Fast Bit Compression and Expansion with Parallel Extract and Parallel Deposit Instructions

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

Abstract

Current microprocessor instruction set architectures are word oriented, with some subword support. Many important applications, however, can realize substantial performance benefits from bit-oriented instructions. We propose the parallel extract (`pex`) and parallel deposit (`pdep`) instructions to accelerate compressing and expanding selections of bits. We show that these instructions can be implemented by the fast inverse butterfly and butterfly network circuits. We evaluate latency and area costs of alternative functional units for implementing subsets of advanced bit manipulation instructions. We show applications exhibiting significant speedup, 3.41× on average over a basic RISC architecture, and 2.48× on average over an instruction set architecture (ISA) that supports extract and deposit instructions.

1. Introduction

Operations on microprocessors are typically word, and more recently subword [1], oriented. However, many important applications benefit from bit-oriented operations. For example, arbitrary *n*-bit permutations take O(*n*) operations using basic instructions such as `and`, `shift` and `or` to move individual bits [2]. A few fixed permutations, such as in ciphers like DES, have been optimized by table lookup [2], still taking tens to hundreds of cycles, due to cache misses. Recent research showed that specialized bit-oriented instructions can permute bits in O(lg *n*) [2-4] or even O(1) operations [5,6]. For example for *n*=64, any one of 64! bit permutations can be achieved in 1 or 2 cycles by butterfly (`bfly`) and inverse butterfly (`ibfly`) permutation instructions [5,6]. Such speedup can enable previously difficult bit manipulation computations to be done much more efficiently.

This work was supported in part by the DoD and Intel. Yedidya Hilewitz is a Hertz Foundation Fellow.

This paper discusses another important class of bit-oriented operations involving selecting and compressing bits, and distributing bits according to different bit patterns. We call these parallel extract (`pex`) and parallel deposit (`pdep`) operations, respectively. `pdep` and `pex` can also be viewed as bit-level *scatter* and *gather* instructions. These operations are important in application domains such as bioinformatics, image processing, steganography, cryptanalysis and coding.

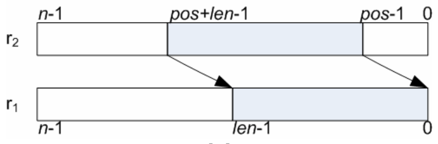
We present the architectural definition of these two novel bit instructions. We show how `pdep` can be implemented using the single-cycle butterfly network datapath. We evaluate alternative new functional units that implement useful subsets of these advanced bit manipulation instructions, and recommend one that is smaller than an ALU with shorter latency. Our performance results indicate that a processor enhanced with `pex` and `pdep` achieves a 5.2× maximum speedup, 3.41× on average, over a basic RISC architecture.

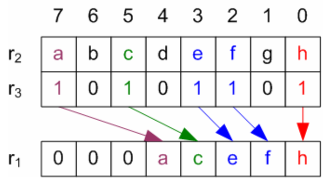
Section 2 describes the new `pex` and `pdep` instructions. Section 3 presents the ISA definitions. Section 4 discusses the implementation and different options for a new functional unit implementing advanced bit-oriented instructions. Section 5 describes applications of these instructions and section 6 their performance. Section 7 concludes the paper.

2. Parallel extract and parallel deposit

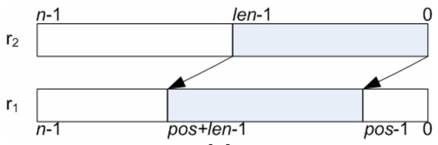
It is often necessary to select non-contiguous bits from data. For example, in pattern matching, many pairs of features may be compared. Then, a subset of these comparison result bits are selected, compressed and used as an index to look up a table. This selection and compression of bits is what a `pex` instruction does (Figure 1(b)). A `pex` instruction can also be viewed as a parallel version of the extract (`extr`) instruction [7, 8]. The `extr` instruction extracts a single field of bits from any position in the source register and right justifies it in the destination register. The `pex` instruction extracts multiple bit fields from the source register, compresses and right justifies them in the destination register. The selected bits are specified by a bit mask. Figure 1 compares `extr` and `pex`.

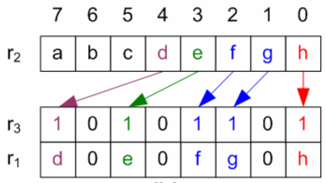
The inverse operations to `extr` and `pex` are `dep` and `pdep`, respectively. The deposit (`dep`) instruction takes a right justified field of bits from the source register and deposits it at any single position in the destination register. The parallel deposit (`pdep`) instruction takes a right justified field of bits from the source register and deposits the bits in different non-contiguous positions indicated by a bit mask. Figure 2 compares `dep` and `pdep`.



(a)


(b)
Figure 1. (a) extr $r_1 = r_2$, pos, len (b) pex.v $r_1 = r_2, r_3$



(a)


(b)
Figure 2. (a) dep $r_1 = r_2$, pos, len (b) pdep.v $r_1 = r_2, r_3$

Feasibility

It is not intuitively clear that `pex` and `pdep` are easy to implement, especially in a single cycle. We define a single cycle as the latency through an ALU of the same width, i.e., with operands the same size as `pex` and `pdep`.

The `pex` instruction can be considered half of a bit permutation primitive, `grp` [9], conserving its most useful properties. It performs a `grp`-like permutation of the source bits, where the bits that are not selected are zeroed out. In our past work on permutation circuits [6], we see that the fastest datapath for arbitrary permutations is the butterfly or inverse butterfly network circuit (Figure 3). Our prior work on accelerating the `grp` permutation instruction shows that it can also be implemented by two inverse butterfly networks operating in parallel, one implementing `grpl` and one implementing `grpr` [9]. Since the `pex` operation is like the `grpr` operation, it can be implemented by one inverse butterfly network.

Since `pdep` is the inverse of `pex`, we claim that it can be implemented by the butterfly circuit, which reverses the stages of the inverse butterfly circuit. But this has to be proved, which we do explicitly in this paper in section 4.

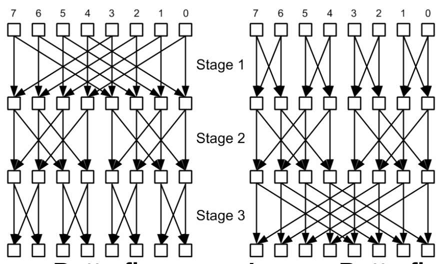


Figure 3. 8-bit butterfly and inverse butterfly networks

3. ISA Definitions

In Figures 1 and 2, we show static versions of `extr` and `dep` but dynamic or *variable* versions of `pex.v` and `pdep.v`. Static versions of `pex` and `pdep` are also very important because mask patterns desired are often known at compile time, and the static versions are much simpler to implement (section 4).

Table I shows the new instructions needed for implementing dynamic, static and loop-invariant versions of `pex` and `pdep`.

Dynamic `pex` and `pdep`. In the `pex.v` instruction, the data bits in GR r_2 selected by the "1" bits in the mask GR r_3 are placed, in the same order, in the destination register GR r_1. In the `pdep.v` instruction, the right justified bits in GR r_2 are placed in the same order in GR r_1, in the positions selected by "1"s in mask GR r_3. For both instructions, the mask r_3 is translated dynamically by a decoder into control bits for the lg(n) stages of an inverse butterfly or butterfly circuit.

Static `pex` and `pdep`. In the static versions of `pex` and `pdep`, some registers associated with the new functional unit are used to hold the control bits for the datapath and these registers must first be loaded by the `mov` instruction. The registers can be the

Table 1. New advanced bit-oriented instructions

Instruction	Description	Cycles
pex.v $r_1 = r_2, r_3$	*Parallel extract, variable*: Data bits in r_2 selected by a dynamically-decoded mask r_3 are extracted, compressed and right-aligned in the result r_1.	3
pdep.v $r_1 = r_2, r_3$	*Parallel deposit, variable*: Right-aligned data bits in r_2 are deposited, in order, in result r_1 in bit positions marked with a "1" in the dynamically-decoded mask r_3.	3
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	1
pex $r_1 = r_2, r_3$, *ar.ib₁*, *ar.ib₂*, *ar.ib₃*	*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, using datapath controls in associated ARs	1
pdep $r_1 = r_2, r_3$, *ar.b₁*, *ar.b₂*, *ar.b₃*	*Parallel deposit, static*: Right-aligned data bits in r_2 are deposited, in order, in result r_1 in bit positions marked with a "1" in the statically-decoded mask r_3, using datapath controls in associated ARs.	1
setib *ar.ib₁*, *ar.ib₂*, *ar.ib₃* = r_3	*Set inverse butterfly circuit controls in associated ARs*, using hardware decoder to translate the mask r_3 to inverse butterfly controls.	2
setb *ar.b₁*, *ar.b₂*, *ar.b₃* = r_3	*Set butterfly circuit controls in associated ARs*, using hardware decoder to translate the mask r_3 to butterfly controls.	2
bfly $r_1 = r_2$, *ar.b₁*, *ar.b₂*, *ar.b₃*	*Perform Butterfly permutation* of data bits using controls in associated ARs	1
ibfly $r_1 = r_2$, *ar.ib₁*, *ar.ib₂*, *ar.ib₃*	*Perform Inverse Butterfly permutation* of data bits using controls in associated ARs	1
grp $r_1 = r_2, r_3$	*Perform Group permutation (variable)*: Data bits in r_2 corresponding to "1"s in r_3 are grouped to the right, while those corresponding to "0"s are grouped to the left.	3

special function unit registers of PA-RISC [7], or the application registers (`ar`) of IA-64 [8]. For the purposes of this paper, we call them application registers, ARs. The `mov ar` instruction in Table 1 is used to move the contents of two general-purpose registers to the application registers. The sub-opcode, *x*, indicates which application registers are written.

In the static version of the `pex` instruction, GR r_2 is `and`'ed with mask GR r_3, then permuted using inverse butterfly application registers ar.ib$_1$, ar.ib$_2$ and ar.ib$_3$, with the result placed in GR r_1. For static `pdep`, GR r_2 is permuted using butterfly application registers ar.b$_1$, ar.b$_2$ and ar.b$_3$, then `and`'ed with mask GR r_3, with the result placed in GR r_1.

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

Table 1 also shows the `bfly` and `ibfly` instructions which can perform arbitrary *n*-bit permutations [5, 6]. In addition, the `grp` bit permutation instruction [3] is also included.

4. IMPLEMENTATION

4.1. Parallel deposit on the butterfly network

The structure of the butterfly (and inverse butterfly) networks are shown in Figure 3. The *n*-bit networks consist of lg(*n*) stages, each stage composed of *n*/2 2-input switches. Each switch is composed of two 2:1 multiplexers for a total of $n \times \lg(n)$ multiplexers. These networks are faster and smaller than an ALU of the same width which also has lg(*n*) stages, but the stages are more complex.

In the *i*th stage, the paired input bits to a switch 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.

Below, we show that any `pdep` operation can be performed using a butterfly circuit:

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

This can be proved by induction on the number of stages. At stage 1, the data bit is moved within *n*/2 positions of its final position. At stage 2, it is moved within *n*/4 positions of its final result, and so on. At stage lg(*n*), it is moved within $n/2^{\lg(n)} = 1$ position of its final result, which is its final result position.

Fact 2: If the mask has k "1"s in it, the k rightmost data bits are selected and moved, i.e., the selected data bits are contiguous. After moving, the selected data bits remain contiguous mod m/2 in each half, where m is the width of the butterfly circuit. They never cross each other in the final result.

This fact is based on the structure of the butterfly network and by definition of the `pdep` instruction.

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

Since the selected data bits never cross each other in the final result (Fact 2), once a bit swaps to L, the selected bits to the left of it must also go to L. Hence, if there is one "1" in the mask, the one selected data bit, d_0, can go to R or L. If there are two "1"s in the mask, the two selected data bits, d_1d_0, can go to RR or LR or LL. (Note that RL is not possible.) *That is, if the data bit on the right stays in R, then the next data bit can go to R or L, but if the data bit on the right goes to L, the next data bit must also go to L.* If there are three "1"s, the three selected data bits, $d_2d_1d_0$, can go to RRR, LRR, LLR or LLL. Hence, there are only k+1 possibilities for k "1"s in the mask.

Fact 4: The selected data bits that have been swapped from R to L, or stayed in L, are all contiguous mod m/2 in L and can be rotated so that they are the rightmost bits of L, and in their original order.

This follows from Facts 2 and 3. At the end of this step, we have two half-sized butterfly networks, L and R, with the selected data bits right-aligned and in order in each of L and R. (The selected data bits that stayed in R are already right aligned in R.)

The above can now be repeated recursively for the half-sized butterfly networks, L and R, until each L and R is a single bit. This is achieved after lg(n) stages of the butterfly network.

As an example, consider the `pdep` operation in Figure 2(b), broken down into steps in Figure 4(a). There are 5 "1"s in the mask, so the rightmost 5 bits, `defgh`, are the selected data bits that will be moved to new positions in the result. There are 3 "1"s in the right half (R) so bits `f`, `g` and `h` stay in R. Bit `e` swaps to L, and bit `d` stays in L, as in Fact 3. Bits `de` are contiguous in L mod 4, and can be rotated right by 3 bits to be right-aligned in L, as in Fact 4 (see the result after stage 1 in Figure 4(a)). The process is repeated for the two 4-bit butterfly networks L and R in stage 2. This is further repeated for the four 2-bit butterfly networks in stage 3, giving the desired `pdep` operation for the 8-bit data input.

The rotations done between stages in Figure 4(a) can be incorporated into the butterfly control bits of each stage by appropriate rotation of the control bits as shown in Figure 4(b). Rotations at stage 1 must be incorporated into the rotations at stage 2, which must be incorporated into the rotations at stage 3, etc.

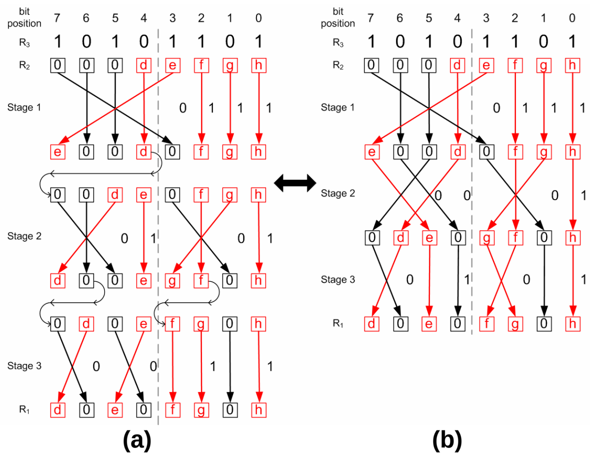


Figure 4. 8-bit pdep operation on the butterfly network (a) with separate rotations between stages, and (b) with rotations incorporated into control bits at each stage

4.2. Bitmask Decoding

The hardest part of the implementation of the `pdep` instruction is the translation of the n-bit mask into the control bits for each stage of the butterfly network. This can be done in software or by a hardware decoder. Figure 5 shows a block diagram of the functional blocks inside such a hardware decoder. Given the complexity of the problem, it is fairly amazing that the decoder can be designed to consist of only two types of subcircuits: 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 LROTC (Left ROTate and Complement) circuits, which are rotators that complement bits upon wraparound.

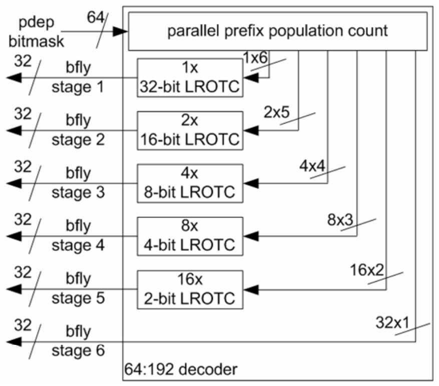


Figure 5. Hardware decoder to translate 64-bit pdep mask to 64/2 × lg(64) bfly control bits.

It turns out that the decoder circuit of Figure 5 is in fact identical to the decoder for `pex` (or `grpr`) in [9] with the caveat that the output directed to butterfly stage *i* for `pdep` is directed to inverse butterfly stage lg(*n*)–*i* for `pex`. This is consistent with the fact that `pex` and `pdep` are inverse operations. Below, we give a brief conceptual description of the decoder circuit, to illustrate its logic blocks.

For circuit optimization, we use a control bit value of "1" to indicate "pass through" and "0" to indicate "swap", counter to the usual convention for switches.

To compute the control bits for the first butterfly stage, count the number *k* of "1"s in the right half (R) of the `pdep` bitmask, and produce a string with "1"s in the *k* rightmost bits. The selected data bits that are swapped in stage 1 to the left half (L) are positioned such that they are rotated left by the number of data bits passed through in R; however, they should be the rightmost bits in L for recursion in the next stage (see Fact 4 above). Rather than rotating the data bits explicitly, we can 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. 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 circuit, while the rotation is done by the LROTC circuits at each stage.

4.3. Functional Units

Suppose we add a new functional unit to support these new bit-oriented instructions. We consider implementing alternative subsets of the instructions listed in Table 1 to show the tradeoffs in hardware cost versus flexibility.

Figure 6 shows a functional unit that supports all the instructions in Table 1, including the `grp` instruction. The latency and size of this unit is essentially determined by the `grp` instruction, which requires two inverse butterfly circuits and two decoders [9]. Note that the inclusion of the application registers for static `pex` and `pdep` allows support for `bfly` and `ibfly` permutation instructions at no extra cost. Thus, also supporting the `grp` permutation is somewhat unnecessary, since a `bfly` followed by an `ibfly` instruction can perform any arbitrary (static) permutation in 2 cycles rather than lg(*n*) cycles. Furthermore, the `grp` instruction can be emulated using the `pex.v` instruction.

Figure 7 removes support for `grp`, using `bfly` and `ibfly`, or `pex`, for permutations. It supports both variable and static `pex` and `pdep`; the variable versions can share the same hardware decoder. Eliminating `grp` yields considerable area savings, since a hardware decoder, an inverse butterfly network and many multiplexers are eliminated.

Figure 8 shows a further simplification by dropping support for the variable versions `pex.v` and `pdep.v`. This eliminates the decoder and the multiplexers for the control bits, further reducing the area. Since most applications require only static versions of `pex` and `pdep`, the elimination of the costly decoder may be justified.

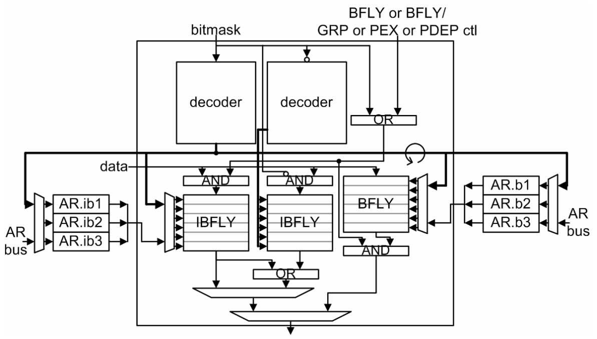


Figure 6. Functional unit supporting grp, pex.v, pdep.v, pex, pdep, ibfly and bfly.

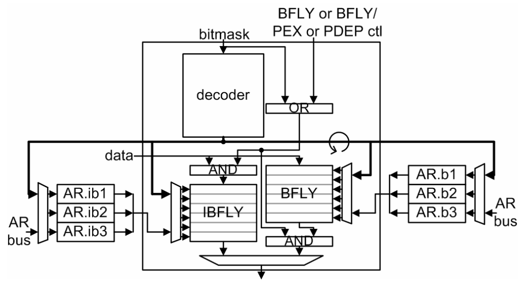


Figure 7. Functional unit supporting pex.v, pdep.v, pex, pdep, ibfly and bfly

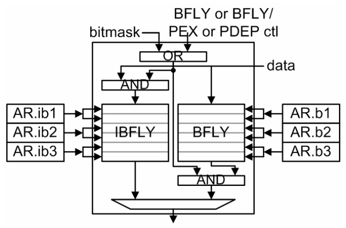


Figure 8. Functional unit supporting pex, pdep, ibfly and bfly

We evaluated these functional units for timing and area. The circuits in Figs. 6 and 7 are implemented with a 3-stage pipeline. The hardware decoder occupies the first 2 pipeline stages due to its slow parallel prefix population counter. The butterfly (or inverse butterfly) network is in the third stage. There is no overlap between the decoder and the routing of the data through the butterfly network for the `pdep` instruction since the control bits for the first stage of the butterfly network depend on the widest population count (Fig. 5), which takes the longest to generate.

The various functional units were coded in Verilog and synthesized using Synopsys Design Compiler mapping to a TSMC 90nm standard cell library [10]. The designs were compiled to optimize timing. The decoder circuit was initially compiled as one stage and then Design Compiler automatically pipelined the subcircuit. Timing and area figures are as reported by Design Compiler. We also synthesized a reference ALU using the same technology library.

Table 2 summarizes the timing and area for the circuits. Table 3 shows the number of different circuit types, to give a sense for why the functional units supporting variable pex, pdep and grp are so much larger. It clearly shows that supporting variable operations comes at a high price. The added complexity is due to the complex decoder combinational logic and to the additional pipeline registers and multiplexer logic. The variable circuits are approximately 15-20% slower, in cycle time latency, due to the decoder complexity and pipeline overhead, and up to three times larger than the static case. The static pex and pdep functional unit (Figure 8) is even smaller and faster than the reference ALU.

Table 2. Latency and area of proposed functional units

Unit	Cycle time	Relative cycle time	Area (NAND gate equiv.)	Relative Area
ALU	0.70 ns	1	10.0K	1
Figure 6: grp	0.81 ns	1.16	30.5K	3.05
Figure 7: pex.v, pdep.v	0.77 ns	1.10	22.1K	2.21
Figure 8: pex, pdep	**0.67 ns**	**0.96**	**7.6K**	**0.76**

Table 3. Number of registers and logical blocks in functional units

Unit	Pipeline Registers	Butterfly and Inverse Butterfly	Decoders	MUXes
Fig 6	~14.5	3	2	14
Fig 7	~9.25	2	1	13
Fig 8	**0**	**2**	**0**	**1**

5. Applications

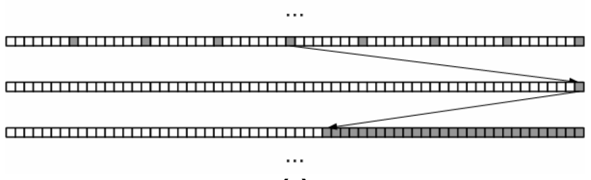
We now describe how the pex and pdep instructions can be used in existing applications, to give currently realizable speedup estimates. Use of these novel instructions in new applications and algorithms is likely to produce even more speedup.

5.1. Bit Compression and Decompression

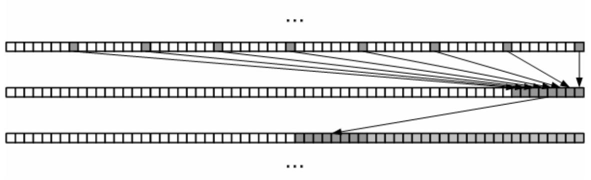
The Itanium [8] and IA-32 [11] parallel compare instructions produce subword masks – the subwords for which the relationship is false contain all zeros and for which the relationship is true contain all ones. This representation is convenient for subsequent subword masking or merging. The SPARC VIS [12] parallel compare instruction produces a bit mask of the results of the comparisons. This representation is convenient if some decision must be made based on the outcome of the multiple comparisons. Converting from the subword representation to the bitmask representation for *k* subwords requires *k* extract instructions to extract a bit from each subword and *k*-1 deposit instructions to concatenate the bits; a single static pex instruction accomplishes the same thing (see figure 9).

The SSE instruction pmovmskb [11] serves a similar purpose; it creates an 8- or 16-bit mask from the most significant bit from each byte of a MMX or SSE register and stores the result in a general purpose register. However, pex offers greater flexibility than the fixed pmovmskb, allowing the mask, for example, to be derived from larger subwords, or from subwords of different sizes packed in the same register. In fact, any arbitrary selection of bits is allowed as described in the general pattern matching scheme in section 2.

Similarly, binary image compression performed by MATLAB's bwpack function [13] benefits from pex. Binary images in MATLAB are typically represented and processed as byte arrays – a byte represents a pixel and has permissible values 0x00 and 0x01. However, certain optimized algorithms are implemented for a bitmap representation, in which a single bit represents a pixel. To produce one 64-bit output word requires 64 extr and 63 dep instructions; only 8 static pex and 7 static dep instructions perform the equivalent function (Figure 9). For decompression, as with the bwunpack function, 64 extr and 56 dep instructions are required to decompress one 64-bit input word; 7 extr and 8 pdep instructions are equivalent.



(a)



(b)

Figure 9. (a) 1 bit requires 1 extr and 1 dep (b) 1 byte requires 1 pex and 1 dep

5.2. Least Significant Bit Steganography

Steganography [14] refers to the process of hiding a secret message, not by directly obscuring the message content as with cryptography, but rather by embedding the message in a larger, innocuous cover message. A simple type of steganography is least significant bit (LSB) steganography in which the least significant bits of the color values of pixels in an image, or the intensity values of samples in a sound file, are replaced by secret message bits. LSB steganography encoding can use a `pdep` instruction to expand the secret message bits and place them at the least significant bit positions of every subword. Decoding uses a `pex` instruction to extract the least significant bits from each subword and reconstruct the secret message.

LSB steganography is an example of an application that utilizes the loop-invariant versions of the `pex` and `pdep` instructions. The sample size and the number of bits replaced are not known at compile time, but they are constant across a single message. Figure 10 depicts an example LSB steganography encoding operation in which the 4 least significant bits from each 16-bit sample of PCM encoded audio is replaced with secret message bits.

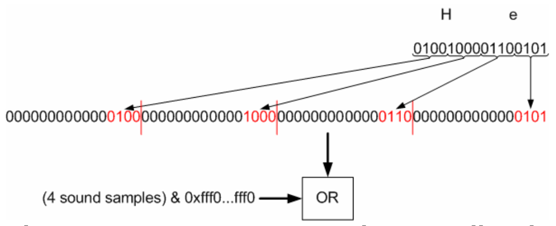


Figure 10. LSB steganography encoding (4 bits per 16-bit PCM encoded audio sample)

5.3. Transfer Coding

Transfer coding is the term applied when arbitrary binary data is transformed to a text string for safe transmission using a protocol that expects only text as its payload. Uuencoding [15] is one such encoding originally used for transferring binary data over email or usenet. In uuencoding, each set of 6 bits is aligned on a byte boundary and 32 is added to each value to ensure the result is in the range of the ASCII printable characters. Without `pdep`, each field is individually extracted and has the value 32 added to it. With `pdep`, 8 fields are aligned at once and a parallel add instruction adds 32 to each byte simultaneously (Figure 11 shows 4 parallel fields). Similarly, for decoding, a parallel subtract instruct deducts 32 from each byte and then `pex` compresses eight 6-bit fields.

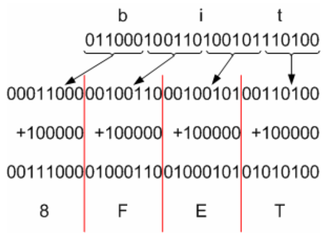


Figure 11. Uuencode of 'bit' using pdep

5.4 Bioinformatics

Pattern matching and bit scatter/gather operations are also found in important bioinformatics applications. For example, the Basic Local Alignment Search Tool (BLAST) is used for determining the similarity of sequences [16]. The BLASTX variant translates a nucleotide sequence to a protein sequence and then compares against a protein database. Each field of 6 bits in the nucleotide sequence is translated into a protein symbol. An efficient algorithm can use `pdep` to distribute eight 6-bit fields on byte boundaries, and then use the result as a set of table indices for a parallel table lookup (`ptlu`) instruction [17] to translate the bytes.

6. Performance Results

We coded kernels for the above applications and simulated them using the SimpleScalar Alpha simulator [18] enhanced to recognize our new instructions. We compared against the baseline Alpha ISA and an Alpha ISA with bit-level `extr` and `dep` instructions. (Alpha's `extract_byte` and `byte_insert` instructions are not general enough for our applications).

Figure 12 show our performance results, normalized to the base ISA cycle counts. The processor with `pex` and `pdep` instructions exhibits speedups over the base ISA ranging from 1.85× to 5.21×, with an average of 3.41×. The speedup over an ISA that has `extr` and `dep` instructions ranges from 1.60× to 4.30×, averaging 2.48× speedup.

The simple bit compression and decompression functions exhibited the greatest speedup as these operations combine 8 extracts and deposits of 1-bit fields into one `pex` or `pdep`. The speedup is lower in the steganography encoding case because there are only 4 fields per word, and also in the uudecode and BLASTX translation case because there are fewer fields overall.

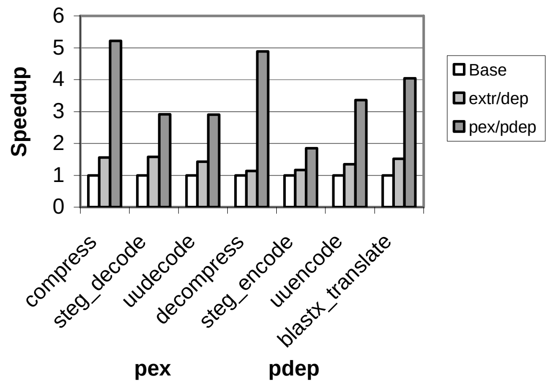


Figure 12. Performance Results

7. Summary and Conclusions

We propose new parallel extract and parallel deposit instructions to accelerate bit compression and expansion operations. These instructions can improve the performance in various applications, achieving a 3.41× average speedup over a basic ISA that has a simple ALU/shifter, and a 2.48× average speedup over an ISA with `extr` and `dep` instructions.

We show that `pdep` can be mapped onto the butterfly permutation circuit. We also propose a new functional unit that performs the `bfly` and `ibfly` permutation instructions in addition to `pex` and `pdep`. We examine alternative functional units that support different subsets of the advanced bit manipulation instructions in Table 1. Our results indicate that support for variable `pex.v` and `pdep.v` instructions in hardware comes at a steep price in area and latency, due mainly to the circuit complexity of the hardware decoder for translating a mask into controls for the butterfly or inverse butterfly datapaths. Also, our applications mostly needed static versions of `pex` and `pdep`; only the LSB steganography application makes use of loop-invariant `pex` and `pdep`.

Hence, we feel that the simplest unit that supports `ibfly` and `bfly` and static `pex` and `pdep` (Figure 8) is the current best choice for both functionality actually needed and cost-effectiveness. In cases where the variable `pex` and `pdep` instructions are required, a software routine can decode the mask and save the resulting control bits for later use by static `pex` and `pdep` instructions. If the operation is loop invariant, this subroutine is invoked only once, with minimal performance overhead.

Areas for future work include new or re-structured algorithms and applications exploiting these fast parallel instructions, and exploration of other advanced bit manipulation instructions such as bit matrix multiply. These advanced bit-oriented instructions are an important ISA extension for word-oriented processors that can provide tangible benefits in many existing and emerging application domains.

8. References

[1] R. B. Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22-32, April 1995.
[2] 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.
[3] 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*, pp.138-148, July 2000.
[4] Xiao Yang and Ruby B. Lee, Fast Subword Permutation Instructions Using Omega and Flip Network Stages, *Proceedings of the International Conference on Computer Design (ICCD 2000)*, pp. 15-22, September 2000.
[5] 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*, August 2002.
[6] 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*, June 2003.
[7] R. Lee, "Precision Architecture", *IEEE Computer*, Vol. 22, No. 1pp.78-91, Jan 1989.
[8] Intel Corporation, *Intel® Itanium® Architecture Software Developer's Manual*, Vol. 1-3., rev. 2.1, Oct. 2002.
[9] Y. Hilewitz, Z. J. Shi, and R. B. Lee, "Comparing Fast Implementations of Bit Permutation Instructions," *Proceedings of the 38th Annual Asilomar Conference on Signals, Systems, and Computers*, Nov. 2004.
[10] Taiwan Semiconductor Manufacturing Corporation, *TCBN90G: TSMC 90nm Core Library Databook*, Oct. 2003.
[11] Intel Corporation, *IA-32 Intel® Architecture Software Developer's Manual*, Vol. 1-2, 2004.
[12] Sun Microsystems, *The VIS™ Instruction Set*, Version 1.0, June 2002.
[13] The Mathworks, Inc., *Image Processing Toolbox User's Guide*: http://www.mathworks.com/access/helpdesk/help/toolbox/images/images.html.
[14] E. Franz, A. Jerichow, S. Möller, A. Pfitzmann, and I. Stierand "Computer Based Steganography," *Information Hiding, Springer Lecture Notes in Computer Science*, vol. 1174, pp. 7–21, 1996.
[15] "Uuencode," Wikipedia: The Free Encyclopedia, http://en.wikipedia.org/wiki/Uuencode.
[16] National Center for Biotechnology Information, BLAST, http://www.ncbi.nlm.nih.gov/BLAST/
[17] A. M. Fiskiran and R. B. Lee, "On-Chip Lookup Tables for Fast Symmetric-Key Encryption," *Proceedings of the IEEE International Conf. on Application-Specific Systems, Architectures and Processors*, pp. 356-363, July 2005.
[18] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, 1997.