`u8u16` - A High-Speed UTF-8 to UTF-16 Transcoder Using Parallel Bit Streams

Robert D. Cameron
Technical Report 2007-18
School of Computing Science
Simon Fraser University

June 21, 2007

Contents

1 Introduction

The *u8u16* program is a high-performance UTF-8 to UTF-16 transcoder using a SIMD programming technique based on parallel bit streams. In essence, character data is processed in blocks of size `BLOCKSIZE`, where `BLOCKSIZE` is the number of bits that are held in a SIMD register. A set of parallel registers each contain one bit per character code unit for `BLOCKSIZE` consecutive code unit positions. For example, in UTF-8 processing, eight parallel registers are used for the eight individual bits of the UTF-8 code units.

The *u8u16* transcoder written in this way takes advantage the pipelining and SIMD capabilities of modern processor architectures to achieve substantially better performance than byte-at-a-time conversion.

The *u8u16* program is open source software written by Professor Rob Cameron of Simon Fraser University. International Characters, Inc., distributes and licenses *u8u16* to the general public under the terms of Open Software License 3.0. The program contains patent-pending technology of International Characters, Inc., licensed under the terms of OSL 3.0. Commercial licenses are also available.

The *u8u16* program is written using the CWEB system for literate programming in C (Donald E. Knuth and Silvio Levy, The CWEB System of Structured Documentation, Addison Wesley, 1993).

2 Idealized SIMD Library

The *u8u16* program is written using a library of idealized SIMD operations. The library simplifies implementation on multiple SIMD instruction set architectures by providing a common set of operations available on each architecture. It also simplifies programming of SIMD algorithms in general, by providing an orthogonal set of capabilities that apply at each field width 2, 4, 8, *ldots*, `BLOCKSIZE`, as well as providing half-operand modifiers to support inductive doubling algorithms. These simplifications lead to substantial reductions in instruction count for key algorithms such as transposition, bit counting and bit deletion.

Beyond simplification of the programming task, the use of the idealized operations anticipates the day that SIMD architectures will provide native support for the inductive doubling extensions, at which point reduced instruction counts ought to translate directly to further performance improvements.

The idealized library defines a type **SIMD_type** as the type of SIMD register values and a set of operations on these registers. Detailed documentation of the idealized library is provided elsewhere, but the following brief description introduces the operations used by *u8u16*.

In general, operations specify a field width as part of the operation name. For example, *r0* = *simd_add_16*(*r1*, *r2*) adds corresponding 16-bit fields of two register values *r1* and *r2* to produce a result register *r0*. Working with 128-bit registers, for example, 8 simultaneous additions are performed. Similarly, *simd_add_2* allows simultaneous addition of 64 sets of 2-bit fields, *simd_add_4* allows simultaneous addition of 32 sets of 4-bit fields and so on. Subtraction follows the same pattern: *simd_sub_8* provides for 16 simultaneous subtractions within 8-bit fields, while *simd_sub_128* provides subtraction of entire register values considered as 128-bit fields.

Shift and rotate operations allow shifting of the field values of one register by independent shift counts in the fields of a second register. For example, $b = \mathit{simd\_rotl\_4}(a, s)$ assigns to b the result of rotating each 4-bit field of a left by the corresponding 4-bit shift value from register s. Similarly $\mathit{simd\_sll\_8}$ and $\mathit{simd\_srl\_2}$ represent shift left logical of 8-bit fields and shift right logical of 2-bit fields. Howevever, shift values are interpreted as modulo field size; the maximum shift within an 8-bit field is thus 7.

Each of the shift operations also has an immediate form, in which all fields are shifted by a particular constant value. Thus $\mathit{simd\_rotli\_8}(a, 2)$ yields a result in which the 8-bit fields of a have each been rotated left 2 positions. These immediate forms are both convenient for programming and support efficient implementation.

The prefix *sisd* (single-instruction single-data) is available for any of the arithmetic and shift operations when the entire register is to be considered as a single field. Thus, $\mathit{sisd\_slli}(a, 1)$ is equivalent to $\mathit{simd\_slli\_64}(a, 1)$ when working with 64-bit SIMD registers (e.g., MMX) or $\mathit{simd\_slli\_128}(a, 1)$ when working with 128-bit registers (Altivec, SSE).

Half-operand modifiers permit convenient transitions between processing of $n/2$ bit fields and n bit fields in inductive algorithms. Given an operation on n bit fields, the l modifier specifies that only the low $n/2$ bits of each input field are to be used for the operation, while the h modifier specifies that the high $n/2$ bits are to be used (shifted into the low $n/2$ positions). For example, $\mathit{cts2} = \mathit{simd\_add\_2\_lh}(a, a)$ specifies that the low bit of each 2-bit field of a is to be added to the high bit of each 2-bit field. Each 2-bit field of $\mathit{cts2}$ is thus the count of the number of bits (0, 1 or 2) in the corresponding 2-bit field of a. Similarly, $\mathit{cts4} = \mathit{simd\_add\_4\_lh}(\mathit{cts2}, \mathit{cts2})$ determines each 4-bit field of $\mathit{cts4}$ as the sum of the low 2-bit count and the high 2-bit count of each 4-bit field of $\mathit{cts2}$. The bit counts in 4-bit fields of a are thus computed after two inductive steps. Further operations $\mathit{cts8} = \mathit{simd\_add\_8\_lh}(\mathit{cts4}, \mathit{cts4})$ and $\mathit{cts16} = \mathit{simd\_add\_16\_lh}(\mathit{cts8}, \mathit{cts8})$ give the bit counts in the 16-bit fields of a after four total steps of inductive doubling.

Merge and pack operations also support inductive doubling transitions. The $\mathit{simd\_mergeh\_4}(a, b)$ operation returns the result of merging alternating 4-bit fields from the high halves of a and b, while $\mathit{simd\_mergel\_4}(a, b)$ performs the complementary merge from the low halves. The $\mathit{simd\_pack\_4}(a, b)$ packs each consecutive pair of 4-bit fields from the concatenation of a and b into a single 4-bit field by unsigned saturation.

Bitwise logical operations are considered as simultaneous logical operations with an implicit field size of 1. The functions $\mathit{simd\_and}$, $\mathit{simd\_or}$, $\mathit{simd\_xor}$, $\mathit{simd\_andc}$ and $\mathit{simd\_nor}$ each take two register values as arguments and return the register value representing the specified bitwise logical combination of the register values. The one-argument function $\mathit{simd\_not}$ provides bitwise negation, while the three-argument function $\mathit{simd\_if}(a, b, c)$ is equivalent to $\mathit{simd\_or}(\mathit{simd\_and}(v, a), \mathit{simd\_andc}(c, a))$.

The $\mathit{simd\_const}$ operations load a specified immediate constant into all fields of a register. Thus, $\mathit{simd\_const\_8}(12)$ loads the value 12 into every byte, while $\mathit{simd\_const\_1}(1)$ loads 1 into every bit.

Loading and storing of registers to and from memory is provided by the $\mathit{sisd\_load\_unaligned}$, $\mathit{sisd\_load\_aligned}$, $\mathit{sisd\_store\_unaligned}$ and $\mathit{sisd\_store\_aligned}$ operations. The $\mathit{sisd\_load\_unaligned}(\mathit{addr})$ operation returns the register value at memory address addr, which may have arbitrary alignment, while $\mathit{sisd\_load\_aligned}(\mathit{addr})$ requires that the address be aligned. The $\mathit{sisd\_store\_unaligned}(\mathit{val}, \mathit{addr})$ operations stores a value at an arbitrary memory address addr, while the aligned version again requires that the address be aligned on a natural boundary.

The actual SIMD library used is selected based on a configuration option `U8U16_TARGET`. Definitions for particular targets are given in later sections.

2 ⟨Import idealized SIMD operations 2⟩ ≡

```
#ifndef U8U16_TARGET
#error "No␣U8U16_TARGET␣defined."
#endif
```

See also chunks 76, 97, and 103.

This code is used in chunk 5.

3 Calling Conventions - iconv-based

The *u8u16* routine uses an interface based on the *iconv* specification (*iconv* - codeset conversion function, The Single UNIX Specification, Version 2, 1997, The Open Group). However, the first argument to *iconv* (the conversion descriptor) is omitted as *u8u16* is specialized to the task of UTF-8 to UTF-16 conversion.

In normal operation, *u8u16* is given an input buffer ∗∗*inbuf* containing ∗*inbytesleft* bytes of UTF-8 data and an output buffer ∗∗*outbuf* having room for ∗*outbytesleft* bytes of UTF-16 output. UTF-8 data is converted and written to the output buffer so long as the data is valid in accord with UTF-8 requirements and neither the input nor output buffer is exhausted.

3
```
size_t u8u16(char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft);
```

¶ On exit, the value of ∗*inbuf* will be adjusted to the first position after the last UTF-8 input sequence processed and the the value ∗*inbytesleft* will be reduced by the number of bytes of UTF-8 data converted. Similarly, the value of ∗*outbuf* will be adjusted to the first position after the last converted UTF-16 unit written to output and the value ∗*outbytesleft* will be reduced to indicate the remaining space available in the output buffer. Contents of the output buffer after the end of converted output (that is, at locations past the final value of ∗*outbuf*) are undefined. Preexisting values in these locations may or may not be preserved.

If it is necessary to ensure that preexisting values of the output buffer past the final value of ∗*outbuf* are preserved, a buffered version of *u8u16* may be used. This routine is also used internally by *u8u16* when the size of the output buffer is potentially too small to hold UTF-16 data corresponding to ∗*inbytesleft* UTF-8 data.

4
```
size_t buffered_u8u16(char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft);
```

¶ Return codes for each of *u8u16* and *buffered_u8u16* are set as follows. If the complete input buffer is successfully converted, ∗*inbytesleft* will be set to 0 and 0 will be returned as the result of the call. Otherwise -1 is returned and *errno* is set according to the encountered error. If conversion terminates due to an an invalid UTF-8 sequence, *errno* is set to `EILSEQ` If conversion terminates because insufficient space remains in the output buffer *errno* is set to `E2BIG`. If the input buffer ends with an incomplete UTF-8 code unit sequence, *errno* is set to `EINVAL`.

For compatibility with *iconv*, if either *inbuf* or ∗*inbuf* is null, the call to *u8u16* is treated as an initialization call with an empty buffer and 0 is returned. If either *outbuf* or ∗*outbuf* is null, the output buffer is treated as having no room.

The top-level structure of *u8u16* implements the initial null checks on the buffer pointers and determines whether the output buffer is of sufficient size to avoid overflow with the aligned output methods of *u8u16*. The efficient blok processing code of *u8u16* is used directly if the output buffer is guaranteed to be big enough; otherwise the buffered version of the converter is invoked.

5
```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
  ⟨ Import idealized SIMD operations 2 ⟩
  ⟨ Type declarations 8 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Endianness definitions 16 ⟩
  size_t u8u16(char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft)
  {
    ⟨ Local variable declarations 9 ⟩
    if (inbuf ∧ *inbuf ∧ outbuf ∧ *outbuf)     /* are all non-NULL */ {
      if (⟨ Output buffer is guaranteed to be big enough 6 ⟩)
        ⟨ Main block processing algorithm of u8u16 24 ⟩
    else return buffered_u8u16(inbuf, inbytesleft, outbuf, outbytesleft);
  }
  else if (inbuf ≡ Λ ∨ ∗inbuf ≡ Λ ∨ ∗inbytesleft ≡ 0) return (size_t) 0;
  else { errno = E2BIG; return (size_t) −1; }
}
```

¶ In the case of aligned output, guaranteeing that the output buffer contains sufficient space to hold the UTF-16 data corresponding to ∗*inbytesleft* UTF-8 code units requires that the buffer contain 2 bytes for each input byte plus any additional space needed to reach the next **BytePack** alignment boundary.

6 **#define** *align_ceil*(*addr*) (*addr* + `PACKSIZE` − 1) % `PACKSIZE`

⟨ Output buffer is guaranteed to be big enough 6 ⟩ ≡

```
  (int) ∗outbuf + ∗outbytesleft ≥ align_ceil((int) ∗outbuf + 2 ∗ (∗inbytesleft))
```

This code is used in chunk 5.

4 Data Representations

4.1 Serial BytePacks and Parallel BitBlocks

The **BytePack** and the **BitBlock** are the two fundamental types used by the *u8u16* program for data held in SIMD registers, representing, respectively, the byte-oriented and bit-oriented views of character data.

8 **#define** `PACKSIZE` **sizeof**(**SIMD_type**)
#define `BLOCKSIZE` (**sizeof**(**SIMD_type**) ∗ 8)

⟨ Type declarations 8 ⟩ ≡

```
  typedef SIMD_type BytePack;
  typedef SIMD_type BitBlock;
```

This code is used in chunk 5.

¶ A block of UTF-8 character data is initially loaded and represented as a series of eight consecutive bytepacks *U8s0*, *U8s1*, ..., *U8s7*. Upon transposition to bit-parallel form, the same data is represented as eight parallel bitblocks *u8bit0*, *u8bit1*, ..., *u8bit7*.

9 ⟨ Local variable declarations 9 ⟩ ≡

```
  BytePack U8s0, U8s1, U8s2, U8s3, U8s4, U8s5, U8s6, U8s7;
  BitBlock u8bit0, u8bit1, u8bit2, u8bit3, u8bit4, u8bit5, u8bit6, u8bit7;
```

See also chunks 10, 12, 13, 14, 25, 37, 42, 46, 59, 83, 89, 92, 95, 100, and 106.
This code is used in chunk 5.

¶ UTF-16 data may then be computed in the form of sixteen parallel bitblocks for each of the individual bits of UTF-16 code units. The registers *u16hi0*, *u16hi1*, ..., *u16hi7* are used to store this data for the high byte of each UTF-16 code unit, while *u16lo0*, *u16lo1*, ..., *u16lo7* are used for the bits of the corresponding low byte. Upon conversion of the parallel bit stream data back to byte streams, registers *U16h0*, *U16h1*, ..., *U16h7* are used for the high byte of each UTF-16 code unit, while *U16l0*, *U16l1*, ..., *U16l7* are used for the corresponding low byte. Finally, the registers *U16s0*, *U16s1*, ..., *U16s15* are then used for UTF-16 data in serial code unit form (2 bytes per code unit) after merging the high and low byte streams.

10 ⟨Local variable declarations 9⟩ +≡

```
BitBlock u16hi0, u16hi1, u16hi2, u16hi3, u16hi4, u16hi5, u16hi6, u16hi7;
BitBlock u16lo0, u16lo1, u16lo2, u16lo3, u16lo4, u16lo5, u16lo6, u16lo7;
BytePack U16h0, U16h1, U16h2, U16h3, U16h4, U16h5, U16h6, U16h7;
BytePack U16l0, U16l1, U16l2, U16l3, U16l4, U16l5, U16l6, U16l7;
BytePack U16s0, U16s1, U16s2, U16s3, U16s4, U16s5, U16s6, U16s7, U16s8, U16s9, U16s10,
    U16s11, U16s12, U16s13, U16s14, U16s15;
```

4.2 BitBlock Control Masks

¶ Several block-based bitmasks are used to control processing. The *input_select_mask* is used to identify with a 1 bit those positions within the block to be included in processing. Normally consisting of a block of all ones during processing of full blocks, *input_select_mask* allows a final partial block to be processed using the logic for full blocks with a masking operation to zero out data positions that are out of range. In some algorithm variations, certain positions in the processing of full blocks may be zeroed out with *input_select_mask* in order to handle alignment or boundary issues.

12 ⟨Local variable declarations 9⟩ +≡

```
BitBlock input_select_mask;
```

¶ When UTF-8 validation identifies errors in the input stream, the positions of these errors are signalled by *error_mask*. Errors in the scope of *input_select_mask* represent and must be reported as actual errors in UTF-8 sequence formation. In processing a final partial block, an error just past the final *input_select_mask* position indicates an incomplete UTF-8 sequence at the end of the input.

13 ⟨Local variable declarations 9⟩ +≡

```
BitBlock error_mask;
```

¶ The generation of UTF-16 data is controlled by *delmask*. One doublebyte code unit is to be generated for each nondeleted position; while no output is generated for deleted positions.

14 ⟨Local variable declarations 9⟩ +≡

```
BitBlock delmask;
```

5 Endianness

¶ Depending on the endianness of the machine, the ordering of bytes within SIMD registers may be from left to right (big endian) or right to left (little endian). Upon transformation to parallel bit streams, the ordering of bit values may similarly vary. To remove the dependencies of core bit-stream algorithms on endianness, logical "shift forward" and "shift back" operations are defined for bitblocks.

16 ⟨Endianness definitions 16⟩ ≡

```
#if BYTE_ORDER ≡ BIG_ENDIAN
#define sisd_sfl(blk, n)sisd_srl (blk, n)
#define sisd_sbl(blk, n)sisd_sll (blk, n)
#define sisd_sfli(blk, n)sisd_srli (blk, n)
#define sisd_sbli(blk, n)sisd_slli (blk, n)
#endif
#if BYTE_ORDER ≡ LITTLE_ENDIAN
#define sisd_sfl(blk, n)sisd_sll (blk, n)
#define sisd_sbl(blk, n)sisd_srl (blk, n)
#define sisd_sfli(blk, n)sisd_slli (blk, n)
```

```
#define sisd_sbli(blk, n)sisd_srli  (blk, n)
#endif
#ifndef PARTITIONED_BLOCK_MODE
#define bitblock_sfl(blk, n)sisd_sfl  (blk, n)
#define bitblock_sbl(blk, n)sisd_sbl  (blk, n)
#define bitblock_sfli(blk, n)sisd_sfli  (blk, n)
#define bitblock_sbli(blk, n)sisd_sbli  (blk, n)
#endif
```

See also chunks 17 and 105.

This code is used in chunk 5.

¶ The *u8u16* program may also be configured to assemble UTF-16 code units in accord with either the UTF-16BE conventions (the default) or those of UTF-16LE. To accomodate these variations, the *u16_merge0* and *u16_merge1* macros are defined to control assembly of UTF-16 doublebyte streams from the individual high and low byte streams.

17 ⟨ Endianness definitions 16 ⟩ +≡

```
#if BYTE_ORDER ≡ BIG_ENDIAN
#ifdef UTF16_LE
#define u16_merge0(a, b)simd_mergeh_8  (b, a)
#define u16_merge1(a, b)simd_mergel_8  (b, a)
#endif
#ifndef UTF16_LE
#define u16_merge0(a, b)simd_mergeh_8  (a, b)
#define u16_merge1(a, b)simd_mergel_8  (a, b)
#endif
#endif
#if BYTE_ORDER ≡ LITTLE_ENDIAN
#ifdef UTF16_LE
#define u16_merge0(a, b)simd_mergel_8  (a, b)
#define u16_merge1(a, b)simd_mergeh_8  (a, b)
#endif
#ifndef UTF16_LE
#define u16_merge0(a, b)simd_mergel_8  (b, a)
#define u16_merge1(a, b)simd_mergeh_8  (b, a)
#endif
#endif
```

6 Transposition Between Serial Byte Streams and Parallel Bit Streams

Core to the *u8u16* transcoder are algorithms for converting serial byte streams of character data to bit parallel form (*s2p*), and the corresponding inverse transformation (*p2s*).

Conversion of serial byte data to and from parallel bit streams is performed using either generic transposition algorithms for the idealized SIMD architecture or algorithms better tuned to specific processor architectures. The generic versions described here are the simplest and most efficient in terms of idealized operations, requiring a mere 24 operations for transposition of a data block in either direction. However, these versions use operations which must be simulated on existing architectures. The appendices provide implementations for common alternative architectures. The specific algorithm to be chosen is specified by the value of the preprocessor constants `S2P_ALGORITHM` and `P2S_ALGORITHM`.

6.1 Serial To Parallel Transposition

The *s2p_ideal* transposition is achieved in three stages. In the first stage, the input stream of serial byte data is separated into two streams of serial nybble data. Eight consecutive registers of byte data *r0*, *r1*, *r2*, *r3*, *r4*, *r5*, *r6*, *r7* are transformed into two sets of four parallel registers: *bit0123_0*, *bit0123_1*, *bit0123_2*, *bit0123_3* for the high nybbles of each byte and *bit4567_0*, *bit4567_1*, *bit4567_2*, *bit4567_3* for the low nybbles of each byte. In the second stage, each stream of serial byte data is transformed into two streams of serial bit pairs. For example, serial nybble data in the four registers *bit0123_0*, *bit0123_1*, *bit0123_2*, and *bit0123_3* is transformed into two sets of parallel registers for the high and low bit pairs, namely *bit01_0* and *bit01_1* for bits 0 and 1 of the original byte data and *bit23_0* and *bit23_1* for bits 2 and 3 of the original byte data. The third stage completes the transposition process by transforming streams of bit pairs into the individual bit streams. Using the idealized architecture, each of these stages is implemented using a set of eight *simd_pack* operations.

```
19  #define s2p_ideal(s0, s1, s2, s3, s4, s5, s6, s7, p0, p1, p2, p3, p4, p5, p6, p7)
            {
              BitBlock bit0123_0, bit0123_1, bit0123_2, bit0123_3, bit4567_0, bit4567_1, bit4567_2,
                  bit4567_3;
              BitBlock bit01_0, bit01_1, bit23_0, bit23_1, bit45_0, bit45_1, bit67_0, bit67_1;
              bit0123_0 = simd_pack_8_hh(s0, s1);
              bit0123_1 = simd_pack_8_hh(s2, s3);
              bit0123_2 = simd_pack_8_hh(s4, s5);
              bit0123_3 = simd_pack_8_hh(s6, s7);
              bit4567_0 = simd_pack_8_ll(s0, s1);
              bit4567_1 = simd_pack_8_ll(s2, s3);
              bit4567_2 = simd_pack_8_ll(s4, s5);
              bit4567_3 = simd_pack_8_ll(s6, s7);
              bit01_0 = simd_pack_4_hh(bit0123_0, bit0123_1);
              bit01_1 = simd_pack_4_hh(bit0123_2, bit0123_3);
              bit23_0 = simd_pack_4_ll(bit0123_0, bit0123_1);
              bit23_1 = simd_pack_4_ll(bit0123_2, bit0123_3);
              bit45_0 = simd_pack_4_hh(bit4567_0, bit4567_1);
              bit45_1 = simd_pack_4_hh(bit4567_2, bit4567_3);
              bit67_0 = simd_pack_4_ll(bit4567_0, bit4567_1);
              bit67_1 = simd_pack_4_ll(bit4567_2, bit4567_3);
              p0 = simd_pack_2_hh(bit01_0, bit01_1);
              p1 = simd_pack_2_ll(bit01_0, bit01_1);
              p2 = simd_pack_2_hh(bit23_0, bit23_1);
              p3 = simd_pack_2_ll(bit23_0, bit23_1);
              p4 = simd_pack_2_hh(bit45_0, bit45_1);
              p5 = simd_pack_2_ll(bit45_0, bit45_1);
              p6 = simd_pack_2_hh(bit67_0, bit67_1);
              p7 = simd_pack_2_ll(bit67_0, bit67_1);
            }
    ⟨ Transpose to parallel bit streams u8bit0 through u8bit7 19 ⟩ ≡
    #if (S2P_ALGORITHM ≡ S2P_IDEAL)
    #if (BYTE_ORDER ≡ BIG_ENDIAN)
      s2p_ideal(U8s0, U8s1, U8s2, U8s3, U8s4, U8s5, U8s6, U8s7,
      u8bit0, u8bit1, u8bit2, u8bit3, u8bit4, u8bit5, u8bit6, u8bit7)
    #endif
    #if (BYTE_ORDER ≡ LITTLE_ENDIAN)
      s2p_ideal(U8s7, U8s6, U8s5, U8s4, U8s3, U8s2, U8s1, U8s0,
      u8bit0, u8bit1, u8bit2, u8bit3, u8bit4, u8bit5, u8bit6, u8bit7)
```

```
#endif
#endif
```

See also chunk 71.

This code is used in chunk 24.

6.2 Parallel to Serial Transposition

The inverse *p2s_ideal* transposition creates serial byte data by successively merging 8 parallel bit streams to become 4 streams of bit pairs, merging these 4 streams of bit pairs to become 2 streams of nybbles, and finally merging the 2 streams of nybbles into a serial byte stream.

```
20 #define p2s_ideal(p0, p1, p2, p3, p4, p5, p6, p7, s0, s1, s2, s3, s4, s5, s6, s7)
       {
         BitBlock bit01_r0, bit01_r1, bit23_r0, bit23_r1, bit45_r0, bit45_r1, bit67_r0, bit67_r1;
         BitBlock bit0123_r0, bit0123_r1, bit0123_r2, bit0123_r3, bit4567_r0, bit4567_r1, bit4567_r2,
             bit4567_r3;
         bit01_r0 = simd_mergeh_1(p0, p1);
         bit01_r1 = simd_mergel_1(p0, p1);
         bit23_r0 = simd_mergeh_1(p2, p3);
         bit23_r1 = simd_mergel_1(p2, p3);
         bit45_r0 = simd_mergeh_1(p4, p5);
         bit45_r1 = simd_mergel_1(p4, p5);
         bit67_r0 = simd_mergeh_1(p6, p7);
         bit67_r1 = simd_mergel_1(p6, p7);
         bit0123_r0 = simd_mergeh_2(bit01_r0, bit23_r0);
         bit0123_r1 = simd_mergel_2(bit01_r0, bit23_r0);
         bit0123_r2 = simd_mergeh_2(bit01_r1, bit23_r1);
         bit0123_r3 = simd_mergel_2(bit01_r1, bit23_r1);
         bit4567_r0 = simd_mergeh_2(bit45_r0, bit67_r0);
         bit4567_r1 = simd_mergel_2(bit45_r0, bit67_r0);
         bit4567_r2 = simd_mergeh_2(bit45_r1, bit67_r1);
         bit4567_r3 = simd_mergel_2(bit45_r1, bit67_r1);
         s0 = simd_mergeh_4(bit0123_r0, bit4567_r0);
         s1 = simd_mergel_4(bit0123_r0, bit4567_r0);
         s2 = simd_mergeh_4(bit0123_r1, bit4567_r1);
         s3 = simd_mergel_4(bit0123_r1, bit4567_r1);
         s4 = simd_mergeh_4(bit0123_r2, bit4567_r2);
         s5 = simd_mergel_4(bit0123_r2, bit4567_r2);
         s6 = simd_mergeh_4(bit0123_r3, bit4567_r3);
         s7 = simd_mergel_4(bit0123_r3, bit4567_r3);
       }
```

```
⟨ Transpose high UTF-16 bit streams to high byte stream 20 ⟩ ≡
#if (P2S_ALGORITHM ≡ P2S_IDEAL)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_ideal(u16hi0, u16hi1, u16hi2, u16hi3, u16hi4, u16hi5, u16hi6, u16hi7,
  U16h0, U16h1, U16h2, U16h3, U16h4, U16h5, U16h6, U16h7)
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
  p2s_ideal(u16hi0, u16hi1, u16hi2, u16hi3, u16hi4, u16hi5, u16hi6, u16hi7,
  U16h7, U16h6, U16h5, U16h4, U16h3, U16h2, U16h1, U16h0)
#endif
```

```
#endif
```

See also chunk 72.

This code is used in chunk 57.

21 ¶ ⟨ Transpose low UTF-16 bit streams to low byte stream 21 ⟩ ≡

```
#if (P2S_ALGORITHM ≡ P2S_IDEAL)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_ideal(u16lo0 , u16lo1 , u16lo2 , u16lo3 , u16lo4 , u16lo5 , u16lo6 , u16lo7 ,
  U16l0 , U16l1 , U16l2 , U16l3 , U16l4 , U16l5 , U16l6 , U16l7 )
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
  p2s_ideal(u16lo0 , u16lo1 , u16lo2 , u16lo3 , u16lo4 , u16lo5 , u16lo6 , u16lo7 ,
  U16l7 , U16l6 , U16l5 , U16l4 , U16l3 , U16l2 , U16l1 , U16l0 )
#endif
#endif
```

See also chunk 73.

This code is used in chunk 57.

¶ When a block of input consists of single and two-byte sequences only, the high 5 bits of the UTF-16 representation are always zero. Transposition of the remaining three bit streams (16*hi5* through *u16hi7* to high UTF-16 bytes is simplified in this case.

22

```
#define p2s_567_ideal(p5 , p6 , p7 , s0 , s1 , s2 , s3 , s4 , s5 , s6 , s7 )
          {
            BitBlock bit45_r0 , bit45_r1 , bit67_r0 , bit67_r1 ;
            BitBlock bit4567_r0 , bit4567_r1 , bit4567_r2 , bit4567_r3 ;
            bit45_r0 = simd_mergeh_1 (simd_const_8 (0), p5 );
            bit45_r1 = simd_mergel_1 (simd_const_8 (0), p5 );
            bit67_r0 = simd_mergeh_1 (p6 , p7 );
            bit67_r1 = simd_mergel_1 (p6 , p7 );
            bit4567_r0 = simd_mergeh_2 (bit45_r0 , bit67_r0 );
            bit4567_r1 = simd_mergel_2 (bit45_r0 , bit67_r0 );
            bit4567_r2 = simd_mergeh_2 (bit45_r1 , bit67_r1 );
            bit4567_r3 = simd_mergel_2 (bit45_r1 , bit67_r1 );
            s0 = simd_mergeh_4 (simd_const_8 (0), bit4567_r0 );
            s1 = simd_mergel_4 (simd_const_8 (0), bit4567_r0 );
            s2 = simd_mergeh_4 (simd_const_8 (0), bit4567_r1 );
            s3 = simd_mergel_4 (simd_const_8 (0), bit4567_r1 );
            s4 = simd_mergeh_4 (simd_const_8 (0), bit4567_r2 );
            s5 = simd_mergel_4 (simd_const_8 (0), bit4567_r2 );
            s6 = simd_mergeh_4 (simd_const_8 (0), bit4567_r3 );
            s7 = simd_mergel_4 (simd_const_8 (0), bit4567_r3 );
          }
```

⟨ Transpose three high UTF-16 bit streams to high byte stream 22 ⟩ ≡

```
#if (P2S_ALGORITHM ≡ P2S_IDEAL)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_567_ideal(u16hi5 , u16hi6 , u16hi7 ,
  U16h0 , U16h1 , U16h2 , U16h3 , U16h4 , U16h5 , U16h6 , U16h7 )
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
```

```
  p2s_567_ideal(u16hi5, u16hi6, u16hi7,
  U16h7, U16h6, U16h5, U16h4, U16h3, U16h2, U16h1, U16h0)
#endif
#endif
```

See also chunk 74.

This code is used in chunk 57.

6.3 Merging of High and Low Byte Streams

The high and low byte streams from parallel to serial conversion must be merged together to form doublebyte streams of UTF-16 data. The *u16_merge0* and *u16merge1* operations perform the merging in endian-dependent fashion.

23 ⟨ Merge high and low byte streams to doublebyte streams 23 ⟩ ≡

```
  U16s0 = u16_merge0(U16h0, U16l0);
  U16s1 = u16_merge1(U16h0, U16l0);
  U16s2 = u16_merge0(U16h1, U16l1);
  U16s3 = u16_merge1(U16h1, U16l1);
  U16s4 = u16_merge0(U16h2, U16l2);
  U16s5 = u16_merge1(U16h2, U16l2);
  U16s6 = u16_merge0(U16h3, U16l3);
  U16s7 = u16_merge1(U16h3, U16l3);
  U16s8 = u16_merge0(U16h4, U16l4);
  U16s9 = u16_merge1(U16h4, U16l4);
  U16s10 = u16_merge0(U16h5, U16l5);
  U16s11 = u16_merge1(U16h5, U16l5);
  U16s12 = u16_merge0(U16h6, U16l6);
  U16s13 = u16_merge1(U16h6, U16l6);
  U16s14 = u16_merge0(U16h7, U16l7);
  U16s15 = u16_merge1(U16h7, U16l7);
```

This code is used in chunk 57.

7 Block Processing Structure for UTF-8 to UTF-16 Conversion

The overall structure of the UTF-8 to UTF-16 conversion algorithm consists of a main loop for processing blocks of UTF-8 byte data. An ASCII short-cut optimization is first applied to process any significant run of UTF-8 data confined to the ASCII subset. When a region of input containing non-ASCII data is identified, it is then subject to block processing using parallel bit streams. After loading and transposing the block to parallel bit streams, UTF-8 validation constraints are checked and decoding to UTF-16 bit streams takes place. These bit streams must then be compressed from *u8-indexed* form (one to four positions per character based on UTF-8 sequence length) to *u16-indexed* form (one position per character for the basic multilingual plane, two positions for supplementary plane characters requiring surrogate pairs). The UTF-16 bit streams are then transposed to doublebyte streams and placed in the output buffer.

24 ⟨ Main block processing algorithm of *u8u16* 24 ⟩ ≡

```
  {
    unsigned char *U8data = (unsigned char *) *inbuf;
    unsigned char *U16out = (unsigned char *) *outbuf;
    int inbytes = *inbytesleft;

    ⟨ Prefetch block data 77 ⟩
    while (inbytes > 0) {
      ⟨ Apply ASCII short-cut optimization and continue 36 ⟩;
      ⟨ Load a block into serial bytepacks U8s0 through U8s7 31 ⟩
      ⟨ Transpose to parallel bit streams u8bit0 through u8bit7 19 ⟩
      ⟨ Apply validation, decoding and control logic on bit streams 28 ⟩
      ⟨ Compress bit streams and transpose to UTF-16 doublebyte streams 57 ⟩
      if (bitblock_has_bit(error_mask)) ⟨ Adjust to error position and signal the error 67 ⟩
      ⟨ Advance pointers and counters 26 ⟩
    }
    ⟨ Determine return values and exit 29 ⟩
  }
```

This code is used in chunk 5.

¶ Local variables *u8advance* and *u16advance* are calculated in each block processing step to represent the number of bytes by which the input and output buffers are expected to advance. When a full block is loaded, the value of *u8advance* is set to `BLOCKSIZE`, possibly reduced by one to three bytes for an incomplete UTF-8 sequence at the end of the block. Otherwise, *u8advance* is set to the remaining *inbytes* when a partial block is loaded. The value of *u16advance* depends on the distribution of different lengths of UTF-8 sequences within the input block.

25 ⟨ Local variable declarations 9 ⟩ +≡

```
  int u8advance, u16advance;
```

¶ When a block is successfully converted, the pointers and counters are updated.

26 ⟨ Advance pointers and counters 26 ⟩ ≡

```
  inbytes −= u8advance;
  U8data += u8advance;
  U16out += u16advance;
```

This code is used in chunks 24, 36, 67, and 81.

¶ Validation, decoding and control logic is divided into three cases corresponding to the three possible maximum byte lengths for UTF-8 blocks containing non-ASCII input. This allows simplified processing in the event that input is confined to two-byte or three-byte sequences. A maximum sequence length of two is frequently found in applications dealing with international texts from Europe, the Middle East, Africa and South America. A maximum sequence length of three accounts for texts confined to the basic multilingual plane of Unicode, including all the normally used characters of languages world-wide. The final case deals with those rare blocks that require the additional logic complexity to process four-byte UTF-8 sequences corresponding to the supplementary planes of Unicode.

28 ¶ ⟨ Apply validation, decoding and control logic on bit streams 28 ⟩ ≡

```
  ⟨ Compute classifications of UTF-8 bytes 38 ⟩
  ⟨ Compute scope classifications for common decoding 43 ⟩
  ⟨ Initiate validation for two-byte sequences 47 ⟩
  ⟨ Perform initial decoding of low eleven UTF-16 bit streams 52 ⟩
  ⟨ Identify deleted positions for basic multilingual plane giving delmask 55 ⟩
  if (⟨ Test whether the block is above the two-byte subplane 39 ⟩) {
    ⟨ Extend scope classifications for three-byte sequences 44 ⟩
    ⟨ Extend validation for errors in three-byte sequences 48 ⟩
    ⟨ Perform initial decoding of high five UTF-16 bit streams 53 ⟩
    if (⟨ Test whether the block is above the basic multilingual plane 40 ⟩) {
      ⟨ Extend scope classifications for four-byte sequences 45 ⟩
      ⟨ Extend validation for errors in four-byte sequences 49 ⟩
      ⟨ Extend decoding for four-byte sequences 54 ⟩
      ⟨ Identify deleted positions for general Unicode giving delmask 56 ⟩
    }
  }
  ⟨ Complete validation by checking for prefix-suffix mismatches 50 ⟩
```

This code is used in chunk 24.

¶ Upon completion of the main block processing loop, all input data up to the cutoff point has been converted and written to the output buffer. Update the external pointers and counters and return.

29 ⟨ Determine return values and exit 29 ⟩ ≡

```
  ∗outbytesleft = (int) U16out − (int) ∗outbuf;
  ∗inbuf = (char ∗) U8data;
  ∗inbytesleft = inbytes;
  ∗outbuf = (char ∗) U16out;
  if (inbytes ≡ 0) return (size_t) 0;
  else return (size_t) −1;
```

This code is used in chunks 24 and 36.

8 Loading Block Data into SIMD Registers

31 ¶⟨ Load a block into serial bytepacks *U8s0* through *U8s7* 31 ⟩ ≡

```
  if (inbytes < BLOCKSIZE) {
    ⟨ Stop prefetch 78 ⟩
    input_select_mask = sisd_sbl(simd_const_8(−1), sisd_from_int(BLOCKSIZE − inbytes));
    ⟨ Load a block fragment 35 ⟩
  }
  else {
    ⟨ Prefetch block data 77 ⟩
    input_select_mask = simd_const_8(−1);
    ⟨ Load a full block of UTF-8 byte data 32 ⟩
  }
```

This code is used in chunk 24.

¶ Generic loading of a full block of UTF-8 byte data assumes that nonaligned loads are available.

32 ⟨ Load a full block of UTF-8 byte data 32 ⟩ ≡

```
#ifdef INBUF_READ_NONALIGNED
  {
    BytePack ∗U8pack = (BytePack ∗) U8data;
    U8s0 = sisd_load_unaligned(&U8pack[0]);
    U8s1 = sisd_load_unaligned(&U8pack[1]);
    U8s2 = sisd_load_unaligned(&U8pack[2]);
    U8s3 = sisd_load_unaligned(&U8pack[3]);
    U8s4 = sisd_load_unaligned(&U8pack[4]);
    U8s5 = sisd_load_unaligned(&U8pack[5]);
    U8s6 = sisd_load_unaligned(&U8pack[6]);
    U8s7 = sisd_load_unaligned(&U8pack[7]);
```

```
      u8advance = BLOCKSIZE;
      ⟨ Apply block shortening 33 ⟩
   }
#endif
```

See also chunks 79 and 107.

This code is used in chunk 31.

¶ A block of UTF-8 data may end in an incomplete UTF-8 sequence with any *u8prefix* at the least significant position, with a *u8prefix3or4* at the second last position, or with a *u8prefix4* at the third last position. If so, *u8advance* is reduced by one, two or three positions, as appropriate.

The logic here is simplified for correct UTF-8 input (assuming only one of these three conditions may be true); the *u8advance* value calculated must not be used until validation is complete.

33
```
#define is_prefix_byte(byte)  (byte ≥ #C0)
#define is_prefix3or4_byte(byte)  (byte ≥ #E0)
#define is_prefix4_byte(byte)  (byte ≥ #F0)
⟨ Apply block shortening 33 ⟩ ≡
  u8advance −= is_prefix_byte(U8data[u8advance − 1])
        + 2 ∗ is_prefix3or4_byte(U8data[u8advance − 2])
        + 3 ∗ is_prefix4_byte(U8data[u8advance − 3]);
```

This code is used in chunks 32, 79, 107, and 108.

8.1 Loading the Final Block Fragment

¶ When loading a block fragment at the end of the input buffer, care must be taken to avoid any possibility of a page fault. For a short fragment, a page fault could occur either by reading across an alignment boundary prior to the first byte or after the last byte.

35
```
#define pack_base_addr(addr)  ((BytePack ∗)(((int)(addr)) & (−PACKSIZE)))
⟨ Load a block fragment 35 ⟩ ≡
#ifdef INBUF_READ_NONALIGNED
  {
    BytePack ∗U8pack = (BytePack ∗) U8data;
    int full_packs = inbytes/PACKSIZE;
    int U8s7_size = inbytes % PACKSIZE;
    int U8data_offset = ((int) U8data) % PACKSIZE;
    int pack = 0;
    switch (full_packs) {
    case 7: U8s0 = sisd_load_unaligned(&U8pack[pack ++]);
    case 6: U8s1 = sisd_load_unaligned(&U8pack[pack ++]);
    case 5: U8s2 = sisd_load_unaligned(&U8pack[pack ++]);
    case 4: U8s3 = sisd_load_unaligned(&U8pack[pack ++]);
    case 3: U8s4 = sisd_load_unaligned(&U8pack[pack ++]);
    case 2: U8s5 = sisd_load_unaligned(&U8pack[pack ++]);
    case 1: U8s6 = sisd_load_unaligned(&U8pack[pack ++]);
    case 0:
      if (U8s7_size ≡ 0) U8s7 = simd_const_8(0);
      else if (U8data_offset + U8s7_size > PACKSIZE)     /∗ unaligned load safe and required. ∗/
        U8s7 = sisd_load_unaligned(&U8pack[pack]);
      else {     /∗ aligned load required for safety ∗/
        U8s7 = sisd_load_aligned(pack_base_addr(&U8pack[pack]));
        U8s7 = sisd_sbl(U8s7, sisd_from_int(8 ∗ U8data_offset));
      }
    }
    input_select_mask = sisd_sbl(simd_const_8(−1), sisd_from_int(BLOCKSIZE − inbytes));
    input_select_mask = sisd_sfl(input_select_mask, sisd_from_int(PACKSIZE ∗ (7 − full_packs)));
    u8advance = inbytes;
  }
#endif
```

See also chunks 80 and 108.

This code is used in chunk 31.

9 ASCII Optimization

Runs of ASCII characters can be converted to UTF-16 using an optimized process that avoids conversion to and from parallel bit streams. Given a bytepack of ASCII characters, two consecutive bytepacks of corresponding UTF-16 output may be produced by merging a bytepack of all zeroes with the ASCII data. Further optimizations are applied for runs consisting of multiple bytepacks: converting to aligned output and using an unrolled loop to handle 4 bytepacks per iteration.

```
36 #define align_offset(addr)  (((int) addr) & (PACKSIZE − 1))
⟨ Apply ASCII short-cut optimization and continue 36 ⟩ ≡
#ifdef GENERIC_ASCII_OPTIMIZATION
  BitBlock vec_0 = simd_const_8(0);
  if (inbytes > PACKSIZE) {
    U8s0 = sisd_load_unaligned((BytePack ∗) U8data);
    if (¬simd_any_sign_bit_8(U8s0)) {
      int fill_to_align = PACKSIZE − align_offset(U16out);
      U16s0 = u16_merge0(vec_0, U8s0);
      sisd_store_unaligned(U16s0, (BytePack ∗) U16out);
      u8advance = fill_to_align/2;
      u16advance = fill_to_align;
      ⟨ Advance pointers and counters 26 ⟩
      while (inbytes > 4 ∗ PACKSIZE) {
        BytePack ∗U8pack = (BytePack ∗) U8data;
        BytePack ∗U16pack = (BytePack ∗) U16out;
        U8s0 = sisd_load_unaligned(U8pack);
        U8s1 = sisd_load_unaligned(&U8pack[1]);
        U8s2 = sisd_load_unaligned(&U8pack[2]);
        U8s3 = sisd_load_unaligned(&U8pack[3]);
        if (simd_any_sign_bit_8(simd_or(simd_or(U8s0, U8s1), simd_or(U8s2, U8s3)))) break;
        sisd_store_aligned(u16_merge0(vec_0, U8s0), U16pack);
        sisd_store_aligned(u16_merge1(vec_0, U8s0), &U16pack[1]);
        sisd_store_aligned(u16_merge0(vec_0, U8s1), &U16pack[2]);
        sisd_store_aligned(u16_merge1(vec_0, U8s1), &U16pack[3]);
        sisd_store_aligned(u16_merge0(vec_0, U8s2), &U16pack[4]);
        sisd_store_aligned(u16_merge1(vec_0, U8s2), &U16pack[5]);
        sisd_store_aligned(u16_merge0(vec_0, U8s3), &U16pack[6]);
        sisd_store_aligned(u16_merge1(vec_0, U8s3), &U16pack[7]);
        u8advance = 4 ∗ PACKSIZE;
        u16advance = 8 ∗ PACKSIZE;
        ⟨ Advance pointers and counters 26 ⟩
      }
      while (inbytes > PACKSIZE) {
        BytePack ∗U16pack = (BytePack ∗) U16out;
        U8s0 = sisd_load_unaligned((BytePack ∗) U8data);
        if (simd_any_sign_bit_8(U8s0)) break;
        sisd_store_aligned(u16_merge0(vec_0, U8s0), U16pack);
        sisd_store_aligned(u16_merge1(vec_0, U8s0), &U16pack[1]);
        u8advance = PACKSIZE;
        u16advance = 2 ∗ PACKSIZE;
        ⟨ Advance pointers and counters 26 ⟩
      }
    }
  }
  if (inbytes ≤ PACKSIZE) {
    int U8data_offset = ((int) U8data) & (PACKSIZE − 1);
    if (U8data_offset + inbytes ≤ PACKSIZE) {
        /∗ Avoid a nonaligned load that could create a page fault. ∗/
      U8s0 = sisd_sbl(sisd_load_aligned((BytePack ∗) pack_base_addr((int) U8data)),
          sisd_from_int(8 ∗ U8data_offset));
    }
    else U8s0 = sisd_load_unaligned((BytePack ∗) U8data);
    U8s0 = simd_and(U8s0, sisd_sbl(simd_const_8(−1), sisd_from_int(8 ∗ (PACKSIZE − inbytes))));
    if (¬simd_any_sign_bit_8(U8s0)) {
      sisd_store_unaligned(u16_merge0(vec_0, U8s0), (BytePack ∗) U16out);
      if (inbytes > PACKSIZE/2)
        sisd_store_unaligned(u16_merge1(vec_0, U8s0), (BytePack ∗) &U16out[PACKSIZE]);
      u8advance = inbytes;
      u16advance = 2 ∗ inbytes;
      ⟨ Advance pointers and counters 26 ⟩
      ⟨ Determine return values and exit 29 ⟩
    }
  }
#endif
```

See also chunk 81.

This code is used in chunk 24.

10 UTF-8 Byte Classification

A set of bit streams are used to classify UTF-8 bytes based on their role in forming single and multibyte sequences. The *u8prefix* and *u8suffix* streams identify bytes that represent, respectively, prefix or suffix bytes of multibyte sequences, while the *u8unibyte* stream identifies those bytes that may be considered single-byte sequences, each representing a character by itself.

Prefix bytes are further classified by whether they code for 2, 3 or 4 byte sequences.

37 ⟨ Local variable declarations 9 ⟩ +≡

```
  BitBlock u8unibyte, u8prefix, u8suffix, u8prefix2, u8prefix3or4, u8prefix3, u8prefix4;
```

¶ These bit streams are computed by straightforward logical combinations reflecting the definition of UTF-8. However, the streams are defined only for valid input positions in accord with *input_select_mask*.

38 ⟨Compute classifications of UTF-8 bytes 38⟩ ≡

```
{
  BitBlock bit0_selected = simd_and(input_select_mask, u8bit0);
  u8unibyte = simd_andc(input_select_mask, u8bit0);
  u8prefix = simd_and(bit0_selected, u8bit1);
  u8suffix = simd_andc(bit0_selected, u8bit1);
  u8prefix3or4 = simd_and(u8prefix, u8bit2);
  u8prefix2 = simd_andc(u8prefix, u8bit2);
  u8prefix3 = simd_andc(u8prefix3or4, u8bit3);
  u8prefix4 = simd_and(u8prefix3or4, u8bit3);
}
```

This code is used in chunk 28.

¶ When a block of UTF-8 input is confined to single-byte or two-byte sequences only, processing may be considerably simplified. A convenient bit test determines whether the logic for three- or four-byte UTF-8 sequences is required.

39 ⟨Test whether the block is above the two-byte subplane 39⟩ ≡

```
bitblock_has_bit(u8prefix3or4)
```

This code is used in chunks 28 and 57.

¶ A similar bit test determines whether the logic sufficient for the basic multilingual plane of Unicode (including up to three-byte sequences) is sufficient, or whether the extended logic for the four-byte sequences is required.

40 ⟨Test whether the block is above the basic multilingual plane 40⟩ ≡

```
bitblock_has_bit(u8prefix4)
```

This code is used in chunk 28.

10.1 Scope streams

¶ Scope streams represent expectations established by prefix bytes. For example, *u8scope22* represents the positions at which the second byte of a two-byte sequence is expected based on the occurrence of two-byte prefixes in the immediately preceding positions. Other scope streams represent combined classifications.

42 ⟨Local variable declarations 9⟩ +≡

```
BitBlock u8scope22, u8scope32, u8scope33, u8scope42, u8scope43, u8scope44;
BitBlock u8lastsuffix, u8lastbyte, u8surrogate;
```

¶ For the decoding operations common to all cases, the *u8lastsuffix* and *u8lastbyte* classifications are needed.

43 ⟨Compute scope classifications for common decoding 43⟩ ≡

```
u8scope22 = bitblock_sfli(u8prefix2, 1);
u8scope33 = bitblock_sfli(u8prefix3, 2);
u8scope44 = bitblock_sfli(u8prefix4, 3);
u8lastsuffix = simd_or(simd_or(u8scope22, u8scope33), u8scope44);
u8lastbyte = simd_or(u8unibyte, u8lastsuffix);
```

This code is used in chunk 28.

¶ When a block is known to include three-byte sequences, the *u8scope32* stream is relevant.

44 ⟨Extend scope classifications for three-byte sequences 44⟩ ≡

```
u8scope32 = bitblock_sfli(u8prefix3, 1);
```

This code is used in chunk 28.

¶ Additional classifications become relevant when a block is known to include four-byte sequences.

45 ⟨Extend scope classifications for four-byte sequences 45⟩ ≡

```
u8scope42 = bitblock_sfli(u8prefix4, 1);
u8scope43 = bitblock_sfli(u8prefix4, 2);
u8surrogate = simd_or(u8scope42, u8scope44);
```

This code is used in chunk 28.

11 UTF-8 Validation

Any UTF-8 errors in a block of input data are identified through the process of UTF-8 validation. The result of the process is an *error_mask* identifying those positions at which an error is positively identified. Blocks are assumed to start with complete UTF-8 sequences; any suffix found at the beginning of a block is marked as an error. An incomplete sequence at the end of the block is not marked as an error if it is possible to produce a legal sequence by adding one or more bytes.

UTF-8 validation involves checking that UTF-8 suffixes match with scope expectations, that invalid prefix codes `#C0`, `#C1`, and `#F5` through `#FF` do not occur, and that constraints on the first suffix byte following certain special prefixes are obeyed, namely `#E0:#A0`–`#BF`, `#ED:#80`–`#9F`, `#F0:#90`–`#BF`, and `#F4:#80`–`#8F`. The variable *suffix_required_scope* is used to identify positions at which a suffix byte is expected.

46 ⟨Local variable declarations 9⟩ +≡

```
BitBlock suffix_required_scope;
```

¶ The logic is staged to initialize *error_mask* and *suffix_required_scope* for errors in two-byte sequences followed by additional logic stages for three-byte sequences and four-byte sequences.

For two-byte sequences, *error_mask* is initialized by a check for invalid prefixes `#C0` or `#C1`, and *suffix_required_scope* is initialized for the suffix position of two-byte sequences.

47 ⟨Initiate validation for two-byte sequences 47⟩ ≡

```
error_mask = simd_andc(u8prefix2, simd_or(simd_or(u8bit3, u8bit4), simd_or(u8bit5, u8bit6)));
suffix_required_scope = u8scope22;
```

This code is used in chunk 28.

¶ Error checking for three-byte sequences involves local variable *prefix_E0ED* to identify occurrences of r `#E0` or `#ED` prefix bytes, and *E0ED_constraint* to indicate positions at which the required suffix constraint holds.

48 ⟨Extend validation for errors in three-byte sequences 48⟩ ≡

```
{
  BitBlock prefix_E0ED, E0ED_constraint;
  prefix_E0ED = simd_andc(u8prefix3, simd_or(simd_or(u8bit6, simd_xor(u8bit4, u8bit7)),
      simd_xor(u8bit4, u8bit5)));
  E0ED_constraint = simd_xor(bitblock_sfli(u8bit5, 1), u8bit2);
  error_mask = simd_or(error_mask, simd_andc(bitblock_sfli(prefix_E0ED, 1), E0ED_constraint));
  suffix_required_scope = simd_or(u8lastsuffix, u8scope32);
}
```

This code is used in chunk 28.

¶ In the case of validation for general Unicode includling four-byte sequences, additional local variables *prefix_F5FF* (for any prefix byte #`F5` through #`FF`), *prefix_F0F4* and *F0F4_constraint* are defined.

49 ⟨ Extend validation for errors in four-byte sequences 49 ⟩ ≡

```
{
  BitBlock prefix_F5FF, prefix_F0F4, F0F4_constraint;
  prefix_F5FF = simd_and(u8prefix4, simd_or(u8bit4, simd_and(u8bit5, simd_or(u8bit6, u8bit7))));
  error_mask = simd_or(error_mask, prefix_F5FF);
  prefix_F0F4 = simd_andc(u8prefix4, simd_or(u8bit4, simd_or(u8bit6, u8bit7)));
  F0F4_constraint = simd_xor(bitblock_sfli(u8bit5, 1), simd_or(u8bit2, u8bit3));
  error_mask = simd_or(error_mask, simd_andc(bitblock_sfli(prefix_F0F4, 1), F0F4_constraint));
  suffix_required_scope = simd_or(suffix_required_scope, simd_or(u8surrogate, u8scope43));
}
```

This code is used in chunk 28.

¶ Completion of validation requires that any mismatch between a scope expectation and the occurrence of a suffix byte be identified.

50 ⟨ Complete validation by checking for prefix-suffix mismatches 50 ⟩ ≡

```
error_mask = simd_or(error_mask, simd_xor(suffix_required_scope, u8suffix));
```

This code is used in chunk 28.

12 UTF-16 Bit Streams

Given validated UTF-8 bit streams, conversion to UTF-16 proceeds by first determining a parallel set of 16 bit streams that comprise a *u8-indexed* representation of UTF-16. This representation defines the correct UTF-16 bit representation at the following UTF-8 positions: at the single byte of a single-byte sequence (*u8unibyte*), at the second byte of a two-byte sequence (*u8scope22*), at the third byte of a three byte sequence (*u8scope33*), and at the second and fourth bytes of a four-byte sequence (*u8scope42* and *u8scope42*). In the case of four byte sequences, two UTF-16 units are produced, comprising the UTF-16 surrogate pair for a codepoint beyond the basic multilingual plane.

The UTF-16 bit stream values at other positions (*u8prefix2*, *u8prefix3*, *u8prefix4*, *u8scope32*, *u8scope43*) are not significant; no UTF-16 output is to be generated from these positions. Prior to generation of output, data bits at these positions are to be deleted using the deletion operations of the subsequent section. These deletions produce the UTF-16 bit streams in *u16-indexed* form.

¶ Decoding is initiated by applying the common logic for the low eleven bit streams identified by the the *u8lastsuffix* and *u8lastbyte* conditions.

52 ⟨ Perform initial decoding of low eleven UTF-16 bit streams 52 ⟩ ≡

```
u16hi5 = simd_and(u8lastsuffix, bitblock_sfli(u8bit3, 1));
u16hi6 = simd_and(u8lastsuffix, bitblock_sfli(u8bit4, 1));
u16hi7 = simd_and(u8lastsuffix, bitblock_sfli(u8bit5, 1));
u16lo0 = simd_and(u8lastsuffix, bitblock_sfli(u8bit6, 1));
u16lo1 = simd_or(simd_and(u8unibyte, u8bit1), simd_and(u8lastsuffix, bitblock_sfli(u8bit7, 1)));
u16lo2 = simd_and(u8lastbyte, u8bit2);
u16lo3 = simd_and(u8lastbyte, u8bit3);
u16lo4 = simd_and(u8lastbyte, u8bit4);
u16lo5 = simd_and(u8lastbyte, u8bit5);
u16lo6 = simd_and(u8lastbyte, u8bit6);
u16lo7 = simd_and(u8lastbyte, u8bit7);
```

This code is used in chunk 28.

¶ For blocks containing three-byte sequences in the basic multilingual plane, the high five UTF-16 bit streams become significant at *u8scope33* positions.

53 ⟨Perform initial decoding of high five UTF-16 bit streams 53⟩ ≡

```
  u16hi0 = simd_and(u8scope33, bitblock_sfli(u8bit4, 2));
  u16hi1 = simd_and(u8scope33, bitblock_sfli(u8bit5, 2));
  u16hi2 = simd_and(u8scope33, bitblock_sfli(u8bit6, 2));
  u16hi3 = simd_and(u8scope33, bitblock_sfli(u8bit7, 2));
  u16hi4 = simd_and(u8scope33, bitblock_sfli(u8bit2, 1));
```

This code is used in chunk 28.

¶ Decoding for 4-byte UTF-8 sequences involves logic for for UTF-16 surrogate pairs at the *u8scope42* and *u8scope44* positions. However, the values for the low ten bit streams at *u8scope44* positions have already been set according to the common pattern for *u8lastsuffix* and *u8lastbyte*, so it is only necessary to extend the definitions of these ten bit streams with the logic for the first UTF-16 code unit of the surrogate pair at the *u8scope42* position. The high six UTF-16 bits are set to a fixed bit pattern of 110110 or 110111 for the respective surrogate pair positions.

54 ⟨Extend decoding for four-byte sequences 54⟩ ≡

```
  {
    BitBlock borrow1, borrow2;
    u16hi0 = simd_or(u16hi0, u8surrogate);
    u16hi1 = simd_or(u16hi1, u8surrogate);
    u16hi3 = simd_or(u16hi3, u8surrogate);
    u16hi4 = simd_or(u16hi4, u8surrogate);
    u16hi5 = simd_or(u16hi5, u8scope44);
    u16lo1 = simd_or(u16lo1, simd_and(u8scope42, simd_not(u8bit3)));
      /* under u8scope42: u16lo0 = u8bit2 - borrow, where borrow = u16lo1 */
    u16lo0 = simd_or(u16lo0, simd_and(u8scope42, simd_xor(u8bit2, u16lo1)));
    borrow1 = simd_andc(u16lo1, u8bit2);      /* borrow for u16hi7. */
    u16hi7 = simd_or(u16hi7, simd_and(u8scope42, simd_xor(bitblock_sfli(u8bit7, 1), borrow1)));
    borrow2 = simd_andc(borrow1, bitblock_sfli(u8bit7, 1));      /* borrow for u16hi6. */
    u16hi6 = simd_or(u16hi6, simd_and(u8scope42, simd_xor(bitblock_sfli(u8bit6, 1), borrow2)));
    u16lo2 = simd_or(u16lo2, simd_and(u8scope42, u8bit4));
    u16lo3 = simd_or(u16lo3, simd_and(u8scope42, u8bit5));
    u16lo4 = simd_or(u16lo4, simd_and(u8scope42, u8bit6));
    u16lo5 = simd_or(u16lo5, simd_and(u8scope42, u8bit7));
    u16lo6 = simd_or(u16lo6, simd_and(u8scope42, bitblock_sbli(u8bit2, 1)));
    u16lo7 = simd_or(u16lo7, simd_and(u8scope42, bitblock_sbli(u8bit3, 1)));
  }
```

This code is used in chunk 28.

13 Compression to U16-Indexed Form by Deletion

As identified in the previous section, the UTF-16 bit streams are initially defined in u8-indexed form, that is, with sets of bits in one-to-one correspondence with UTF-8 bytes. However, only one set of UTF-16 bits is required for encoding two or three-byte UTF-8 sequences and only two sets are required for surrogate pairs corresponding to four-byte UTF-8 positions. The *u8lastbyte* (*unibyte*, *u8scope22*, *u8scope33*, and *u8scope44*) and *u8scope42* streams mark the positions at which the correct UTF-16 bits are computed. The bit sets at other positions must be deleted to compress the streams to u16-indexed form. In addition, any positions outside the *input_select_mask* must also be deleted.

In the case of input confined to the basic multilingual plane, there are no *u8scope42* positions to consider in forming the deletion mask.

55 ⟨ Identify deleted positions for basic multilingual plane giving *delmask* 55 ⟩ ≡

```
  delmask = simd_not(simd_and(input_select_mask, u8lastbyte));
```

This code is used in chunk 28.

¶ For general Unicode, however, *u8scope42* positions must be not be deleted, provided that the the full 4-byte sequence (including the corresponding *u8scope44* position) is within the selected input area.

56 ⟨ Identify deleted positions for general Unicode giving *delmask* 56 ⟩ ≡

```
  {
    BitBlock scope42_selected = bitblock_sbli(simd_and(u8scope44, input_select_mask), 2);
    delmask = simd_not(simd_and(input_select_mask, simd_or(u8lastbyte, scope42_selected)));
  }
```

This code is used in chunk 28.

¶ Several algorithms to delete bits at positions marked by *delmask* are possible. Preprocessor configuration options allow selection from available alternatives for particular architectures. In each case, however, the *u8u16* program is designed to perform the initial deletion operations within fields of size `PACKSIZE`/2. Within each such field, then, non-deleted bits become compressed together at the front end of the field, followed by zeroes for the deleted bits at the back end. Upon transposition to doublebyte streams of UTF-16 code units, each `PACKSIZE`/2-bit field becomes a single bytepack of UTF-16 data. After writing each such bytepack to output, the output pointer is advanced only by the number of nondeleted bytes. In this way, the final compression to continuous u16-indexed code unit streams is achieved as part of the output process.

In the context of this general deletion strategy, algorithm variations achieve deletion within `PACKSIZE`/2 fields by different methods. In each case, the deletion process is controlled by deletion information computed from *delmask*. Based on this information, deletion operations may be applied to bit streams and/or as byte stream transformations.

The following code describes the general structure, also incorporating an optimization for the two-byte subplane (UTF-8 inputs confined to one or two-byte sequences). In this case, the high five bits of the UTF-16 representation are always zero, so bit deletion operations for these streams can be eliminated.

57 ⟨ Compress bit streams and transpose to UTF-16 doublebyte streams 57 ⟩ ≡

```
  ⟨ Determine deletion information from delmask 60 ⟩
  ⟨ Apply bit deletions to low eleven UTF-16 bit streams 63 ⟩
  if (¬⟨ Test whether the block is above the two-byte subplane 39 ⟩) {
    ⟨ Transpose three high UTF-16 bit streams to high byte stream 22 ⟩
  }
  else {
    ⟨ Apply bit deletions to high five UTF-16 bit streams 62 ⟩
    ⟨ Transpose high UTF-16 bit streams to high byte stream 20 ⟩
  }
  ⟨ Transpose low UTF-16 bit streams to low byte stream 21 ⟩
  ⟨ Apply byte stream transformations 64 ⟩
  ⟨ Merge high and low byte streams to doublebyte streams 23 ⟩
  ⟨ Write compressed UTF-16 data 65 ⟩
```

This code is used in chunk 24.

13.1 Deletion by Central Result Induction

¶ The default implementation of deletion within `PACKSIZE`/2 fields is designed for a `BLOCKSIZE` of 128 and hence a `PACKSIZE` of 16. Deletion within 8-bit fields requires three operations per bit stream: conversion of 2-bit central deletion results to 8-bit central deletion results in two steps of deletion by rotation (central result induction), followed by conversion to 8-bit front-justified results by a back-shift operation.

59 ⟨ Local variable declarations 9 ⟩ +≡

```
  unsigned char u16_bytes_per_reg[16]__attribute__((aligned(16)));
#if ((DOUBLEBYTE_DELETION ≡ FROM_LEFT8) ∨ (BIT_DELETION ≡ ROTATION_TO_LEFT8))
  BitBlock delcounts_2, delcounts_4, delcounts_8, u16_advance_8, u16_bytes_8;
#endif
#if (BIT_DELETION ≡ ROTATION_TO_LEFT8)
  BitBlock rotl_2, rotl_4, sll_8;
#endif
```

60 ¶ ⟨ Determine deletion information from *delmask* 60 ⟩ ≡

```
#if ((DOUBLEBYTE_DELETION ≡ FROM_LEFT8) ∨ (BIT_DELETION ≡ ROTATION_TO_LEFT8))
  delcounts_2 = simd_add_2_lh(delmask, delmask);
  delcounts_4 = simd_add_4_lh(delcounts_2, delcounts_2);
  delcounts_8 = simd_add_8_lh(delcounts_4, delcounts_4);
  sisd_store_aligned(simd_slli_8(simd_sub_8(simd_const_8(8), delcounts_8), 1), (BytePack *)
        &u16_bytes_per_reg[0]);
#endif
#if (BIT_DELETION ≡ ROTATION_TO_LEFT8)
  rotl_2 = simd_if(himask_4, delmask, vec_srli(delmask, 1));
  rotl_4 = simd_if(himask_8, simd_sub_2(vec_0, delcounts_2), simd_srli(delcounts_2, 2));
  sll_8 = simd_srli(delcounts_4, 4);
#endif
```

See also chunks 82, 91, 96, and 99.

This code is used in chunk 57.

13.2 Apply Deletions to Bit Streams

¶ Perform the two rotations and one shift operation to yield left-justified data within 8-bit fields.

62 ⟨ Apply bit deletions to high five UTF-16 bit streams 62 ⟩ ≡

```
#if (BIT_DELETION ≡ ROTATION_TO_LEFT8)
  u16hi0 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi0, rotl_2), rotl_4), sll_8);
  u16hi1 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi1, rotl_2), rotl_4), sll_8);
  u16hi2 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi2, rotl_2), rotl_4), sll_8);
  u16hi3 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi3, rotl_2), rotl_4), sll_8);
  u16hi4 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi4, rotl_2), rotl_4), sll_8);
#endif
```

See also chunks 84, 93, and 101.

This code is used in chunk 57.

63 ¶ ⟨ Apply bit deletions to low eleven UTF-16 bit streams 63 ⟩ ≡

```
#if (BIT_DELETION ≡ ROTATION_TO_LEFT8)
  u16hi5 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi5, rotl_2), rotl_4), sll_8);
  u16hi6 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi6, rotl_2), rotl_4), sll_8);
  u16hi7 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16hi7, rotl_2), rotl_4), sll_8);
  u16lo0 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo0, rotl_2), rotl_4), sll_8);
  u16lo1 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo1, rotl_2), rotl_4), sll_8);
  u16lo2 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo2, rotl_2), rotl_4), sll_8);
  u16lo3 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo3, rotl_2), rotl_4), sll_8);
  u16lo4 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo4, rotl_2), rotl_4), sll_8);
  u16lo5 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo5, rotl_2), rotl_4), sll_8);
  u16lo6 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo6, rotl_2), rotl_4), sll_8);
  u16lo7 = simd_sll_8(simd_rotl_4(simd_rotl_2(u16lo7, rotl_2), rotl_4), sll_8);
#endif
```

See also chunks 85, 94, and 102.

This code is used in chunk 57.

64 ¶ ⟨Apply byte stream transformations 64⟩ ≡

```
    /* No byte stream transformations are required in the default algorithm. */
```

See also chunk 86.

This code is used in chunk 57.

65 ¶

```
#define unaligned_output_step(reg, bytes)
        sisd_store_unaligned(reg, (BytePack *) &U16out[u16advance]);
        u16advance += bytes;
```

⟨Write compressed UTF-16 data 65⟩ ≡

```
#ifdef OUTBUF_WRITE_NONALIGNED
  u16advance = 0;
  unaligned_output_step(U16s0, u16_bytes_per_reg[0])
  unaligned_output_step(U16s1, u16_bytes_per_reg[1])
  unaligned_output_step(U16s2, u16_bytes_per_reg[2])
  unaligned_output_step(U16s3, u16_bytes_per_reg[3])
  unaligned_output_step(U16s4, u16_bytes_per_reg[4])
  unaligned_output_step(U16s5, u16_bytes_per_reg[5])
  unaligned_output_step(U16s6, u16_bytes_per_reg[6])
  unaligned_output_step(U16s7, u16_bytes_per_reg[7])
  unaligned_output_step(U16s8, u16_bytes_per_reg[8])
  unaligned_output_step(U16s9, u16_bytes_per_reg[9])
  unaligned_output_step(U16s10, u16_bytes_per_reg[10])
  unaligned_output_step(U16s11, u16_bytes_per_reg[11])
  unaligned_output_step(U16s12, u16_bytes_per_reg[12])
  unaligned_output_step(U16s13, u16_bytes_per_reg[13])
  unaligned_output_step(U16s14, u16_bytes_per_reg[14])
  unaligned_output_step(U16s15, u16_bytes_per_reg[15])
#endif
```

See also chunk 88.

This code is used in chunk 57.

14 Error Identification and Reporting

¶ When a validation error is identified, the end of the last complete UTF-8 sequence prior to the error must be determined as the basis for calculating *u8advance* and *u16advance*. The pointers and counters may then be updated and the error return made.

```
67 ⟨Adjust to error position and signal the error 67⟩ ≡
  {
    if (¬bitblock_has_bit(simd_and(error_mask, input_select_mask))) {
        /* Error is not in block; must be at end of input. */
      errno = EINVAL;
      do {
        u8advance−−;
      } while (¬is_prefix_byte(U8data[u8advance]));
    }
    else {
      BitBlock cutoff_mask;
      int errpos;
      errno = EILSEQ;
      ⟨Adjust error_mask to mark UTF-8 prefix positions 68⟩;
      errpos = count_leading_zeroes(error_mask);
      cutoff_mask = sisd_sfl(simd_const_8(−1), sisd_from_int(errpos));
      input_select_mask = simd_andc(input_select_mask, cutoff_mask);
      u8advance = bitblock_bit_count(input_select_mask);
#ifdef PARTITIONED_BLOCK_MODE
      if (errpos ≥ BLOCKSIZE/2) u8advance −= half_block_adjust;
#endif
      u16advance = 2 ∗ (bitblock_bit_count(simd_and(u8lastbyte, input_select_mask)) +
          bitblock_bit_count(simd_and(u8scope44, input_select_mask)));
    }
    ⟨Advance pointers and counters 26⟩
    ∗outbytesleft = (int) U16out − (int) ∗outbuf;
    ∗inbytesleft = inbytes;
    ∗inbuf = (char ∗) U8data;
    ∗outbuf = (char ∗) U16out;
    return (size_t) −1;
  }
```

This code is used in chunk 24.

¶ Validation may locate errors at any position within a UTF-8 sequence. In order to identify the beginning of the UTF-8 sequences containing errors, any error at a suffix position must be propagated to the corresponding prefix position.

```
68 ⟨Adjust error_mask to mark UTF-8 prefix positions 68⟩ ≡
  error_mask = simd_or(simd_or(error_mask,
      simd_and(u8prefix, bitblock_sbli(error_mask, 1))),
      simd_or(simd_and(u8prefix3or4, bitblock_sbli(error_mask, 2)),
      simd_and(u8prefix4, bitblock_sbli(error_mask, 3))));
```

This code is used in chunk 67.

15 Buffered Version

¶ The *buffered_u8u16* routine uses an internal buffer for assembling UTF-16 code units prior to copying them to the specified output buffer.

```
70 #define is_suffix_byte(byte) (byte ≥ #80 ∧ byte ≤ #BF)
```

```
size_t buffered_u8u16(char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft)
{
  if (inbuf ∧ *inbuf ∧ outbuf ∧ *outbuf)      /* are all non-NULL */ {
    unsigned char *inbuf_start = (unsigned char *) *inbuf;
    size_t max_inbytes = min(3 * (*outbytesleft)/2, *inbytesleft);
    size_t inbytes_start = *inbytesleft;
    size_t internal_space = 2 * (*inbytesleft);
    size_t internal_space_left = internal_space;
    char *internal_buf_start = malloc(PACKSIZE + internal_space);
    char *internal_buf = internal_buf_start;
    size_t return_code = u8u16(inbuf, &max_inbytes, &internal_buf, &internal_space_left);
    size_t u16advance = internal_space − internal_space_left;
    size_t u8advance = (int)(*inbuf) − (int) inbuf_start;
    if (u16advance > *outbytesleft) {
      errno = E2BIG;
      return_code = (size_t) −1;
      do {
        do {
          u8advance−−;
        } while (is_suffix_byte(inbuf_start[u8advance]));
        if (is_prefix4_byte(inbuf_start[u8advance])) u16advance −= 4;
        else u16advance −= 2;
      } while (u16advance > *outbytesleft);
    }
    memcpy(*outbuf, internal_buf_start, u16advance);
    free(internal_buf_start);
    *inbuf = (char *) inbuf_start + u8advance;
    *inbytesleft −= u8advance;
    *outbuf += u16advance;
    *outbytesleft −= u16advance;
    return return_code;
  }
  else if (inbuf ≡ Λ ∨ *inbuf ≡ Λ ∨ *inbytesleft ≡ 0) return (size_t) 0;
  else { errno = E2BIG; return (size_t) −1; }
}
```

16 Alternative Transposition Algorithms Using Byte Packing/Merging

In the event that byte-level pack and merge operations represent the finest granularity level available on a particular SIMD target architecture, transposition using the generic algorithms uses simulated implementations of pack and merge operations at the bit, bit pair and nybble levels. Better performance can be achieved by restructured algorithms that directly use byte-level pack and merge.

In the case of serial to parallel to serial transposition, the restructured algorithm uses three stages of packing data from consecutive bytes. In the first stage, individual bits from consecutive bytes are paired up to produce two parallel streams comprising the even bits and the odd bits of the original byte data. In the second stage, pairs of bits from consecutive bytes are paired up to give runs of 4. In the final stage, runs of 4 are paired up to generate bit streams.

71
```
#define s2p_step(s0, s1, hi_mask, shift, p0, p1)
        {
          BitBlock t0, t1;
```

```
            t0 = simd_pack_16_hh(s0, s1);
            t1 = simd_pack_16_ll(s0, s1);
            p0 = simd_if(hi_mask, t0, simd_srli_16(t1, shift));
            p1 = simd_if(hi_mask, simd_slli_16(t0, shift), t1);
        }
#define s2p_bytepack(s0, s1, s2, s3, s4, s5, s6, s7, p0, p1, p2, p3, p4, p5, p6, p7)
        {
            BitBlock bit00224466_0, bit00224466_1, bit00224466_2, bit00224466_3;
            BitBlock bit11335577_0, bit11335577_1, bit11335577_2, bit11335577_3;
            BitBlock bit00004444_0, bit22226666_0, bit00004444_1, bit22226666_1;
            BitBlock bit11115555_0, bit33337777_0, bit11115555_1, bit33337777_1;
            s2p_step(s0, s1, mask_2, 1, bit00224466_0, bit11335577_0)
            s2p_step(s2, s3, mask_2, 1, bit00224466_1, bit11335577_1)
            s2p_step(s4, s5, mask_2, 1, bit00224466_2, bit11335577_2)
            s2p_step(s6, s7, mask_2, 1, bit00224466_3, bit11335577_3)
            s2p_step(bit00224466_0, bit00224466_1, mask_4, 2, bit00004444_0, bit22226666_0)
            s2p_step(bit00224466_2, bit00224466_3, mask_4, 2, bit00004444_1, bit22226666_1)
            s2p_step(bit11335577_0, bit11335577_1, mask_4, 2, bit11115555_0, bit33337777_0)
            s2p_step(bit11335577_2, bit11335577_3, mask_4, 2, bit11115555_1, bit33337777_1)
            s2p_step(bit00004444_0, bit00004444_1, mask_8, 4, p0, p4)
            s2p_step(bit11115555_0, bit11115555_1, mask_8, 4, p1, p5)
            s2p_step(bit22226666_0, bit22226666_1, mask_8, 4, p2, p6)
            s2p_step(bit33337777_0, bit33337777_1, mask_8, 4, p3, p7)
        }
```

⟨Transpose to parallel bit streams *u8bit0* through *u8bit7* 19⟩ +≡

```
#if (S2P_ALGORITHM ≡ S2P_BYTEPACK)
  {
    BitBlock mask_2 = simd_himask_2;
    BitBlock mask_4 = simd_himask_4;
    BitBlock mask_8 = simd_himask_8;
#if (BYTE_ORDER ≡ BIG_ENDIAN)
    s2p_bytepack(U8s0, U8s1, U8s2, U8s3, U8s4, U8s5, U8s6, U8s7,
    u8bit0, u8bit1, u8bit2, u8bit3, u8bit4, u8bit5, u8bit6, u8bit7)
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
    s2p_bytepack(U8s7, U8s6, U8s5, U8s4, U8s3, U8s2, U8s1, U8s0,
    u8bit0, u8bit1, u8bit2, u8bit3, u8bit4, u8bit5, u8bit6, u8bit7)
#endif
  }
#endif
```

¶ Parallel to serial transposition reverses the process.

72

```
#define p2s_step(p0, p1, hi_mask, shift, s0, s1)
        {
            BitBlock t0, t1;
            t0 = simd_if(hi_mask, p0, simd_srli_16(p1, shift));
            t1 = simd_if(hi_mask, simd_slli_16(p0, shift), p1);
            s0 = simd_mergeh_8(t0, t1);
            s1 = simd_mergel_8(t0, t1);
        }
```

```
#define p2s_bytemerge(p0, p1, p2, p3, p4, p5, p6, p7, s0, s1, s2, s3, s4, s5, s6, s7)
        {
          BitBlock bit00004444_0, bit22226666_0, bit00004444_1, bit22226666_1;
          BitBlock bit11115555_0, bit33337777_0, bit11115555_1, bit33337777_1;
          BitBlock bit00224466_0, bit00224466_1, bit00224466_2, bit00224466_3;
          BitBlock bit11335577_0, bit11335577_1, bit11335577_2, bit11335577_3;
          p2s_step(p0, p4, simd_himask_8, 4, bit00004444_0, bit00004444_1)
          p2s_step(p1, p5, simd_himask_8, 4, bit11115555_0, bit11115555_1)
          p2s_step(p2, p6, simd_himask_8, 4, bit22226666_0, bit22226666_1)
          p2s_step(p3, p7, simd_himask_8, 4, bit33337777_0, bit33337777_1)
          p2s_step(bit00004444_0, bit22226666_0, simd_himask_4, 2, bit00224466_0, bit00224466_1)
          p2s_step(bit11115555_0, bit33337777_0, simd_himask_4, 2, bit11335577_0, bit11335577_1)
          p2s_step(bit00004444_1, bit22226666_1, simd_himask_4, 2, bit00224466_2, bit00224466_3)
          p2s_step(bit11115555_1, bit33337777_1, simd_himask_4, 2, bit11335577_2, bit11335577_3)
          p2s_step(bit00224466_0, bit11335577_0, simd_himask_2, 1, s0, s1)
          p2s_step(bit00224466_1, bit11335577_1, simd_himask_2, 1, s2, s3)
          p2s_step(bit00224466_2, bit11335577_2, simd_himask_2, 1, s4, s5)
          p2s_step(bit00224466_3, bit11335577_3, simd_himask_2, 1, s6, s7)
        }
⟨ Transpose high UTF-16 bit streams to high byte stream 20 ⟩ +≡
#if (P2S_ALGORITHM ≡ P2S_BYTEMERGE)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_bytemerge(u16hi0, u16hi1, u16hi2, u16hi3, u16hi4, u16hi5, u16hi6, u16hi7,
  U16h0, U16h1, U16h2, U16h3, U16h4, U16h5, U16h6, U16h7)
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
  p2s_bytemerge(u16hi0, u16hi1, u16hi2, u16hi3, u16hi4, u16hi5, u16hi6, u16hi7,
  U16h7, U16h6, U16h5, U16h4, U16h3, U16h2, U16h1, U16h0)
#endif
#endif
```

73 ¶ ⟨ Transpose low UTF-16 bit streams to low byte stream 21 ⟩ +≡

```
#if (P2S_ALGORITHM ≡ P2S_BYTEMERGE)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_bytemerge(u16lo0, u16lo1, u16lo2, u16lo3, u16lo4, u16lo5, u16lo6, u16lo7,
  U16l0, U16l1, U16l2, U16l3, U16l4, U16l5, U16l6, U16l7)
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
  p2s_bytemerge(u16lo0, u16lo1, u16lo2, u16lo3, u16lo4, u16lo5, u16lo6, u16lo7,
  U16l7, U16l6, U16l5, U16l4, U16l3, U16l2, U16l1, U16l0)
#endif
#endif
```

¶ When a block of input consists of single and two-byte sequences only, the high 5 bits of the UTF-16 representation are always zero. Transposition of the remaining three bit streams (16*hi5* through *u16hi7* to high UTF-16 bytes is simplified in this case.

74

```
#define p2s_halfstep(p1, hi_mask, shift, s0, s1)
        {
          BitBlock t0, t1;
```

```
            t0 = simd_andc(sisd_srli(p1, shift), hi_mask);
            t1 = simd_andc(p1, hi_mask);
            s0 = simd_mergeh_8(t0, t1);
            s1 = simd_mergel_8(t0, t1);
          }
#define p2s_567_bytemerge(p5, p6, p7, s0, s1, s2, s3, s4, s5, s6, s7)
          {
            BitBlock bit22226666_0, bit22226666_1;
            BitBlock bit11115555_0, bit33337777_0, bit11115555_1, bit33337777_1;
            BitBlock bit00224466_0, bit00224466_1, bit00224466_2, bit00224466_3;
            BitBlock bit11335577_0, bit11335577_1, bit11335577_2, bit11335577_3;
            p2s_halfstep(p5, simd_himask_8, 4, bit11115555_0, bit11115555_1)
            p2s_halfstep(p6, simd_himask_8, 4, bit22226666_0, bit22226666_1)
            p2s_halfstep(p7, simd_himask_8, 4, bit33337777_0, bit33337777_1)
            p2s_halfstep(bit22226666_0, simd_himask_4, 2, bit00224466_0, bit00224466_1)
            p2s_step(bit11115555_0, bit33337777_0, simd_himask_4, 2, bit11335577_0, bit11335577_1)
            p2s_halfstep(bit22226666_1, simd_himask_4, 2, bit00224466_2, bit00224466_3)
            p2s_step(bit11115555_1, bit33337777_1, simd_himask_4, 2, bit11335577_2, bit11335577_3)
            p2s_step(bit00224466_0, bit11335577_0, simd_himask_2, 1, s0, s1)
            p2s_step(bit00224466_1, bit11335577_1, simd_himask_2, 1, s2, s3)
            p2s_step(bit00224466_2, bit11335577_2, simd_himask_2, 1, s4, s5)
            p2s_step(bit00224466_3, bit11335577_3, simd_himask_2, 1, s6, s7)
          }
```

⟨Transpose three high UTF-16 bit streams to high byte stream 22⟩ +≡

```
#if (P2S_ALGORITHM ≡ P2S_BYTEMERGE)
#if (BYTE_ORDER ≡ BIG_ENDIAN)
  p2s_567_bytemerge(u16hi5, u16hi6, u16hi7,
  U16h0, U16h1, U16h2, U16h3, U16h4, U16h5, U16h6, U16h7)
#endif
#if (BYTE_ORDER ≡ LITTLE_ENDIAN)
  p2s_567_bytemerge(u16hi5, u16hi6, u16hi7,
  U16h7, U16h6, U16h5, U16h4, U16h3, U16h2, U16h1, U16h0)
#endif
#endif
```

17 Altivec-Specific Implementation

76 ¶ ⟨Import idealized SIMD operations 2⟩ +≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
#include "../lib/altivec_simd.h"
#endif
```

77 ¶ ⟨Prefetch block data 77⟩ ≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  vec_dst(U8data, (8 ≪ 24) + (2 ≪ 16) + BLOCKSIZE, 2);
  vec_dstst(U16out, (16 ≪ 24) + (2 ≪ 16) + 2 * BLOCKSIZE, 1);
#endif
```

This code is used in chunks 24 and 31.

78 ¶ ⟨ Stop prefetch 78 ⟩ ≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  vec_dss(2);
  vec_dss(1);
#endif
```

This code is used in chunk 31.

79 ¶ ⟨ Load a full block of UTF-8 byte data 32 ⟩ +≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  {
    BitBlock r0, r1, r2, r3, r4, r5, r6, r7, r8;
    BitBlock input_shiftl = vec_lvsl(0, U8data);
    r0 = vec_ld(0, U8data);
    r1 = vec_ld(16, U8data);
    r2 = vec_ld(32, U8data);
    U8s0 = vec_perm(r0, r1, input_shiftl);
    r3 = vec_ld(48, U8data);
    U8s1 = vec_perm(r1, r2, input_shiftl);
    r4 = vec_ld(64, U8data);
    U8s2 = vec_perm(r2, r3, input_shiftl);
    r5 = vec_ld(80, U8data);
    U8s3 = vec_perm(r3, r4, input_shiftl);
    r6 = vec_ld(96, U8data);
    U8s4 = vec_perm(r4, r5, input_shiftl);
    r7 = vec_ld(112, U8data);
    U8s5 = vec_perm(r5, r6, input_shiftl);      /* Do not load beyond known input area (bytes 0 to 127). */
    r8 = vec_ld(127, U8data);
    U8s6 = vec_perm(r6, r7, input_shiftl);
    U8s7 = vec_perm(r7, r8, input_shiftl);
    u8advance = BLOCKSIZE;
    ⟨ Apply block shortening 33 ⟩
  }
#endif
```

¶ Load a block fragment as a full block with possible junk after the fragment end position. Make sure to avoid any access past the end of buffer.

80

```
#define min(x, y)  ((x) < (y) ? (x) : (y))
⟨ Load a block fragment 35 ⟩ +≡
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  {
    BitBlock r0, r1, r2, r3, r4, r5, r6, r7, r8;
    BitBlock input_shiftl = vec_lvsl(0, U8data);
    int last_byte = inbytes − 1;
    r0 = vec_ld(0, U8data);
    r1 = vec_ld(min(16, last_byte), U8data);
    r2 = vec_ld(min(32, last_byte), U8data);
    U8s0 = vec_perm(r0, r1, input_shiftl);
    r3 = vec_ld(min(48, last_byte), U8data);
    U8s1 = vec_perm(r1, r2, input_shiftl);
    r4 = vec_ld(min(64, last_byte), U8data);
    U8s2 = vec_perm(r2, r3, input_shiftl);
    r5 = vec_ld(min(80, last_byte), U8data);
    U8s3 = vec_perm(r3, r4, input_shiftl);
    r6 = vec_ld(min(96, last_byte), U8data);
    U8s4 = vec_perm(r4, r5, input_shiftl);
    r7 = vec_ld(min(112, last_byte), U8data);
    U8s5 = vec_perm(r5, r6, input_shiftl);
    r8 = vec_ld(min(127, last_byte), U8data);
    U8s6 = vec_perm(r6, r7, input_shiftl);
    U8s7 = vec_perm(r7, r8, input_shiftl);
    u8advance = inbytes;
  }
#endif
```

81 ¶⟨ Apply ASCII short-cut optimization and continue 36 ⟩ +≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  BitBlock vec_0 = simd_const_8(0);
  if (inbytes > PACKSIZE) {
    BitBlock r0, r1, r2, r3, r4;
    BitBlock input_shiftl = vec_lvsl(0, U8data);
    U8s0 = vec_perm(vec_ld(0, U8data), vec_ld(15, U8data), input_shiftl);
    if (¬simd_any_sign_bit_8(U8s0)) {
      int fill_to_align = PACKSIZE − align_offset(U16out);
      U16s0 = u16_merge0(vec_0, U8s0);
      pending = vec_perm(pending, U16s0, vec_lvsr(0, U16out));
      vec_st(pending, 0, U16out);
      u8advance = fill_to_align/2;
      u16advance = fill_to_align;
      ⟨ Advance pointers and counters 26 ⟩
      input_shiftl = vec_lvsl(0, U8data);
      r0 = vec_ld(0, U8data);
      while (inbytes > 4 ∗ PACKSIZE) {
        BytePack ∗U16pack = (BytePack ∗) U16out;
        r1 = vec_ld(16, U8data);
        r2 = vec_ld(32, U8data);
        U8s0 = vec_perm(r0, r1, input_shiftl);
        r3 = vec_ld(48, U8data);
        U8s1 = vec_perm(r1, r2, input_shiftl);
        r4 = vec_ld(64, U8data);
        U8s2 = vec_perm(r2, r3, input_shiftl);
        U8s3 = vec_perm(r3, r4, input_shiftl);
        if (simd_any_sign_bit_8(simd_or(simd_or(U8s0, U8s1), simd_or(U8s2, U8s3)))) break;
        sisd_store_aligned(u16_merge0(vec_0, U8s0), U16pack);
        sisd_store_aligned(u16_merge1(vec_0, U8s0), &U16pack[1]);
        sisd_store_aligned(u16_merge0(vec_0, U8s1), &U16pack[2]);
        sisd_store_aligned(u16_merge1(vec_0, U8s1), &U16pack[3]);
        sisd_store_aligned(u16_merge0(vec_0, U8s2), &U16pack[4]);
        sisd_store_aligned(u16_merge1(vec_0, U8s2), &U16pack[5]);
        sisd_store_aligned(u16_merge0(vec_0, U8s3), &U16pack[6]);
```

```
          pending = u16_merge1(vec_0, U8s3);
          sisd_store_aligned(pending, &U16pack[7]);
          u8advance = 4 * PACKSIZE;
          u16advance = 8 * PACKSIZE;
          ⟨ Advance pointers and counters 26 ⟩
          r0 = r4;
        }
        while (inbytes > PACKSIZE) {
          BytePack *U16pack = (BytePack *) U16out;
          r1 = vec_ld(16, U8data);
          U8s0 = vec_perm(r0, r1, input_shiftl);
          if (simd_any_sign_bit_8(U8s0)) break;
          sisd_store_aligned(u16_merge0(vec_0, U8s0), U16pack);
          pending = u16_merge1(vec_0, U8s0);
          sisd_store_aligned(pending, &U16pack[1]);
          u8advance = PACKSIZE;
          u16advance = 2 * PACKSIZE;
          ⟨ Advance pointers and counters 26 ⟩
          r0 = r1;
        }
      }
    }
#endif
```

17.1 Deletion by Central Result Induction/Packed Permutation Vector

Permutation vectors allow selection of arbitrary sets of bytes in a single *vec_perm* operation. For example, select all the nondeleted bytes into leftmost positions.

Packed permutation vectors consist of 2 consecutive 32-byte vectors packed into a single vector of 32 nybbles. Permutation values are confined to the range 0..15.

Packed permutation vectors can be computed by deleting indices of deleted elements. These deletions are applied in nybble space, operating on 32 elements at a time. This provides a 4:1 advantage over applying operations in doublebyte space, and a 2:1 advantage over applying operations in byte space.

Given a 128-bit delmask, the following logic computes 4 32-position packed permutation vectors that can be used to compute 8-position left deletion results.

82 ⟨ Determine deletion information from *delmask* 60 ⟩ +≡

```
#if (BYTE_DELETION ≡ BYTE_DEL_BY_PERMUTE_TO_LEFT8)
  {
    BitBlock d0, d1, q0, q1, p0, p1;
    BitBlock delmask_hi4 = simd_srli_8(delmask, 4);      /* Step 1. 2-¿4 central deletion */
    d0 = vec_perm(del2_4_shift_tbl, del2_4_shift_tbl, delmask_hi4);
    d1 = vec_perm(del2_4_shift_tbl, del2_4_shift_tbl, delmask);
    q0 = vec_mergeh(d0, d1);      /* 0A00 0B00 0C00 0D00 pattern 0..63 */
    q1 = vec_mergel(d0, d1);      /* 0A00 0B00 0C00 0D00 pattern 64 .. 128 */
    p0 = simd_srli_8(q0, 4);      /* 0000 0A00 0000 0C00 pattern 0..63 */
    p1 = simd_srli_8(q1, 4);      /* 0000 0A00 0000 0C00 pattern 64 .. 128 */
    l8perm0 = simd_rotl_8(packed_identity, vec_mergeh(p0, q0));
    l8perm1 = simd_rotl_8(packed_identity, vec_mergel(p0, q0));
    l8perm2 = simd_rotl_8(packed_identity, vec_mergeh(p1, q1));
    l8perm3 = simd_rotl_8(packed_identity, vec_mergel(p1, q1));      /* Step 2. 4-¿8 central deletion */
    d0 = vec_perm(del4_8_rshift_tbl, del4_8_rshift_tbl, delmask_hi4);
    d1 = vec_perm(del4_8_lshift_tbl, del4_8_lshift_tbl, delmask);
    p0 = vec_mergeh(d0, d1);      /* -4*(A+B), 4*(C+D), -4*(E+F) for 0..63 */
    p1 = vec_mergel(d0, d1);
    l8perm0 = simd_rotl_16(l8perm0, vec_unpackh(p0));
    l8perm1 = simd_rotl_16(l8perm1, vec_unpackl(p0));
    l8perm2 = simd_rotl_16(l8perm2, vec_unpackh(p1));
    l8perm3 = simd_rotl_16(l8perm3, vec_unpackl(p1));      /* Step 3. 8 central -¿ 8 left deletion */
    d0 = vec_perm(del8_shift_tbl, del8_shift_tbl, delmask_hi4);
    p0 = vec_unpackh(d0);
    p1 = vec_unpackl(d0);
    l8perm0 = simd_rotl_32(l8perm0, vec_unpackh(p0));
    l8perm1 = simd_rotl_32(l8perm1, vec_unpackl(p0));
    l8perm2 = simd_rotl_32(l8perm2, vec_unpackh(p1));
    l8perm3 = simd_rotl_32(l8perm3, vec_unpackl(p1));
  }
#endif
#if (DOUBLEBYTE_DELETION ≡ ALTIVEC_FROM_LEFT8)
  {
    BitBlock delmask_hi4 = simd_srli_8(delmask, 4);
    delcounts_8 = simd_add_8(vec_perm(bits_per_nybble_tbl, bits_per_nybble_tbl, delmask_hi4),
        vec_perm(bits_per_nybble_tbl, bits_per_nybble_tbl, delmask));
    u16_bytes_8 = simd_slli_8(simd_sub_8(simd_const_8(8), delcounts_8), 1);      /* 2 × (8 − d) */
  }
#endif
```

¶ Tables for computing deletion info.

83 ⟨Local variable declarations 9⟩ +≡

```
#if (BYTE_DELETION ≡ BYTE_DEL_BY_PERMUTE_TO_LEFT8)
  BitBlock bits_per_nybble_tbl = (BitBlock)(0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);
  BitBlock packed_identity = (BitBlock)(#01, #23, #45, #67, #89, #AB, #CD, #EF, #01, #23, #45, #67, #89,
      #AB, #CD, #EF);
  BitBlock del2_4_shift_tbl = (BitBlock)(0, 0, 4, 4, #40, #40, #44, #44, 0, 0, 4, 4, #40, #40, #44, #44);
  BitBlock del4_8_rshift_tbl = (BitBlock)(0, −4, −4, −8, 0, −4, −4, −8, 0, −4, −4, −8, 0, −4, −4, −8);
  BitBlock del4_8_lshift_tbl = (BitBlock)(0, 0, 0, 0, 4, 4, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8);
  BitBlock del8_shift_tbl =      /* 4 * bitcount */
  (BitBlock)(0, 4, 4, 8, 4, 8, 8, 12, 4, 8, 8, 12, 8, 12, 12, 16);
  BitBlock l8perm0, l8perm1, l8perm2, l8perm3;
#endif
#if (DOUBLEBYTE_DELETION ≡ ALTIVEC_FROM_LEFT8)
  BitBlock delcounts_8, u16_bytes_8;
#endif
```

84 ¶ ⟨Apply bit deletions to high five UTF-16 bit streams 62⟩ +≡

```
#if (BYTE_DELETION ≡ BYTE_DEL_BY_PERMUTE_TO_LEFT8)
  {      /* No operations on bit streams. */
  }
#endif
```

85 ¶ ⟨Apply bit deletions to low eleven UTF-16 bit streams 63⟩ +≡

```
#if (BYTE_DELETION ≡ BYTE_DEL_BY_PERMUTE_TO_LEFT8)
  {    /* No operations on bit streams. */
  }
#endif
```

86 ¶

```
#define unpack_packed_permutation(packed, high_perm, low_perm)
        {
          BitBlock even_perms = simd_srli_8(packed, 4);
          BitBlock odd_perms = simd_andc(packed, simd_himask_8);
          high_perm = simd_mergeh_8(even_perms, odd_perms);
          low_perm = simd_mergel_8(even_perms, odd_perms);
        }
```

⟨Apply byte stream transformations 64⟩ +≡

```
#if (BYTE_DELETION ≡ BYTE_DEL_BY_PERMUTE_TO_LEFT8)
  {
    BitBlock high_perm, low_perm;
    unpack_packed_permutation(l8perm0, high_perm, low_perm)
    U16l0 = vec_perm(U16l0, U16l0, high_perm);
    U16h0 = vec_perm(U16h0, U16h0, high_perm);
    U16l1 = vec_perm(U16l1, U16l1, low_perm);
    U16h1 = vec_perm(U16h1, U16h1, low_perm);
    unpack_packed_permutation(l8perm1, high_perm, low_perm)
    U16l2 = vec_perm(U16l2, U16l2, high_perm);
    U16h2 = vec_perm(U16h2, U16h2, high_perm);
    U16l3 = vec_perm(U16l3, U16l3, low_perm);
    U16h3 = vec_perm(U16h3, U16h3, low_perm);
    unpack_packed_permutation(l8perm2, high_perm, low_perm)
    U16l4 = vec_perm(U16l4, U16l4, high_perm);
    U16h4 = vec_perm(U16h4, U16h4, high_perm);
    U16l5 = vec_perm(U16l5, U16l5, low_perm);
    U16h5 = vec_perm(U16h5, U16h5, low_perm);
    unpack_packed_permutation(l8perm3, high_perm, low_perm)
    U16l6 = vec_perm(U16l6, U16l6, high_perm);
    U16h6 = vec_perm(U16h6, U16h6, high_perm);
    U16l7 = vec_perm(U16l7, U16l7, low_perm);
    U16h7 = vec_perm(U16h7, U16h7, low_perm);
  }
#endif
```

87 ¶

```
#define output_step(vec, vec_num)
        {
          BitBlock rshift, lshift;
          rshift = vec_lvsr(u16advance, U16out);
          vec_stl(vec_perm(pending, vec, rshift), u16advance, U16out);
          lshift = vec_add(vec_0__15, vec_splat(u16_bytes_8, vec_num));
          pending = vec_perm(pending, vec, lshift);
          u16advance += dbyte_count[vec_num];
        }
```

88 ¶⟨ Write compressed UTF-16 data 65 ⟩ +≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  {
    u16advance = 0;
    BitBlock vec_0__15 = vec_lvsl1(0);
    unsigned char ∗dbyte_count = (unsigned char ∗) &u16_bytes_8;
    output_step(U16s0, 0)
    output_step(U16s1, 1)
    output_step(U16s2, 2)
    output_step(U16s3, 3)
    output_step(U16s4, 4)
    output_step(U16s5, 5)
    output_step(U16s6, 6)
    output_step(U16s7, 7)
    output_step(U16s8, 8)
    output_step(U16s9, 9)
    output_step(U16s10, 10)
    output_step(U16s11, 11)
    output_step(U16s12, 12)
    output_step(U16s13, 13)
    output_step(U16s14, 14)
    output_step(U16s15, 15)
    vec_st(vec_perm(pending, simd_const_8(0), vec_lvsl1(16 − (#0F & ((int) &U16out[u16advance])))),
        u16advance − 1, U16out);
  }
#endif
```

¶ If the initial value of ∗*outbuf* is not on an aligned boundary, the existing data between the boundary and ∗*outbuf* must be loaded into the *pending* output data register.

89 ⟨ Local variable declarations 9 ⟩ +≡

```
#if (U8U16_TARGET ≡ ALTIVEC_TARGET)
  BitBlock start_of_output_existing = vec_ld(0, ∗outbuf);
  BitBlock pending = vec_perm(start_of_output_existing, start_of_output_existing, vec_lvsl(0, ∗outbuf));
#endif
```

18 MMX-Specific Implementation

¶ To right-justify within 4-bit fields, bits move at most three positions. For each bit position, determine the 2-bit coding for the amount to move as *del4_rshift2* and *del4_rshift1*. Initially, *del4_rshift1* is the *delmask* parity of the two positions immediately to the right (within the 4-bit field). One step of the parallel prefix method completes the calculation.

91 ⟨ Determine deletion information from *delmask* 60 ⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT4)
  del4_rshift1 = simd_xor(simd_slli_4(delmask, 1), simd_slli_4(delmask, 2));
  del4_rshift1 = simd_xor(del4_rshift1, simd_slli_4(del4_rshift1, 2));
    /∗ Transition to even delcount: odd delcount to right, this one deleted. ∗/
  del4_trans2 = simd_and(del4_rshift1, delmask);      /∗ Odd number of transition positions to right. ∗/
  del4_rshift2 = simd_xor(simd_slli_4(del4_trans2, 1), simd_slli_4(del4_trans2, 2));
  del4_rshift2 = simd_xor(del4_rshift2, simd_slli_4(del4_rshift2, 2));
      /* Only move bits that are not deleted. */
  del4_rshift1 = simd_andc(del4_rshift1, delmask);
  del4_rshift2 = simd_andc(del4_rshift2, delmask);
      /* Update del4_rshift2 to apply after del4_rshift1. */
  del4_rshift2 = simd_add_4(simd_and(del4_rshift1, del4_rshift2), del4_rshift2);
#endif
```

92 ¶ ⟨Local variable declarations 9⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT4)
  BitBlock del4_rshift1, del4_trans2, del4_rshift2;
#endif
```

¶ Right shift within 4-bit fields with the combination of a single-bit shift for bits that must move an odd number of positions and a 2-bit shift for bits that must move 2 or 3 positions.

93

```
#define do_right4_shifts(vec, rshift1, rshift2)
        {
          BitBlock s2;
          vec = simd_sub_8(vec, sisd_srli(simd_and(rshift1, vec), 1));
          s2 = simd_and(rshift2, vec);
          vec = simd_or(sisd_srli(s2, 2), simd_xor(vec, s2));
        }
```

⟨Apply bit deletions to high five UTF-16 bit streams 62⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT4)
  do_right4_shifts(u16hi0, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi1, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi2, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi3, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi4, del4_rshift1, del4_rshift2)
#endif
```

94 ¶ ⟨Apply bit deletions to low eleven UTF-16 bit streams 63⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT4)
  do_right4_shifts(u16hi5, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi6, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16hi7, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo0, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo1, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo2, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo3, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo4, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo5, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo6, del4_rshift1, del4_rshift2)
  do_right4_shifts(u16lo7, del4_rshift1, del4_rshift2)
#endif
```

95 ¶ ⟨Local variable declarations 9⟩ +≡

```
#if (DOUBLEBYTE_DELETION ≡ FROM_LEFT4)
  BitBlock delcounts_2, delcounts_4, u16_bytes_4;
#endif
```

96 ¶ ⟨Determine deletion information from *delmask* 60⟩ +≡

```
#if (DOUBLEBYTE_DELETION ≡ FROM_LEFT4)
  delcounts_2 = simd_add_2_lh(delmask, delmask);
  delcounts_4 = simd_add_4_lh(delcounts_2, delcounts_2);
  u16_bytes_4 = sisd_slli(simd_sub_8(simd_const_4(4), delcounts_4), 1);
#if BYTE_ORDER ≡ BIG_ENDIAN
  sisd_store_aligned(simd_mergeh_4(simd_const_4(0), u16_bytes_4), &u16_bytes_per_reg[0]);
  sisd_store_aligned(simd_mergel_4(simd_const_4(0), u16_bytes_4), &u16_bytes_per_reg[8]);
#endif
#if BYTE_ORDER ≡ LITTLE_ENDIAN
  sisd_store_aligned(simd_mergel_4(simd_const_4(0), u16_bytes_4), &u16_bytes_per_reg[0]);
  sisd_store_aligned(simd_mergeh_4(simd_const_4(0), u16_bytes_4), &u16_bytes_per_reg[8]);
#endif
#endif
```

97 ¶ ⟨Import idealized SIMD operations 2⟩ +≡

```
#if (U8U16_TARGET ≡ MMX_TARGET)
#include "../lib/mmx_simd.h"
#endif
```

19 SSE-Specific Implementation

¶ To right-justify within 8-bit fields, bits move at most seven positions. For each bit position, determine the 3-bit coding for the amount to move as *del8_rshift4*, *del8_rshift2*, and *del8_rshift1*. Initially, *del8_rshift1* is the *delmask* parity of the two positions immediately to the right (within the 8-bit field). Two steps of the parallel prefix method complete the calculation.

99 ⟨Determine deletion information from *delmask* 60⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT8)
  del8_rshift1 = simd_xor(simd_slli_8(delmask, 1), simd_slli_8(delmask, 2));
  del8_rshift1 = simd_xor(del8_rshift1, simd_slli_8(del8_rshift1, 2));
  del8_rshift1 = simd_xor(del8_rshift1, simd_slli_8(del8_rshift1, 4));
     /* Transition to even delcount: odd delcount to left, this one deleted. */
  del8_trans2 = simd_and(del8_rshift1, delmask);     /* Odd number of transition positions to left. */
  del8_rshift2 = simd_xor(simd_slli_8(del8_trans2, 1), simd_slli_8(del8_trans2, 2));
  del8_rshift2 = simd_xor(del8_rshift2, simd_slli_8(del8_rshift2, 2));
  del8_rshift2 = simd_xor(del8_rshift2, simd_slli_8(del8_rshift2, 4));
     /* Transition positions: odd del2count to left, this one a transition to even. */
  del8_trans4 = simd_and(del8_rshift2, del8_trans2);
  del8_rshift4 = simd_xor(simd_slli_8(del8_trans4, 1), simd_slli_8(del8_trans4, 2));
  del8_rshift4 = simd_xor(del8_rshift4, simd_slli_8(del8_rshift4, 2));
  del8_rshift4 = simd_xor(del8_rshift4, simd_slli_8(del8_rshift4, 4));
     /* Only move bits that are not deleted. */
  del8_rshift1 = simd_andc(del8_rshift1, delmask);
  del8_rshift2 = simd_andc(del8_rshift2, delmask);
  del8_rshift4 = simd_andc(del8_rshift4, delmask);
    /* Update del8_rshift2 to apply after del8_rshift1. */
  del8_rshift2 = simd_sub_8(del8_rshift2, simd_srli_16(simd_and(del8_rshift1, del8_rshift2), 1));
    /* Update del8_rshift4 to apply after del8_rshift2 and del8_rshift1. */
  del8_rshift4 = simd_sub_8(del8_rshift4, simd_srli_16(simd_and(del8_rshift1, del8_rshift4), 1));
  {
    BitBlock shift_bits = simd_and(del8_rshift2, del8_rshift4);
    del8_rshift4 = simd_or(simd_srli_16(shift_bits, 4), simd_xor(del8_rshift4, shift_bits));
  }
#endif
```

100 ¶ ⟨Local variable declarations 9⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT8)
  BitBlock del8_rshift1, del8_trans2, del8_rshift2, del8_trans4, del8_rshift4;
#endif
```

¶ Right shift within 8-bit fields with the combination of a single-bit shift for bits that must move an odd number of positions and a 2-bit shift for bits that must move 2, 3, 6 or 7 positions and a 4-bit shift for bits that must move 4 or more positions.

101

```
#define do_right8_shifts(vec, rshift1, rshift2, rshift4)
          {
            BitBlock s2;
            vec = simd_sub_8(vec, simd_srli_16(simd_and(rshift1, vec), 1));
            s2 = simd_and(rshift2, vec);
            vec = simd_or(simd_srli_16(s2, 2), simd_xor(vec, s2));
            s2 = simd_and(rshift4, vec);
            vec = simd_or(simd_srli_16(s2, 4), simd_xor(vec, s2));
          }
```

⟨Apply bit deletions to high five UTF-16 bit streams 62⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT8)
  do_right8_shifts(u16hi0, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi1, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi2, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi3, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi4, del8_rshift1, del8_rshift2, del8_rshift4)
#endif
```

102 ¶ ⟨Apply bit deletions to low eleven UTF-16 bit streams 63⟩ +≡

```
#if (BIT_DELETION ≡ SHIFT_TO_RIGHT8)
  do_right8_shifts(u16hi5, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi6, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16hi7, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo0, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo1, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo2, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo3, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo4, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo5, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo6, del8_rshift1, del8_rshift2, del8_rshift4)
  do_right8_shifts(u16lo7, del8_rshift1, del8_rshift2, del8_rshift4)
#endif
```

103 ¶ ⟨Import idealized SIMD operations 2⟩ +≡

```
#if (U8U16_TARGET ≡ SSE_TARGET)
#include "../lib/sse_simd.h"
#endif
```

¶ Although SSE provides 128-bit registers and hence allows bit-parallel processing of 128 character code units at a time, the largest bit-level shift operation available is only 64 bits. This creates boundary crossing issues at the block midpoint. This could be addressed by simulating full-register shift operations. Alternatively, the following subblock shortening strategy is used. When the first 64-position subblock is found to terminate with an incomplete UTF-8 sequence, the second subblock is loaded from the point of the prefix of this sequence.

105 ¶ ⟨Endianness definitions 16⟩ +≡

```
#ifdef PARTITIONED_BLOCK_MODE
#if BYTE_ORDER ≡ LITTLE_ENDIAN
#define bitblock_sfl(blk, n)simd_sll_64 (blk, n)
#define bitblock_sbl(blk, n)simd_srl_64 (blk, n)
#define bitblock_sfli(blk, n)simd_slli_64 (blk, n)
#define bitblock_sbli(blk, n)simd_srli_64 (blk, n)
#endif
#endif
```

106 ¶ ⟨Local variable declarations 9⟩ +≡

```
#ifdef PARTITIONED_BLOCK_MODE
  int half_block_adjust;
#endif
```

107 ¶ ⟨Load a full block of UTF-8 byte data 32⟩ +≡

```
#ifdef PARTITIONED_BLOCK_MODE
  {
    BitBlock ∗U8pack = (BitBlock ∗) U8data;
    U8s0 = sisd_load_unaligned(&U8pack[0]);
    U8s1 = sisd_load_unaligned(&U8pack[1]);
    U8s2 = sisd_load_unaligned(&U8pack[2]);
    U8s3 = sisd_load_unaligned(&U8pack[3]);
    if (is_prefix4_byte(U8data[BLOCKSIZE/2 − 3])) half_block_adjust = 3;
    else if (is_prefix3or4_byte(U8data[BLOCKSIZE/2 − 2])) half_block_adjust = 2;
    else if (is_prefix_byte(U8data[BLOCKSIZE/2 − 1])) half_block_adjust = 1;
    else half_block_adjust = 0;
    U8pack = (BitBlock ∗)(U8data − half_block_adjust);
    U8s4 = sisd_load_unaligned(&U8pack[4]);
    U8s5 = sisd_load_unaligned(&U8pack[5]);
    U8s6 = sisd_load_unaligned(&U8pack[6]);
    U8s7 = sisd_load_unaligned(&U8pack[7]);
    u8advance = BLOCKSIZE − half_block_adjust;
    ⟨ Apply block shortening 33 ⟩
  }
#endif
```

¶ When loading a block fragment at the end of the input buffer, care must be taken to avoid any possibility of a page fault. For a short fragment, a page fault could occur either by reading across an alignment boundary prior to the first byte or after the last byte.

108 ⟨ Load a block fragment 35 ⟩ +≡

```
#ifdef PARTITIONED_BLOCK_MODE
  {
    BytePack *U8pack = (BytePack *) U8data;
    int full_packs = inbytes/PACKSIZE;
    int remaining_packs = full_packs;
    int U8data_offset, U8s7_size;
    BitBlock half_mask1 = simd_const_8(0);
    BitBlock half_mask2;
    int pack = 0;
    half_block_adjust = 0;
    if (full_packs ≥ 4) {
      U8s0 = sisd_load_unaligned(&U8pack[0]);
      U8s1 = sisd_load_unaligned(&U8pack[1]);
      U8s2 = sisd_load_unaligned(&U8pack[2]);
      U8s3 = sisd_load_unaligned(&U8pack[3]);
      if (is_prefix4_byte(U8data[BLOCKSIZE/2 − 3])) half_block_adjust = 3;
      else if (is_prefix3or4_byte(U8data[BLOCKSIZE/2 − 2])) half_block_adjust = 2;
      else if (is_prefix_byte(U8data[BLOCKSIZE/2 − 1])) half_block_adjust = 1;
      U8pack = (BitBlock *)(&U8data[BLOCKSIZE/2] − half_block_adjust);
      full_packs = (inbytes + half_block_adjust)/PACKSIZE − 4;
      half_mask1 = sisd_sbli(simd_const_8(−1), BLOCKSIZE/2);
    }
    else half_mask1 = simd_const_8(0);
    switch (full_packs) {
    case 4:     /* Must treat as a full block */
      U8s4 = sisd_load_unaligned(&U8pack[pack++]);
      U8s5 = sisd_load_unaligned(&U8pack[pack++]);
      U8s6 = sisd_load_unaligned(&U8pack[pack++]);
      U8s7 = sisd_load_unaligned(&U8pack[pack++]);
      u8advance = inbytes − (inbytes + half_block_adjust) % BLOCKSIZE;
      ⟨ Apply block shortening 33 ⟩
      input_select_mask = simd_const_8(−1);
      break;
    case 3: U8s4 = sisd_load_unaligned(&U8pack[pack++]);
    case 2: U8s5 = sisd_load_unaligned(&U8pack[pack++]);
    case 1: U8s6 = sisd_load_unaligned(&U8pack[pack++]);
    case 0: U8data_offset = (inbytes − half_block_adjust) % PACKSIZE;
      U8s7_size = (inbytes + half_block_adjust) % PACKSIZE;
      U8data_offset = ((int) U8pack) % PACKSIZE;
      if (U8s7_size ≡ 0) U8s7 = simd_const_8(0);
      else if (U8data_offset + U8s7_size > PACKSIZE)    /* unaligned load safe and required. */
        U8s7 = sisd_load_unaligned(&U8pack[pack]);
      else {    /* aligned load required for safety */
        U8s7 = sisd_load_aligned(pack_base_addr(&U8pack[pack]));
        U8s7 = sisd_sbl(U8s7, sisd_from_int(8 * U8data_offset));
      }
      half_mask2 = simd_and(sisd_sbl(simd_const_8(-1), sisd_from_int(PACKSIZE - U8s7_size)),
          sisd_sfl(simd_const_8(-1), sisd_from_int(PACKSIZE * (7 - full_packs))));
      input_select_mask = simd_or(half_mask1, half_mask2);
      u8advance = inbytes;
    }
  }
#endif
```

Index

List of Refinements

⟨ Adjust to error position and signal the error 67 ⟩ Used in chunk 24.
⟨ Adjust *error_mask* to mark UTF-8 prefix positions 68 ⟩ Used in chunk 67.
⟨ Advance pointers and counters 26 ⟩ Used in chunks 24, 36, 67, and 81.
⟨ Apply ASCII short-cut optimization and continue 36, 81 ⟩ Used in chunk 24.
⟨ Apply bit deletions to high five UTF-16 bit streams 62, 84, 93, 101 ⟩ Used in chunk 57.
⟨ Apply bit deletions to low eleven UTF-16 bit streams 63, 85, 94, 102 ⟩ Used in chunk 57.
⟨ Apply block shortening 33 ⟩ Used in chunks 32, 79, 107, and 108.
⟨ Apply byte stream transformations 64, 86 ⟩ Used in chunk 57.
⟨ Apply validation, decoding and control logic on bit streams 28 ⟩ Used in chunk 24.
⟨ Complete validation by checking for prefix-suffix mismatches 50 ⟩ Used in chunk 28.
⟨ Compress bit streams and transpose to UTF-16 doublebyte streams 57 ⟩ Used in chunk 24.
⟨ Compute classifications of UTF-8 bytes 38 ⟩ Used in chunk 28.
⟨ Compute scope classifications for common decoding 43 ⟩ Used in chunk 28.
⟨ Determine deletion information from *delmask* 60, 82, 91, 96, 99 ⟩ Used in chunk 57.
⟨ Determine return values and exit 29 ⟩ Used in chunks 24 and 36.
⟨ Endianness definitions 16, 17, 105 ⟩ Used in chunk 5.
⟨ Extend decoding for four-byte sequences 54 ⟩ Used in chunk 28.
⟨ Extend scope classifications for four-byte sequences 45 ⟩ Used in chunk 28.
⟨ Extend scope classifications for three-byte sequences 44 ⟩ Used in chunk 28.
⟨ Extend validation for errors in four-byte sequences 49 ⟩ Used in chunk 28.
⟨ Extend validation for errors in three-byte sequences 48 ⟩ Used in chunk 28.
⟨ Identify deleted positions for basic multilingual plane giving *delmask* 55 ⟩ Used in chunk 28.
⟨ Identify deleted positions for general Unicode giving *delmask* 56 ⟩ Used in chunk 28.
⟨ Import idealized SIMD operations 2, 76, 97, 103 ⟩ Used in chunk 5.
⟨ Initiate validation for two-byte sequences 47 ⟩ Used in chunk 28.
⟨ Load a block fragment 35, 80, 108 ⟩ Used in chunk 31.
⟨ Load a block into serial bytepacks *U8s0* through *U8s7* 31 ⟩ Used in chunk 24.
⟨ Load a full block of UTF-8 byte data 32, 79, 107 ⟩ Used in chunk 31.
⟨ Local variable declarations 9, 10, 12, 13, 14, 25, 37, 42, 46, 59, 83, 89, 92, 95, 100, 106 ⟩ Used in chunk 5.
⟨ Main block processing algorithm of *u8u16* 24 ⟩ Used in chunk 5.
⟨ Merge high and low byte streams to doublebyte streams 23 ⟩ Used in chunk 57.
⟨ Output buffer is guaranteed to be big enough 6 ⟩ Used in chunk 5.
⟨ Perform initial decoding of high five UTF-16 bit streams 53 ⟩ Used in chunk 28.
⟨ Perform initial decoding of low eleven UTF-16 bit streams 52 ⟩ Used in chunk 28.
⟨ Prefetch block data 77 ⟩ Used in chunks 24 and 31.
⟨ Stop prefetch 78 ⟩ Used in chunk 31.
⟨ Test whether the block is above the basic multilingual plane 40 ⟩ Used in chunk 28.
⟨ Test whether the block is above the two-byte subplane 39 ⟩ Used in chunks 28 and 57.
⟨ Transpose high UTF-16 bit streams to high byte stream 20, 72 ⟩ Used in chunk 57.
⟨ Transpose low UTF-16 bit streams to low byte stream 21, 73 ⟩ Used in chunk 57.
⟨ Transpose three high UTF-16 bit streams to high byte stream 22, 74 ⟩ Used in chunk 57.
⟨ Transpose to parallel bit streams *u8bit0* through *u8bit7* 19, 71 ⟩ Used in chunk 24.
⟨ Type declarations 8 ⟩ Used in chunk 5.
⟨ Write compressed UTF-16 data 65, 88 ⟩ Used in chunk 57.