Performing Advanced Bit Manipulations Efficiently in General-Purpose Processors

Yedidya Hilewitz and Ruby B. Lee
Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA
{hilewitz, rblee}@princeton.edu

Abstract

This paper describes a new basis for the implementation of a shifter functional unit. We present a design based on the inverse butterfly and butterfly datapath circuits that performs the standard shift and rotate operations, as well as more advanced extract, deposit and mix operations found in some processors. Additionally, it also supports important new classes of even more advanced bit manipulation instructions recently proposed: these include arbitrary bit permutations, bit scatter and bit gather instructions. The new functional unit's datapath is comparable in latency to that of the classic barrel shifter. It replaces two existing functional units - shifter and mix - with a much more powerful one.

Keywords: shifter, rotations, permutations, bit manipulations, arithmetic, processor

1. Introduction

Computer arithmetic for integers and floating-point numbers is a well-developed and mature field. Many books and papers describe the design of integer arithmetic and floating-point arithmetic units. Less extensively studied is the design of shifters, and functional units for advanced bit manipulations in general-purpose, word-oriented microprocessors. Simple bit-parallel operations like AND, OR, XOR and NOT are supported in word-oriented processors. Denoted "logical" operations, they are typically implemented with the integer arithmetic operations in the Arithmetic-Logic Unit (ALU), which is the most basic functional unit in a processor.

In current microprocessors, only very few bit operations that are not bit-parallel are supported. These are shifts and rotates, where each bit moves in the same relative amount as every other bit in the word (register). A separate Shifter functional unit is typically used to implement these operations. A few processors also have Extract_field and Deposit_field operations, which can be viewed as variants of Shift_Right and Shift_Left operations, with certain bits masked out and set to zeros or replicated-sign bits.

There are many emerging applications, such as cryptography, imaging and biometrics, where more advanced bit manipulation operations are needed. While these can be built from the simpler logical and shift operations, 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, performing multiple bit-field extract operations in parallel, and performing multiple bit-field deposit operations in parallel. We call these permutation (perm), parallel extract (pex) or bit gather, and parallel deposit (pdep) or bit scatter operations, respectively. They will be further described in section 2. It has been shown that these operations can be implemented in a single new Permutation functional unit, utilizing two simple datapaths: an inverse butterfly circuit and a butterfly circuit [1].

In this paper, we show that we can perform both existing and newly proposed bit manipulation instructions with a single, simple functional unit, rather than two (or more) separate units. Also, instead of starting with the existing Shifter functional unit and extending it so that it also performs bit permutations, bit gather and bit scatter operations, we propose to start with the more powerful Permutation functional unit and perform all the operations previously done by the Shifter functional unit. Hence, our new Shift-Permute functional unit can perform all the useful bit manipulations beyond the simple bit-parallel logical operations already done by the ALU.

The contributions of this paper are:

- A proposal for a new basis for the design of Shifters, based on the inverse butterfly circuit, that is much more powerful, with only small or no impact on cycle-time and area.
- A recursive algorithm for determining the

control bits for Rotate, Shift, Extract, Deposit and Mix operations on the inverse butterfly and butterfly datapath circuits.

- Demonstration of the implementation of a powerful Shift-Permute unit, comparing its complexity with that of an ordinary Shifter functional unit.

In section 2, we list the instructions supported by our proposed new functional unit. In section 3, we show how to obtain the control bits for the inverse butterfly and butterfly datapaths. In section 4, the implementation of the new functional unit is described and compared to that of the barrel and log shifter. Section 5 concludes the paper.

2. Existing and Newly-Proposed Bit Manipulation Instructions

We now detail the operations supported by two existing microprocessor functional units – the Shifter and the Mix multimedia functional units - as well as a recently proposed Permutation functional unit [1].

2.1. Shifter and Mix Operations

The first 4 groups of instructions in Table 1 are the rotate and shift instructions supported in most microprocessors. These include right or left rotates, right or left shifts (with zero or sign propagation). The last 3 groups of instructions exist in a few Instruction Set Architectures (ISAs) such as PA-RISC [2] and IA-64 [3]. These include extract, deposit and mix instructions.

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

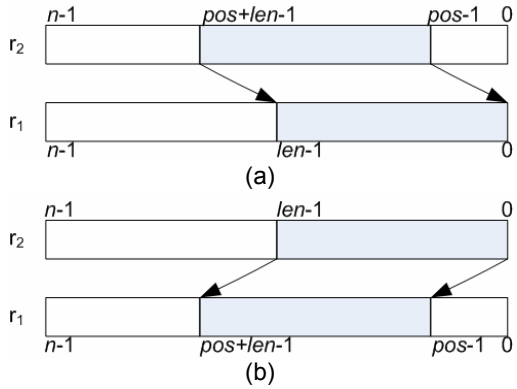

Figure 1. (a) extr.u r1 = r2, pos, len
(b) dep.z r1 = r2, pos, len

The mix operation selects the left subwords or right subwords from each pair of subwords alternating between the two source registers r_2 and r_3 (Figure 2). The mix instruction was first introduced in PA-RISC for multimedia acceleration [4], and also appears in

Table 1. Existing Shifter and Mix Operations

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.

IA-64 (Itanium) [3], 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 [5]. 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 datapath processor.

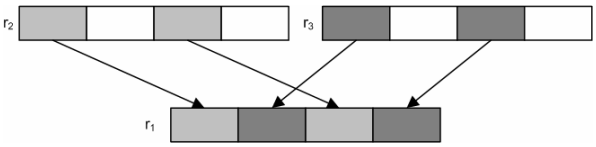

Figure 2. Mix left operation

2.2. Permute Operations

The permute operations are a new class of bit manipulation instructions that have been proposed for accelerating various application domains ranging from cryptography to bioinformatics. These instructions include the butterfly and inverse butterfly permutation instructions and the parallel extract and parallel deposit instructions, which are like bit gather and bit scatter instructions (Table 2). Without loss of generality, we consider only the simpler *static* versions of these instructions in this paper, where software "pre-decodes" the mask in the second source register into control bits for the datapath, and moves these into Application Registers (ARs), which are registers associated with the Permute functional unit. Dynamic mask decoding by hardware involves a complicated decoder circuit that we have found unnecessary for most applications [1].

The butterfly (bfly) and inverse butterfly (ibfly) instructions route their inputs through butterfly and inverse butterfly circuits, respectively [6]. The concatenation of these two circuits forms a Benes circuit, a general permutation network [7]. 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 [8].

The structure of the circuits is shown in Figure 3, which shows 8-bit circuits. 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, which we normalize to a latency of one processor cycle. Furthermore, each switch is composed of two 2:1 multiplexers, totaling $n \times \lg(n)$ multiplexers for each circuit, leading to small overall 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].

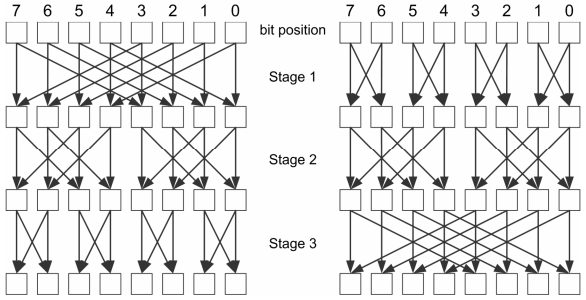

Figure 3. 8-bit Butterfly and Inverse Butterfly Circuits

The parallel extract (pex) and parallel deposit (pdep) instructions are generalizations of the extract and deposit instructions [1]. 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 Figure 4(a)). The

Table 2. Recently Proposed Bit Permutation Instructions

Instruction	Description
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

rest of the bits in the result register are cleared to "0"s. 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 Figure 4(b)). The remaining bits in the result register are cleared to "0"s.

Parallel extract and parallel deposit are equivalent to masked inverse butterfly and butterfly permutations, respectively. The pex or pdep bit mask in the second operand can be decoded by hardware [1] or software to generate the $n/2 \times \lg(n)$ inverse butterfly or butterfly control bits. Only the static variants for pex and pdep are listed in Table 2, where the application registers used to control the inverse butterfly and butterfly circuits are first loaded with pex or pdep control bits generated by software. The bit mask is still needed to mask the permutation to produce the pex or pdep result.

The goal of this work is to show that such a Permutation functional unit based on the butterfly and inverse butterfly circuits in Figure 3 can also support the existing Rotate, Shift, Extract, Deposit and Mix instructions listed in Table 1.

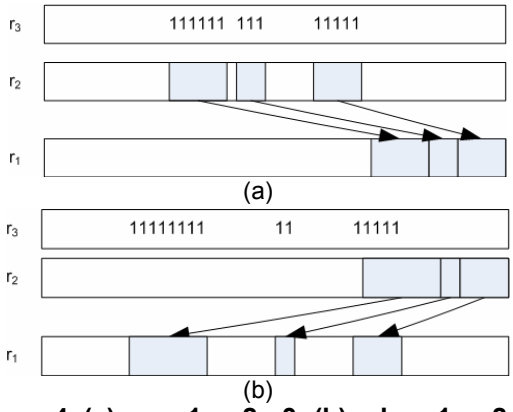

Figure 4. (a) pex r1 = r2, r3; (b) pdep r1 = r2, r3

3. Control Bits for Shift Operations on Butterfly and Inverse Butterfly Networks

In this section, we show that the n-bit inverse butterfly and butterfly circuits can achieve any of the Shifter and Mix operations in Table 1. The hard part is determining how the controls of the $\lg(n)$ stages of the circuit should be set, and we give a definitive algorithm for this.

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

Proof: This is a known property of inverse butterfly circuits (see [9] for example, where rotations are called cyclic shifts).

Corollary 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. 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 precomputed as either zero or the sign bit or the bits of the second source operand depending on the desired operation.

Corollary 2: Theorem 1 and Corollary 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 Figure 5). Hence, a butterfly circuit can also achieve any rotation of its inputs. As in Corollary 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.

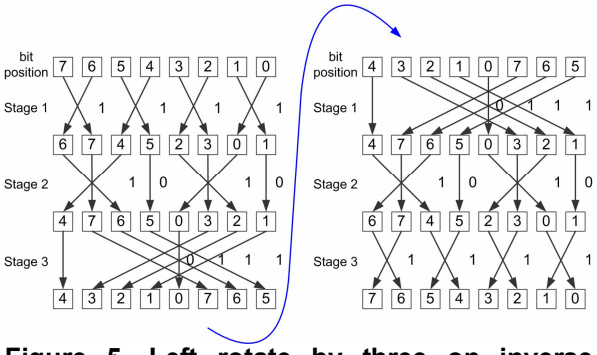

Figure 5. Left rotate by three on inverse butterfly is equivalent to right rotate by three on butterfly

We first show how control bits are obtained for rotations in section 3.1, then for the other operations in section 3.2.

3.1. Determining the Control Bits for Rotations

To achieve a rotation by s positions, s = 0, 1, 2 … n-1, using the n-bit wide inverse butterfly circuit with lg(n) stages, the input must be rotated by s mod 2^j within each 2^j-bit wide inverse butterfly circuit with j stages contained in the full n-bit circuit. This is because stage j+1 on can only move bits at granularities larger than 2^j positions.

Consider the full n-bit circuit, which 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. 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:

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

as the last stage can only move bits by n/2 positions.

When the rotation amount s is less than n/2 then the bits that *wrapped around* in the (lg(n)-1)-stage circuits must be swapped in the final stage to yield the input right_rotated by s (Figure 6(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 must be swapped in the final stage to yield the input right_rotated by s (Figure 6(b)):

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

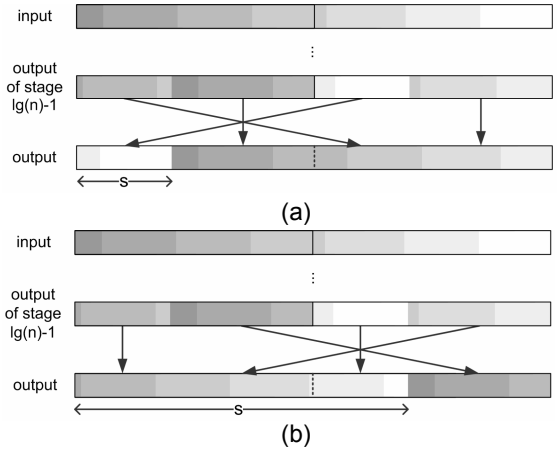

Figure 6. (a) rotation by $s<n/2$ and (b) by $s \geq n/2$

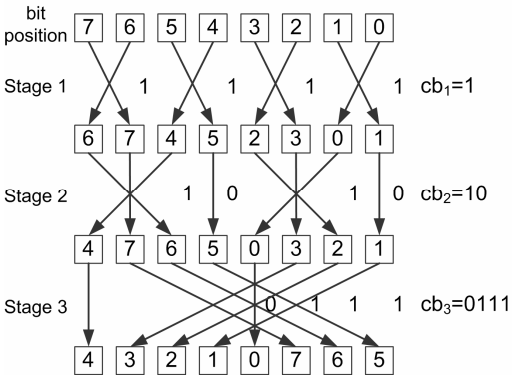

Figure 7. Right rotate by 5 on 8-bit, 3-stage inverse butterfly network

For example, consider the 8-bit inverse butterfly network with right rotation amount s = 5, depicted in Figure 7. 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 2^{2-1} = 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 mod 2^1 = 5 mod 2 = 1, is equal to than 2^{1-1} = 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, … lg(n), for achieving rotations on an inverse butterfly datapath. These equations yield a compact circuit (shown in Figure 8) for the (rotation) control bit generator shown in Figure 9 and Figure 10.

From (1)-(3) and Figure 6, we observe that the control bits pattern 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 $s = s$ mod n/2 when $s < n/2$ and s-n/2 = s mod 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 mod 2^j, with the most significant bit denoted s_{j-1}. The condition s mod $2^j < 2^{j-1}$ is equivalent to s_{j-1} being equal to 0 and the condition s mod $2^j \geq 2^{j-1}$ is equivalent to s_{j-1} being equal to 1:

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

(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 mod $k \leq k$-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":

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

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

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 mod 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 mod $2^j = s$ mod 2^{j-1}:

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

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

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

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

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

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

$$f(s, j+1) = 1^{2^{j-1}} \,||\, f(s, j) \quad (14)$$

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

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

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

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 (+) and ANDed (•) with 1 and 0, respectively:

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

$$f(s, j+1) = \left(f(s, j) + s_{j-1}\right) ||\, s_{j-1} \,||\, \left(f(s, j) \bullet s_{j-1}\right) \quad (16)$$

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

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

Figure 8 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 Figure 7, the 8-bit inverse butterfly network with right rotation amount s = 5 ($s_2 s_1 s_0$ = 101). The first stage control bit, cb_1, replicated for the four 2-bit circuits, is given by:

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

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

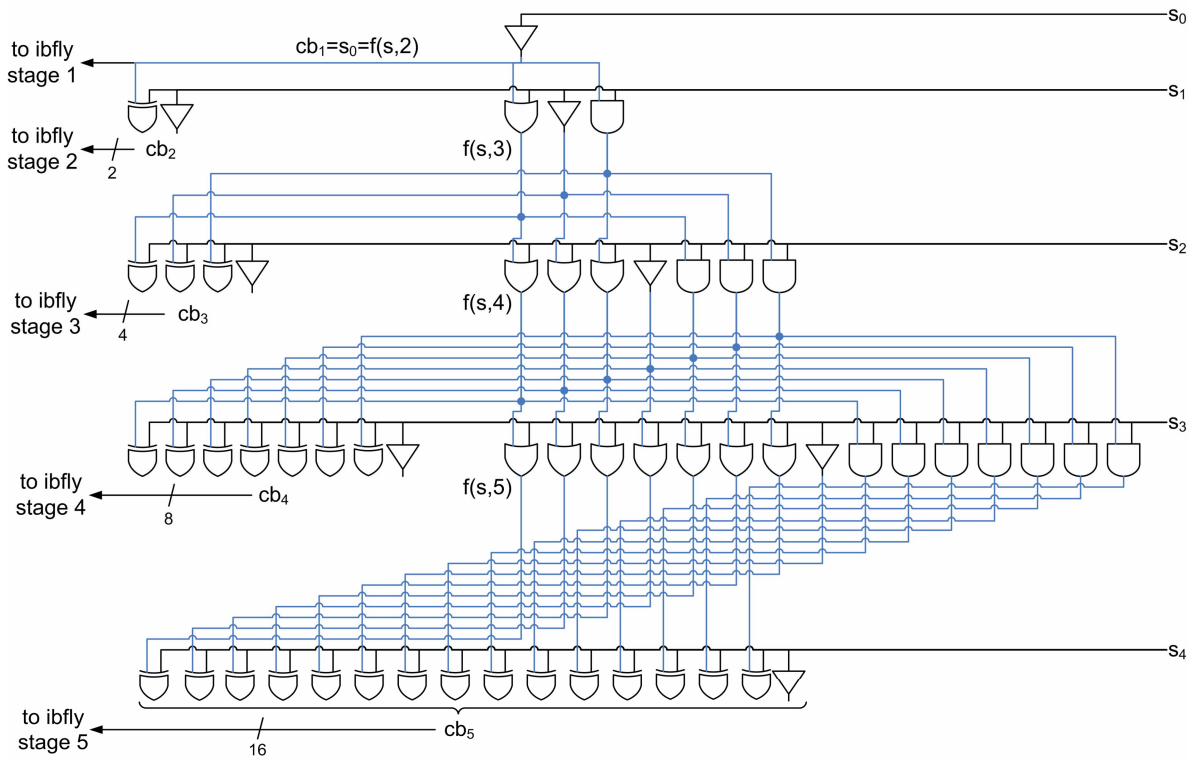

Figure 8. Control bit generator circuit for rotations on inverse butterfly (first 5 stages)

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

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

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

Figure 7 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).

3.2. 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 (see Figure 9). 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: shifts, extracts, deposits and mix's.

For a right shift by s operation, 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 in the source register. 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) to the high order bits. Note that $\{f(\text{len}, \lg(n)+1) \,||\, 0\}$ is $1^{\text{len}}||0^{n-\text{len}}$. So reversing left-to-right $\{f(\text{len}, \lg(n)+1) \,||\, 0\}$ 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. $\{f(\text{pos+len},\lg(n)+1) \,||\, 0\}$ is $1^{\text{pos+len}}||0^{\text{n-pos-len}}$. 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).

4. New Shift-Permute Functional Unit Implementation

A circuit block diagram of the combined shift-permute functional unit is shown in Figure 9. The 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 pre-masking for the pex operation [1]); 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 the order of the stages is reversed, following Corollary 2. Also note that the butterfly circuit is considered optional and hence is shown in dotted lines as the functionality can be emulated using the inverse butterfly network at the cost of some performance. 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. Thus the inverse butterfly datapath is faster, since its control bit generation latency is overlapped with its datapath latency.

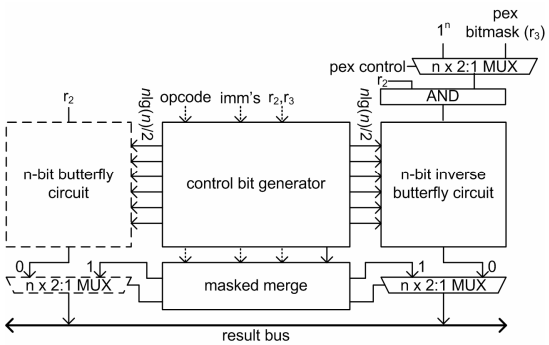

Figure 9. Combined shift-permute functional unit based on inverse butterfly circuit

Figure 10 shows the control bit generator block in more detail. The source of the control bits can be from a pex/pdep decoder [1], the rotation control bit generator (Figure 8) 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 decoder is large and has long latency (2 cycles), and is only needed for variable pex/pdep instructions which we found were rarely used [1]. Hence, it is shown in dotted lines. The rotation control bit generator is that of Figure 8, extended to produce $cb_{\lg(n)}$. The shift amount is s for right rotates, right

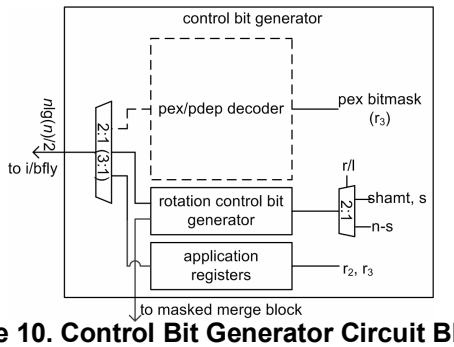

Figure 10. Control Bit Generator Circuit Block

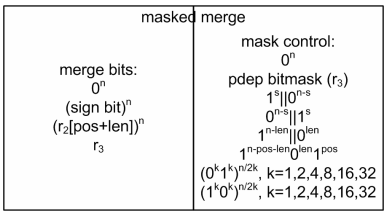

Figure 11. Masked Merge Block

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.

Figure 11 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 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 and
- various strings of "1"s and "0"s for shifts, extract, deposit and mix, as described in Section 3.2.

4.1. Comparison to Barrel and Log 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 (Figure 12). 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.

A second popular shifter architecture is 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 version as is from the previous stage (Figure 13). The advantages are that only $n\times\lg(n)$ or $n\times\log_4(n)$ elements are required and that 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 are performed by implementing a single, say right, shifter; 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.

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 also implemented a 64-bit log shifter using 3 stages of 4:1 multiplexers.

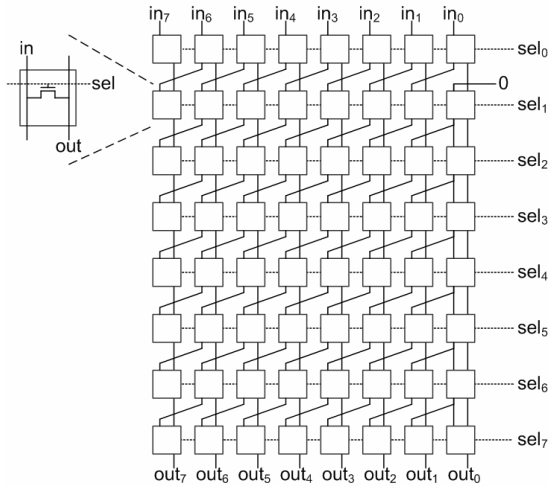

Figure 12. 8-bit Barrel Shifter with detail of pass transistor switch element

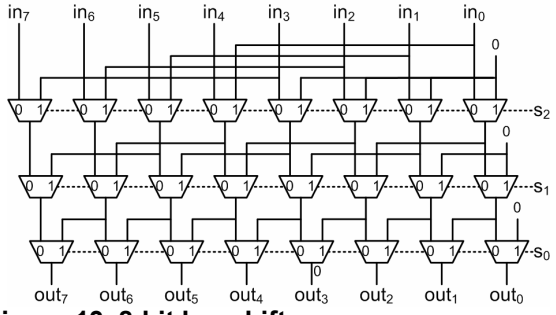

Figure 13. 8-bit log shifter

We used the method of logical effort [10] to compare the delay along the critical paths for the barrel shifter, the log shifter and the inverse butterfly shifter. 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 Shift-Permute functional unit, we consider the delay from the input to the rotation control bit generator (Figure 8) through the output of the inverse butterfly circuit. We assume that any preprocessing of the shift amount and post-processing for masked merge is approximately equal for the two designs.

The critical path for the barrel shifter extends through the 3:8 decoder to the select lines of the first shifter, to the input to the second shifter, to the output. For the log shifter, the critical path is through the first stage select lines, to the second stage input, to the third stage input, to the output. The critical path for our inverse butterfly shifter extends through cb_2 generation, to the stage 2 select lines, to the remainder of the inverse butterfly network. 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. Using our new shifter may adversely affect cycle time for processors that use the log shifter implementation depending on the slack in the shifter critical path.

5. Conclusions

We have described a new basis for Shifter and Mix functional units based on the inverse butterfly datapath (with optional butterfly datapath). Our new Shift-Permute functional unit is a much more powerful functional unit: it performs the existing Shifter operations and multimedia subword-permutation operations (shift, rotate, extract, deposit and mix operations) as well as the newly proposed advanced bit manipulation instructions (bfly, ibfly, parallel extract and parallel deposit) [1].

We have shown how to determine the control bits to configure the inverse butterfly (and butterfly) circuits to perform rotations; this is given by a simple recursive function of the shift amount. Furthermore we have shown how to compute the merge bits and mask control for turning rotations into shifts, extract, deposit and mix operations, and how the mask control uses the same recursive function.

Additionally, we have compared the complexity of our new functional unit to that of the classic barrel shifter and log shifter using logical effort. Our proposed Shift-Permute unit has comparable latency to that of the barrel shifter and is 19% slower than the log shifter while supporting a much more powerful set of advanced permutation operations (as given in Table 2) as well as existing shifter and mix operations (as given in Table 1.) There is essentially no area overhead since the existing Shifter unit is replaced. Since the mix operation is currently supported as a separate multimedia functional unit in [3], we may have *reduced* area requirements by replacing two existing functional units (Shifter, Multimedia-mix) with one new unit.

In summary, our proposed new Shift-Permute functional unit enables processors to support advanced bit permutations efficient, without any area overhead and with only minor or no cycle-time latency impact. We recommend its use in future microprocessors, and hope to have stimulated further research into optimal implementations of shifters and important new bit manipulation operations.

6. References

[1] Yedidya Hilewitz and Ruby B. Lee, "Fast Bit Compression and Expansion with Parallel Extract and Parallel Deposit Instructions", *Proceedings of the IEEE 17th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 65-72, September 11-13, 2006.

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

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

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

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

[6] Zhijie Shi, Xiao Yang and Ruby 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.

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

[8] Ruby B. Lee, Zhijie Shi, and Xiao Yang, How a Processor can Permute n bits in O(1) cycles, *Proceedings of Hot Chips 14 - A Symposium on High Performance Chips*, August 2002.

[9] F. Thompson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, 1992.

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