-merge block for use in computing the masked-merge controls. The final source of the control bits is from the application registers for use in the bfly and ibfly permutation instructions or static pex and pdep instructions.

Fig. 16 shows an overview of the masked merge block and lists the bit patterns generated for the merge bits and the mask control bits. The merge bits can be:

- the zero string for unsigned right shift, left shift, unsigned extract, deposit-and-zero, and parallel deposit;
- the sign bit for arithmetic right shift;
- the sign bit of the extracted field (bit pos+len-1 of the source operand) for signed extract or
- the second source operand r_3 for deposit-and-merge and mix.

The mask control bits are "0" when selecting the rotated bits output from the butterfly or inverse butterfly circuit and "1" when selecting the merge bits. The patterns are:

- the zero string for right and left rotate, bfly and ibfly permutation instructions and parallel extract (which do not merge bits);
- the second source operand r_3 for parallel deposit or
- various strings of "1"s and "0"s for shifts, extract, deposit and mix, as described in Section 3.1.

4 Evolution of Shifters

One popular shifter architecture is the barrel shifter. The barrel shifter essentially is an n-bit wide n:1 multiplexer that selects the input shifted by s positions, where s = 0,1,2,…, n-1 (Fig. 17). The advantage of this design is that there is only a single gate delay between the input and output. The disadvantages are that n^2 switch elements (pass transistors or transmission gates) are required, and long delays due to capacitance as each input fans out to n elements, each output fans in from n elements and the shift amount needs to be decoded.

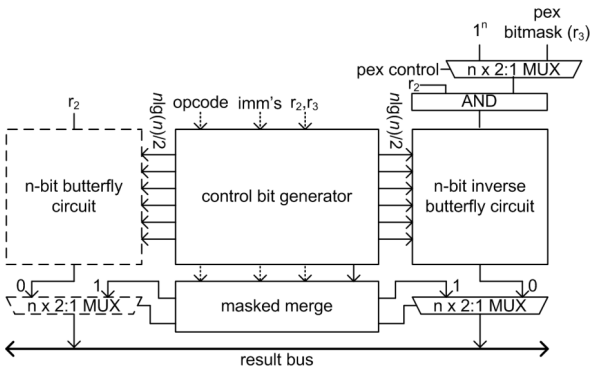

Fig. 14. New shift-permute functional unit

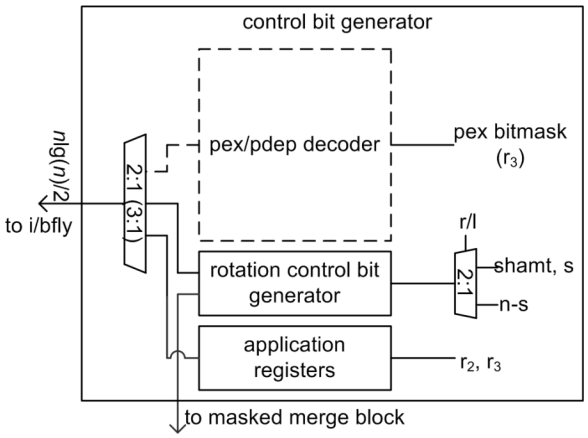

Fig. 15. Control bit generator circuit block

masked merge	
	mask control:
merge bits:	rot, bfly, ibfly, pex: 0^n
shr.u, shl, extr.u, dep.z, pdep: 0^n	pdep: bitmask (r_3)
shr: (sign bit)^n	shr: $1^s \| \| 0^{n-s}$
extr: (r_2[pos+len-1])^n	shl: $0^{n-s} \| \| 1^s$
dep, mix: r_3	extr: $1^{n-len} \| \| 0^{len}$
	dep: $1^{n-pos-len} 0^{len} 1^{pos}$
	mix.r: $(0^k 1^k)^{n/2k}$, k=1,2,4,8,16,32
	mix.l: $(1^k 0^k)^{n/2k}$, k=1,2,4,8,16,32

Fig. 16. Masked merge block

Due to the high capacitance of wires, a second popular shifter architecture emerged – the log shifter. The log shifter shifts the input by decreasing powers of two or four and selects at each stage the shifted version or the pass-through version from the previous stage (Fig. 18). The advantages are that only n×lg(n) or n×$\log_4(n)$ elements are required and the shift amount directly controls the multiplexer elements. The disadvantage is that there are lg(n) or $\log_4(n)$ gates between the input and output.

Left and right shifts can be performed by implementing two datapaths to perform left and right shifts separately or alternatively a single-direction shifter, e.g., only right shifting; the left shift is performed by subtracting

the left shift amount from the bit width, n, (with the appropriate logic to ensure proper zero propagation). Arithmetic right shift is accomplished by conditionally propagating the sign bit rather than a zero bit. Additionally, the shifters easily support rotations by wrapping around the bits.

Table 2 presents a high level comparison of the 3 shifter designs. The first two lines contain the components that contribute to area. Both the log shifter and our new ibfly-based shifter have $n\times\lg(n)$ elements, while the barrel shifter has n^2 elements. The log shifter also has the fewest control lines ($\lg(n)$) while our new shifter design has the most as each switch, or pair of elements, requires an independent control bit. Thus we might expect that our new design has slightly larger area than the log shifter.

The next two lines pertain to latency. The datapath of the barrel shifter has a single gate delay while the log shifter and our new design have $\lg(n)$ gate delay. However, both the log shifter and our new design utilize narrow multiplexers with lower capacitance at output nodes.

We first used the method of logical effort [24] to compare the delay along the critical paths for the barrel shifter, the log shifter and the inverse butterfly shifter. This estimates the critical path in terms of FO4 gate equivalents, which is the delay of an inverter driving 4 similar inverters. We compare the latency of only the basic shifter operations on these datapaths. As the 64-bit barrel shifter is impractical due to the capacitance on the output lines, we implemented a 64-bit shifter as an 8-byte barrel shifter followed by an 8-bit barrel shifter, which limits the number of transmission gates tied together to 8. We consider the delay only from the input to the decoder through the two shifter levels for the barrel shifter and through the three shifter levels for the log shifter.

For our proposed ibfly-based Shifter, we consider the delay from the input to the control bit generator (i.e., the *rotation* control bit generator of Fig. 9) through the output of the inverse butterfly circuit. According to the logical effort calculations, the delay for the barrel shifter is 15.1 FO4 and the delay for the log shifter is 13.0 FO4, while the delay for inverse butterfly shifter is 15.5 FO4. Thus the delay along the critical path for the barrel shifter and our new proposed shifter is comparable, and our new shifter is 19% slower than a log shifter.

As the log shifter is the faster and more compact of the two current shifter designs we implemented it and our new ibfly-based shifter design using a standard cell library. We synthesized all designs to gate level, optimizing for shortest latency, using Design Compiler mapping to a TSMC 90 nm standard cell library [25]. The results are summarized in Tables 3-5.

Table 3 shows the result of a basic shifter that only implements shift and rotate instructions. For the log shifter, we implemented parallel datapaths for left and right shifts. Our new shifter has 1.18× the latency of the log shifter, which is similar to the logical effort calculation. The new design is also smaller than the log shifter, at approximately 70% of the area. (Note that a single datapath log shifter would have even smaller area. Furthermore, accounting for the wires will increase the area for the ibfly-based shifter relative to the log shifter, as mentioned.) For comparison, we also implemented an ALU (supporting add, subtract, and, or, not, xor with register or immediate operands) synthesized using the same standard cell library. Our new shifter is faster (92% latency) and smaller (52% area) than this ALU.

Table 4 shows the result when both shifter architectures are enhanced to support extract, deposit and mix instructions (i.e., the top half of Table 1). The critical path of the log shifter is now through the extract sign bit propagation, so the latency is now comparable to that of the ibfly-based shifter. Our new ibfly-based design is still only 83% of the area of the log shifter. We include the results for an ALU of similar latency, which turns out to have comparable area, in NAND gate equivalents.

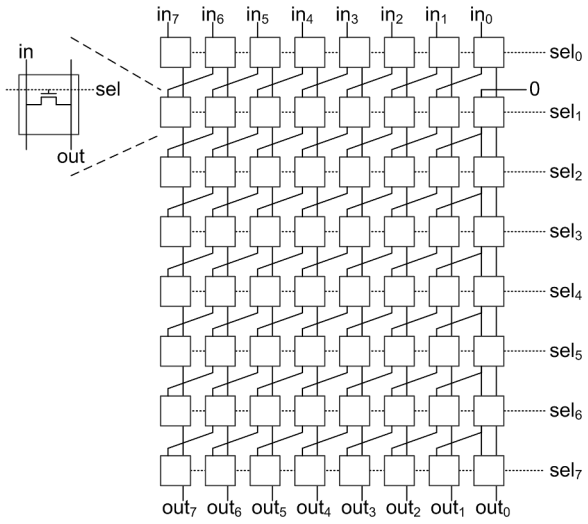

Fig. 17. 8-bit barrel shifter

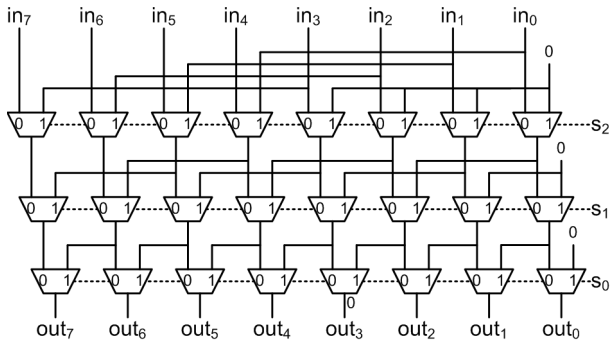

Fig. 18. 8-bit log shifter

TABLE 2
COMPARISON OF SHIFTER DESIGNS

	Barrel Shifter	Log Shifter	Our IBFLY Shifter
# Elements	n^2	$n\times\lg(n)$ or $n\times\log_4(n)$	$n\times\lg(n)$
# Control Lines	n	$\lg(n)$	$n/2\times\lg(n)$
Gate delay (Datapath)	1	$\lg(n)$ or $\log_4(n)$	$\lg(n)$
Mux width (Capacitance)	n	2 or 4	2

Table 5 shows the results when we add support for advanced bit manipulation operations to our new ibfly-based shifter, but not to the log-shifter. The first line is the log shifter circuit from Table 4, included as the baseline. The second line is a unit that supports the ibfly and static pex instructions. We considered this unit in [9] as the functionality of the bfly circuit can be emulated using ibfly, albeit with a multi-cycle penalty. The latency increases are due to extra multiplexing for the control bits and output. The area increases due to the ARs, the extra multiplexers and the pex masking. This unit has 1.18× the latency and 1.29× the area of the log shifter.

The third line is a unit that also supports bfly and static pdep. The latency increases slightly due to output multiplexing and the area increases due to the second (butterfly) datapath and second set of three ARs. This unit has 1.20× the latency and 1.87× the area of the log shifter. Alternatively, we can add a separate unit to perform just bfly and pdep (line 4), thereby enabling simultaneous superscalar execution with the ibfly-pex-shifter unit (line 2). We see that our full shift-permute unit (line 3) can be split into two units at no additional increase in area. We also include the results for an ALU of similar latency, which is now smaller than the log shifter due to the relaxed latency constraint. The full shift-permute unit (line 3) now has 2.25× the area of the ALU.

We remark that full custom designs of the ALU, log shifter and our new shift-permute unit should be done, since standard cell implementations may not reflect a fair or accurate comparison – especially between the shifters and the ALU, which is typically highly optimized by custom circuit design. Such circuit design is more appropriately done by microprocessor custom circuit designers according to implementation-specific needs and the process technology used.

TABLE 3
LATENCY AND AREA OF UNITS PERFORMING SHIFT AND ROTATE

Functional Unit	Latency (ns)	Relative Latency	Area (NAND gates)	Relative Area
Log Shifter	0.30	1	6.2K	1
Our IBFLY shifter	0.35	1.18	4.3K	0.69
ALU	0.38	1.27	8.2K	1.31

TABLE 4
LATENCY AND AREA OF UNITS PERFORMING SHIFT, ROTATE, EXTRACT, DEPOSIT AND MIX

Functional Unit	Latency (ns)	Relative Latency	Area (NAND gates)	Relative Area
Log Shifter	0.41	1	6.4K	1
Our IBFLY shifter	0.42	1.03	5.3K	0.83
ALU	0.40	0.99	6.6K	1.03

TABLE 5
LATENCY AND AREA OF UNITS PERFORMING ADVANCED BIT OPERATIONS

Functional Unit	Latency (ns)	Relative Latency	Area (NAND gates)	Relative Area
Log Shifter	0.41	1	6.4K	1
ibfly, pex	0.48	1.18	8.2K (1.2K for ARs)	1.29
ibfly, bfly, pex, pdep	0.49	1.20	11.9K (2.5K for ARs)	1.87
bfly, pdep (no shifter)	0.48	1.19	3.6K (1.5K for ARs)	0.57
ALU	0.49	1.20	5.3K	0.83

5 CONCLUSION

We described a new basis for Shifter and Mix functional units based on the inverse butterfly datapath. Our new Shift-Permute functional unit is a much more powerful functional unit: it performs the existing Shifter operations (shift, rotate, extract, deposit) and multimedia subword-permutation operations (mix operations) as well as the newly proposed advanced bit manipulation operations (bfly, ibfly, parallel extract and parallel deposit) and mix operations down to bit-level.

We showed how to configure an inverse butterfly circuit to achieve rotations; this is given by a simple recursive function of the shift amount. We also showed how to compute the merge bits and mask control for turning rotations into shifts, extract, deposit and mix operations, using the same recursive function.

We also showed how to determine the control bits to configure the inverse butterfly (and butterfly) circuits to perform parallel extract (and parallel deposit) operations.

We also put all the implementation details together for the entire Shift-Permute unit based on the inverse butterfly (and optional butterfly) datapaths.

Additionally, we compared the complexity of our new functional unit to that of the existing barrel shifter and log shifter. Our proposed Shift-Permute unit is about the same latency as a barrel shifter but slower than a log shifter, using logical effort estimations. With standard cell implementations of the faster log shifter and our proposed shifter, we find our shifter 3% slower and 17% *smaller* than the log shifter, for the circuit supporting shifts, rotates, extract, deposit and mix. The full circuit is 20% slower and 87% larger than the log shifter while supporting a much more powerful set of advanced bit manipulation operations (including ibfly, bfly, pex and pdep) as well as existing shifter and mix operations. Since the mix operation is currently supported as a separate multimedia functional unit in [3], we may have potentially *reduced* overall area requirements by replacing two existing functional units (Shifter, Multimedia-mix) with one new unit. In contrast to adding an advanced bit manipulation unit, this yields a significant reduction in area.

In summary, our proposed new Shift-Permute functional unit enables processors to support advanced bit manipulations efficiently, in addition to existing shifter instructions, with only minor cycle-time latency and area overhead. We recommend its use in future microprocessors. Unlike adders and multipliers, shifters have not

evolved as much in the past few decades. We hope to have stimulated further research into optimal implementations of shifters, and also into important applications of the new, high performance, advanced bit manipulation operations.

ACKNOWLEDGMENT

This work was supported in part by a grant from the Department of Defense. Y. Hilewitz holds NSF and Hertz Foundation fellowships.

REFERENCES

[1] Intel Corporation, *IA-32 Intel® Architecture Software Developer's Manual*, vol. 2, 2004.

[2] R.B. Lee, "Precision Architecture", *IEEE Computer*, Vol. 22, No. 1, pp.78-91, Jan. 1989.

[3] Intel Corporation, *Intel® Itanium® Architecture Software Developer's Manual*, vol. 3, rev. 2.2, Jan. 2006.

[4] IBM Corporation, *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors*, ver. 2.0, June 2003.

[5] R.B. Lee and J. Huck, "64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture," *Proceedings of IEEE Compcon*, pp. 152-160, Feb. 1996.

[6] R.B. Lee, "Subword Parallelism with MAX-2", *IEEE Micro*, Vol. 16 No. 4, pp. 51-59, Aug. 1996.

[7] H.S. Warren Jr., *Hackers's Delight*, Boston: Addison-Wesley Professional, 2002.

[8] Y. Hilewitz and R.B. Lee, "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*, pp. 65-72, Sept. 2006.

[9] Y. Hilewitz and R.B. Lee, "Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors," *Journal of Signal Processing Systems*, vol. 53, no. 1/2, Nov. 2008.

[10] Y. Hilewitz and R.B. Lee, "Performing Advanced Bit Manipulations Efficiently in General-Purpose Processors," *Proceedings of 18th IEEE Symposium on Computer Arithmetic*, June 2007.

[11] R.B. Lee, A.M. Fiskiran, and A. Bubshait, "Multimedia Instructions in IA-64," *Proceedings of the IEEE International Conference on Multimedia and Expo*, pp. 281-284, Aug. 2001.

[12] Z. Shi and R.B. Lee, "Subword Sorting with Versatile Permutation Instructions," *Proceedings of the International Conference on Computer Design*, pp. 234-241, Sept. 2002.

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

[14] R.B. Lee, Z. Shi, and X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56-69, Dec. 2001.

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

[16] R.B. Lee, Z. Shi, and X. Yang, "How a Processor can Permute n bits in O(1) Cycles," *Proceedings of Hot Chips 14 - A Symposium on High Performance Chips*, Aug. 2002.

[17] Z. Shi, X. Yang and R.B. Lee, "Arbitrary Bit Permutations in One or Two Cycles", *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 237-247, June 2003.

[18] R.B. Lee, X. Yang and Z.J. Shi, "Single-Cycle Bit Permutations with MOMR Execution," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 577-585, Sept. 2005.

[19] J.P. McGregor and R.B. Lee, "Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications," *Proceedings of the International Conference on Computer Design*, pp. 453-461, Sept. 2001.

[20] V.E. Beneš, "Optimal Rearrangeable Multistage Connecting Networks", *Bell System Technical Journal*, vol. 43, no. 4, pp. 1641-1656, July 1964.

[21] Y. Hilewitz, Z.J. Shi, and R.B. Lee, "Comparing Fast Implementations of Bit Permutation Instructions," *Proc. of the 38th Asilomar Conference on Signals, Systems, and Computers*, Nov. 2004.

[22] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, San Francisco: Morgan Kaufmann Publishers, 1992.

[23] R.B. Lee and Y. Hilewitz, "Fast Pattern Matching with Parallel Extract Instructions," *Princeton University Department of Electrical Engineering Technical Report CE-L2005-002*, Feb. 2005.

[24] I. Sutherland, B. Sproull and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, San Francisco: Morgan Kaufmann, 1999.

[25] Taiwan Semiconductor Manufacturing Corp., *TCBN90GTHP: TSMC 90nm Core Library Databook*, ver 1.1, Dec. 2006.

Yedidya Hilewitz received a B.Eng. degree in Electrical Engineering from The Cooper Union in 2003 and a Ph.D. in Electrical Engineering from Princeton University in 2008, where he explored the architecture and implementation of bit manipulation instructions in commodity microprocessors. He currently is a design engineer at the Massachusetts Microprocessor Design Center, Intel Corporation. He is a member of the IEEE and ACM.

Ruby B. Lee received an A.B. with distinction from Cornell University, where she was a College Scholar, a M.S. in Computer Science and a Ph.D. in Electrical Engineering both from Stanford University. Dr. Lee served as chief architect at Hewlett-Packard, responsible at different times for processor architecture, multimedia architecture and security architecture. She was a key architect of PA-RISC used in HP workstations and servers and introduced multimedia instructions for microprocessors in HP and Intel processor ISAs. Concurrent with her full-time HP employment, she was also Consulting Professor of Electrical Engineering at Stanford University. Currently, she 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, defining hardware trust anchors, multicore security, hardware defenses against software side-channel attacks, resiliency to Internet-scale epidemics, and innovative ISA for achieving supercomputer performance for advanced bit manipulations. She is a Fellow of the ACM and of the IEEE, Associate Editor-in-Chief of IEEE Micro and Editorial Board member of IEEE Security and Privacy. She holds over 120 U.S. and international patents.