cadence®

Lempel Ziv Compression for Solid State Storage

Application Note

Cadence Design Systems, Inc.
2566 Seely Ave.
San Jose, CA 95134

www.cadence.com

December 2012 Doc Number: AN-12-645-00

Document Change History:

Published November 2012

November 2015 reformatted

Contents

Tables

Abstract

Data compression is a key feature of Solid State Disk Drive (SSD) controller designs. For most SSD controller applications, the high data throughput rates and resulting high computational requirements of the data compression algorithms typically dictate that the compression function be implemented in a dedicated hardware block in the controller System on Chip (SOC) design. The downsides of dedicated hardware compression blocks include (a) inflexibility (non-programmability) which limits the range of SSD applications in which a single controller design can be used, and (b) high development risk associated with implementing complex algorithms purely in finite state machines. The high risk and low range of applicable usage for a hardware-based approach to SSD Controller design often leads to unacceptable project return on investment (ROI).

This application note illustrates an alternative design approach – the use of customized, optimized Xtensa dataplane processors (DPUs) that are specifically tuned to both the algorithmic and data throughput requirements of compression functions for SSD controller SOCs. With customized Xtensa DPUs it is possible to implement not only the compression functionality but also many other SSD dataplane functions in programmable DPUs, a methodology alternative that decreases SOC design time, decreases SOC verification efforts, and provides post-silicon flexibility. Lower design costs plus greater re-use of the SOC design in multiple SSD drive products can lead to dramatically better ROI.

This document provides a data point to illustrate this concept, showing an implementation of LZ77 data compression optimization improved by 550% (compared to a baseline 3-issue VLIW Xtensa CPU configuration) with a small added silicon area of only 16.5 kGates.

Note: The baseline Xtensa configuration, the Diamond Standard 570T controller CPU is comparable in general application performance to other high-end, real-time controller CPU designs on the market today.

Note that this illustration does not represent the maximum data compression performance possible in a single Xtensa core. Significantly higher levels of performance are attainable through the use of additional parallelization of the computation load and through aggressive use of Tensilica's patented unconventional designer-defined I/O data streams. The specific optimization illustrated in this application note is kept to the lower-end of possible acceleration to aid in your quick comprehension, and is designed to illustrate the concepts of compression algorithm acceleration without the complexity of instruction parallelism (VLIW techniques employing the Xtensa FLIX instruction feature) or data parallelism using SIMD techniques. Your Cadence applications engineer will gladly assist you in exploring how these more aggressive techniques can be applied to your specific SSD compression algorithmic requirements.

In addition to this document, an associated Tensilica software development tools workspace for the Xtensa Xplorer GUI development environment is available that includes source code for all examples cited herein. Please contact your local Cadence sales office for access to the example code workspace or view the appendix for details.

1. Introduction

Data compression is commonly used to increase the effective storage capacity of devices. For Flash storage, data compression has the added benefit of reducing Write Amplification –the extra data that has to be written because whole blocks must be written even when only a few bytes of data are actually changed.

This document briefly describes the Lempel Ziv compression techniques and shows how the LZSS variant can be implemented very efficiently on an Xtensa DPU using Tensilica's instruction customization capability, the Tensilica Instruction Extension (TIE) language and methodology.

2. Lempel Ziv (LZ) Algorithms

This section contains information from the Wikipedia article :
http://en.wikipedia.org/wiki/LZ77_and_LZ78

and from Michael Dipperstein's LZSS site:
http://michael.dipperstein.com/lzss/

There are many variants of LZ algorithms, which all work by parsing the input sequence into distinct phrases. Each phrase is encoded by reference to some previous phrase and perhaps some additional information. This exploits redundancy that might exist in the input sequence.

2.1 LZ77 Algorithms

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the input (uncompressed) data stream. A match is encoded by a pair of numbers called a *length-distance pair*, which is equivalent to the statement "each of the next *length* characters is equal to the characters exactly *distance* characters behind it in the uncompressed stream". (The "distance" is sometimes called the "offset" instead.)

To spot matches, the encoder must keep track of some amount of the most recent data, such as the last 2 kB, 4 kB, or 32 kB. The structure in which this data is held is called a *sliding window*, which is why LZ77 is sometimes called **sliding window compression**. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to. The larger the sliding window is, the longer back the encoder may search for creating references.

It is not only acceptable, but also frequently useful to allow length-distance pairs to specify a length that actually exceeds the distance. Tackling one byte at a time, there is no problem serving this request, because as a byte is copied over, it may be fed again as input to the copy command. When the copy-from position makes it to the initial destination position, it is consequently fed data that was pasted from the *beginning* of the copy-from position. The operation is thus equivalent to the statement "copy the data you were given and repetitively paste it until it fits". As this type of pair repeats a single copy of data multiple times, it can be used to incorporate a flexible and easy form of run-length encoding.

2.2 LZSS Algorithms

In their original LZ77 algorithm, Lempel and Ziv proposed that all strings be encoded as a length and offset, even strings with no match. Storer and Szymanski observed that individual unmatched symbols or matched strings of one or two symbols take up more space to encode than they do to leave uncoded.

For example, encoding a string from a dictionary of 4096 symbols, and allowing for matches of up to 15 symbols, requires 16 bits to store an offset and a length. A single character is only 8 bits. Using this method, encoding a single character doubles the required storage.

Storer and Szymanski proposed preceding each symbol with a coded/uncoded flag. Using the above example it now takes 9 bits for each uncoded symbol and 17 bits for each encoded string.

Storer and Szymanski also observed that if you are not encoding strings of length 0 to *M*, then *M* may be used as an offset to the length of the match. Applying this observation, instead of using 4 bits to represent match lengths of 0 to 15, the same 4 bits may be used to represent match lengths of *M* to (15 +*M*).

3. LZSS Implementations

This document uses and presents parts of the “LZSS” library from Michael Dipperstein that can be found at http://michael.dipperstein.com/lzss/, available under the GNU Lesser General Public License.

3.1 Decoding / Decompression

Decoding an LZSS compressed file is fairly straightforward and thus is not illustrated in this document. The source code for decoding is included in the associated Xplorer Workspace.

3.2 Encoding / Compression

There are a couple of common ways to implement the sliding window search function (called *FindMatch* in the reference code). This is the function that searches the sliding window for a match and is what takes up the most time.

Which approach to use usually depends on whether additional memory can be used to store a hash table or not. The following section explains how the Brute Force approach (no use of local memory storage) as well as the hash table approach (using additional memory) can be implemented with Xtensa processors.

4. Compression using the Brute Force Approach

The brute force approach is comparable to methods employed when built as *a stand-alone RTL accelerator.*

This brute force method does not require the use of any additional memory, but takes more cycles. For more information, see the file called *brute.c* in the reference code.

4.1 Analysis and Profiling

Starting with the original reference C code for *brute*, we profiled the code running on the reference Diamond Standard 570T CPU. Our profile results show the following (XCC compiler flags: -O2 –g):

TABLE 1. PROFILED CODE RESULTS

Function	Cycles (% of all)
Total	55,956,903 (100%)
FindMatch	55,067,020 (98.4%)
EncodeLZSSByFile	130,773 (0.23%)
fgetc	121,621 (0.21%)
DecodeLZSSByFile	111,926 (0.20%)
BitFilePutBit	57,986 (0.10%)
BitFileGetBit	45,894 (0.08%)

The FindMatch function consumes the vast majority of the cycles and is where we focused our acceleration.

The FindMatch() function is shown below for reference. The full source code is available in the accompanying Xplorer Workspace (the TIE additions are removed to simplify the display here)..

FindMatch() function Reference 'C' Source Code:

```
/************************************************************************
*   Function    : FindMatch
*   Description: This function will search through the slidingWindow
*                dictionary for the longest sequence matching the MAX_CODED
*                long string stored in uncodedLookahed.
*   Parameters : windowHead - head of sliding window
*                uncodedHead - head of uncoded lookahead buffer
*   Effects     : None
*   Returned    : The sliding window index where the match starts and the
*                 length of the match.  If there is no match a length of
*                 zero will be returned.
************************************************************************/
encoded_string_t FindMatch(unsigned int windowHead, unsigned int uncodedHead) {

  encoded_string_t matchData; // Contiguous matches and window offset
  unsigned int wp,     // Window position
    nm,                // Number of continguous matches
    wIndex, wData,     // Sliding Window index and char data
    uIndex, uData;     // Uncoded Head index and char data

  unsigned char ucodedHead;   // Beginning of the uncoded data
```

```
  // 1. Initializations
  matchData.length = nm = 0;    // No matches
  matchData.offset = 0;        // At start
  wp = windowHead;             // Set to beginning of sliding window
  ucodedHead = uncodedLookahead[uncodedHead];

  /*
   * 2. Find the longest sequence that matches the uncoded data
   * within the sliding window.
   */
  while(TRUE) {
   // Is next byte a match?
   if(slidingWindow[wp] == uncodedHead /*uncodedLookahead[uncodedHead]*/) {

    // 3. We matched one - find further contiguous matches
    nm = 1;    // Initialize to one match

    while(TRUE) {
      // 4a. Next indices in sliding window and uncoded data
      wIndex = Wrap((wp + nm), WINDOW_SIZE);
      uIndex = Wrap((uncodedHead + nm), MAX_CODED);

      wData = slidingWindow[wIndex];     // Next window char
      uData = uncodedLookahead[uIndex];  // Next uncoded char

      if(wData != uData) break;   // No match, exit inner loop

      // 5a. Increment number of matches
      if(nm >= MAX_CODED) break;  // Over limit, exit inner loop
      nm++;     // Matched - increment counter
    }


    // 6. Is the last match sequence the longest yet?
    if(nm > matchData.length) {   // Yes, update
      matchData.length = nm;
      matchData.offset = wp;
    }
   }

   // 7. Set up to start at the next point in the sliding window
   if(nm >= MAX_CODED) {             // Over limit?
      matchData.length = MAX_CODED;   // All chars matched
      break;                          // Exit outer loop
   }

   // 8a. Increment wp for starting at next location in window
   wp = Wrap((wp + 1), WINDOW_SIZE); // Increment
   if(wp == windowHead) break;       // At start, exit outer loop
  }

  return matchData;      // Contains the longest matching sequence and offset
}
```

4.2 TIE Implementation

Following are the areas that were studied and implemented in TIE for this document:

- Searching for the next match in the sliding window
- Wrapping a circular buffer

- Getting the next character in window with wrapping
- Incrementing the number of matches with maximum limit
- Advancing the starting position for FindMatch in the sliding window

4.2.1 Searching for the Next Match in the Sliding Window

From the reference 'C' code:

```
while(TRUE) {
 // 2a. Is next byte a match?
 if(slidingWindow[wp] == uncodedHead /*uncodedLookahead[uncodedHead]*/) {

  // 3. We matched one - find further contiguous matches
  .
  .
  .
}
```

This compares each byte in the sliding window with the uncoded byte until it finds a match.

The resulting assembly code (DC_570T, -O2), which requires 6 cycles looks like the following:

```
60002549: loop    a0, 600025bf
6000254c: add.n   a9, a7, a12
6000254e: l8ui    a9, a9, 0
60002551: bne  a9, a15, 60002598

60002554: movi.n a6, 1
```

This performs a single byte read, yet most processors load at least 4 bytes at a time as a 32-bit word. This can be optimized to compare each of the word's 4 bytes in turn after a single load. This reduces the number of loads considerably, but adds complexity in the comparison as one has to deal with unaligned bytes in the word, and find the first byte that matches. Replicating the byte four times into a word, comparing the whole word and then finding the first one still takes many cycles and may not end up saving many cycles.

With an Xtensa processor, we add TIE to do the comparison of multiple bytes and update the window pointer to the first one that matched in a single cycle, as follows:

```
/* ------------------------------------------------------------------------
 * Sliding window read and compare
 *   Read multiple bytes (characters) at once and compare (4 bytes in this
example)
 *   If match is found:
 *     stop reading
 *     update the window pointer to the first matched byte position
 *   Otherwise
 *     update the window pointer to one byte before next position to be read
 *
 */

operation SlidingRead
{out BR match, inout AR wp, in AR *slidingwindow, in AR ucodedHead}
{out VAddr, in MemDataIn32, in WINDOWSIZE, inout NUMMATCHES}
{

   wire [31:0] addr = slidingwindow + wp;
   wire [1:0] align = addr[1:0];

   // 1. Fetch the next 4 bytes
```

```
   assign VAddr = {addr[31:2], 2'b00}; // 32-bit aligned
   wire [31:0] data = MemDataIn32;

   // 2. Alignment
   wire align0 = (align == 2'b00);
   wire align1 = (align == 2'b01);
   wire align2 = (align == 2'b10);
   wire align3 = (align == 2'b11);

   // 3. Compare data
   wire b0match = ucodedHead[7:0] == data[7:0];
   wire b1match = ucodedHead[7:0] == data[15:8];
   wire b2match = ucodedHead[7:0] == data[23:16];
   wire b3match = ucodedHead[7:0] == data[31:24];
   wire [3:0] bmatch = {b3match, b2match, b1match, b0match};

   // 4. For each byte position, is it at the end of the window?
   wire top1 = (wp == WINDOWSIZE);            // Last byte
   wire top2 = ((wp+32'd1) == WINDOWSIZE);    // 2 bytes from end
   wire top3 = ((wp+32'd2) == WINDOWSIZE);    // 3 bytes from end

   // 5. Set valid bitmask for bytes that should be compared per alignment
   wire [3:0] v0 = top1 ?       4'b0001:
                   top2 ?       4'b0011:
                   top3 ?       4'b0111:
                          4'b1111;
   wire [3:0] v1 = top1 ?       4'b0010:
                   top2 ?       4'b0110:
                          4'b1110;
   wire [3:0] v2 = top1 ?       4'b0100:
                                4'b1100;
   wire [3:0] v3 =        4'b1000;

   // 6. Compare just the bytes that are valid for this alignment
   wire [3:0] valid = TIEsel(align0, v0, align1, v1, align2, v2, align3, v3);
   wire [3:0] validmatch = bmatch & valid;

   // 7. Was there a match?
   assign match = (validmatch != 4'b0);
   assign NUMMATCHES =  (validmatch != 4'b0) ? 1 : NUMMATCHES;

   // 8. Calculate extra increment for wp (1 is added by default in caller)
   // - set increment for each alignment in inc0 thru inc3.
   // - prevent wrapping over end using topN
   wire [2:0] inc0 = (validmatch[0] | top1) ?       3'b000:
                     (validmatch[1] | top2) ?       3'b001:
                     (validmatch[2] | top3) ?       3'b010:
                                                    3'b011;
   wire [2:0] inc1 = (validmatch[1] | top1) ?       3'b000:
                     (validmatch[2] | top2) ?       3'b001:
                                                    3'b010;
   wire [2:0] inc2 = (validmatch[2] | top1) ?       3'b000:
                                                    3'b001;
   wire [2:0] inc3 =                                3'b000;

   // 9. Set the increment for the current alignment
   wire [2:0] wpinc = TIEsel(    align0, inc0, align1, inc1,
                                 align2, inc2, align3, inc3);
   wire [31:0] wpinc32 = {29'b0, wpinc};

   assign wp = wp + wpinc32; // Increment
}
```

The multiple-line complexity shown in Sections 5 and 7 of the code is for dealing with the four possible alignments of the current position of the byte pointer `wp`.

The updated 'C' source code is:

```
while(TRUE) {
 // 2b. Find first match to uncoded head in next group of window bytes
 SlidingRead(matchb, wp, &slidingWindow, ucodedHead);
 if (matchb) { // One of the bytes matched, wp points to matched byte
  // 3. We matched one - find further contiguous matches
   .
   .
   .
}
```

The Xplorer workspace shows this 32-bit example, as well as a version that reads 64 bits at a time for even fewer cycles when wider loads are configured. Note that the Xtensa processor can be configured with Load/Store units up to 512 bits wide. Thus the acceleration examples shown here (32 bits) can easily be scaled by an additional factor of 16x.

4.2.2 Circular Buffer Wrapping

There are two circular buffers in this implementation. One for the input character stream being compressed and the other for the sliding window holding previously compressed input data. These buffers can be of different lengths and are accessed frequently during the compression.

The macro for the "Wrap" function is as follows:

```
Wrap(value, limit)   (((value) < (limit)) ? (value) : ((value) - (limit)))
```

This results in the following assembler instruction sequence (DC_570T, -O2):

```
60002598: { addi a9, a7, 1; bltu.w18   a11, a6, 600025a8 <FindMatch+0x80> }
600025a0: bltu   a8, a9, 600025b2 <FindMatch+0x8a>
600025a3: j   600025b7 <FindMatch+0x8f>

600025a6 <FindMatch+0x7e>:
   ...

600025a8 <FindMatch+0x80>:
600025a8: { l32i a2, a1, 0; movi       a3, 18; nop }
600025b0: retw.n

600025b2 <FindMatch+0x8a>:
600025b2: l32r   a9, 6000192c <_stext+0x54>
600025b5: add.n  a9, a7, a9
600025b7: { mov.n   a7, a9; beq.w18    a9, a2, 600025c8 <FindMatch+0xa0> }

600025bf <FindMatch+0x97>:
600025bf: j   60002549 <FindMatch+0x21>
```

This can take 2 to 4 cycles to execute on a traditional processor. With the addition of a custom instruction, this can be done in a single cycle with an Xtensa processor.

We created a general purpose TIE function that can be used for wrapping to different buffer lengths – this is called by the `L8Wrap` and `IncWP` operations (shown in the following sections of this application note).

The TIE code is shown as follows:

```
/* ------------------------------------------------------------------------
 * Add offset to base and wrap them if over the boundary (circular buffer)
 *
 * result == (base + offset) % wlen
 */
function [31:0] wrap ([31:0] base, [31:0] offset, [31:0] wlen) shared
{
     wire [31:0] tmp = base + offset;
     wire [31:0] newOffset = (tmp < wlen)? tmp : tmp - wlen;
     assign wrap = newOffset;
}
```

This is called using the “Wrap” intrinsic, as shown in the updated source code in the following sections of this application note.

4.2.3 Get Next Char in Window with Wrapping

The reference ‘C’ source code:

```
      // 4a. Next indices in sliding window and uncoded data
      wIndex = Wrap((wp + nm), WINDOW_SIZE);
      uIndex = Wrap((uncodedHead + nm), MAX_CODED);

      wData = slidingWindow[wIndex];     // Next window char
      uData = uncodedLookahead[uIndex];  // Next uncoded char
```

produces the following assembly code (DC_570T; -O2) that consumes 4 to 8 cycles:

```
6000255e: add.n  a9, a7, a6
60002560: { mov.n   a10, a9; bltu.w18  a8, a9, 6000256b <FindMatch+0x3b> }
60002568: j   6000256e <FindMatch+0x3e>

6000256b <FindMatch+0x3b>:
6000256b: addmi  a10, a9, 0xfffff000

6000256e: { add  a9, a3, a6; add       a4, a10, a12; nop }
60002576: { l8ui a4, a4, 0; bgeu.w18   a11, a9, 60002581 <FindMatch+0x51> }

6000257e: addi   a9, a9, -18
```

For each of the sliding window and uncoded data, this code calculates a wrapped index with the same offset (`nm`) into the different circular buffers using a common macro “Wrap” and then gets the data at that index.

Here is the single-cycle TIE implementation that does the offset, wrapping and returning of the data for one circular buffer. In this case, the common offset “nm” is stored in a state register “NUMMATCHES” and thus is not an explicit operand of the `L8Wrap` function.

```
/* ------------------------------------------------------------------------
 * Load next char (8 bits) from wrapped offset to circular buffer base (addr)
 * This takes an isSlidingWindow parameter to determine the wrap point
 *
 * Effect:
 * limit = WINDOWSIZE if isSlidingWindow, otherwise MAXCODED
 * data = addr[ (index+NUMMATCHES)%limit ]
 */
operation L8Wrap {out AR data, in AR *addr, in AR index, in SW
isSlidingWindow}
{in WINDOWSIZE, in MAXCODED, in NUMMATCHES, out VAddr, in MemDataIn8}
{
     wire [31:0] limit = isSlidingWindow ? WINDOWSIZE : MAXCODED;
```

```
    wire [31:0] offset = wrap(index, NUMMATCHES, limit); // Add with wrap
    wire [31:0] memaddr = addr + offset;          // Offset into the buffer
    assign VAddr = memaddr;                         // Issue the data fetch

    assign data = {24'b0, MemDataIn8};                  // Fetched data
}
```

The new ‘C’ source code to replace the reference code is:

```
      wData = L8Wrap(&slidingWindow[0], wp, 1);
      uData = L8Wrap(&uncodedLookahead[0], uncodedHead, 0);
```

This updated source code produces the following assembly code (DC_570T + TIE) that now consumes only 2 cycles:

```
60002598: l8wrap a4, a8, a3, 0
6000259b: l8wrap a15, a7, a6, 1
```

4.2.4 Increment the Number of Matches with Max Limit

The Reference ‘C’ source code:

```
      // 5a. Increment number of matches
      if(nm >= MAX_CODED) break;  // Over limit, exit inner loop
      nm++;                       // Matched - increment counter
```

produces the following assembly code (DC_570T; -O2):

```
60002589: bltu   a11, a6, 60002595
6000258c: { addi a6, a6, 1; j   6000255e
```

This requires 5 cycles in the case where the limit is reached and 3 cycles if nm is incremented.

Here is the single-cycle TIE code that increments up to the MAXCODED limit:

```
/* ------------------------------------------------------------------------
 * Increment the number of matches, up to the MAXCODED limit
 */
operation IncNUMMATCHES {out BR exit}
{in MAXCODED, inout NUMMATCHES}
{
    wire limit = NUMMATCHES >= MAXCODED;    // At/over the limit?
    assign NUMMATCHES = NUMMATCHES + 32'b1; // Increment
    assign NUMMATCHES_kill = limit;         // Don't inc if over/at limit
    assign exit = NUMMATCHES >= MAXCODED;   // Return the exit status
}
```

In the new source ‘C’ code:

```
      // 5b. Increment NUMMATCHES state unless over MAX_CODED limit
      done = IncNUMMATCHES();          // Sets done if over limit
      if (done) break;                 // Exit inner loop when over limit
      }
```

the updated source code now produces the following assembly code (570T + TIE) that only consumes 1 to 3 cycles:

```
6000259e: incnummatches  b2, a15, a4
600025a1: bt     b2, 6000257b <FindMatch+0x2b>
```

4.2.5 Advance the Starting Position for FindMatch in the Sliding Window

The reference source code:

```
    // 7. Set up to start at the next point in the sliding window
    if(nm >= MAX_CODED) {              // Over limit?
      matchData.length = MAX_CODED;    // All chars matched
      break;                           // Exit outer loop
    }
```

has a similar performance to the previously optimized code, taking 3 to 5 cycles.

The TIE code:

```
/* ----------------------------------------------------------------------
 * Increment circular window position (wp) with wrapping
 * Don't increment if reached coding limit or wrapped to beginning (windowHead)
 *
 * Effect:
 * if((wp+1)!=windowHead && NUMMATCHES<MAXCODED) {
 *          wp = wp + 1;
 *          exit = false;
 * } else {
 *          if(NUMMATCHES>MAXCODED) nm = MAXCODED;
 *          exit = true;
 * }
 */
operation IncWP {out AR nm, out BR exit, inout AR wp, in AR windowHead}
{in WINDOWSIZE, in NUMMATCHES, in MAXCODED, inout WRAPAROUND}
{
     // 1. Set earlyexit if we've reached the encoding limit
     wire earlyexit = NUMMATCHES >= MAXCODED;
     assign nm = MAXCODED;
     assign nm_kill = ~earlyexit;  // Don't update nm if we're not exiting

     // 2. Increment index and wrap over end of search window
     wire [31:0] next = wrap(wp, 32'b1, WINDOWSIZE);
     assign wp = next;
     assign wp_kill = earlyexit;  // Don't update wp if we need to exit

   // 3. Detect wraparound but keep current state in case already wrapped
   wire nextwrap = (next == 32'b0) ? 1'b1 : WRAPAROUND
   assign WRAPAROUND = nextwrap;

     // 4. Set exit if past end of search window or reached max encoding limit
     assign exit = (next==windowHead) | (nextwrap & (next>windowHead)) |
earlyexit;
}
```

produces new ‘C’ source code as follows that only requires 1 to 3 cycles:

```
    // 7. Set up to start at the next point in the sliding window
    // 8b. Increment wp with window wrap, set done if wrapped to start
    IncWP(matchData.length, done, wp, windowHead);
    if(done) break;                    // At start, exit outer loop
```

4.3 Updated 'C' Source Code

Following is the new 'C' Source Code reflecting all previous changes in this section:

```
/*****************************************************************************
*   Function   : FindMatch
*   Description: This function will search through the slidingWindow
*                dictionary for the longest sequence matching the MAX_CODED
*                long string stored in uncodedLookahed.
*   Parameters : windowHead - head of sliding window
*                uncodedHead - head of uncoded lookahead buffer
*   Effects    : None
*   Returned   : The sliding window index where the match starts and the
*                length of the match.  If there is no match a length of
*                zero will be returned.
*****************************************************************************/
encoded_string_t FindMatch(unsigned int windowHead, unsigned int uncodedHead) {

  encoded_string_t matchData; // Contiguous matches and window offset
  unsigned int wp,     // Window position
    nm,                // Number of continguous matches
    wIndex, wData,     // Sliding Window index and char data
    uIndex, uData;     // Uncoded Head index and char data

  unsigned char ucodedHead;   // Beginning of the uncoded data

  // 1. Initializations
  matchData.length = nm = 0;      // No matches
  matchData.offset = 0;        // At start
  wp = windowHead;             // Set to beginning of sliding window
  ucodedHead = uncodedLookahead[uncodedHead];

  xtbool done;                // Used to early-terminate loops
  xtbool matchb;              // Used to identify any matched bytes
  // State registers
  WUR_WINDOWSIZE(WINDOW_SIZE);
  WUR_MAXCODED(MAX_CODED);
  WUR_NUMMATCHES(nm);
  WUR_WRAPAROUND(0);

  /*
   * 2. Find the longest sequence that matches the uncoded data
   * within the sliding window.
   */
  while(TRUE) {
   // 2b. Find first match to uncoded head in next group of window bytes
   SlidingRead(matchb, wp, &slidingWindow, ucodedHead);
   if(matchb) {
    // 3. We matched one - find further contiguous matches

    while (TRUE) {
    // 4b. Next char in sliding window and uncoded data
    // L8Wrap adds the NUMMATCHES state to get the address
    wData = L8Wrap(&slidingWindow[0], wp, 1);
    uData = L8Wrap(&uncodedLookahead[0], uncodedHead, 0);

    ////if (wData != uData) break;       // No match, exit inner loop

    // 5b. Increment NUMMATCHES state unless over MAX_CODED limit
    done = IncNUMMATCHES ();            // Sets done if over limit
    if (done) break;                    // Exit inner loop when over limit
    }
    nm = RUR_NUMMATCHES();              // Number of matches from state
```

```
    // 6. Is the last match sequence the longest yet?
    if(nm > matchData.length) {    // Yes, update
      matchData.length = nm;
      matchData.offset = wp;
    }
  }

   // 7. Set up to start at the next point in the sliding window
   // 8b. Increment wp with window wrap, set done if wrapped to start
   IncWP(matchData.length, done, wp, windowHead);
   if (done) break;                    // At start, exit outer loop
  }

  return matchData; // Contains the longest matching sequence and offset
}
```

The LZSS workspace includes the whole code with conditional #define's for running the original or TIE instructions.

The following table shows the profile comparison with all changes discussed previously in this section (XCC compiler option: `-O2 -g`):

TABLE 2. PROFILE COMPARISON

Cycle Count	DC_570T (Reference Core)	DC_570T (with TIE 15.6 kGates)	Differences
Total	55,956,903	10,800,786	5.1x
FindMatch	55,067,020	9,909,137	5.6x

As Table 2 shows, the FindMatch function is accelerated about 5.6x and the overall algorithm by over 5x.

5. Compression using the Hash Table Approach

This approach is typically used when implementing this algorithm on a processor.

This approach uses a hash table to reduce time searching the window for matching sequences. This approach requires the use of memory to store the tables, but is faster than the Hardware (brute force) approach discussed in Section 4.

This document uses and presents parts of the "LZSS" library from Michael Dipperstein that can be found at: http://michael.dipperstein.com/lzss/, available under the GNU Lesser General Public License.

The hash-based implementation of LZSS compression still requires a sliding window search as discussed previously in this document. Following is the source code for FindMatch using a hash-based approach:

```
encoded_string_t FindMatch(unsigned int windowHead, unsigned int uncodedHead) {
    encoded_string_t matchData;
    unsigned int i, j;

    matchData.length = 0;
    matchData.offset = 0;

    i = hashTable[HashKey(uncodedHead, TRUE)];
    j = 0;

    while (i != NULL_INDEX) {
        if (slidingWindow[i] == uncodedLookahead[uncodedHead]) {
            /* we matched one how many more match? */
            j = 1;
            while(slidingWindow[Wrap((i + j), WINDOW_SIZE)] ==
                uncodedLookahead[Wrap((uncodedHead + j),MAX_CODED)]) {
                if (j >= MAX_CODED) {
                    break;
                }
                j++;
            }
            if (j > matchData.length) {
                matchData.length = j;
                matchData.offset = i;
            }
        }

        if (j >= MAX_CODED) {
            matchData.length = MAX_CODED;
            break;
        }

        i = next[i];    /* try next in list */
    }

    return matchData;
}
```

As in Section 4, the same acceleration used previously applies in this approach as the most time-consuming part; the matching of bytes in the sliding window (see the highlighted source code above).

Results for this approach show a similar performance increase of 5.5x and uses up to 4 kBytes of data for the hash table and the same 16.5 kGates of TIE logic as in the previous example.

6. Conclusions

This document explains how adding 16.5kGates of logic to a 140kGate processor (a 12% increase) improves performance by over 5x when processing 4 bytes of data at a time.

Note that further improvements can be made by processing 8, 16, 32 or 64 bytes at a time – something that can be done on all Xtensa processors due to the wide memory interfaces available to the local memories. This further increases the TIE code size, but provides a proportionally higher performance increase.

7. Appendix

7.1 The LZSS Xplorer Workspace

Xtensa Xplorer is Tensilica's Eclipse-based IDE for software and TIE development. You can import existing projects into workspaces that include all of the source files and processors in order to view and further develop.

The LZSS workspace, which includes the source code used in the examples in this document, can be downloaded from within your support account or from:

http://ip.cadence.com/storage

This workspace contains:

- A buildable and runnable project with targets for running with and without TIE (ReleaseTIE and Release respectively) so the results can be compared
- The "LZSS Brute" Launch: use to set the default command line arguments
- `LZSSbrute.tie`: the file containing all of the TIE that has been included in this document
- SSD570T_LZSS processor configuration: configuration built with the TIE
- Running the project with the LZSS Brute launch will:
  - Compress "test.txt" to "test_compressed.txt"
  - Show the number of cycles taken
  - Decompress "test_compressed.txt" to "test_decompressed.txt"
  - Compare the decompressed file with the original and display pass/fail

Instructions for importing workspaces can be found in the Xtensa Xplorer Tutorial, Installing Xplorer Workspaces, accessible from the Help menu.

To run the project code after importing the workspace:

1. Select Project "LZSS_brute".
2. Select Configuration "SSD570_LZSS".
3. Select Target "Release" or "ReleaseTIE" for running without or with TIE respectively.
4. Select Run Launch "LZSS Brute".