Architectural Support for SWAR Text Processing with Parallel Bit Streams: The Inductive Doubling Principle

Robert D. Cameron Dan Lin
School of Computing Science, Simon Fraser University
{cameron, lindanl}@cs.sfu.ca

Abstract

Parallel bit stream algorithms exploit the SWAR (SIMD within a register) capabilities of commodity processors in high-performance text processing applications such as UTF-8 to UTF-16 transcoding, XML parsing, string search and regular expression matching. Direct architectural support for these algorithms in future SWAR instruction sets could further increase performance as well as simplifying the programming task. A set of simple SWAR instruction set extensions are proposed for this purpose based on the principle of systematic support for inductive doubling as an algorithmic technique. These extensions are shown to significantly reduce instruction count in core parallel bit stream algorithms, often providing a 3X or better improvement. The extensions are also shown to be useful for SWAR programming in other application areas, including providing a systematic treatment for horizontal operations. An implementation model for these extensions involves relatively simple circuitry added to the operand fetch components in a pipelined processor.

Categories and Subject Descriptors C.1.2 [*PROCESSOR ARCHITECTURES*]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms Design, Performance

Keywords inductive doubling, parallel bit streams, SWAR

1. Introduction

In the landscape of parallel computing research, finding ways to exploit intrachip (multicore) and intraregister (SWAR) parallelism for text processing and other non-numeric applications is particularly challenging. Indeed, in documenting this landscape, a widely cited Berkeley study [1] identifies the finite-state machine algorithms associated with text processing to be the hardest of the thirteen "dwarves" to parallelize, concluding that nothing seems to help. Indeed, the study even speculates that applications in this area may simply be "embarrassingly sequential," easy to tackle for traditional sequential processing approaches suitable for uniprocessors, but perhaps fundamentally unsuited to parallel methods.

One approach that shows some promise, however, is the method of parallel bit streams, recently applied to UTF-8 to UTF-16 transcoding [2, 3], XML parsing [4, 6] and amino acid sequencing[5]. In this method, byte-oriented character data is first transposed to eight parallel bit streams, one for each bit position within the character code units (bytes). Loading bit stream data into 128-bit registers, then, allows data from 128 consecutive code units to be represented and processed at once. Bitwise logic and shift operations, bit scans, population counts and other bit-based operations are then used to carry out the work.

In application to UTF-8 to UTF-16 transcoding, a 3X to 25X speed-up is achieved in using parallel bit stream techniques on SWAR-capable uniprocessors employing the SSE or Altivec instruction sets[3]. In the broader context of XML parsing, further applications of these techniques demonstrate the utility of parallel bit stream techniques in delivering performance benefits through a significant portion of the web technology stack. In an XML statistics gathering application, including the implementation of XML well-formedness checking, an overall 3X to 10X performance improvement is achieved in using the parallel bit stream methods in comparison with a similarly coded application using such well known parsers as Expat and Xerces [4]. In an application involving transformation between different XML formats (GML and SVG), an implementation using parallel bit stream technology required a mere 15 cycles per byte, while a range of other technologies required from 25 to 200 cycles per byte [6]. Ongoing work is further applying the parallel bit stream methods to parallel hash value computation and parallel regular expression matching for the pur-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-215-3/09/03. . . $5.00

pose of validating XML datatype declarations in accord with XML Schema [4].

Given these promising initial results in the application of parallel bit stream methods, what role might architectural support play in further enhancing this route to parallelization of text processing? This paper addresses this question through presentation and analysis of a constructive proposal: a set of SWAR instruction set features based on the principle of systematic support for inductive doubling algorithms. Inductive doubling refers to a general property of certain kinds of algorithm that systematically double the values of field widths or other data attributes with each iteration. In essence, the goal of the proposed features is to support such algorithms with specific facilities to transition between successive power-of-2 field widths. These transitions are quite frequent in parallel bit stream programming as well as other applications. The specific features presented herein will be referred to as IDISA: inductive doubling instruction set architecture.

The remainder of this paper is organized as follows. The second section of this paper introduces IDISA and the SWAR notation used throughout this paper. The third section moves on to discuss an evaluation methodology for IDISA in comparison to two reference architectures motivated by the SSE and Altivec instruction sets. The fourth section provides a short first example of the inductive doubling principle in action through the case of population count. Sections 5 through 7 then address the application of IDISA to core algorithms in text processing with parallel bit streams. The eighth section then considers the potential role of IDISA in supporting applications beyond parallel bit streams. Section 9 addresses IDISA implementation while Section 10 concludes the paper with a summary of results and directions for further work.

2. Inductive Doubling Architecture

This section presents IDISA as an idealized model for a SWAR instruction set architecture designed specifically to support inductive doubling algorithms. The architecture is idealized in the sense that we concentrate on only the necessary features for our purpose, without enumerating the additional operations that would be required for SWAR applications in other domains. The goal is to focus on the principles of inductive doubling support in a way that can accommodate a variety of realizations as other design constraints are brought to bear on the overall instruction set design. First we introduce a simple model and notation for SWAR operations in general and then present the four key features of IDISA.

IDISA supports typical SWAR integer operations using a *three-register model* involving two input registers and one output register. Each register is of size $N = 2^K$ bits, for some integer K. Typical values of K for commodity processors include $K = 6$ for the 64-bit registers of Intel MMX and Sun VIS technology, $K = 7$ for the 128-bit registers of SSE and Altivec technology and $K = 8$ for the upcoming Intel AVX technology. The registers may be partitioned into N/n fields of width $n = 2^k$ bits for some values of $k \leq K$. Typical values of k used on commodity processors include $k = 3$ for SWAR operations on 8-bit fields (bytes), $k = 4$ for operations on 16-bit fields and $k = 5$ for operations on 32-bit fields. Whenever a register r is partitioned into n-bit fields, the fields are indexed $r_n[0]$ through $r_n[N/n-1]$. Field $r_n[i]$ consists of bits $i \times n$ through $(i+1) \times n - 1$ of register r, using big-endian numbering.

Let `simd<n>` represent the class of SWAR operations defined on fields of size n using C++ template syntax. Given a binary function F_n on n-bit fields, we denote the SWAR version of this operation as `simd<n>::F`. Given two input registers `a` and `b` holding values a and b, respectively, the operation `r=simd<n>::F(a,b)` stores the value r in the output register `r` as determined by the simultaneous calculation of individual field values in accord with the following equation.

$$r_i = F_n(a_i, b_i) \quad (1)$$

For example, addition(`add`), subtraction (`sub`) and shift left logical (`sll`) may be defined as binary functions on n-bit unsigned integers as follows.

$$\mathrm{add}_n(a,b) = (a+b) \bmod 2^n \quad (2)$$
$$\mathrm{sub}_n(a,b) = (a-b+2^n) \bmod 2^n \quad (3)$$
$$\mathrm{sll}_n(a,b) = a \times 2^{b \bmod n} \bmod 2^n \quad (4)$$

The Altivec instruction set includes each of these operations for 8, 16 and 32-bit fields directly following the three-register model. The SSE set uses a two-register model with the result being copied back to one of the input operands. However, the C language intrinsics commonly used to access these instructions reflect the three-register model. The SSE set extends these operations to include operations on 64-bit fields, but constrains the shift instructions, requiring that all field shifts by the same amount.

Given these definitions and notation, we now present the four key elements of an inductive doubling architecture. The first is a definition of a core set of binary functions on n-bit fields for all field widths $n = 2^k$ for $0 \leq k \leq K$. The second is a set of *half-operand modifiers* that allow the inductive processing of fields of size $2n$ in terms of combinations of n-bit values selected from the fields. The third is the definition of packing operations that compress two consecutive registers of n-bit values into a single register of $n/2$-bit values. The fourth is the definition of merging operations that produce a set of $2n$ bit fields by concatenating corresponding n-bit fields from two parallel registers. Each of these features is described below.

For the purpose of direct and efficient support for inductive doubling algorithms, the provision of a core set of operations at field widths of 2 and 4 as well as the more traditional field widths of 8, 16 and 32 is key. In essence, inductive doubling algorithms work by establishing some base property at

either single or 2-bit fields. Each iteration of the algorithm then goes on to establish the property for the power-of-2 field width. In order for this inductive step to be most conveniently and efficiently expressed, the core operations needed for the step should be available at each field width. In the case of work with parallel bit streams, the operations `add`, `sub`, `sll`, `srl` (shift right logical), and `rotl` (rotate left) comprise the core. In other domains, additional operations may be usefully included in the core depending on the work that needs to be performed at each inductive doubling level.

Note that the definition of field widths $n = 2^k$ for $0 \leq k \leq K$ also includes fields of width 1. These are included for logical consistency, but are easily implemented by mapping directly to appropriate bitwise logic operations, which we assume are also available. For example, `simd<1>::add` is equivalent to `simd_xor`, the bitwise exclusive-or operation.

The second key facility of the inductive doubling architecture is the potential application of half-operand modifiers to the fields of either or both of the operands of a SWAR operation. These modifiers select either the low $n/2$ bits of each n-bit field (modifier "`l`") or the high $n/2$ bits (modifier "`h`"). When required, the modifier "`x`" means that the full n bits should be used, unmodified. The semantics of these modifiers are given by the following equations.

$$l(r_n) = r_n \bmod 2^{n/2} \quad (5)$$
$$h(r_n) = r_n / 2^{n/2} \quad (6)$$
$$x(r_n) = r_n \quad (7)$$

In our notation, the half-operand modifiers are specified as optional template (compile-time) parameters for each of the binary functions. Thus, `simd<4>::add<h,l>(a,b)` is an operation which adds the 2-bit quantity found in the high 2-bits of each 4-bit field of its first operand (`a`) together with the corresponding 2-bit quantity found in the low 2-bits of its second operand (`b`). In general, the purpose of the half-operand modifiers in support of inductive doubling is to allow the processing of n-bit fields to easily expressed in terms of combination of the results determined by processing $n/2$ bit fields.

The third facility of the inductive doubling architecture is a set of pack operations at each field width $n = 2^k$ for $1 \leq k \leq K$. The field values of `r=simd<n>::pack(a,b)` are defined by the following equations.

$$r_{n/2}[i] = \text{conv}(a_n[i], n/2), \text{for } i < N/n \quad (8)$$
$$r_{n/2}[i] = \text{conv}(b_n[i - N/n], n/2), \text{for } i \geq N/n \quad (9)$$

Here conv is a function which performs conversion of an n-bit value to an $n/2$ bit value by signed saturation (although conversion by unsigned saturation would also suit our purpose).

Half-operand modifiers may also be used with the pack operations. Thus packing with conversion by masking off all but the low $n/2$ bits of each field may be be performed using the operation `simd<n>::pack<l,l>`.

The final facility of the inductive doubling architecture is a set of merging operations that produce $2n$-bit fields by concatenating corresponding n-bit fields from the operand registers. The operations `r=simd<n>::mergeh(a,b)` and `s=simd<n>::mergel(a,b)` are defined by the following equations.

$$r_{2n}[i] = a[i] \times 2^n + b[i] \quad (10)$$
$$s_{2n}[i] = a[i + N/(2n)] \times 2^n + b[i + N/(2n)] \quad (11)$$

Both SSE and Altivec provide versions of pack and merge operations for certain field widths. The pack operations are provided with operands having 16-bit or 32-bit fields on each platform, although with some variation in how conversion is carried out. The merge operations are provided at 8-bit, 16-bit and 32-bit field widths on both architectures and also at the 64-bit level on SSE.

This completes the description of IDISA. As described, many of the features are already available with the SWAR facilities of existing commodity processors. The extensions enumerated here are relatively straightforward. The innovation is to specifically tackle the design of facilities to offer systematic support for transitions between power-of-2 field widths. As we shall show in the remainder of this paper, these facilities can dramatically reduce instruction count in core parallel bit stream algorithms, with a factor of 3 reduction being typical.

3. Evaluation Methodology

IDISA represents a set of instruction set features that could potentially be added to any SWAR processor. The goal in this paper is to evaluate these features independent of artifacts that may be due to any particular realization, while still considering realistic models based on existing commodity instruction set architectures. For the purpose of IDISA evaluation, then, we define two reference architectures. For concreteness, IDISA and the two reference architectures will each be considered as 128-bit processors employing the three-register SWAR model defined in the previous section.

Reference architecture A (RefA) consists of a limited register processor providing a set of core binary operations defined for 8, 16, 32 and 64 bit fields. The core binary operations will be assumed to be those defined by the SSE instruction set for 16-bit fields. In addition, we assume that shift immediate operations for each field width exist, e.g., `simd<8>::srli<1>(x)` for a right logical shift of each 8-bit field by 1. We also assume that a constant load operation `simd::constant<n>(c)` loads the constant value c into each n bit field. The pack and merge facilities of SSE will also be assumed.

Reference architecture B (RefB) consists of a register-rich processor incorporating all the operations of reference architecture A as well as the following additional facilities inspired by the Altivec instruction set. For each of the 8,

16, 32 and 64 bit widths, a binary rotate left logical instruction `simd<n>::rotl(a,b)` rotates each field of a by the rotation count in the corresponding field of b. A three-input `simd<1>::if(a,b,c)` bitwise logical operator implements the logic $r = a \wedge b \vee \neg a \wedge c$, patterned after the Altivec `vec_sel` operation. Finally, a `simd<8>::permute(a,b,c)` selects an arbitrary permutation of bytes from the concatenation of a and b based on the set of indices in c.

Two versions of IDISA are assessed against these reference architectures as follows. IDISA-A has all the facilities of RefA extended with half-operand modifiers and all core operations at field widths of 2, 4 and 128. IDISA-B is similarly defined and extended based on RefB. Algorithms for both RefA and IDISA-A are assessed assuming that any required constants must be loaded as needed; this reflects the limited register assumption. On the other, assessment for both RefB and IDISA-B will make the assumption that sufficiently many registers exist that constants can be kept preloaded.

In each case, the processors are assumed to be pipelined processors with a throughput of one SWAR instruction each processor cycle for straight-line code free of memory access. This assumption makes for straightforward performance evaluation based on instruction count for straight-line computational kernels. Furthermore, the assumption also eliminates artifacts due to possibly different latencies in reference and IDISA architectures. Because the same assumption is made for reference and IDISA architectures, determination of the additional circuit complexity due to IDISA features is unaffected by the assumption.

In the remainder of this paper, then, IDISA-A and IDISA-B models are evaluated against their respective reference architectures on straight-line computational kernels used in parallel bit stream processing and other applications. As XML and other sequential text processing applications tend to use memory in an efficient streaming model, the applications tend to be compute-bound rather than IO-bound. Thus, the focus on computational kernels addresses the primary concern for performance improvement of these applications.

The additional circuit complexity to realize IDISA-A and IDISA-B designs over their reference models will be addressed in the penultimate section. That discussion will focus primarily on the complexity of implementing half-operand modifier logic, but will also address the extension of the core operations to operate on 2-bit, 4-bit and 128-bit fields, as well.

4. Population Count

As an initial example to illustrate the principle of inductive doubling in practice, consider the problem of *population count*: determining the number of one bits within a particular bit field. It is important enough for such operations as calculating Hamming distance to be included as a built-in instruction on some processors. For example, the SPU of the

```
c = (x & 0x55555555) + ((x >> 1) & 0x55555555);
c = (c & 0x33333333) + ((c >> 2) & 0x33333333);
c = (c & 0x0F0F0F0F) + ((c >> 4) & 0x0F0F0F0F);
c = (c & 0x00FF00FF) + ((c >> 8) & 0x00FF00FF);
c = (c & 0x0000FFFF) + ((c >>16) & 0x0000FFFF);
```

Figure 1. Population Count Reference Algorithm

```
c = simd<2>::add<h,l>(x, x);
c = simd<4>::add<h,l>(c, c);
c = simd<8>::add<h,l>(c, c);
c = simd<16>::add<h,l>(c, c);
c = simd<32>::add<h,l>(c, c);
```

Figure 2. IDISA Population Count

Cell Broadband Engine has a SWAR population count instruction `si_cntb` for simultaneously determining the number of 1 bits within each byte of a 16-byte register. In text processing with parallel bit streams, population count has direct application to keeping track of line numbers for error reporting, for example. Given a bit block identifying the positions of newline characters within a block of characters being processed, the population count of the bit block can be used to efficiently and conveniently be used to update the line number upon completion of block processing.

Figure 1 presents a traditional divide-and-conquer implementation for a 32-bit integer `x` adapted from Warren [11], while Figure 2 shows the corresponding IDISA implementation for a vector of 32-bit fields. Each implementation employs five steps of inductive doubling to produce population counts within 32 bit fields. The traditional implementation employs explicit masking and shifting operations, while these operations are implicit within the semantics of the inductive doubling instructions shown in Figure 2. In each implementation, the first step determines the the population counts within 2-bit fields by adding the high bit of each such field to the low bit to produce a set of 2-bit counts in `c`. In the second step, the counts within 4-bit fields of `c` are determined by adding the counts of the corresponding high and low 2-bit subfields. Continuing in this fashion, the final population counts within 32-bit fields are determined in five steps.

With the substitution of longer mask constants replicated for four 32-bit fields, the implementation of Figure 1 can be directly adapted to SWAR processing using 128-bit registers. Each binary operator is replaced by a corresponding binary SWAR operation. Without the IDISA features, a straightforward RefA implementation of population count for 32-bit fields thus employs 10 operations to load or generate mask constants, 10 bitwise-and operations, 5 shifts and 5 adds for a total of 30 operations to complete the task. Employing optimization identified by Warren, this can be reduced to 20 operations, 5 of which are required to generate mask constants. At the cost of register pressure, it is possible that these

constants could be kept preloaded in long vector processing. In accord with our evaluation model, the RefB cost is thus 15 operations. As the IDISA implementation requires no constants at all, both the IDISA-A and IDISA-B cost is 5 operations. At our assumed one CPU cycle per instruction model, IDISA-A offers a 4X improvement over RefA, while IDISA-B offers a 3X improvement over its comparator.

The pattern illustrated by population count is typical. An inductive doubling algorithm of n steps typically applies mask or shift operations at each step for each of the two operands being combined in the step. In general, the mask constants shown in Figure 1 recur; these are termed "magic masks" by Knuth [7]. If the algorithm employs a single operation at each step, then a total of $3n$ operations are the required in a RefB implementation, and possibly $4n$ for a RefA implementation including the cost of loading masks. IDISA-A and IDISA-B implementations typically eliminate the explicit mask and shift operations through appropriate half-operand modifiers, reducing the total instruction count to n. Thus a 3X to 4X improvement obtains in these cases.

5. Transposition to Parallel Bit Streams

In this section, we consider the first major application of IDISA: transposition of byte stream data to parallel bit stream form. Of course, this operation is critical to the method of parallel bit streams and all applications of the method can benefit from a highly efficient transposition process. Before considering how the IDISA supports this transposition process, however, we first consider algorithms on existing architectures. Two algorithms are presented; the best of these requires 72 SWAR operations under the RefB model to perform transposition of eight serial registers of byte stream data into eight parallel registers of bit stream data.

We then go on to show how the transposition problem can be solved using IDISA-A or IDISA-B with a mere 24 three-register SWAR operations. We also show that this is optimal for any three-register instruction set model.

Figure 3 illustrates the input-output requirements of the transposition problem. We assume that inputs and outputs are each SWAR registers of size $N = 2^K$ bits. The input consists of N bytes of serial byte data, stored consecutively in eight SWAR registers each holding $N/8$ bytes. The output consists of eight parallel registers, one each for the eight individual bit positions within a byte. Upon completion of the transposition process, each output register is to hold the N bits corresponding to the selected bit position in the sequence of N input bytes.

5.1 Bit Gathering Algorithm

One straightforward algorithm for implementing the transposition process takes advantage of SWAR bit gathering operations that exist on some architectures. This operation gathers one bit per byte from a particular position within

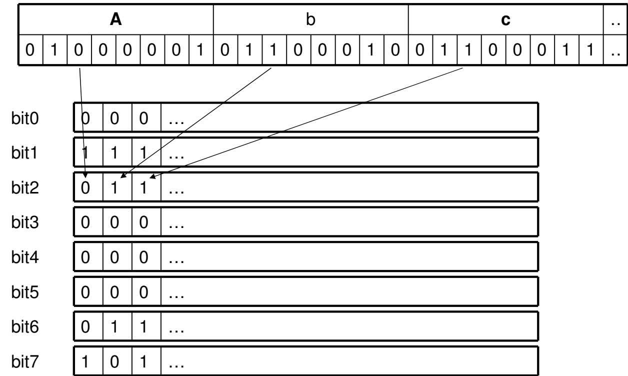

Figure 3. Serial to Parallel Transposition

each byte of a register. For example, the `pmovmskb` operation of the Intel SSE instruction set forms a 16-bit mask consisting of the high bit of each byte. Similarly, the `si_gbb` operation of the synergistic processing units of the Cell Broadband Engine gathers together the low bit of each byte.

Using bit gathering, each bit stream of output is assembled 16 positions at a time. Bits from the input register must be shifted into position, the gather operation performed and the result inserted into position in the output register. For the 8 streams, this requires at least 22 operations for 16 positions, or 176 operations to complete the transposition task.

5.2 BytePack Algorithm

A more efficient transposition algorithm on commodity SWAR architectures involves three stages of binary division transformation. This is similar to the three stage bit matrix inversion described by Warren [11], although modified to use SWAR operations. In each stage, input streams are divided into two half-length output streams. The first stage separates the bits at even numbered positions from those at odd number positions. The two output streams from the first stage are then further divided in the second stage. The stream comprising even numbered bits from the original byte stream divides into one stream consisting of bits from positions 0 and 4 of each byte in the original stream and a second stream consisting of bits from positions 2 and 6 of each original byte. The stream of bits from odd positions is similarly divided into streams for bits from each of the positions 1 and 5 and bits from positions 2 and 6. Finally, each of the four streams resulting from the second stage are divided into the desired individual bit streams in the third stage.

The binary division transformations are accomplished in each stage using byte packing, shifting and masking. In each stage, a transposition step combines each pair of serial input registers to produce a pair of parallel output registers. Figure 4 shows a stage 1 transposition step in a Ref B implementation. Using the permute facility, the even and odd bytes, respectively, from two serial input registers `s0` and `s1`

```
even={0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30};
odd ={1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31};
mask = simd<8>::constant(0xAA);
t0 = simd<8>::permute(s0, s1, even);
t1 = simd<8>::permute(s0, s1, odd);
p0 = simd_if(mask, t0, simd<16>::srli<1>(t1));
p1 = simd_if(mask, simd<16>::slli<1>(t0), t1);
```

Figure 4. RefB Transposition Step in BytePack Stage 1

are packed into temporary registers `t0` and `t1`. The even and odd bits are then separated into two parallel output registers `p0` and `p1` by selecting alternating bits using a mask. This step is applied four times in stage 1; stages 2 and 3 also consist of four applications of a similar step with different shift and mask constants. Overall, 6 operations per step are required, yielding a total of 72 operations to transpose 128 bytes to parallel bit stream form in the RefB implementation.

In a RefA implementation, byte packing may also be achieved by the `simd<16>::pack` with 4 additional operations to prepare operands. Essentially, the RefB implementation uses single permute instructions to implement the equivalent of `simd<16>::pack<h,h>(s0, s1)` and `simd<16>::pack<l,l>(s0, s1)`. The RefA implementation also requires 3 logic operations to implement each `simd_if`. Assuming that mask loads are only need once per 128 bytes, a total of 148 operations are required in the RefB implementation.

5.3 Inductive Halving Algorithm

Using IDISA, it is possible to design a transposition algorithm that is both easier to understand and requires many fewer operations than the the BytePack algorithm described above. We call it the inductive halving algorithm for serial to parallel transposition, because it proceeds by reducing byte streams to two sets of nybble streams in a first stage, dividing the nybble streams into streams of bit-pairs in a second stage and finally dividing the bit-pair streams into bit streams in the third stage.

Figure 5 shows one step in stage 1 of the inductive halving algorithm, comprising just two IDISA-A operations. The `simd<8>::pack<h,h>` operation extracts the high nybble of each byte from the input registers, while the `simd<8>::pack<l,l>` operation extracts the low nybble of each byte. As in the BytePack algorithm, this step is applied 4 times in stage 1, for a total of 8 operations.

Stage 2 of the inductive halving algorithm reduces nybble streams to streams of bit pairs. The basic step in this algorithm consists of one `simd<4>::pack<h,h>` operation to extract the high pair of each nybble and one `simd<4>::pack<l,l>` operation to extract the low pair of each nybble. Four applications of this step complete stage 2.

Stage 3 similarly uses four applications of a step that uses a `simd<2>::pack<h,h>` operation to extract the high bit of each pair and a `simd<2>::pack<l,l>` to extract the low bit

```
p0 = simd<8>::pack<h,h>(s0, s1);
p1 = simd<8>::pack<l,l>(s0, s1);
```

Figure 5. Step in Inductive Halving Algorithm Stage 1

of each pair. Under either IDISA-A or IDISA-B models, the complete algorithm to transform eight serial byte registers s0 through s7 into the eight parallel bit stream registers bit0 through bit7 requires a mere 24 instructions per 128 input bytes.

5.4 Optimality of the Inductive Halving Algorithm

Here we show that the inductive halving algorithm presented in the previous subsection is optimal in the following sense: no other algorithm on any 3-register SWAR architecture can use fewer than 24 operations to transform eight serial registers of byte stream data into eight parallel registers of bit stream data. By 3-register SWAR architecture, we refer to any architecture that uses SWAR instructions consistent with our overall model of binary operations using two input register operands to produce one output register value.

Observe that the N data bits from each input register must be distributed $N/8$ each to the 8 output registers by virtue of the problem definition. Each output register can effectively be given a 3-bit address; the partitioning problem can be viewed as moving data to the correct address. However, each operation can move results into at most one register. At most this can result in the assignment of one correct address bit for each of the N input bits. As all $8N$ input bits need to be moved to a register with a correct 3-bit address, a minimum of 24 operations is required.

5.5 End-to-End Significance

In a study of several XML technologies applied to the problem of GML to SVG transformation, the parabix implementation (parallel bit streams for XML) was found to the fastest with a cost of approximately 15 CPU cycles per input byte [6]. Within parabix, transposition to parallel bit stream form requires approximately 1.1 cycles per byte [4]. All other things being equal, a 3X speed-up of transposition alone would improve end-to-end performance in a complete XML processing application by more than 4%.

6. Parallel to Serial Conversion

Parallel bit stream applications may apply string editing operations in bit space to substitute, delete or insert parallel sets of bits at particular positions. In such cases, the inverse transform that converts a set of parallel bit streams back into byte space is needed. In the example of UTF-8 to UTF-16 transcoding, the inverse transform is actually used twice for each application of the forward transform, to separately compute the high and low byte streams of each UTF-16 code unit. Those two byte streams are subsequently merged to form the final result.

Algorithms for performing the inverse transform mirror those of the forward transform, employing SWAR merge operations in place of pack operations. The best algorithm known to us on the commodity SWAR architectures takes advantage of versions of the simd<8>::mergeh and simd<8>::mergel operations that are available with each of the SSE and Altivec instruction sets. To perform the full inverse transform of 8 parallel registers of bit stream data into 8 serial registers of byte stream data, a RefA implementation requires 120 operations, while a RefB implementation reduces this to 72.

An algorithm employing only 24 operations using IDISA-A/B is relatively straightforward.. In stage 1, parallel registers for individual bit streams are first merged with bit-level interleaving using simd<1>::mergeh and simd<8>::mergel operations. For each of the four pairs of consecutive even/odd bit streams (bit0/bit1, bit2/bit3, bit4/bit5, bit6/bit7), two consecutive registers of bit-pair data are produced. In stage 2, simd<2>::mergeh and simd<2>::mergel are then applied to merge to bit-pair streams to produce streams of nybbles for the high and low nybble of each byte. Finally, the nybble streams are merged in stage 3 to produce the desired byte stream data. The full inductive doubling algorithm for parallel to serial transposition thus requires three stages of 8 instructions each. The algorithm is again optimal, requiring the fewest operations of any possible algorithm using any 3-register instruction set model.

The existence of high-performance algorithms for transformation of character data between byte stream and parallel bit stream form in both directions makes it possible to consider applying these transformations multiple times during text processing applications. Just as the time domain and frequency domain each have their use in signal processing applications, the byte stream form and parallel bit stream form can then each be used at will in character stream applications.

7. Parallel Bit Deletion

Parallel bit deletion is an important operation that allows string editing operations to be carried out while in parallel bit stream form. It is also fundamental to UTF-8 to UTF-16 transcoding using parallel bit streams, allowing the excess code unit positions for UTF-8 two-, three- and four-byte sequences to be deleted once the sixteen parallel bit streams of UTF-16 have been computed [3].

Parallel bit deletion is specified using a deletion mask. A deletion mask is defined as a bit stream consisting of 1s at positions identifying bits to be deleted and 0s at positions identifying bits to be retained. For example, consider an 8-bit deletion mask 10100010 and two corresponding 8-element parallel bit streams abcdefgh and ABCDEFGH. Parallel deletion of elements from both bit streams in accordance with the mask yields two five element streams, i.e., bdefh and BDEFH.

Bit deletion may be performed using the parallel-prefix compress algorithm documented by Warren and attributed to Steele [11]. This algorithm uses only logic and shifts with a constant parameter to carry out the deletion process. However, it requires k^2 preprocessing steps for a final field width parameter of size 2^k, as well as 4 operations per deletion step per stream. Using the inductive doubling instruction set architecture it is possible to carry out bit deletion much more efficiently.

Deletion within fixed size fields or registers may produce results that are either left justified or right-justified. For example, a five-element stream bdefh within an eight-element field may be represented as either bdefhxxx or xxxbdefh, with don't care positions marked 'x'. Concatenating an adjacent right-justified result with a left-justified result produces an important intermediate form known as a *central deletion result*. For example, xxbd and efhx may be respective right-justified and left-justified results from the application of the 4-bit deletion masks 1010 and 0010 to the two consecutive 4-element stream segments abcd and efgh. Concatenation of xxbd and efhx produces the central result xxbdefhx, which may easily be converted to a either a left or a right justified 8-element result by an appropriate shift operation.

The observation about how two n-bit central deletion results can combine to yield a $2n$ central deletion result provides the basis for an inductive doubling algorithm. Figure 6 illustrates the inductive process for the transition from 8-bit central deletion results to 16-bit central deletion results. The top row shows the original deletion mask, while the second row shows the original bit stream to which deletions are to be applied, with deleted bits zeroed out. The third row shows the central result for each 8-bit field as the result of the previous inductive step.

To perform the $8 \rightarrow 16$ central deletion step, we first form the population counts of 4-bit fields of the original deletion mask as shown in row 4 of Figure 6. Note that in right-justifying an 8-bit central result, we perform a right shift by the population count of the low half of the field. Similarly, left-justification requires a left-shift by the population count in the high half of the field.

The left and right shifts can be performed simultaneously using a rotate left instruction. Right justification by shifting an n bit field i positions to the right is equivalent to a left rotate of $n - i$ positions. Given a register value c8 preloaded with the value 8 in each 8-bit field, the right rotation amounts are computed by the operation rj=simd<8>::sub<x,l>(c8, cts_4) producing values shown in row 5, except that don't care fields (which won't be subsequently used) are marked XX.

The left shift amounts are calculated by lj=simd<8>::srli<4>(cts_4) producing the values shown in row 6, and are then combined with

delmask	1001	1100	0100	1111	0111	0010	0011	0010
bits	0bc0	00gh	i0kl	0000	q000	uv0x	yz00	CD0F
rslt_8	00bcgh00		0ikl0000		000quvx0		00yzCDF0	
cts_4	2	2	1	4	3	1	2	1
rj	6		XX		7		XX	
lj	XX		1		XX		2	
rot_8	6		1		7		2	
rslt_16	0000bcghikl00000				0000quvxyzCDF000			

Figure 6. Example $8 \rightarrow 16$ Step in Deletion by Central Result Induction

```
y = simd<2>::xor<h,l>(x, x);
y = simd<4>::xor<h,l>(y, y);
y = simd<8>::xor<h,l>(y, y);
y = simd<16>::xor<h,l>(y, y);
y = simd<32>::xor<h,l>(y, y);
```

Figure 7. IDISA Parity Implementation

the right shift amounts by the selection operation `rot_8=simd_if(mask0xFF00, rj, lj)` as shown in row 7. Using these computed values, the inductive step is completed by application of the operation `rslt_16=simd<8>::rotl(rslt_8, rot_8)` as shown in row 8.

At each inductive doubling level, it requires 4 operations to compute the required deletion information and one operation per bit stream to perform deletion. Note that, if deletion is to be applied to a set of eight parallel bit streams, the computed deletion information is used for each stream without recomputation, thus requiring 12 operations per inductive level.

In comparison to the parallel-prefix compress method, the method of central deletion results using the inductive doubling architecture has far fewer operations. The total preprocessing cost is $4k$ for k steps of deletion by central result induction versus $4k^2$ for the parallel-prefix method. Using the computed deletion operation, only a single SWAR rotate operation per bit stream per level is needed, in comparison with 4 operations per level for parallel-prefix compress.

8. Beyond Parallel Bit Streams

IDISA has a variety of applications in domains beyond text processing with parallel bit streams. We present a number of examples in this section, including, most significantly, a full general solution to the problem of supporting *horizontal* SWAR operations.

8.1 Parity

Parity has important applications for error-correcting codes such as the various Hamming codes for detecting and correcting numbers of bit errors dependent on the number of parity bits added. Figure 7 shows an IDISA-A parity implementation with only 5 operations required for 32-bit fields, slightly more than a 2X improvement over the 11 operations required in a RefB implementation following Warren [11]. The improvement is less than 3X seen in other cases because one of the operands need not be modified before applying the exclusive-or operation.

8.2 Bit Reverse

Bit reverse is an important operation needed in a number of low level codecs. Following Warren's inductive doubling implementation using masks and shifts [11], a RefA implementation on 32-bit fields requires 28 operations, while a straightforward IDISA-A implementation using `simd<n>::rotli` at each inductive doubling level requires only 5 operations.

8.3 Packed DNA Representation

DNA sequences are often represented as strings consisting of the four nucleotide codes A, C, G and T. Internally, these sequences are frequently represented in internal form as packed sequences of 2-bit values. The IDISA `simd<8>:pack` and `simd<4>:pack` operations allow these packed representations to be quickly computed from byte-oriented string values by two steps of inductive halving. Similarly, conversion back to string form can use two steps of inductive merging. Without direct support for these pack and merge operations, the SWAR implementations of these conversions require the cost of explicit masking and shifting in combination with the 16-bit to 8-bit packing and 8-bit to 16-bit merging operations supported by existing SWAR facilities on commodity processors.

8.4 String/Decimal/Integer Conversion

Just as DNA sequences represent an important use case for SWAR operations on 2-bit fields, packed sequences of decimal or hexadecimal digits represent a common use case for 4-bit fields. These representations can be used both as an intermediate form in numeric string to integer conversion and as a direct representation for packed binary coded decimal.

Figure 8 shows a three-step inductive doubling implementation for conversion of 32-bit packed BCD values to integer form. The 32-bit value consists of 8 4-bit decimal digits. Pairs of digits are first combined by multiplying the

```
b=(d & 0x0F0F0F0F) + 10 * ((d >> 4) & 0x0F0F0F0F)
b=(d & 0x00FF00FF) + 100 * ((d >> 8) & 0x00FF00FF)
b=(d & 0x0000FFFF) + 10000 * (d >> 16)
```

Figure 8. BCD to Integer Reference Algorithm

```
t1=simd<8>:constant(10)
t2=simd<16>:constant(100)
t3=simd<32>:constant(10000)
b=simd<8>::add<x,l>(simd<8>::mult<h,x>(d,t1), d)
b=simd<16>::add<x,l>(simd<16>::mult<h,x>(b,t2), b)
b=simd<32>::add<x,l>(simd<32>::mult<h,x>(b,t3), b)
```

Figure 9. IDISA BCD to Integer

higher digit of the pair by 10 and adding. Pairs of these two-digit results are then further combined by multiplying the value of the higher of the two-digit results by 100 and adding. The final step is to combine four-digit results by multiplying the higher one by 10000 and adding. Overall, 20 operations are required for this implementation as well as the corresponding RefA implementation for sets of 32-bit fields. Under the RefB model, preloading of 6 constants into registers for repeated use can reduce the number of operations to 14 at the cost of register pressure.

The IDISA implementation of this algorithm is shown in Figure 9. This implementation shows an interesting variation in the use of half-operand modifiers, with only one operand of each of the addition and multiplication operations modified at each level. Overall, the IDISA-A implementation requires 9 operations, while the IDISA-B model requires 6 operations with 3 preloaded registers. In either case, this represents more than a 2X reduction in instruction count as well as a 2X reduction in register pressure.

8.5 Further Applications

Further applications of IDISA can often be found by searching for algorithms employing the magic masks `0x55555555`, `0x33333333`, and so on. Examples include the bit-slice implementation of AES [9] and integer contraction and dilation for quadtrees and octrees[10] and Morton-ordered arrays [8]. Pixel packing from 32 bit fields into a 5:5:5 representation is a further application of parallel bit deletion.

8.6 Systematic Support for Horizontal SWAR Operations

In SWAR parlance, *horizontal* operations are operations which combine values from two or more fields of the same register, in contrast to the normal *vertical* operations which combine corresponding fields of different registers. Horizontal operations can be found that combine two (e.g., `haddpd` on SSE3), four (e.g, `si_orx` on SPU), eight (e.g, `psadbw` on SSE) or sixteen values (e.g., `vcmpequb` on Altivec). Some horizontal operations have a vertical component as well. For example, `psadbw` first forms the absolute value of the difference of eight corresponding byte fields before performing horizontal add of the eight values, while `vsum4ubs` on Altivec performs horizontal add of sets of four unsigned 8-bit fields within one register and then combines the result horizontally with corresponding 32-bit fields of a second registers.

The space of potential horizontal operations thus has many dimensions, including not only the particular combining operation and the operand field width, but also the number of fields being combined, whether a vertical combination is applied and whether it is applied before or after the horizontal operation and what the nature of the vertical combining operation is. Within this space, commodity SWAR architectures tend to support only a very few combinations, without any particular attempt at systematic support for horizontal operations in general.

In contrast to this *ad hoc* support on commodity processors, IDISA offers a completely systematic treatment of horizontal operations without any special features beyond the inductive doubling features already described. In the simplest case, any vertical operation `simd<n>::F` on n-bit fields gives rise to an immediate horizontal operation `simd<n>::F<h,l>(r, r)` for combining adjacent pairs of $n/2$ bit fields. For example, `simd<16>::add<h,l>` adds values in adjacent 8 bit fields to produce 16 bit results, while `simd<32>::min<h,l>` can produce the minimum value of adjacent 16-bit fields. Thus any binary horizontal operation can be implemented in a single IDISA instruction making use of the `<h,l>` operand modifier combination.

Horizontal combinations of four adjacent fields can also be realized in a general way through two steps of inductive doubling. For example, consider the or-across operation `si_orx` of the SPU, that performs a logical or operation on four 32-bit fields. This four field combination can easily be implemented with the following two operations.

```
t = simd<64>::or<h,l>(x, x)
t = simd<128>::or<h,l>(t, t)
```

In general, systematic support for horizontal combinations of sets of 2^h adjacent fields may be realized through h inductive double steps in a similar fashion. Thus, IDISA essentially offers systematic support for horizontal operations entirely through the use of `<h,l>` half-operand modifier combinations.

Systematic support for general horizontal operations under IDISA also creates opportunity for a design tradeoff: offsetting the circuit complexity of half-operand modifiers with potential elimination of dedicated logic for some *ad hoc* horizontal SWAR operations. Even if legacy support for these operations is required, it may be possible to provide that support through software or firmware rather than a full hardware implementation. Evaluation of these possibilities in the context of particular architectures is a potential area for further work.

9. Implementation

IDISA may be implemented as a software abstraction on top of existing SWAR architectures or directly in hardware. In this section, we briefly discuss implementation of IDISA libraries before moving on to consider hardware design. Although a full realization of IDISA in hardware is beyond our current capabilities, our goal is to develop a sufficiently detailed design to assess the costs of IDISA implementation in terms of the incremental complexity over the RefA and RefB architectures. The cost factors we consider, then, are the implementation of the half-operand modifiers, and the extension of core operations to the 2-bit, 4-bit and 128-bit field widths. In each case, we also discuss design tradeoffs.

9.1 IDISA Libraries

Implementation of IDISA instructions using template and macro libraries has been useful in developing and assessing the correctness of many of the algorithms presented here. Although these implementations do not deliver the performance benefits associated with direct hardware implementation of IDISA, they have been quite useful in providing a practical means for portable implementation of parallel bit stream algorithms on multiple SWAR architectures. However, one additional facility has also proven necessary for portability of parallel bit stream algorithms across big-endian and little-endian architectures: the notion of shift-forward and shift-back operations. In essence, shift forward means shift to the left on little-endian systems and shift to the right on big-endian systems, while shift back has the reverse interpretation. Although this concept is unrelated to inductive doubling, its inclusion with the IDISA libraries has provided a suitable basis for portable SWAR implementations of parallel bit stream algorithms. Beyond this, the IDISA libraries have the additional benefit of allowing the implementation of inductive doubling algorithms at a higher level abstraction, without need for programmer coding of the underlying shift and mask operations.

9.2 IDISA Model

Figure 10 shows a block diagram for a pipelined SWAR processor implementing IDISA. The SWAR Register File (SRF) provides a file of $R = 2^A$ registers each of width $N = 2^K$ bits. IDISA instructions identified by the Instruction Fetch Unit (IFU) are forwarded for decoding to the SWAR Instruction Decode Unit (SIDU). This unit decodes the instruction to produce signals identifying the source and destination operand registers, the half-operand modifiers, the field width specification and the SWAR operation to be applied.

The SIDU supplies the source register information and the half-operand modifier information to the SWAR Operand Fetch Unit (SOFU). For each source operand, the SIDU provides an A-bit register address and two 1-bit signals h and l indicating the value of the decoded half-operand modifiers for this operand. Only one of these values may

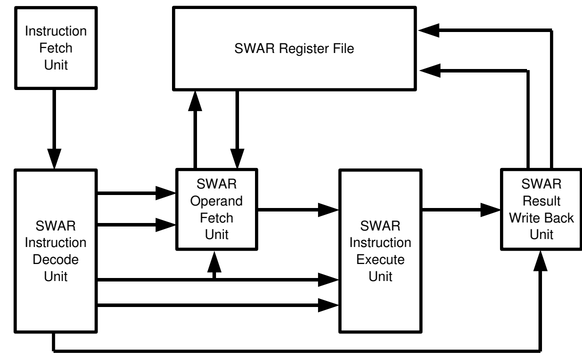

Figure 10. IDISA Block Diagram

be 1; both are 0 if no modifier is specified. The SIDU also supplies decoded field width signals w_k for each field width 2^k to both the SOFU and to the SWAR Instruction Execute Unit (SIEU). Only one of the field width signals has the value 1. The SIDU also supplies decoded SWAR opcode information to SIEU and a decoded A-bit register address for the destination register to the SWAR Result Write Back Unit (SRWBU).

The SOFU is the key component of the IDISA model that differs from that found in a traditional SWAR processor. For each of the two A-bit source register addresses, SOFU is first responsible for fetching the raw operand values from the SRF. Then, before supplying operand values to the SIEU, the SOFU applies the half-operand modification logic as specified by the h, l, and field-width signals. The possibly modified operand values are then provided to the SIEU for carrying out the SWAR operations. A detailed model of SOFU logic is described in the following subsection.

The SIEU differs from similar execution units in current commodity processors primarily by providing SWAR operations at each field width $n = 2^k$ for $0 \leq k \leq K$. This involves additional circuitry for field widths not supported in existing processors. In our evaluation model, IDISA-A adds support for 2-bit, 4-bit and 128-bit field widths in comparison with the RefA architecture, while IDISA-B similarly extends RefB.

When execution of the SWAR instruction is completed, the result value is then provided to the SRWBU to update the value stored in the SRF at the address specified by the A-bit destination operand.

9.3 Operand Fetch Unit Logic

The SOFU is responsible for implementing the half-operand modification logic for each of up to two input operands fetched from SRF. For each operand, this logic is implemented using the decoded half-operand modifiers signals h and l, the decoded field width signals w_k and the 128-bit

operand value r fetched from SRF to produce a modified 128-bit operand value s following the requirements of equations (4), (5) and (6) above. Those equations must be applied for each possible modifier and each field width to determine the possible values $s[i]$ for each bit position i. For example, consider bit position 41, whose binary 7-bit address is 0101001. Considering the address bits left to right, each 1 bit corresponds to a field width for which this bit lies in the lower $n/2$ bits (widths 2, 16, 64), while each 0 bit corresponds to a field width for which this bit lies in the high $n/2$ bits. In response to the half-operand modifier signal h, bit $s[41]$ may receive a value from the corresponding high bit in the field of width 2, 16 or 64, namely $r[40]$, $r[33]$ or $r[9]$. Otherwise, this bit receives the value $r[41]$, in the case of no half-operand modifier, or a low half-operand modifier in conjunction with a field width signal w_2, w_{16} or w_{64}. The overall logic for determining this bit value is thus given as follows.

$$\begin{aligned} s[41] &= h \wedge (w_2 \wedge r[40] \vee w_{16} \wedge r[33] \vee w_{64} \wedge r[9]) \\ &\quad \vee \neg h \wedge (\neg l \vee w_2 \vee w_{16} \vee w_{64}) \wedge r[41] \end{aligned}$$

Similar logic is determined for each of the 128 bit positions. For each of the 7 field widths, 64 bits are in the low $n/2$ bits, resulting in 448 2-input and gates for the $w_k \wedge r[i]$ terms. For 120 of the bit positions, or gates are needed to combine these terms; $441 - 120 = 321$ 2-input or gates are required. Another 127 2-input and gates combine these values with the h signal. In the case of a low-half-operand modifier, the or-gates combining w_k signals can share circuitry. For each bit position $i = 2^k + j$ one additional or gate is required beyond that for position j. Thus 127 2-input or gates are required. Another 256 2-input and gates are required for combination with the $\neg h$ and $r[i]$ terms. The terms for the low and high half-operand modifiers are then combined with an additional 127 2-input or gates. Thus, the circuitry complexity for the combinational logic implementation of half-operand modifiers within the SOFU is 1279 2-input gates per operand, or 2558 gates in total.

The gate-level complexity of half-operand modifiers as described is nontrivial, but modest. However, one possible design tradeoff is to differentiate the two operands, permitting a high half-operand modifier to be used only with the first operand and a low-modifier to be used only with the second operand. This would exclude `<h,h>` and `<l,l>` modifier combinations and also certain combinations for noncommutative core operations. The principal consequence for the applications considered here would be with respect to the pack operations in forward transposition, but it may be possible to address this through SIEU circuitry. If this approach were to be taken, the gate complexity of half-operand modification would be reduced by slightly more than half.

9.4 2-Bit and 4-Bit Field Widths

Beyond the half-operand modifiers, extension of core SWAR operations to 2-bit and 4-bit field widths is critical to inductive doubling support. The principal operations that need to be supported in this way are addition, pack, merge merge, and rotate.

Addition for 4-bit fields in a 128-bit SWAR processor may be implemented as a modification to that for 8-bit fields by incorporating logic to disable carry propagation at the 16 mid-field boundaries. For 2-bit fields, disabling carry propagation at 32 additional boundaries suffices, although it may be simpler to directly implement the simple logic of 2-bit adders.

Pack and merge require bit selection logic for each field width. A straightforward implementation model for each operation uses 128 2-input and gates to select the desired bits from the operand registers and another 128 2-input or gates to feed these results into the destination register.

Rotation for 2-bit fields involves simple logic for selecting between the 2 bits of each field of the operand being rotated on the basis of the low bit of each field of the rotation count. Rotation for 4-bit fields is more complex, but can also be based on 1-of-4 selection circuitry involving the low 2 bits of the rotation count fields.

9.5 128-Bit Field Widths

For completeness, the IDISA model requires core operations to be implemented at the full register width, as well as power-of-2 partitions. This may be problematic for operations such as addition due to the inherent delays in 128-bit carry propagation. However, the primary role of 128 bit operations in inductive doubling is to combine two 64-bit fields using `<h,l>` operand combinations. In view of this, it may be reasonable to define hardware support for such combinations to be based on 64-bit logic, with support for 128-bit logic implemented through firmware or software.

9.6 Final Notes and Further Tradeoffs

In order to present IDISA as a concept for design extension of any SWAR architecture, our discussion of gate-level implementation is necessarily abstract. Additional circuitry is sure to be required, for example, in implementation of SIDU. However, in context of the 128-bit reference architectures studied, our analysis suggests realistic IDISA implementation well within a 10,000 gate budget.

However, the additional circuitry required may be offset by elimination of special-purpose instructions found in existing processors that could instead be implemented through efficient IDISA sequences. These include examples such as population count, count leading and/or trailing zeroes and parity. They also include specialized horizontal SWAR operations. Thus, design tradeoffs can be made with the potential of reducing the chip area devoted to special purpose instructions in favor of more general IDISA features.

10. Conclusions

In considering the architectural support for SWAR text processing using the method of parallel bit streams, this paper has presented the principle of inductive doubling and a realization of that principle in the form of IDISA, a modified SWAR instruction set architecture. IDISA offers support for SWAR operations at all power-of-2 field widths, including 2-bit and 4-bit widths, in particular, as well as half-operand modifiers and pack and merge operations to support efficient transition between successive power-of-two field widths. The principal innovation is the notion of half-operand modifiers that eliminate the cost associated with the explicit mask and shift operations required for such transitions.

Several algorithms key to parallel bit stream methods have been examined and shown to benefit from dramatic reductions in instruction count compared to the best known algorithms on reference architectures. This includes both a reference architecture modeled on the SWAR capabilities of the SSE family as well as an architecture incorporating the powerful permute or shuffle capabilities found in Altivec or Cell BE processors. In the case of transposition algorithms to and from parallel bit stream form, the architecture has been shown to make possible straightforward inductive doubling algorithms with a 3X speedup over the best known versions on permute-capable reference architectures, achieving the lowest total number of operations of any possible 3-register SWAR architecture.

Applications of IDISA in other areas have also been examined. The support for 2-bit and 4-bit field widths in SWAR processing is beneficial for packed DNA representations and packed decimal representations respectively. Additional inductive doubling examples are presented and the phenomenon of power-of-2 transitions discussed more broadly. Most significantly, IDISA supports a fully general approach to horizontal SWAR operations, offering a considerable improvement over the *ad hoc* sets of special-purpose horizontal operations found in existing SWAR instruction sets.

An IDISA implementation model has been presented employing a customized operand fetch unit to implement the half-operand modifier logic. Gate-level implementation of this unit and operations at the 2-bit and 4-bit field widths have been analyzed and found to be quite reasonable within a 10,000 gate budget. Design tradeoffs to reduce the cost have also been discussed, possibly even leading to a net complexity reduction through elimination of instructions that implement special-case versions of inductive doubling.

Future research may consider the extension of inductive doubling support in further ways. For example, it may be possible to develop a pipelined architecture supporting two or three steps of inductive doubling in a single operation.

Acknowledgments

This research was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Robert D. Cameron. `u8u16` – a high-speed UTF-8 to UTF-16 transcoder using parallel bit streams. Technical Report TR 2007-18, Simon Fraser University, Burnaby, BC, Canada, 2007.

[3] Robert D. Cameron. A case study in SIMD text processing with parallel bit streams. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, Utah, 2008.

[4] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance XML parsing using parallel bit stream technology. In *CASCON '08: Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario , Canada, 2008.

[5] James R. Green, Hanan Mahmoud, and Michel Dumontier. Towards real time protein identification using the Cell BE. In *Workshop on Cell BE and Heterogeneous Multicore Systems: Architectures and Applications at CASCON '08*, Toronto, Ontario , Canada, 2008.

[6] Kenneth S. Herdy, David S. Burggraf, and Robert D. Cameron. High performance GML to SVG transformation for the visual presentation of geographic data in web-based mapping systems. In *Proceedings of SVG Open 2008*, Nuremburg, Germany, 2008.

[7] Donald E. Knuth. The Art of Computer Programming Volume 4 Pre-Fascicle 1A: Bitwise Tricks and Techniques. Draft of 22 December 2008, Stanford University.

[8] R. Raman and D.S. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, 2008.

[9] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In David Pointcheval, Yi Mu, and Kefei Chen, editors, *CANS*, volume 4301 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2006.

[10] Leo Stocco and Günther Schrack. Integer dilation and contraction for quadtrees and octrees. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 426–428, Victoria, B.C., 1995.

[11] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.