Comparing Fast Implementations of Bit Permutation Instructions

Yedidya Hilewitz[1], Zhijie Jerry Shi[2] and Ruby B. Lee[1]

Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA, {hilewitz, rblee}@princeton.edu

Abstract - **Recently, a number of candidate instructions have been proposed to efficiently compute arbitrary bit permutations. Among these, GRP is the most attractive, having utility for other applications in addition to permutation such as sorting and having good inherent cryptographic properties. However, the current implementation of GRP is the slowest of the candidates; BFLY, on the other hand, is the fastest. In this paper, we examine the possibility of executing GRP on a butterfly or an inverse butterfly network.**

I. Introduction

Bit permutation operations are very useful in the design of block ciphers. However, current microprocessors do not directly support arbitrary permutations and thus such operations are slow when implemented with software instructions [1]. Recently, a number of candidate instructions have been proposed in order to efficiently compute arbitrary permutations on a general purpose microprocessor. These include BFLY, IBFLY [2, 3], CROSS [1, 4], OMFLIP [1, 5], PPERM [1], SWPERM with SIEVE [6] and GRP [1, 7].

The fastest proposed permutation instruction is the BFLY/IBFLY pair. These instructions route the data bits through complete butterfly and inverse butterfly networks, respectively. BFLY and IBFLY are the fastest in two senses. First, only a single BFLY instruction followed by an IBFLY instruction is required to compute any arbitrary permutation. Thus, any one of the n! possible permutations of n bits can be performed in two instructions using BFLY and IBFLY; the other permutation instructions require a sequence of lg(n) instructions to achieve any arbitrary permutation.

More importantly, the BFLY and IBFLY instructions have simple circuit implementations that exhibit a lower latency than CROSS, OMFLIP or GRP [2, 8, 9]. This latency is less than that of an ALU of the same width. Since a processor's cycle time is usually determined by the ALU latency, each of the BFLY and IBFLY instructions can be accomplished in one cycle.

GRP, on the other hand, has the longest latency of these proposed bit permutation instructions. GRP divides its data bits into two subsets depending on the corresponding control bits: if a control bit is 1, that data bit is grouped right (an *R bit*); if a control bit is 0, that data bit is grouped left (an *L bit*) (Fig. 1). The relative ordering within each subgroup is maintained. GRP is slower than BFLY or IBFLY. It requires up to lg(n) instructions to compute an arbitrary permutation. Furthermore, the current GRP implementation utilizes a series of linear shift network that has a much greater latency than that of BFLY or IBFLY, taking two to three cycles to compute (see section V for a detailed discussion of original GRP circuit) [8, 9].

However, there is an impediment to using BFLY and IBFLY instructions – the need to supply $n$lg(n)/2 control bits in addition to the n bits to be permuted for each of these instructions. Thus, for n=64 in a 64-bit microprocessor, four 64-bit source registers are required for BFLY or IBFLY, while the typical processor architecture supports only two source operands per instruction. On the other hand, GRP has additional desirable features aside from its use in performing arbitrary permutations – GRP can be used to perform hardware radix sorting [10] and has strong inherent differential cryptographic properties [11].

Consequently, we examine whether the GRP operation can be implemented on the significantly faster butterfly or inverse butterfly networks. While GRP would still require lg(n) instructions to compute an arbitrary permutation, a faster implementation would make GRP much more attractive.

We show that GRP cannot be performed on a butterfly or inverse butterfly network but that two inverse butterfly networks may be used to group the R bits and L bits in parallel. The outputs of the two networks are merged to complete the GRP operation. We show the circuit that dynamically decodes the n GRP control bits to the $n$lg(n)/2 IBFLY control bits. This circuit has significant latency and offsets the speed of the faster inverse butterfly network. However, the new design is a viable alternative to the original GRP circuit. In addition, a fixed GRP operation can bypass the decoder and directly use the fast IBFLY network; the "decoding" can be done statically by the compiler.

The paper is organized as follows: Section II examines the possibility of performing GRP on a butterfly or inverse

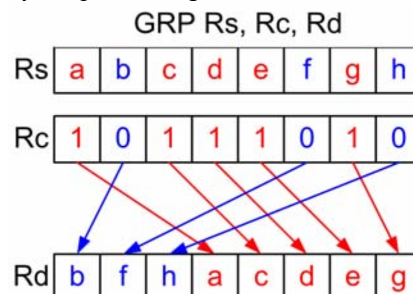

Fig. 1: GRP operation on 8 bits. `bfh` are L bits; `acdeg` are R bits. Instruction is **`GRP Rs,Rc,Rd`**, where Rs is the source register, Rc the control register and Rd the destination register.

[1]This work is supported in part by NSF ITR 0326372; Yedidya Hilewitz holds a Hertz Foundation/Princeton Fellowship and an NSF Fellowship.

[2]Z. J. Shi is now with the Computer Science and Engineering Department, University of Connecticut, Storrs, CT 06269 USA, zshi@engr.uconn.edu

Yedidya Hilewitz, Zhijie Jerry Shi, and Ruby B. Lee, "Comparing Fast Implementations of Bit Permutation Instructions," *Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers*, November 2004.

butterfly network. Section III discusses the n:$n$lg(n)/2 decoder and specifies the decoder circuit. Section IV analyzes the critical path of the circuit using the method of logical effort. Section V compares the new GRP implementation to the original one. Section VI concludes the paper.

II. Analysis of GRP on BFLY and IBFLY Networks

We define two new operations: GRPR selects the R bits moving them to the right of the result register and zeros the L bits. GRPL selects the L bits moving them to the left of the result register and zeros the R bits. Then, the GRP operation can be considered as the combination of GRPR and GRPL. We believe that GRPR may itself be a useful instruction that functions as a generalized EXTRACT, suited to gathering and right justifying an offset or table index that may have bits scattered across a word.

GRPR and GRPL are similar to the packing operation described in the context of packet routing networks [12]. The packing operation can be performed on an inverse butterfly network, but not on a butterfly network. Furthermore, the GRP operation involving the simultaneous packing into two groups is not possible with one pass through either an inverse butterfly network or butterfly network. We propose replicating the inverse butterfly network, performing GRPR and GRPL in parallel and combining the results (Fig. 2), similar to the approach with linear shift networks taken for the original GRP circuit [8, 9].

We can show that GRP cannot be performed on a butterfly or inverse butterfly circuit. Also, GRPR or GRPL cannot be performed on butterfly due to path conflicts (contention for a multiplexer node or wire in Fig. 3). Butterfly and inverse butterfly networks are composed of a number of stages, where at each stage two bits in a pair of bits are either swapped or passed through to the next stage. In this paper, a control bit of “0” indicates swapping and “1” indicates passing through for each pair of bits. Each successive stage is composed of two disjoint subnetworks, each subnetwork a butterfly or inverse butterfly network that is half the size. As these subnetworks are disjoint, there exists a unique path from any input to any output. If two inputs are to be simultaneously routable, the unique paths to their respective outputs must be through different subnetworks; otherwise, a path conflict will exist.

To show GRP cannot be achieved on the butterfly network, consider the case shown in Fig. 3. Bits d and h are the only bits in the R group and are destined for positions 1 and 0, respectively. These bits are paired in the first stage and the paths to their outputs are both through the lower subnetwork. Thus they are not routable conflict-free. For the inverse butterfly network, consider the case shown in Fig. 4. Bit c is destined for the even subnetwork (bit position 6 in result). Bit d is also destined for the even subnetwork (bit position 2 in result). As they must both be routed to the even subnetwork after stage 1, there is a path conflict.

To show GRPR also cannot be achieved on a butterfly circuit, we can just use the same counterexample as for GRP (Fig. 3).

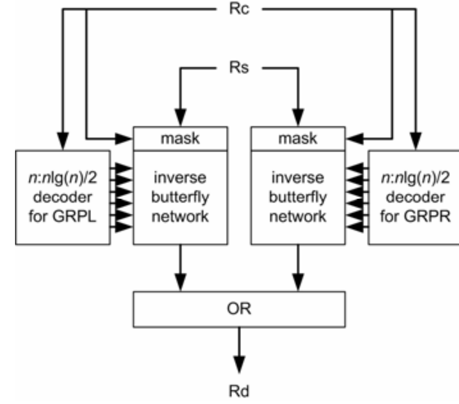

Fig. 2: Overview of GRP operation using parallel IBFLY circuits.

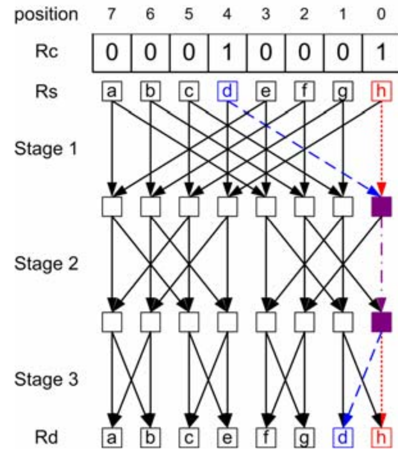

Fig. 3: Example of conflicting greedy paths on 8-bit butterfly network when attempting to route GRP operation with Rc = 00010001.

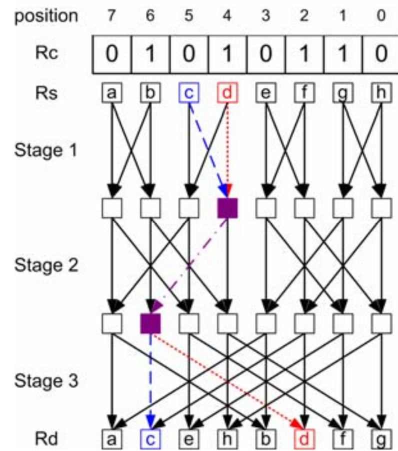

Fig. 4: Example of conflicting greedy paths on 8-bit inverse butterfly network when attempting to route GRP operation with Rc = 01010110.

GRPR on inverse butterfly. Both GRPR and GRPL can be achieved using the inverse butterfly circuit. Fig. 5 shows an 8-bit example. We provide the basis of an inductive proof by first describing how GRPR is done at stage $k+1$, assuming both the right half circuit and the left half circuit through stage k have performed GRPR on their respective data bits. The result from the left half circuit of stage k is then rotated right by the number of zeroed L bits in the right half. At level k+1,

the bits in the left half that wrap are swapped into the most significant bits of the right half, via the inverse butterfly operation at this stage. This completes the GRPR operation for stage *k+1*.

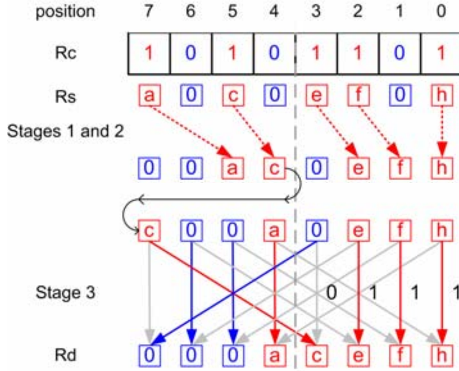

Fig. 5: Example GRPR operation on an 8-bit inverse butterfly network. The output from stage 2 is the GRPR operation within the left and right parts: `00ac`, `0efh`. The left part is rotated right by the number of zeros in the right part: `00ac`→`c00a`. Bit c is then swapped (control bit = "0") into the right half to produce the output `000acefh`.

III. Decoding the GRP Control Bits

A. Decoder Description

We first introduce some terminology. The inverse butterfly network is composed of *subcircuits*, where a subcircuit is a set of overlapping switches (for example, bit positions 0–3 in stage 2 of Fig. 4); the switch either passes through or swaps the bits based on whether the control bit of the switch is "1" or "0", respectively. Stage i has $n/2^i$ subcircuits each 2^i bits wide. The right half of the inputs to the switches of a subcircuit is called the *right part* of the subcircuit (bit positions 0–1 in stage 2 of Fig. 4) and the left half is called the *left part* (bit positions 2–3 in stage 2 of Fig. 4). Each of the right and left parts of stage i is 2^{i-1} bits wide.

The method of computing the $n\lg(n)/2$ control bits for the inverse butterfly network from the n GRP control bits follows the procedure described above. In order for any subcircuit to perform GRPR, the R bits in the right part are passed-through and the zeroed L bits are swapped out (to swap in the R bits from the left half) as shown in the example in Fig. 5. The control bits indicate this by taking the value 1 for the k least significant switches and the value 0 for the remaining switches, where k is the number of R bits in the right part. This bit pattern is precisely the GRPR of the original control bits of the GRP instruction corresponding to right part of the subcircuit. This uses the GRP control bits as both the data and the control bits. For example, observe in Fig. 5 that the GRP control bits for the right part are "1101" and GRPR("1101", "1101") = "0111", the inverse butterfly control bits for stage 3. This pattern is also equivalent to a unary encoding of the population count of the ones (POPCNT) in the right part GRP control bits.

The unary encoding of the POPCNT can be achieved using a "left rotate and complement on wrap (LROTC)" operation (Fig. 6). This operation is a standard left rotation except that bits are complemented whenever they wrap. A LROTC operation of the zero string "0…0" by the POPCNT will produce a one in the least significant bit for each rotation by one position, thus expressing a unary encoding of the value.

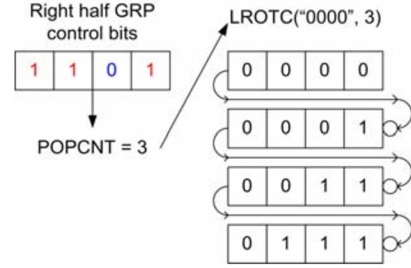

Fig. 6: Left rotate and complement on wrap of "0000" by POPCNT("1101") = 3 produces the result "0111."

This method determines the control bits for a subcircuit in isolation. However, when considering a subcircuit in context of the entire inverse butterfly network, each subcircuit, except for the rightmost subcircuit in each stage, feeds a left part subcircuit in some subsequent stages. Left part subcircuits perform GRPR rotated right by the number of L bits in their corresponding right part subcircuits. This is the population count of the zeroes (ZEROCNT) in the right part GRP control bits. This "right rotate by ZEROCNT" operation can be replaced by a "left rotate by POPCNT" since ZEROCNT+POPCNT equals the total number of bits in the right part, which is equal to the number of bits rotated in the left part (see Fig. 5).

To achieve the desired rotation of the data bits, the control bits for GRPR specified above must be modified. In general, to perform a rotation of a permutation π by m positions on an inverse butterfly network, the right part circuit and left part circuit through stage k rotate their respective parts of π by m positions. In order to complete the rotation at stage k+1, the control bits at that stage are also rotated by m positions in order to keep a control bit associated with its paired data bits; however, the control bits are complemented upon wrap, reversing the routing of the data bits (Fig. 7). Thus a rotate and complement (ROTC) operation of the control bits is needed for a rotation of the data bits. Note that in order to rotate π by m positions at a given stage, the same rotation must have been performed at the previous stage. Thus the rotation is propagated up and performed at each stage of the inverse butterfly network.

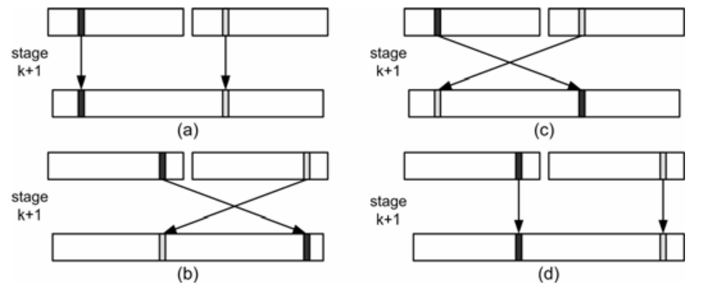

Fig. 7: Performing rotation at level k+1 assuming rotation through level k. Fig. 7a shows the case that two bits are not swapped in the original permutation. Fig. 7b shows that, if these bits are rotated m positions and wrap, completing the rotation requires swapping the bits. Fig. 7c and 7d show the case where the two bits are swapped in the original permutation.

Consequently, the left rotation of the left part data bits is accomplished via LROTC of the inverse butterfly control bits of all previous stages of the left part by the POPCNT of the right half GRP control bits. Thus the inverse butterfly control bits for a subcircuit are generated by an LROTC of "0…0" by the POPCNT of the right part GRP control bits followed by an LROTC of that pattern by the POPCNT of all the right parts from the subsequent stages. The composition of these LROTC operations can be reduced to a single LROTC of "0…0" by the total POPCNT extending from the most significant bit of the right part circuit down to position 0. To generate the inverse butterfly control bits for all stages, we need to calculate all such POPCNT values. We calculate these POPCNT values in parallel, using a *parallel prefix popcount* circuit.

Thus, we present the algorithm to decode the n GRP control bits into the $n\lg(n)/2$ inverse butterfly control bits. This algorithm is implemented in hardware for GRP instructions with dynamically determined control bits, and is utilized by the compiler for GRP instructions using static control bits.

Algorithm 1: To generate the $n\lg(n)/2$ inverse butterfly control bits from the n GRP control bits.

Let $x \| y$ indicate the concatenation of bit patterns x and y. control[x] refers to bit x of the original GRP control bits. sel is a $\lg(n) \times n/2$ bit matrix that represents the inverse butterfly control bits. LROTC(a, rot) is a "left rotate and complement on wrap" operation, where a is the input and rot is the rotation amount. PPC[a] is the prefix POPCNT of position a, i.e., POPCNT of the GRP control bits from bit 0 to bit a (we use POPCNT to refer to the population count of a field and PPC[a] to refer to the prefix population count with respect to position a). 0^k indicates a bit-string of k zeros.

1. Calculate the prefix popcounts:
   PPC[0] = control[0]
   For i = 1, 2, …, n–2
      PPC[i] = PPC[i–1] + control[i]
2. Calculate the inverse butterfly control bits for each subcircuit by performing LROTC("0…0", PPC[m]), where m is the most significant bit of the right part of the subcircuit:
   sel = {}
   For i = 1, 2, …, lg(n) //for each stage
      $k = 2^{i-1}$ //number of bits in right part circuit
      For j = 0, 1, …, $n/2^i-1$ //for each subcircuit
         temp = LROTC(0^k, PPC[$j*2^i + k - 1$])
         sel[i] = temp $\|$ sel[i]

Step 2 is more intuitively understood by referring to the inverse butterfly circuit structure in Fig. 4. This circuit has lg(n) = 3 stages. For stage 1, k = 1, and j runs through the values 0, 1, 2, 3. That is, there are 4 subcircuits in stage 1, and the right part of each subcircuit is 1 bit wide. We take the PPC of the most significant bit in the right part of each of these 4 subcircuits; this is the PPC of bits 0, 2, 4, 6. For stage 2, k = 2, and j runs through 0, 1. That is, there are 2 subcircuits in stage 2, and the right part of each subcircuit is 2 bits wide. We take the PPC of bits 1 and 5. For stage 3, k = 4, and j runs through just 0. That is, there is just 1 subcircuit in stage 3, and the right part is 4 bits. We take the PPC of bit 3. Hence, we only need the 1-bit PPC values of bits 0, 2, 4 and 6, the 2-bit PPC values of bits 1 and 5, and the full 3-bit PPC value of only bit 3.

Note, the full POPCNT value (of lg(n) bits) is not needed except for the last stage. In earlier stages, only the least significant bits are needed. Specifically, the number of bits of the POPCNT values required to generate the control bits for the inverse butterfly network is equal to the stage number. The $n/2$ right part circuits in the first stage require only the least significant bit of the POPCNT values. Also, since 1-bit wide POPCNT values are the same as the outputs of the 1-bit LROTC operations (LROTC("0", 0) = "0" and LROTC("0", 1) = "1"), these 1-bit LROTC operations can be eliminated, thus simplifying the implementation (Fig. 8).

Thus, the $n{:}n\lg(n)/2$ hardware decoder that realizes Algorithm 1 consists of two stages: 1) a circuit that, in parallel, counts the number of GRP control bits that are "1"s from position 0 to every position (except n–1). This circuit is a parallel prefix POPCNT unit; 2) For each inverse butterfly stage i, i>1, a 2^{i-1}-bit LROTC (left rotate and complement) circuit for the $n/2^i$ i-bit POPCNT values to generate the $n/2$ control bits for that stage. Fig. 8 presents the block diagram of the GRPR circuit for n = 64 with details of the decoder. The decoder produces lg(64)*32 = 192 control bits.

B. GRP Circuit

The GRP circuit is composed of parallel circuits that perform GRPR and GRPL with the results ORed together to produce GRP (Fig. 2). The circuit that produces GRPR is shown in Fig. 8. The circuit for GRPL is similar, except that the decoder is the mirror image of that for GRPR and that the control bits are inverted. The decoder operates in parallel to the routing. The control bits of the earlier stages are calculated first and the routing through the first stages of the inverse butterfly network is in parallel with the calculation of the control bits for the later stages.

The decoder consists of 2 stages: a parallel-prefix population counter followed by LROTC circuits for each stage. The POPCNT values are generated by a parallel prefix network with carry-save addition being the operation at each node (Fig. 9). The architecture resembles a radix-3 Han-Carlson network. The radix-3 stems from the carry-save addition. The resemblance to a Han-Carlson network stems from the replication of the basic network fragment depicted in Fig. 9 for only odd i. The even counts are all 1-bit wide and are deferred until the final stage as they are simply an XOR with the least significant bit of a neighboring count.

The first stage (PPC1) of the circuit divides its 8 input bits into sets of 3, 2 and 3, and sums these sets producing three 2-bit sums. The second stage (PPC2) adds these three sums to produce a redundant sum of 8 bits represented as a 2-bit sum and a 3-bit carry (with the least significant bit of the carry being 0).

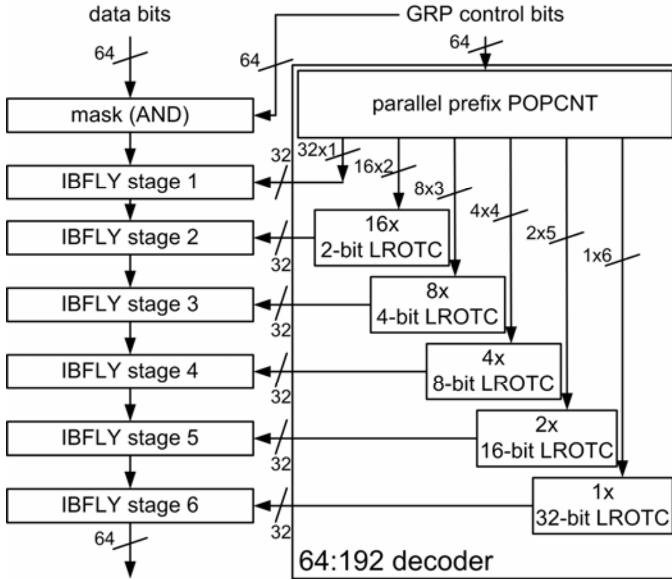

Fig. 8: 64-bit GRPR circuit with decoder detail.

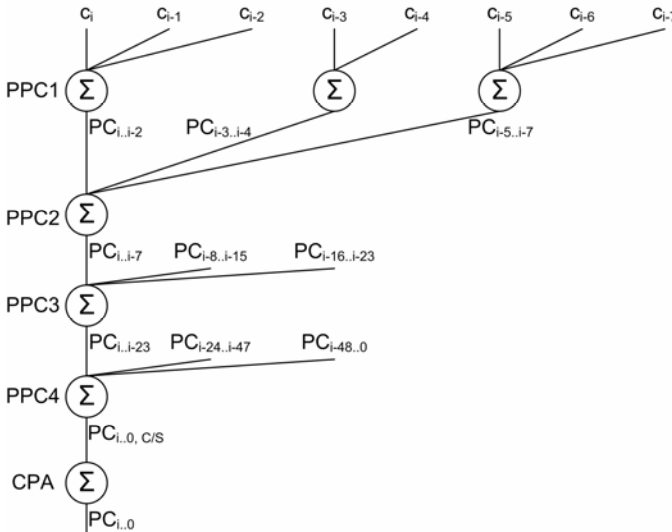

Fig. 9: The basic network fragment of the parallel prefix popcount circuit, with the network being truncated for small *i*. $PC_{i..j}$ refers to the POPCNT of positions *i* to *j*.

The third stage (PPC3) adds to three redundant sums of 8 bits to produce the sums of 24 bits. This stage is a compound stage as there are actually six input operands – three sums and three carries. Dual carry-save adders add the three sums and three carries from the previous stage thereby reducing the six input operands to four; a concatenation of the sum of the sums and the carry of the carries reduces the four operands to three; and a second carry-save stage reduces the sum to two 4-bit operands (with the two least significant bits of the carry being zero).

The fourth stage (PPC4) of the circuit adds the appropriate partial sums to produce a single redundant carry/save sum of the POPCNT. This stage input has two or three sums and one, two or three carries depending on *i*. Thus this stage consists of a 6:2, 5:2, 4:2 or 3:2 adder as appropriate. The final stage is a carry-propagate adder (CPA) that produces the final POPCNT result. The POPCNT values are only calculated to the required bit lengths as described above. Note that computing POPCNT values may require fewer than 4 PPC stages for small *i* or truncated values. Also note that as the low bits of the carries entering the final PPC stage are zeros, the low bits of a POPCNT value may be fully calculated before the CPA stage.

The output from the population counters controls the LROTC circuits. Each LROTC circuit can be realized as a barrel rotator modified to complement the bits that wrap around (Fig. 10a). However, while a standard 2^k-bit rotator has *k* stages and control bits, this shifter has *k*+1 stages and control bits. The final stage selects between its input and the complement, as the bits wrap 2^k positions, back to the same spot. Propagating the zeros at the input can greatly simplify the circuit (Fig. 10b). The outputs from the rotate circuits are routed directly to the appropriate inverse butterfly switches they control.

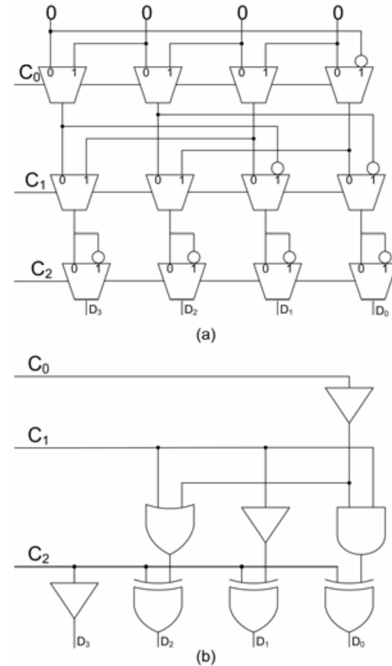

Fig. 10a: Barrel rotator implementation of 4-bit LROTC circuit.
Fig. 10b: Simplified circuit obtained from propagating zeros at input.

IV. Implementation Analysis

We now perform the logical effort analysis of the critical path of the 64-bit GRP circuit. Logical effort is a technology-independent method to estimate the delay of a CMOS circuit [13]. The result of a logical effort analysis gives the estimated delay in units of fan-out of four (FO4), the delay of an inverter that drives four identical inverters. For a full discussion of the method of logical effort see Appendix A. The circuit is assumed to drive a copy of itself (H = 1). Table I shows the equivalent capacitance for a wire spanning the width of the standard cells from the TSMC 90nm library we use in the circuit [8, 14].

TABLE I: EQUIVALENT CAPACITANCE OF WIRE SPAN

Cell	Equivalent load	Cell	Equivalent load
MUXI	0.33	FA	1.04
2-XOR	0.38	HA	0.58
3-XOR	0.63	Cell height/ 9 routing tracks	0.33
4-XOR	0.96		

The carry-save adders in the POPCNT units consist of parallel full adders (FA). Each FA is composed of an asymmetric 3-input XOR gate and an asymmetric 3-input inverting majority gate. The XOR gate has logical effort $g_{ax*} = 12$, $g_{bx*} = 6$ and $g_{cx*} = 6$ for the three input bundles, where a bundle is composed of the complement and uncomplemented input signal, and the majority gate has logical effort of $g_{am'} = 2$, $g_{bm'} = 4$ and $g_{cm'} = 4$ for the three complemented inputs. Both gates have a parasitic delay $p_m = p_x = 6$ [13]. In order to limit the effort of any single input, the XOR input with the highest effort, *ax**, is tied to the majority input with the lowest effort, *am'*, yielding an effort $g_{a*} = 14$ for the bundle and $g_{a'} = 8$ for the complemented input, and $g_{b*} = g_{c*} = 10$ and $g_{b'} = g_{c'} = 7$. Each FA is driven by a 2:1 fork stage that generates the complement and uncomplemented inputs. We assume each fork inverter is 4x drive strength.

The inverse butterfly network is composed of n 2:1 inverting multiplexers (MUXI) at each level. The logical effort of any MUXI input is 2 and of any select signal is 4, and the parasitic delay of the MUXI is 4 [13]. Other gates have logical effort and parasitic delays as in [13].

The various paths through the circuit tradeoff time spent decoding the bits and time propagating through the inverse butterfly network. The 1-bit POPCNT values, which require the least effort to compute, control the first stage of the inverse butterfly network. Wider POPCNT values control later stages and thus paths through those counts experience smaller delay through the inverse butterfly network. We calculated the path delays for the most significant position of each POPCNT length: $i = 62$ for the 1-bit count, $i = 61$ for the 2-bit count, $i = 59$ for the 3-bit count, $i = 55$ for the 4-bit count, $i = 47$ for the 5-bit count and $i = 31$ for the 6-bit count. The critical path is through the most significant bit of the 4-bit count PPC[55], the final stage of the 8-bit LROTC circuit and the last three stages of the inverse butterfly network (IBFLY4 thru IBFLY6). Specifically, the path through the POPCNT circuit consists of 6 FAs to reduce the input to two operands followed by a 2-bit CPA with no carry out. The results are summarized in Table II.

The total effort can be calculated as:

$$F = GBH = \Pi g^* \, \Pi b = 7.1 \times 10^9 \times 5.0 \times 10^7 = 3.5 \times 10^{17}$$

The optimal number of stages is:

$$N = \log_{3.6} F = 31$$

As there are 23 stages, eight inverters need to be added along the path to drive the large loads. The delay of the path can be calculated as:

$$D = N \times F^{1/N} + P = 31 \times F^{1/31} + (70 + 8) = 192.2$$

When divided by five, the delay is about 38.4 FO4. The latency can be decomposed as 27.8 FO4 through the decoder (24.7 FO4 through the parallel prefix POPCNT unit), 7.4 FO4 through the second half of the inverse butterfly network and 3.2 FO4 due to branching to the GRPR and GRPL circuit and combining the results. The GRP on IBFLY latency of 38.4 FO4 is much greater than the 13.0 FO4 latency of the original inverse butterfly network due to the high latency through the decoder. This latency can be attributed to the high delay of full adders. Each full adder level contributes 2.5–3.0 FO4 delay. Additionally, the branching required by the parallel prefix architecture together with the large equivalent capacitance for a wire spanning a full adder causes a large branching effort at each stage of the unit.

TABLE II: LOGICAL EFFORT OF GRP ON IBFLY

Stage	*Gate*	*Load*	***b***	***g***	***p***	***# stages***
PPC1	FA[a]	1 FA + track	(2*4+6.61)/4	8	7	2[c]
PPC2	FA[b]	3 FA + track	(6*4+34.35)/4	7	7	2[c]
PPC3	FA[a]	FA	2	8	7	2[c]
	FA[a]	2 FA + track	(4*4 + 28.79)/4	8	7	2[c]
PPC4	FA[b]	FA	2	7	7	2[c]
	FA[b]	HA	(4+4/3)/(4/3)	7	7	2[c]
CPA	HA Carry	Sum (XOR)	1	4/3	2	1
	SUM	7 XOR+ INV+track	(7*4+1+2.73)/4	4	5	2[c]
LROTC8	2-XOR	2 MUXI.sel + track	(2*4+3.54)/2	4	5	2[c]
IBFLY4	MUXI.sel	2 MUXI.in + Track	(2*2 + 7.08)/2	2	5	2[c]
IBFLY5	MUXI.in	2 MUXI.in + Track	(2*2 +14.17)/2	2	4	1
IBFLY6	MUXI.in	NOR+Track	(5/3+13.0)/ (5/3)	2	4	1
	NOR	INV	1	5/3	2	1
	INV	6 FA + 2 HA + buffer + Track	(12*4+2*16/3+ 2+86.5)/4	1	1	1
Total			7.1×10^9	5.0×10^7	70	23

[a] *The critical path through these adders is through a'.*
[b] *The critical path through these adders is through b' or c'.*
[c] *Includes complement generation stage.*

V. COMPARISON TO ORIGINAL GRP CIRCUIT

A. Original GRP Circuit

The original GRP circuit [8, 9] is similar in structure to Fig. 2, with the routing network a linear shift network and the decoder also a similar linear shift network. The basic operation computes GRPR of n bits using the GRPR of the right half $n/2$ bits and the left half $n/2$ bits. The linear shift network produces $n/2+1$ outputs. These outputs are the shifts of the left half GRPR pattern onto the right half pattern by k positions, with k equal to the possible number of zeroed L bits in the right half and thus ranging from 0 to $n/2$. Another circuit produces a one-hot encoding of the actual number of zeroed L bits. The one-hot encoding acts as the select signals to a bank of transmission gates, passing to the next stage only the output that corresponds to the correct shift amount. The circuit that produces the one hot encoding is an adder of the one hot encodings of the number of L bits in the two substages that feed the right half stage. A one-hot encoding adder is also a linear shift network with one encoding the select and the other the data input shifted onto the all zeros pattern.

The original analysis of the circuit in [8, 9] did not consider that the linear shift network actually forms an $n/2+1$:1

multiplexer, and thus did not account for the high capacitance present on the output nodes of the transmission gates. A simple fix is to split this wide mux into a multi-stage mux, each stage composed of smaller muxes. The incoming select signals remain valid for the first level of muxes and the later stage select signals are computed using simple logical gates – the select signal of a second stage leg is the NOR of the select signals from all the first stage legs whose output is input to that leg. This method generates a one low encoding of the select signals for the second stage. The later stage signals are calculated similarly, alternating NAND and NOR to ensure that the select signals are always one hot or one low. The revised latency of this original GRP circuit is 38.1 FO4.

B. Comparison of Linear Shift GRP vs. GRP on IBFLY

The GRP on IBFLY latency of 38.4 FO4 is comparable to 38.1 FO4 latency of the original GRP with liner shift circuits. This difference is approximately 1% and given the coarseness of the wire load model, it is difficult to attribute any significance to the difference. Given a typical microprocessor cycle time of 14-24 FO4 [15], either circuit has a 2 or 3 cycle latency.

We synthesized both circuits using a TSMC 90nm standard cell library [14]. Table III compares the latency from logical effort estimates, the latency from synthesis and the area from synthesis. The latency from synthesis verifies the logical effort result, with both circuits having comparable latency. However, the area results clearly favor the GRP on IBFLY implementation.

TABLE III: RESULTS OF LOGICAL EFFORT AND SYNTHESIS

Circuit	Latency, Logical Effort	Latency, Synthesis	Area (NAND gates)
Original GRP	38.1 FO4	39.1 FO4	68.6K
GRP on IBFLY	38.4 FO4	37.3 FO4	19.7K

VI. CONCLUSION

In this paper, we examine the possibility of performing the GRP operation on a butterfly or inverse butterfly network. We show that GRP cannot be routed on either network but that GRPR and GRPL can be routed on the inverse butterfly network. We design a decoder circuit that can produce the required $n\lg(n)/2$ butterfly control bits from the n GRP control bits. However, the latency through this decoder is quite large and thus the benefit of the fast routing inverse butterfly network is negated. The overall latency, however, is comparable to that of the original GRP circuit and this circuit has the benefit of using a more general purpose routing circuit. Furthermore, if we wish to perform a static GRP operation, the compiler can decode the bits in advance and produce control bits for the fast inverse butterfly network directly, bypassing the hardware decoder (this requires adding a bypass multiplexer in Fig. 8).

Alternatively, we could remove the GRPL circuit and add a butterfly network thereby enabling arbitrary permutations to be computed using BFLY and IBFLY and at the same time support the GRPR functionality. Such a scheme would have a substantial savings in area over the proposed circuit, which is already significantly smaller than the original implementation, and the delay of multiplexing control signals to the inverse butterfly network would be offset by the removal of the branching to and combining of GRPR and GRPL. For these reasons, we believe that the GRP on an inverse butterfly circuit is the preferred implementation of the GRP instruction.

ACKNOWLEDGEMENT

The authors wish to thank David Harris of Harvey Mudd College for his time and valuable suggestions.

REFERENCES

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

[2] 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.

[3] 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*, June 2003.

[4] Xiao Yang, Manish Vachharajani and Ruby B. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks," *Proceedings of Media Processors 1999 IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, pp. 80-86, January 2000.

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

[6] John P. McGregor and Ruby B. Lee, "Architectural Techniques for Accelerating Subword Permutations with Repetitions," IEEE Transactions on Very Large Scale Integration Systems, vol. 11, no. 3, pp. 325-335, June 2003.

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

[8] Zhijie Jerry Shi and Ruby B. Lee, "Implementation Complexity of Bit Permutation Instructions," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[9] Zhijie Shi, "Bit Permutation Instructions: Architecture, Implementation, and Cryptographic Properties," PhD Thesis, Princeton University, June 2004.

[10] 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.

[11] R. B. Lee, R. L. Rivest, M.J.B. Robshaw, Z.J. Shi, and Y.L. Yin, "On Permutation Operations in Cipher Design," *Proceedings of the International Conference on Information Technology (ITCC)*, vol. 2, pp. 569 - 577, April 2004.

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

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

[14] Taiwan Semiconductor Manufacturing Corporation, TCBN90G: TSMC 90nm Core Library Databook, October 2003.

[15] Francois Labonte, "Microprocessors through the Ages," available online: http://www-vlsi.stanford.edu/group/chips_micropro.html, 23 Nov 2004.

APPENDIX A

Logical effort is a technology-independent method to estimate the delay of a CMOS circuit [13]. The method also aids in determining the optimum number of logical stages used

and in sizing transistors in logic gates. It uses the following concepts:

Logical effort g : The ratio of input capacitance of a logic gate to that of an equal drive strength inverter.

Electrical effort h: The ratio of output capacitance of a gate to its input capacitance.

Branching effort b: The ratio of total capacitive load on one logic gate's output to the gate capacitance of the next gate on the path examined.

Parasitic delay p: The total diffusion capacitance on the output node of a gate relative to that of a minimum-sized inverter.

The delay of a single gate can be calculated as:

$$d = gh + p. \qquad \text{(A1)}$$

To find the delay along a path, we first calculate the total path effort:

$$F = GBH \qquad \text{(A2)}$$

where $G = \Pi g$, $B = \Pi b$, and $H = \Pi h$. Πg means the product of the logical effort of all the gates along the path. Similarly, Πb is for the total branch effort and Πh for the total electrical effort. The total electrical effort $H = \Pi h$ reduces to the ratio of the output capacitance loading the last gate to the gate capacitance of the first gate on the path. Normally, we assume a circuit drives a copy of itself, so $H = 1$.

Once the path effort has been calculated, the ideal number of stages required to achieve the logical function can be estimated as:

$$N = \log_{3.6} F \qquad \text{(A3)}$$

where 3.6 is the stage effort achieving the best performance [13]. N is then rounded to the nearest integer that is reasonable for the path, and the effort delay for each stage can be calculated as:

$$\alpha = F^{1/N}. \qquad \text{(A4)}$$

α can be used to decide the transistor size in each stage along the path. The basic idea is to estimate the number of stages using the ideal stage effort α=3.6, and then calculate the real α from the estimated number of stages. Finally, the total delay of the path can be calculated as:

$$D = N\alpha + P, \qquad \text{(A5)}$$

where $P = \sum p$. The results in (A5) are in τ, the basic time unit used in logical effort, which is independent of process technology. Dividing D in (A5) by five gives the estimated delay in terms of fan-out of four (FO4), the delay of an inverter that drives four identical inverters.