Advanced Bit Manipulation Instruction Set Architecture

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

1. Introduction

Microprocessor instruction set architectures currently offer support for operations on full processor words and, more recently, on subwords of 8, 16, 32 or 64 bits. However, there is little support for bit-oriented operations beyond the bitwise logical operators and shift operators. Some ISAs, such as PA-RISC and Itanium, offer more advanced instructions such as extract and deposit that combine mask and shift operations in one instruction.

We believe that numerous applications domains benefit from instructions that operate on bits. We propose instructions (Table 1) to manipulate bits – permutation instructions to reorder bits and subwords, and other instructions to compact and expand bit patterns, as well as instructions to efficiently compute the parity of bits. We have examined applications domains ranging from cryptography to bioinformatics, from cryptanalysis to channel coding and shown that advanced bit operations have relevance.

We are soliciting your help to broaden our application analysis. We present our proposed instructions and ask how can these instructions help your domain? Can your algorithms be slightly modified to take advantage of our proposed instructions? Can our instructions be slightly modified to enable performance improvements for your applications?

Table 1: Summary of Advance Bit Manipulation Instructions

Instruction		**Mnemonic**	**Description**	**Instruction Latency in Cycles (relative to single cycle ALU)**
Butterfly		bfly	Performs butterfly permutation of input	1
Inverse Butterfly		ibfly	Performs inverse butterfly permutation of input	1
Group		grp	Divides input into two groups – the bits marked with a "1" are grouped to the right and the bits marked with a "0" are grouped to the left	3
Parallel Permute		pperm	A subword of the output is assigned the values of the bits of the input with positions given in a list of indices	1
Mix		mix	Even or odd-indexed subwords are selected alternately from the two source registers.	1
Check		check	Subwords are picked alternately from the two source registers in a checkerboard pattern.	1
Excheck		excheck	Subwords are picked alternately from the two source registers in a reversed checkerboard pattern.	1
Mux		mux	Performs specific subword permutations	1
Parallel Extract	Static	pex	The bits in the input marked with a "1" are extracted and grouped to the right.	1
	Variable	pex.v	The bits in the input marked with a "1" are extracted and grouped to the right. The bit marking is variable.	3

Parallel Deposit	Static	pdep	A right justified field of bits is expanded and deposited in bit positions marked with a “1”.	1
	Variable	pdep.v	A right justified field of bits is expanded and deposited in bit positions marked with a “1”. The bit marking is variable.	3
Population Count		popcount	The count of the number of bits in the input set to “1”.	2
Dot Product		dotprod	Dot product of the inputs viewed as bit vectors.	1
Bit Matrix Multiply	$n \times n$	bmm.1rR	Multiplication of $1 \times n$ bit vector by $n \times n$ bit matrix.	1 ($n = 64$)
	$k \times n$	bmm.2rR	Multiplication of $2n$-bit matrix by kn-bit matrix.	1
	$1 \times n$	bmm.2r	Multiplication of n-bit matrix by n-bit matrix.	1
Parallel Table Lookup		ptlu	Each byte of a word is independently used as an offset into parallel on-chip tables. The table lookups are performed in parallel and the results are combined.	1

2. Permutation Instructions

2.1 Butterfly (`bfly`) and Inverse Butterfly (`ibfly`) [6, 8, 10, 11, 12]

- **Format**: bfly $r_1 = r_2$, *ar.b*$_1$, *ar.b*$_2$, *ar.b*$_3$

  ibfly $r_1 = r_2$, *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$
- **Description**: The `bfly` and `ibfly` instructions permute their inputs, r_2, using butterfly and inverse butterfly circuits, respectively (Fig. 1 – each box is a switch that either passes through or swaps its inputs depending on the value of a configuration bit). The concatenation of these circuits forms a Benes̑ network, a permutation network. Consequently, a single execution of both instructions can yield any of the $n!$ permutations of n bits.
- **Advantages**: The circuits have single cycle latency (i.e., latency less than of an ALU of similar width). The circuits are small in area, each consisting solely of $n\lg(n)$ 2:1 multiplexers.
- **Disadvantages**: Each circuit requires $n\lg(n)/2$ configuration bits. For $n = 64$, three registers of configuration bits are necessary, in addition to register needed for the data bits. Typical architecture support only two or three input operands. Thus, we propose to use special registers, ar.b_1-b_3 and ar.ib_1-ib_3 (a la PA-RISC special functional unit registers or Itanium application registers), dedicated to supplying the bits to the functional units. In addition to this extra state, there is extra overhead in loading these registers if the permutation is changing.
- **Example Usage**: P-Box permutation in DES cipher round function.

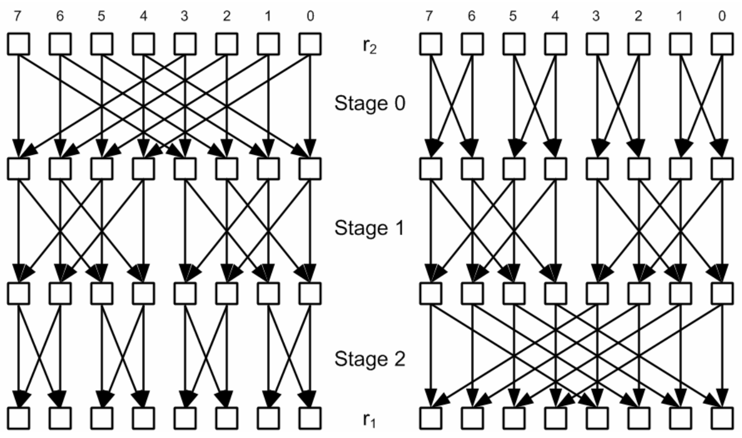


Fig. 1: 8-bit butterfly and inverse butterfly networks.

2.2 Group (`grp`) [3, 5, 10, 12]

- **Format**: grp $r_1 = r_2, r_3$
- **Description**: The `grp` instruction permutes its data input, r_2, by grouping to the right those bits flagged with a 1 in a configuration input, r_3, and grouping to the left those bits flagged with a 0 in the configuration input, maintaining the order of the bits within each group (Fig. 2). A series of lg(*n*) `grp` instruction can yield any permutation.
- **Advantages**: `grp` has been used in a fast radix-2 sorting algorithm. The use of `grp` as a cryptographic primitive has also been explored.
- **Disadvantages**: The current implementation of `grp` consists of parallel inverse butterfly circuits and parallel hardware decoders that translate the configuration input to the inverse butterfly control bits. These decoders are over 2.5 times the size of a butterfly or inverse butterfly network. Thus this circuit is much larger than just a single inverse butterfly circuit. Furthermore, the decoder is slow, causing the latency of `grp` to be 2 or 3 cycles.
- **Example Usage**: P-Box permutation in DES cipher round function.

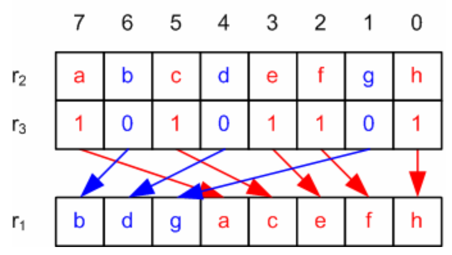


Fig. 2: 8-bit `grp` operation.

2.3 Parallel Permute (`pperm`/`pperm3r`) [3, 5]

- **Format**: pperm.x $r_1 = r_2, r_3$

  pperm3r.x $r_1 = r_2, r_3, r_4$

- **Description**: `pperm` (and `pperm3r`) permutes its data input, r_2, according to an explicit list of source indices found in a configuration input, r_3. A sub-opcode, x, specifies at what subword offset in the destination the bits are written. Each index requires lg(n) bits, so at most n/lg(n) indices can be stored per word. As n indices must be specified, a minimum of lg(n) instructions is required to yield any arbitrary permutation. `pperm` has only two inputs and zeros the remaining positions in the output (and thus requires additional `or` instructions to combine results). `pperm3r` has three inputs and keeps the bits in the remaining positions from the third input, r_4.
- **Advantages**: `pperm` can additionally permute bits with repetition, simply by repeating a source index.
- **Disadvantages**: `pperm` requires the full sequence of instructions even for simple permutations. `pperm` requires more than lg(n) instructions for an arbitrary permutation (8 instructions for n = 64 if each index is byte aligned).
- **Example Usage**: Expansion permutation in DES cipher round function.

2.4 Mix (`mix`) [1, 2]

- **Format**: mix.sw.l/r $r_1 = r_2, r_3$
- **Description**: `mix` performs a permutation on the 1, 2, 4-bit, 1, 2 or 4-byte subwords in r_2 and r_3. The subwords in each source register are grouped in pairs and one element from each pair is selected for the output, alternating between the source registers (Fig. 3). The left form (.l) selects the left element of each pair and the right form (.r) selects the right element from each pair. Currently only 1-byte and larger mix instructions exist in PA-RISC and IA-64.
- **Advantages**: `mix` can be used for efficiently reshaping subword matrices (e.g., to transform from column major to row major). `mix` can also be used to expand the width of subwords (by mixing with the zero register).
- **Example Usage**: Reshaping subword matrices in IDCT.

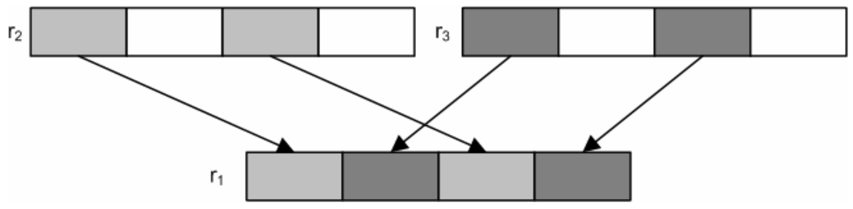


Fig. 3: mix.2.l operation (for 64-bit words)

2.5 Check (`check`) and Excheck (`excheck`) [2]

- **Format**: ex/check.sw $r_1 = r_2, r_3$
- **Description**: `check` and `excheck` perform a permutation on the 1, 2 or 4-byte subwords in r_2 and r_3. The subwords in each source register are grouped in pairs and one

element from each pair is selected for the output, alternating between the source registers (Fig. 4). `check` selects the left element of each pair from r2 and the right element of each pair from r3, forming a checkerboard pattern. `excheck` reverses the pattern. Bit-level check and excheck can also be defined, if deemed necessary.

- **Example Usage**: check and excheck can be used to achieve the rearrangements of area-mapped 2 × 2 matrices.

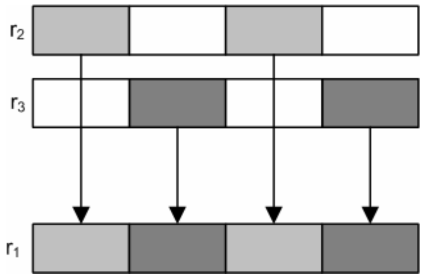


Fig. 4a: check.2 operation

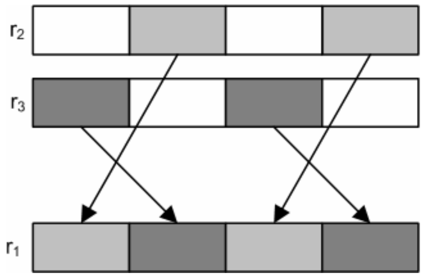


Fig. 4b: excheck.2 operation

2.6 Mux (`mux`) [23]

- **Format**: mux.sw.{rev,mix,shuf,alt,brcst} $r_1 = r_2$
- **Description**: `mux` performs a set of five byte permutations of r_2 (Fig. 5). `rev` reverse the bytes; `mix` performs a mix operation on the two halves of r_2; `shuf` performs a shuffle operation on the two halves of r_2; `alt` performs an alternate operation on the two halves of r_2; `brcst` replicates the least significant byte (or 2 bytes) of r_2 to all subwords. Bit-level mux is performed by the bit-level permutation instructions.
- **Advantages**: Compact specification of specific common byte-level permutations.
- **Example Usage**: mux.brcst can be used to create a vector of constant values.

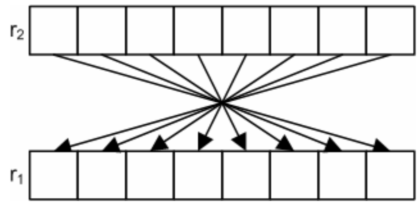


Fig. 5a: mux.1.rev operation

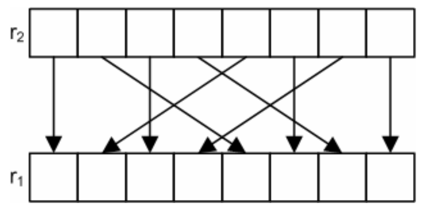


Fig. 5b: mux.1.mix operation

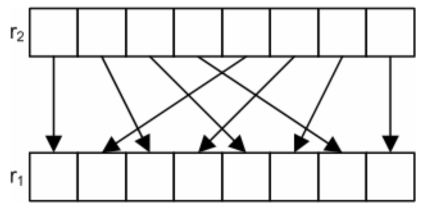


Fig. 5c: mux.1.shuf operation

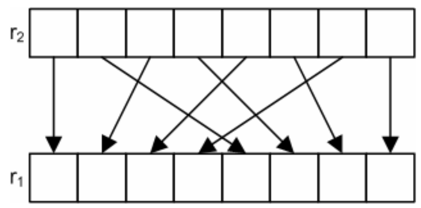


Fig. 5d: mux.1.alt operation

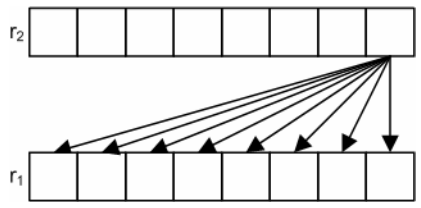


Fig. 5e: mux.1.brcst operation

3. Bit Compression and Expansion

3.1 Parallel Extract (`pex`) [12-14]

- **Format**: pex $r_1 = r_2, r_3,$ *ar.ib*$_1$, *ar.ib*$_2$, *ar.ib*$_3$
- **Description**: The `pex` instruction groups to the right the bits in its data input, r_2, flagged with a 1 in a configuration input, r_3 (Fig. 6). The remaining bit positions in the output are zeroed. `pex` is a generalization of the extract (`extr`) instruction found in PA-RISC and Itanium. `extr` performs a mask and shift operation – it right justifies a single field of bits from its input. `pex` compresses and right justifies any subset of bits in its input. `pex` can also be viewed as the right half of a `grp` operation.
- **Advantages**: The `pex` implementation uses an inverse butterfly circuit. The `ibfly` control bits are calculated in advance and `pex` functions similarly to `ibfly` and shares all of its advantages. (Note that the `pex` instruction actually performs a masked `ibfly` operation and can be used as such by setting control bits ar.ib$_1$-ib$_3$ so as to perform an operation not corresponding to `pex` with mask r_3.)

- **Disadvantages**: As `pex` uses the inverse butterfly circuit, it has the same disadvantages – extra registers and overhead to set those registers for static `pex`.
- **Example Usage**: Least significant bit steganography (the least significant bits of image or sound data are replaced by secret message bits) decoding.

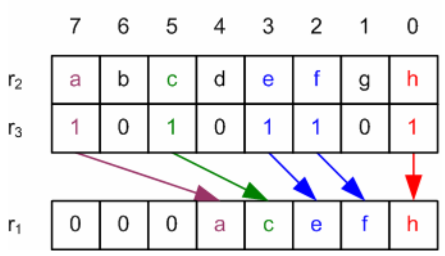


Fig. 6: 8-bit `pex` operation

3.2 Parallel Extract, variable (`pex.v`) [12-14]

- **Format**: pex $r_1 = r_2, r_3$
- **Description**: The `pex` instruction groups to the right the bits in its data input, r_2, flagged with a 1 in a configuration input, r_3. The inverse butterfly control bits are dynamically decoded from r_3.
- **Advantages**: The inverse butterfly control bits are dynamically decoded from data dependent mask r_3 using a dedicated hardware decoder (the same decoder as used for `grp`). If the `pex` operation is loop independent, then the decoder can write its output to the inverse butterfly control registers and the static, fast `pex` instruction can be executed in the loop.
- **Disadvantages**: Dynamic `pex` requires a sizable and slow decoder.
- **Example Usage**: Least significant bit steganography decoding in which only a random subset of subwords are selected for embedding of message bits.

3.3 Parallel Deposit (`pdep`) [12, 14]

- **Format**: pdep $r_1 = r_2, r_3, \mathit{ar.b_1}, \mathit{ar.b_2}, \mathit{ar.b_3}$
- **Description**: The `pdep` instruction expands a right justified field of bits in its data input, r_2 to the positions flagged by 1s in a configuration input, r_3 (Fig. 7). The remaining bit positions in the output are zeroed. `pdep` is a generalization of the deposit (`dep`) instruction found in PA-RISC and Itanium. `dep` performs a mask and shift operation – it shifts a single right justified field of bits in the input to any position in the output. `pdep` expands a single right justified field of bits to any arbitrary set of positions in the output.
- **Advantages**: The `pdep` implementation uses a butterfly circuit. The `bfly` control bits are calculated in advance and `pdep` functions similarly to `bfly` and shares all of its advantages. (Note that the `pdep` instruction actually performs a masked `bfly` operation and can be used as such by setting control bits $ar.ib_1$-ib_3 so as to perform an operation not corresponding to `pdep` with mask r_3.)

- **Disadvantages**: As pdep uses the inverse butterfly circuit, it has the same disadvantages – extra registers and overhead to set those registers for static pdep. Functionally, dep can maintain the values of the bits not overwritten in the destination. For pdep, a third input operand would be required to support this feature.
- **Example Usage**: Least significant bit steganography encoding.

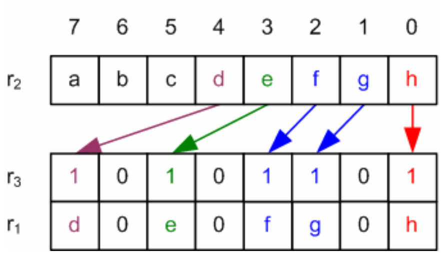


Fig. 7: 8-bit pdep operation.

3.4 Parallel Deposit, variable (pdep.v) [12, 14]

- **Format**: pdep $r_1 = r_2, r_3$
- **Description**: The pdep instruction expands a right justified field of bits in its data input, r_2 to the positions flagged by 1s in a configuration input, r_3. The remaining bit positions in the output are zeroed. pdep is a generalization of the deposit (dep) instruction found in PA-RISC and Itanium. dep performs a mask and shift operation – it shifts a single right justified field of bits in the input to any position in the output. pdep expands a single right justified field of bits to any arbitrary set of positions in the output.
- **Advantages**: The inverse butterfly control bits are dynamically decoded from data dependent mask r_3 using a dedicated hardware decoder (the same decoder as used for grp except the control bits are fed to butterfly circuit reversing the order of the stages). If the pdep operation is loop independent, then the decoder can write its output to the inverse butterfly control registers and the static, fast pdep instruction can be executed in the loop.
- **Disadvantages**: Dynamic pdep requires a sizable and slow decoder. Functionally, dep can maintain the values of the bits not overwritten in the destination. For pdep, a third input operand would be required to support this feature.
- **Example Usage**: Least significant bit steganography encoding in which only a random subset of subwords are selected for embedding of message bits.

4. Dot Product

4.1 Population Count (popcnt) [23]

- **Format**: popcount $r_1 = r_2$
- **Description**: The popcnt instruction counts the number of bits set to "1" in input r_2. This instruction exists in ISAs such as Itanium.

- **Advantages**: In addition its normal usage, `popcnt` can be used to compute parity as the least significant bit of the count is the parity of the word.
- **Example Usage**: Error correcting coding.

4.2 Bit Vector Dot Product (`dotprod`)

- **Format**: dotprod $r_1 = r_2, r_3$
- **Description**: The `dotprod` instruction `and`s its two inputs, r_2 and r_3, and then computes the parity of the result: $r_1 = \bigoplus_{i=0}^{n-1} r_2\{i\} \cdot r_3\{i\}$ where $\{i\}$ indicates the *i*th bit of the operand and '$\cdot$' indicates bitwise and.
- **Advantages**: `dotprod` replaces a sequence of
    - `and` of the two inputs,
    - `popcnt` of the result and
    - `and` to isolate the least significant bit.
- **Example Usage**: Error correcting coding.

4.3 Bit Matrix Multiply (`bmm`) [18, 19]

- **Format**: bmm $r_1 = r_2, R_{bmm}$ for $1 \times n$ bit vectors r_1 and r_2, and $n \times n$ bit matrix R_{bmm}
- **Description**: `bmm` multiplies a $1 \times n$ bit vector by an $n \times n$ bit matrix: $r_1 = r_2 \times R_{bmm}^T$ mod 2 (where $\times$ is standard matrix multiplication) (Fig. 8). A single bit, *i*, of r_1 is the bit vector dot product of r_2 and the *i*th row of R_{bmm}. A single vector-matrix multiplication calculates *n* dot products in a single instruction.

  We are also exploring multiplication primitives for decomposing the bmm operation into multiplication of submatrices.
- **Advantages**: In addition to parity/dot product operations, `bmm` is also a superset of all the bit manipulation instructions listed. A single "1" in the *i*th row of R_{bmm}, say in the *j*th column, shifts the bit in the *j*th column of r_2 to the *i*th column of r_1. Thus, with proper configuration of the B matrix, all permutations, with repetition and masking, are computable using `bmm`.

  Another possible use of `bmm` is bit matrix transpose (`bmt`). If *n* r_2 vectors form the $n \times n$ identity matrix then the corresponding *n* r_1 vectors contain the transpose of R_{bmm}.
- **Disadvantages**: In order to efficiently compute `bmm`, the entire R_{bmm} matrix must be read in a single cycle. This requires extra storage to hold the matrix and overhead to load this storage. Additionally, the cost of dedicating storage to hold the entire matrix may prove too high. This is why the smaller primitives are being explored.
- **Example Usage**: Error correcting coding.

5. Table Lookup

5.1 Parallel Table Lookup (`ptlu`) [20-22]

- **Format** (generic): ptlu $r_1 = r_2, r_3$
- **Description**: `ptlu` interprets the bytes of a word, r_1, as independent indices into a set of tables and then looks up the values in these tables in parallel and combines the result with the second input, r_2, in the output word (Fig. 8). The size of the entries and the method of combining the table output (logical, concatenation, etc.) is flexible (our current description is optimized for AES and thus has 32-bit entries and can XOR, concatenate or select individual entries).

  While `ptlu` is not a bit manipulation instruction, we list it due to the synergistic effects of using bit manipulation instruction together with `ptlu`. For example, the bytes of a word might need to be permuted prior to lookup. Alternatively, the data is less than a byte wide, so the indices need to be expanded to byte boundaries prior to lookup.
- **Advantages**: `ptlu` can greatly speed up algorithms that use table lookups. Not only are 8 lookups performed in parallel in a 64-bit machine, but the tables are stored in dedicated on-chip memory (essentially another block of L1 cache), allowing for single-cycle latency and the elimination of cache misses that could potentially plague lookups to main memory.
- **Disadvantages**: A dedicated block of on-chip memory is required to support this operation. Efficiently managing multiple tables is still an open issue.
- **Example Usage**: Fast AES implementation.

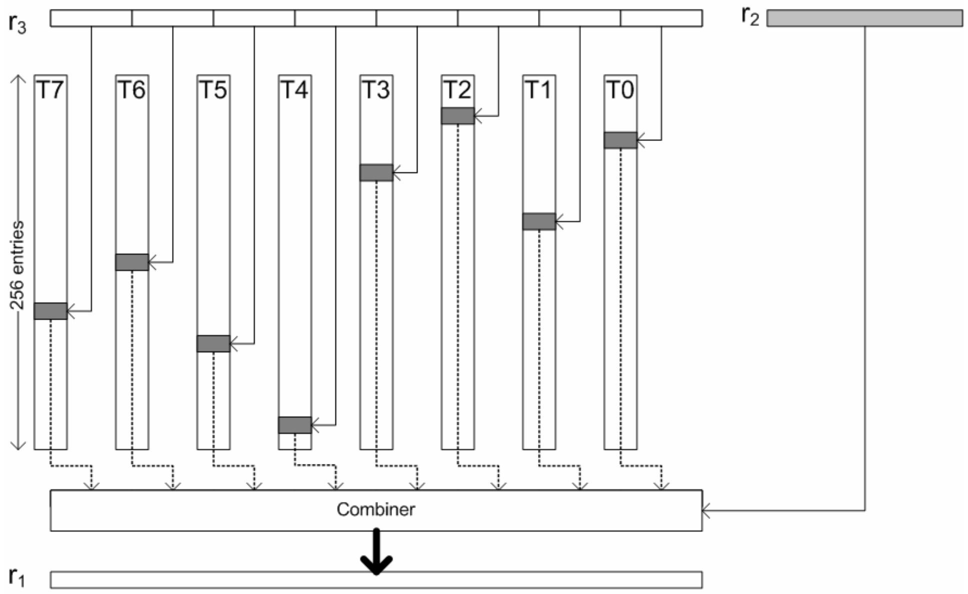


Fig. 8: 64-bit `ptlu`

6. Example Applications

We present a number of examples of applications that benefit from bit-oriented operations in the ISA. We hope that these examples provide insight for the analysis of the relevance of bit-oriented operations to your own application domain.

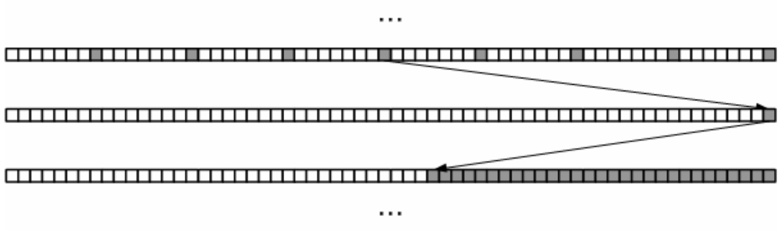
6.1 Cryptography

A number of popular ciphers, such as DES, have permutations as primitive operations. We have previously shown that the inclusion of permutation instructions such as `bfly`/`ibfly` or `grp` can greatly improve performance of the inner loop of these functions [3, 5]. Also see [9].

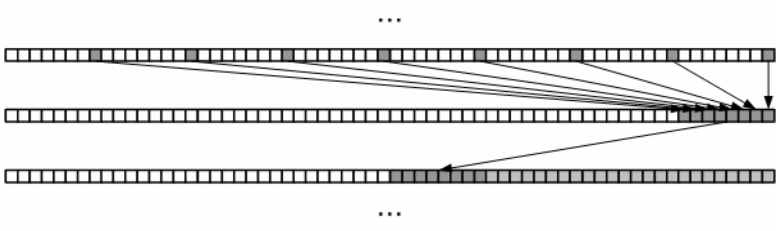
6.2 Binary Compression and Decompression

The Itanium [23] and AltiVec [24] parallel compare instruction 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 [25] 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 Itanium representation to the VIS 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 `pex` instruction accomplishes the same thing. The SSE instruction `pmovmskb` [26], which 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, serves a similar purpose. However, `pex` offers greater flexibility than the fixed `pmovmskb`, allowing the mask, for example, to be derived from larger subwords.

Similarly, binary image compression performed by MATLAB's `bwpack` function [27] 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; 8 `pex` and 7 dep instructions are equivalent (Fig. 9). For decompression, as with `bwunpack`, 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)

Fig. 9: (a) 1 **bit** of output requires 1 extr and 1 dep; (b): 1 **byte** of output requires 1 pex and 1 deposit

6.3 Least Significant Bit Steganography

Steganography refers to the process of hiding an 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) [28] 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 utilizes 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 re-form the secret message. Fig. 10 depicts an example LSB steganography encoding operation in which the 2 least significant bits from each 16-bit sample of PCM encoded audio is replaced with secret message bits.

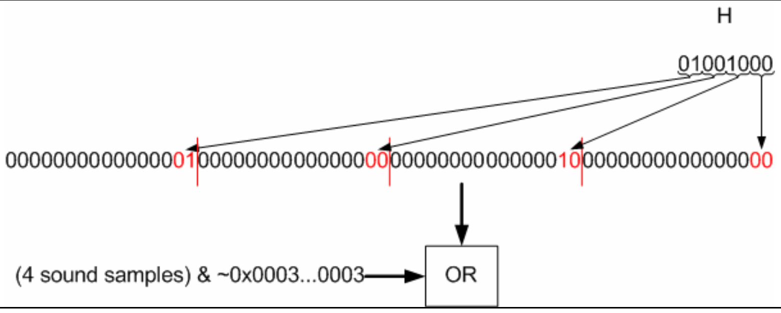


Fig. 10: LSB steganography encoding (2 bits per 16-bit PCM encoded audio sample)

6.4 Binary Image Morphology

Binary image morphology is a collection of techniques for binary image processing such as erosion, dilation, opening, closing, thickening, thinning, etc. The bwmorph function [27] in MATLAB implements these techniques primarily through one or more table lookups applied to the 3 × 3 neighborhood surrounding each pixel (i.e. the value of 9 pixels is used as index into a 512 entry table). In its current implementation, bwmorph processes byte array binary images, not bitmap images, possibly due to the difficulty in extracting the neighborhoods in the bitmap form. The relative pixel position is used as index into a 3 × 3 table that gives the actual index weight for that position. The index weights for the 9 pixels are added and the result is used as the index into the pixel transformation lookup table. If the images are processed in bitmap form, a single `pex` instruction extracts the entire index at once (assuming a 64-bit word contains an 8 × 8 block of 1-bit pixels, Fig. 11). As the neighborhood changes for each pixel, the dynamic `pex.v` instruction may be needed. Alternatively, the data can be shifted so that the neighborhood to be extracted remains in the same bit positions.

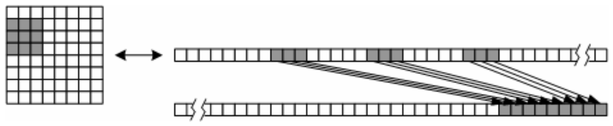


Fig. 11: Using pex to extract a 3 × 3 neighborhood of pixels

6.5 Transfer Coding

Transfer coding is the term applied when binary data is encoded to allow for safe transmission using a text-only protocol. Uuencoding [29] 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 subword simultaneously (Fig. 12). Similarly, for decoding, a parallel subtract instruct deducts 32 from each subword and then pex compresses 8 6-bit fields.

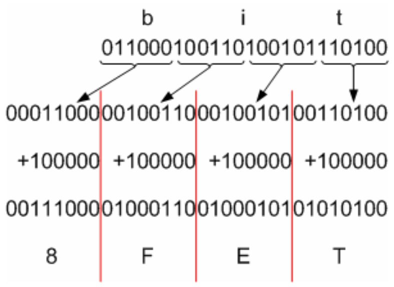


Fig. 12: Uuencode of the word 'bit' using pdep

A similar transfer encoding is Base64 [30], which is the binary data transfer coding for the Multipurpose Internet Mail Extensions (MIME) protocol which describes the format of email messages. Currently, binary email attachments are most likely encoded using base64. Base64 uses each 6-bit value from the input stream as a lookup into the string "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" (note that this string is sometimes modified due to context).

Some popular email applications, such as PINE or KDE Kmail, perform base64 encoding in a straightforward fashion by extracting each 6-bit field and use the field as a lookup into the string. Other applications, such as Mozilla Thunderbird, determine the output directly as a function of the input value (i.e. if the input is in the range 0-25, the output equals the input plus 65 (the value of ASCII 'A'), etc.). pex/pdep, together with our parallel table lookup instruction (ptlu), can greatly accelerate either process. For encoding, pdep is used to align each 6-bit index on a byte boundary and ptlu is used to perform 8 lookups in parallel or multimedia (subword parallel arithmetic) instructions are used to perform the comparisons and additions. For decoding, ptlu is used to perform the reverse lookups in parallel or multimedia instructions are used to compute the original indices and then pex compresses the eight 6-bit results.

6.6 Bioinformatics

Bioinformatics is the field of analysis of genetic information. DNA, the genetic code contained within the nucleus of each cell, is a strand of the nucleotides adenine, cytosine, guanine and thymine. These nucleotides are typically represented by an ASCII string using the characters A, C, G and T. However, a 2-bit (rather than 8-bit ASCII) encoding of the nucleotides [31] is more efficient and can significantly increase performance of matching and searching operations on large genomic sequences (the human genome contains 3.2 billion nucleotides). The ASCII codes for the characters A, C, G and T differ in bit positions 2 and 1 (A: 00, C: 01, G: 11, T: 10) and these two bits can be used to encode each nucleotide. Thus a fourfold compression of a genomic sequence simply requires a single `pex` instruction to select bits 2 and 1 of each byte of a word (Fig. 13). Without `pex`, a sequence of 8 extract instructions is required to isolate each 2-bit field individually.

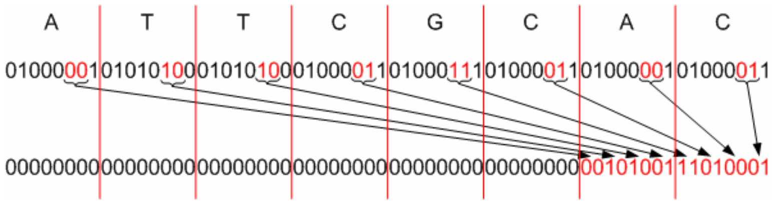


Fig. 13: Compression of sequence ATTCGCAC using `pex`

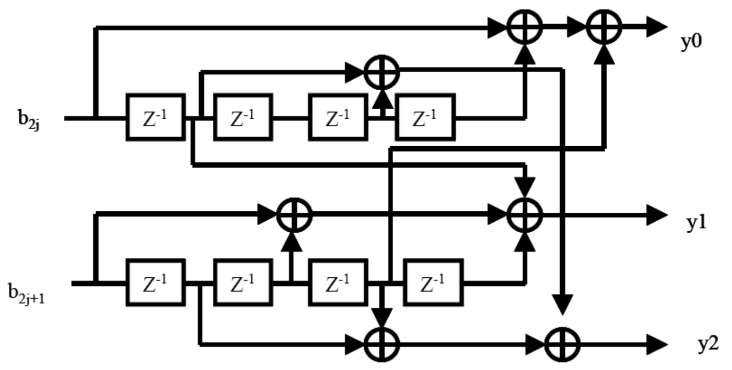
A strand of DNA is a double helix – there are really two strands with the complementary nucleotides, A↔T and C↔G, aligned. When performing analysis on a DNA string, often the complementary string is analyzed as well. To obtain the complementary string, the bases are complemented and the entire string is reversed, as the complement string is read from the other end. The reversal of the DNA string amounts to a reversal of the ordering of the pairs of bits in a word. This is a straightforward `bfly` or `ibfly` permutation. Without these instructions, a sequence of `and`s, `shift`s and `or`s (and a `mux@rev`) are required to perform the permutation.

The DNA sequence is transcribed by the cell into a sequence of amino acids or a protein. Often the analysis of the genetic data is more accurate when performed on a protein basis. A set of three nucleotides, or 6 bits of data, corresponds to a protein codon. Translating the nucleotides to a codon requires a table lookup operation using each set of 6 bits as an index. This process is identical to the base64 encoding described in the previous section and thus is accelerated by the `pdep` instruction.

6.7 Error Correcting Codes

Error correcting codes are used in virtually all real-world digital communication settings to correct bit errors introduced by channel noise. They are also used in storage settings such as ECC RAM to correct for soft bit errors or RAID to prevent data loss in case of physical hard disk failure. In error correcting coding the parity of subsets of message bits are computed and sent along with the message. The knowledge of the parity bits allows for the detection and possibly correction of bits flipped to due channel noise. In Hamming codes, the parity bits are created by direct multiplication of the symbols with a generator matrix – a bit matrix multiply operation. Convolutional coding processes the input as a continuous stream. One or more shift registers store some number of previous input bits and the outputs are the parity of subsets of the stored bits and the next input bits. This type of coding can also be mapped to bit matrix multiply by replicating and shifting the matrix containing the generator polynomials across the bit matrix to emulate the effect of the shift register memory. For example, the IEEE 802.11g 22 Mbit and 33

Mbit convolutional coder is shown in Fig. 14(a) [32]. The equivalent B matrix is shown in Fig. 14(b).



(a)

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} & & & & & \\ G & & & & & \\ & & & & & \\ 0 & 0 & & & & \\ 0 & 0 & G & & & \\ 0 & 0 & & & & \\ 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & G & \\ 0 & 0 & 0 & 0 & & \\ & & & & & \ddots \end{bmatrix}$$

(b)

Fig. 14: (a) IEEE 802.11g 22Mbit and 33Mbit convolutional coder (b) Equivalent B matrix with generator matrix G replicated and shifted.

References:

PALMS papers are available for download at: http://palms.ee.princeton.edu/publications.html

Permutation Instructions:
[1] R. Lee. "Subword Parallelism with MAX-2," *IEEE Micro*. vol. 16, no. 4, pp.51-59, August 1996.
[2] R. B. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2000)*, pp. 3-14, July 2000.
[3] Z. Shi and R. 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.
[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, "Efficient Permutation Instructions for Fast Software Cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56-69, December 2001.
[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Z. J. Shi, *Bit Permutation Instructions: Architecture, Implementation and Cryptographic Properties*, Phd. Thesis, Princeton University, June 2004.

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

[12] Y. Hilewitz and R. B. Lee, "Advanced Bit Manipulation Instructions," *Princeton University Department of Electrical Engineering Technical Report*, in progress.

Bit Compression and Expansion:

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

[14] Y. Hilewitz and R. 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 (Best Paper Award).

Also [12]

Permutation Functional Unit Implementation:

[15] Zhijie Jerry Shi and Ruby B. Lee, "Implementation Complexity of Bit Permutation Instructions," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, pp. 879-886, November 2003 (Nominated for Best Student Paper Award).

[16] 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*, pp. 1856-1863, November 2004.

[17] Y. Hilewitz and R. B. Lee, "Performing Advanced Bit Manipulations Efficiently in General-Purpose Processors," *Princeton University Department of Electrical Engineering Technical Report CE-L2006-003*, October 2006 (*in submission to ARITH-18*).

Also [14]

Dot Product:

[18] William Lee, Gary J. Geissler, Steven J. Johnson, Alan J. Schiffleger, *Vector Bit-Matrix Multiply Functional Unit*, US patent 5,170,370, to Cray Research, Inc., Patent and Trademark Office, 1990.

[19] Y. Hilewitz and R. B. Lee, "Towards Supercomputer Performance on Commodity Microprocessors – Implementation of Bit Matrix Multiply in the Itanium Instruction Set Architecture," *Princeton University Department of Electrical Engineering Technical Report*, in progress.

Parallel Table Lookup:

[20] A. Murat Fiskiran and Ruby B. Lee, On-Chip Lookup Tables for Fast Symmetric-Key Encryption, *Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 356-363, July 23-25, 2005.

[21] A. Murat Fiskiran and Ruby B. Lee, Fast Parallel Table Lookups to Accelerate Symmetric-Key Cryptography, *Proceedings of the International Conference on Information Technology Coding and Computing (ITCC), Embedded Cryptographic Systems Track,* pp. 526-531, April 2005.

[22] A. Murat Fiskiran, *Instruction Set Architecture for Accelerating Cryptographic Processing in Wireless Computing Devices*, PhD. Thesis, Princeton University, August 2005.

Commercial ISAs:

[23] Intel Corporation, *Intel® Itanium® Architecture Software Developer's Manual*, Vol. 1-3, rev. 2.2, Jan. 2006.

[24] IBM Corporation, *PowerPC Microprocessor Family: AltiVec™ Technology Programming Environments Manual*, Version 2.0, July 2003.

[25] Sun Microsystems, *The VIS™ Instruction Set*, Version 1.0, June 2002.

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

Applications:

[27] The Mathworks, Inc., *Image Processing Toolbox User's Guide*: http://www.mathworks.com/access/helpdesk/help/toolbox/images/images.html.

[28] 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.

[29] "Uuencode," Wikipedia: The Free Encyclopedia, http://en.wikipedia.org/wiki/Uuencode, 14 Oct 2005.

[30] Internet Engineering Task Force, "The Base16, Base32, and Base64 Data Encodings," RFC 3548, July 2003.

[31] Cray Corporation, Man Page Collection: Bioinformatics Library Procedures, 2004, available online: http://www.cray.com/craydoc/manuals/S-2397-21/S-2397-21.pdf.

[32] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification, IEEE Std. 802.11g-2003.