

Project Zipline Decompression Top

Micro Architecture Specification

**Authors:**

**Microsoft Corporation**

**Broadcom Corporation**

**Revision History**

|  |  |
| --- | --- |
| Date | Description |
| 04/03/2019 | Version 1.0 |
|  |  |

**License**

Contributions to this Specification are made under the terms and conditions set forth in the Open Web Foundation Contributor License Agreement (“OWF CLA 1.0”) (“Contribution License”) by:

Microsoft Corporation

Broadcom Corporation

Usage of this Specification is governed by the terms and conditions set forth in Open Web Foundation Final Specification Agreement (“OWFa 1.0”) (“Specification License”).

**Note**: The following clarifications, which distinguish technology licensed in the Contribution License and/or Specification License from those technologies merely referenced (but not licensed), were accepted by the Incubation Committee of the OCP:

**None.**

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

[1 Overview 7](#_Toc5180722)

[2 Top-Level Block Diagram 9](#_Toc5180723)

[3 Detailed Description 10](#_Toc5180724)

[3.1 Front End Block 10](#_Toc5180725)

[3.1.1 Overview 10](#_Toc5180726)

[3.1.2 Top-Level Block Diagram 10](#_Toc5180727)

[3.1.3 Interface Signals 11](#_Toc5180728)

[3.1.4 Detailed Functional Description 16](#_Toc5180729)

[3.1.5 Errors 27](#_Toc5180730)

[3.2 Huffman Table Formatter 28](#_Toc5180731)

[3.2.1 Overview 28](#_Toc5180732)

[3.2.2 Interfaces 29](#_Toc5180733)

[3.2.3 Detailed Description 36](#_Toc5180734)

[3.2.4 Errors 47](#_Toc5180735)

[3.3 Symbol Data Decoder 47](#_Toc5180736)

[3.3.1 Overview 47](#_Toc5180737)

[3.3.2 Interfaces 49](#_Toc5180738)

[3.3.3 Detailed Description 54](#_Toc5180739)

[3.3.4 Errors 64](#_Toc5180740)

[3.4 Move to Front Replacer 66](#_Toc5180741)

[3.4.1 Overview 66](#_Toc5180742)

[3.4.2 Interfaces 66](#_Toc5180743)

[3.4.3 Detailed Description 69](#_Toc5180744)

[3.5 LZ77 Decompression 71](#_Toc5180745)

[3.5.1 Overview 71](#_Toc5180746)

[3.5.2 Interface Signals 71](#_Toc5180747)

[3.5.3 LZ77 Decompression Block Diagram 74](#_Toc5180748)

[3.5.4 Local Statistic Counters 84](#_Toc5180749)

[3.5.5 Error Tokens to Back end 85](#_Toc5180750)

[3.5.6 Debug Error Registers (Maskable Interrupts) 85](#_Toc5180751)

[3.6 Back End Block 85](#_Toc5180752)

[3.6.1 Overview 85](#_Toc5180753)

[3.6.2 Interface Signals 86](#_Toc5180754)

[3.6.3 Detailed Description 87](#_Toc5180755)

[3.6.4 Errors 88](#_Toc5180756)

[3.7 Debug Features 89](#_Toc5180757)

[3.7.1 Interface Monitors 89](#_Toc5180758)

[3.7.2 Table Monitors 90](#_Toc5180759)

[3.7.3 Huffman Decoder Bypass 92](#_Toc5180760)

[3.7.4 Performance and Statistics Events 92](#_Toc5180761)

[3.8 Table of Errors in the Decompression Block 95](#_Toc5180762)

[3.8.1 Obsolete Errors (removed from the original list) 100](#_Toc5180763)

[3.9 SRAM Estimate 101](#_Toc5180764)

[3.10 Flop Estimate 101](#_Toc5180765)

[4 Appendix 104](#_Toc5180766)

[4.1 LZ77 Decompression example 104](#_Toc5180767)

[4.2 Pointer processing Example 1 104](#_Toc5180768)

[4.3 Pointer Processing Example 2 108](#_Toc5180769)

Table of Figures

[Figure 1 Top-level block diagram 9](#_Toc3903409)

[Figure 2 Front end block diagram 11](#_Toc3903410)

[Figure 3 Frame Format To/From Decompressor 17](#_Toc3903411)

[Figure 4 Latency FIFO/Data Aligner 20](#_Toc3903412)

[Figure 5 Huffman Table Formatter High-Level Block Diagram 29](#_Toc3903413)

[Figure 6 Decoder Table Formats 37](#_Toc3903414)

[Figure 7 Table Formatter Logic Pipelining 45](#_Toc3903415)

[Figure 8 Symbol Data Decoder Top Level Block Diagram 48](#_Toc3903416)

[Figure 9 Window Feeder 56](#_Toc3903417)

[Figure 10 XP9/XP10 Decode Lane 57](#_Toc3903418)

[Figure 11 Huffman Symbol Decoder 58](#_Toc3903419)

[Figure 12 Symbol Selector 60](#_Toc3903420)

[Figure 13 Symbol Walker 62](#_Toc3903421)

[Figure 14 Symbol Packer 63](#_Toc3903422)

[Figure 15 Symbol Packer Example 63](#_Toc3903423)

[Figure 16 Example MTF update after MTF miss 69](#_Toc3903424)

[Figure 17 Example MTF update after MTF hit 70](#_Toc3903425)

[Figure 18 LZ77 Decompression Block 74](#_Toc3903426)

[Figure 19 Prefix Control and Prefix/History Buffer 76](#_Toc3903427)

[Figure 20 Lane Literal/Pointer Processing 79](#_Toc3903428)

Table of Tables

[Table 1 Huffman Table Formatter Interfaces 35](#_Toc3903399)

[Table 3 Example Huffman Alphabet Sorted by Codeword 38](#_Toc3903400)

[Table 4 Example Huffman Alphabet Sorted by Symbol 38](#_Toc3903401)

[Table 5 Base Code Table Example 39](#_Toc3903402)

[Table 6 Start Address Table Example 40](#_Toc3903403)

[Table 7 Symbol Lookup Table Example 40](#_Toc3903404)

[Table 8 Initial Parameters on a pointer decode 105](#_Toc3903405)

[Table 9 First Read from the history buffer 106](#_Toc3903406)

[Table 10 Second Read from the history buffer 106](#_Toc3903407)

[Table 11 Third Read from the history buffer 107](#_Toc3903408)

# Overview

The Project Zipline Decompression block is responsible for decompressing the compressed data. The compressed data is encapsulated in a Project Zipline frame. The Project Zipline frame has a variable length header stack up that is encoded as Type/Length/Value (TLVs) followed by the compressed frame and finally footer stack up again encoded as TLVs.

XP10 provides 63 sets of pre-defined Huffman tables that are not stored with the compressed data. This frame format also allows the use of pre-defined or user defined prefix data that is used as part of the history buffer in the LZ77 decompression. Both the pre-defined Huffman table and prefix data, if present, are inserted in the header stack up of the Project Zipline frame as TLVs by an upstream block before it is sent to the Decompression block.

The Front End Block has three sub-blocks, the Frame Header Parser, the Latency FIFO & Data Aligner, and the Block Header Parser. The Frame Header Parser is responsible for parsing the Project Zipline TLVs and forwarding the necessary information to the downstream blocks. The Frame Header Parser extracts the prefix data and sends it to the LZ77 block via a side band bus so that this data is available by the time the actual compressed data makes its way to the LZ77 block. There are two kinds of prefix data, pre-defined and user-defined. Pre-defined prefix data is always 1K in size, whereas user defined prefix can be anywhere from 1KB to 64KB, always in 1K increments. Pre-defined prefix data will be stored in dedicated 1K buffers in the LZ77 block, so the LZ77 block can continuously accept the prefix data from the Frame Header Parser. The user defined data will be stored directly in the history buffer. Since the LZ77 block has to wait for the current frame processing to complete before it can start storing the user defined prefix data, the Frame Header parser may be paused until the LZ77 block is ready to accept all the data. Pausing the Frame Header Parser will in turn result in the datapath to the Decompression block being paused, impacting performance.

The Frame Header Parser also extracts the pre-defined Huffman table from the Project Zipline frame and forwards it to the Huffman Table Formatter.

The compressed data in the Project Zipline frame is then forwarded to the Latency FIFO/Data Aligner (LFA). The LFA is responsible for sending the header data to the Block Header Parser and the actual compressed data to the Symbol Data Decoder. The Block Header Parser extracts the symbol table from the header data and forwards it to the Huffman Table Formatter.

The Huffman Table Formatter builds the Huffman decoder tables from the symbol table information that it receives from the Block Header Processor. The Huffman Table Formatter may receive pre-determined Huffman bit length sets from the Frame Header Parser (XP10 only), or compressed symbol tables from the Block Header Parser (XP9/XP10, or DEFLATE).

The Symbol Data Decoder uses the Huffman tables built by the Huffman Table Formatter to decode the Huffman encoded LZ77 symbols in the compressed data stream. The LFA block sends the compressed data stream to the SDD. The SDD performs a parallel decode on the incoming bit stream and fully resolves multiple symbols per cycle. For XP9/XP10 frame formats, the LFA knows the exact number of compressed data bits to send to the SDD. So, if the LFA has multiple frames in the FIFO, it can always send the header data to the Huffman Table Formatter to start building the Huffman decoder tables for the next frame as the current frame’s data is being decoded by the SDD. For Deflate frames, since the end of the block information is encoded in the compressed data stream, LFA has to wait for acknowledgement from SDD after it has decoded the EOB symbol, before it can send the next block’s header data to the Huffman Table Formatter.

The Move to Front Replacer (MTF) in the decompressor operates on XP9/XP10 frames from the SDD, replacing MTF back-references with PTR back-references, using a cache the 4 most recently used LZ77 back-reference offsets. When a back-reference (PTR or MTF) is encountered, the associated offset is moved to the front of the cache, shifting the other entries down.

The LZ77 block receives literals and pointers from the MTF block. As pointers and literals are processed, the decompressed data is sent to both the Back End block and written to a history buffer. Although the output to the back end from the LZ77 is 8 bytes per clock (50Gbps), 16 bytes of data is accumulated and written back to the history buffer, to speed up reads from the history buffer on future pointer decodes.

Literals don’t need any processing. They are sent as-is to the output and history buffer.

Pointers are in the form of a {offset, length} pair. The offset indicates the starting position of the data to copy from the history buffer with respect to the current decompressed position. The length indicates the number of bytes to copy from the offset.

If a frame has prefix data included, the prefix data is treated as part of the history buffer.

The Decompressor Back End block combines the decompressed data from the LZ77 block with the header and footer data from the Front End block. It also checks the CRC32/64 or Adler32 of the frame as the case maybe.

For XP9 and GZIP frames, it checks if the size of the original uncompressed data matches the size of the decompressed data from the LZ77 block.

The decompressor supports minimum match lengths of 3 and 4, for both XP9 and XP10 frames.

# Top-Level Block Diagram



Figure 1 Top-level block diagram

# Detailed Description

## Front End Block

### Overview

The Front End block is the first block in the Decompression pipeline. This block has three main components, the Frame Header Parser block, the Block Header Parser block and the Latency FIFO/Data Aligner block.

* **Frame Header Parser (FHP)**

This block is responsible for parsing the Project Zipline frame TLVs. It extracts the relevant information from the frame header fields and forwards them to the appropriate downstream blocks. The prefix data is extracted and sent to the LZ77 block. If the pre-determined Huffman data is present in the frame, it is sent to the Huffman Table Formatter. The actual frame data is forwarded to the Latency FIFO/Data Aligner block.

* **Latency FIFO/Data Aligner (LFA)**

This block sends the header/symbol table data to the Block Header Parser (BHP) and the compressed data to the Symbol Data Decoder (SDD). The Latency FIFO will store the compressed data until the symbol tables are decoded by the Huffman Table Formatter (HTF). Since the frame/block header, symbol table and compressed data are all packed together, the Data Aligner block keeps track of the symbol table boundary and the End of block boundary to align the header and compressed data to 64 bit boundaries to send to the BHP and the SDD.

* **Block Header Parser (BHP)**

This block parses the compression format specific header to forward the symbol tables to the Huffman Table Formatter (HTF). It also extracts the MTF header offsets from XP9/XP10 blocks and forwards them to the MTF block. After the symbol tables are decoded in the Huffman Table Formatter, the HTF acknowledges with the total bits used for the symbol table decode from the incoming data stream. The BHP sends this information to the Latency FIFO/Data Aligner along with the total compressed bits (XP9/XP10 only) to send to the Symbol Data Decoder.

### Top-Level Block Diagram

Below is a top level block diagram for the Front End block.



Figure 2 Front end block diagram

### Interface Signals

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **Input Interface** | | | |
| xp10\_decomp\_ib\_in.tid | I | 1 | Transaction ID. |
| xp10\_decomp\_ib\_in.tvalid | I | 1 | Data valid |
| xp10\_decomp\_ib\_in.tlast | I | 1 | Set on the last beat of the Completion message. |
| xp10\_decomp\_ib\_in.tdata | I | 64 | 64 bit datain |
| xp10\_decomp\_ib\_in.tstrb | I | 8 | Data bytes valid on the bus  0: Bits [7:0]  1: Bits [15:8]  …  7: Bits [63:56] |
| xp10\_decomp\_ib\_in.tuser | I | 8 | 8 bits defined; undefined bits are reserved |
| xp10\_decomp\_ib\_out.tready | I | 1 | Decompressor ready to accept data |
| **Block Header Parser <-> Huffman Table Formatter** | | | |
| bhp\_htf\_hdr\_dp\_valid | O | 1 | Indicates if the data input is valid |
| htf\_bhp\_hdr\_dp\_ready | I | 1 | Indicates if the HTF is ready to receive symbol table data or bit length |
| bhp\_htf\_hdr\_dp\_bus.data | O | 64 | Contains block-level symbol table. Data is direct from the Decompressor DTB. |
| bhp\_htf\_hdr\_dp\_bus.last | O | 1 | Indicates the end of a symbol table. |
| bhp\_htf\_hdr\_dp\_bus.trace\_bit | O | 1 | Indicates if the statistics should be counted for this frame. |
| bhp\_htf\_hdrinfo\_valid | O | 1 | Indicates if the type indication input is valid. |
| htf\_bhp\_hdrinfo\_ready | I | 1 | Indicates if the HTF is ready to receive type indication. |
| bhp\_htf\_hdrinfo\_bus.fmt | O | 3 | Type of the data on the symbol table data  bus:  0 = XP9 block  1 = XP10 block  2 = DEFLATE block dynamic  3\* = DEFLATE block fixed  4\* = RAW block  3 and 4 will have no associated data on the symbol table data bus |
| bhp\_htf\_hdrinfo\_bus.sub\_fmt | O | 2 | 0= GZIP  1= CHU4k/ZLIB  2= CHU8k |
| bhp\_htf\_hdrinfo\_bus.winsize | O | 3 | The number of bit lengths encoded in the symbol table is variable for XP10 and depends on the window size. Window size less than 4 is only support for XP10. |
| bhp\_htf\_hdrinfo\_bus.min\_ptr\_len | O | 1 | Minimum match length for XP9/XP10 PTR:  0 = 3  1 = 4 |
| bhp\_htf\_hdrinfo\_bus.min\_mtf\_len | O | 1 | Minimum match length for XP9 MTF:  0 = 2  1 = 3 |
| bhp\_htf\_hdrinfo\_bus.phd\_present | O | 1 | Indicates if the pre-determined Huffman table was loaded as part of this frame. |
| bhp\_htf\_hdrinfo\_bus.trace\_bit | O | 1 | Trace bit indicator |
| bhp\_htf\_hdrinfo\_bus.sof\_blk | O | 1 | Indicates if this is the first block of the frame. |
| bhp\_htf\_hdrinfo\_bus.hdr\_bits\_in | O | 23 | Total number of frame/block header bits consumed for this block by the BHP. |
| bhp\_htf\_hdrinfo\_bus.rqe\_sched\_handle | O | 16 | Scheduler handle information passed from RQE tlv to the scheduler block |
| bhp\_htf\_hdrinfo\_bus.tlv\_frame\_num | O | 11 | Frame number from the DATA TLV |
| bhp\_htf\_hdrinfo\_bus.tlv\_eng\_id | O | 4 | Engine ID from the DATA TLV |
| bhp\_htf\_hdrinfo\_bus.tlv\_seq\_num | O | 8 | TLV Sequence number from the DATA TLV. |
| htf\_bhp\_status\_valid | I | 1 | Indicates the symbol table size information is valid. This message is only sent in response to types 1, 2, and 3 (see bhp\_htf\_hdrinfo\_fmt above). |
| htf\_bhp\_status\_bus.size | I | 13 | Size of the received symbol table in bits.Includes both short and long tables for  XP9/XP10. |
| bhp\_htf\_status\_ready | O | 1 | Block ready to accept data |
| htf\_bhp\_status\_bus.error | I | 8 | Error status of symbol table processing: Please see Project Zipline error codes for the error information. |
| **Latency FIFO/Data Aligner <-> Symbol Data Decoder** | | | |
| lfa\_sdd\_dp\_valid | O | 1 | Compressed data to Symbol Data Decoder is valid |
| lfa\_sdd\_dp\_bus.data | O | 64 | Compressed data to Symbol Data Decoder |
| lfa\_sdd\_dp\_bus.numbits | O | 7 | Number of bits valid in the lfa\_sdd\_data.  For XP9/XP10, this indicates the number of valid bits in the last EOB word.  For Deflate, this is meaningful only for the last word of the last block. From the Data frame size TLV, we know the size of the last word of the last block. |
| lfa\_sdd\_dp\_bus.sob | O | 1 | Start of Block Indication. |
| lfa\_sdd\_dp\_bus.eob | O | 1 | End of Block Indication.  This is always sent for XP9 and XP10.  For Deflate, LFA sends the EOB indicator when it has reached the end of data for the frame, which is the last word of the last block. |
| lfa\_sdd\_dp\_bus.eof | O | 1 | End of Frame Indication.  For XP9/XP10, this is sent with the EOB of the last block.  For Deflate, this is sent after SDD has sent the EOB of the block after data decode. |
| lfa\_sdd\_dp\_bus.error | O | 8 | Error code is qualified by EOF. Non-zero value means that an error has been detected. |
| lfa\_sdd\_dp\_bus.trace\_bit | O | 1 | Trace bit indicator |
| lfa\_sdd\_dp\_bus.frame\_bytes\_in | O | 28 | Total number of bytes received in this frame. |
| lfa\_sdd\_dp\_bus.last\_frame | O | 1 | Last frame indicator for compound commands |
| sdd\_lfa\_dp\_ready | I | 1 | Symbol Data Decoder is ready to receive data |
| sdd\_lfa\_ack\_valid | I | 1 | If 1, indicates valid acknowledge message. |
| sdd\_lfa\_ack\_bus.eob | I | 1 | End of Block Indication from SDD . For XP9/XP10, indicates that final symbol in block was decoded. For Deflate, indicates that EOB symbol was decoded. |
| sdd\_lfa\_ack\_bus.err | I | 1 | Symbol Data Decoder found an error in the current frame. |
| sdd\_lfa\_ack\_bus.numbits | I | 7 | Number of bits consumed by the Symbol Data Decoder. This is valid when sdd\_lfa\_ack\_valid is asserted. LFA uses this to advance the tail pointer. |
| **Latency FIFO/Data Aligner -> Back End Block** | | | |
| lfa\_be\_crc\_valid | O | 1 | CRC/Adler valid |
| lfa\_be\_crc\_bus.crc\_frm\_fmt | O | 3 | 3’b000 : XP9  3’b001 : XP10\_CRC32  3’b010 : XP10\_CRC64  3’b011 : ZLIB  3’b100 : GZIP  3’b101 : Deflate  3’b110 : CHU with CRC  3’b111 : CHU without CRC |
| lfa\_be\_crc\_bus.isize | O | 32 | For formats that indicate the input size, this will have the input size extracted from the frame. |
| lfa\_be\_crc\_bus.data | O | 64 | The frame format in the footer is used to determine if the data has the Adler or CRC.  [63:0] if CRC64. CRC32/CRC64 for XP10 frame indication is in the frame header.  [31:0] if CRC32  [31:0] if Adler32 |
| **Pre-Determined Huffman to HTF** | | | |
| fhp\_htf\_bl\_valid | O | 1 | Indicates if the data input is valid. |
| htf\_fhp\_bl\_ready | I | 1 | Indicates if the HTF is ready to receive bit  length data. |
| fhp\_htf\_bl\_bus.data | O | 64 | Contains pre-determined Huffman bit lengths (from an XP10 TLV). Data is direct from the Decompressor DTB |
| fhp\_htf\_bl\_bus.last | O | 1 | Indicates the end of a bit length set. |
| fhp\_htf\_bl\_bus.trace\_bit | O | 1 | Trace bit Indicator. |
| **Frame Parser to LZ77** | | | |
| fhp\_lz\_prefix\_hdr\_valid | O | 1 | Valid Indicator |
| fhp\_lz\_prefix\_hdr\_bus.data\_sz | O | 7 | Size in 1K chunks |
| fhp\_lz\_prefix\_hdr\_bus.prefix\_type | O | 1 | 0: pre-defined  1: user defined |
| fhp\_lz\_prefix\_hdr\_bus.trace\_bit | O | 1 | Trace bit indicator |
| lz\_fhp\_prefix\_hdr\_ready | I | 1 | LZ77 block is ready to accept prefix information on SOF. If all three the 1K prefix buffers are in use, ready will be de-asserted until at least one of them frees up. |
| **Frame Parser to LZ77 (Prefix Data)** | | | |
| fhp\_lz\_prefix\_valid | O | 1 | Frame Parser to LZ77 prefix data valid |
| fhp\_lz\_prefix\_dp\_bus.data | O | 64 | Frame Parser to LZ77 data |
| fhp\_lz\_prefix\_dp\_bus.sof | O | 1 | Start of prefix data. |
| fhp\_lz\_prefix\_dp\_bus.last | O | 1 | Last beat of prefix data. |
| fhp\_lz\_prefix\_dp\_bus.prefix\_type | O | 1 | 0: pre-defined  1: user defined |
| lz\_fhp\_pre\_prefix\_ready | I | 1 | LZ77 is ready to accept predetermined prefix data |
| lz\_fhp\_usr\_prefix\_ready | I | 1 | LZ77 is ready to accept user prefix data. |
| **Block Header Parser <-> MTF** | | | |
| bhp\_mtf\_hdr\_bus.exp0 | O | 5 | MTF Exponent0 |
| bhp\_mtf\_hdr\_bus.offset0 | O | 16 | MTF Header Offset0 |
| bhp\_mtf\_hdr\_bus.exp1 | O | 5 | MTF Exponent1 |
| bhp\_mtf\_hdr\_bus.offset1 | O | 16 | MTF Header Offset1 |
| bhp\_mtf\_hdr\_bus.exp2 | O | 5 | MTF Exponent2 |
| bhp\_mtf\_hdr\_bus.offset2 | O | 16 | MTF Header Offset2 |
| bhp\_mtf\_hdr\_bus.exp3 | O | 5 | MTF Exponent3 |
| bhp\_mtf\_hdr\_bus.offset3 | O | 16 | MTF Header Offset 3 |
| bhp\_mtf\_hdr\_bus.ptr\_last | O | 1 | This bit is set if the previous match is PTR or MTF. This is used only in Xpress9. |
| bhp\_mtf\_hdr\_bus.format | O | 1 | 0 = Xpress9  1 = Xpress10 |
| bhp\_mtf\_hdr\_bus.present | O | 1 | Indicates that the MTF cache should be updated with the new header info. Otherwise, retain the current cache state. |
| bhp\_mtf\_hdr\_bus.trace\_bit | O | 1 | Trace bit indicatior. |
| bhp\_mtf\_valid | O | 1 | Indicates MTF header info is valid. |
| mtf\_bhp\_ready | I | 1 | Handshake data receive ready (active high). |
| **Frame Header Parser to LZ77 Debug Path** | | | |
| fhp\_lz\_dbg\_data\_valid | 1 | O | Debug data path from FHP to LZ is valid |
| fhp\_lz\_dbg\_data\_bus.data | 64 | O | 64 bit data. |
| lz\_fhp\_dbg\_dp\_ready | 1 | I | LZ is ready to accept debug data |

### Detailed Functional Description

#### Frame Header Parser Block

The frame header parser block parses the Project Zipline TLVs. The Pre-determined Huffman, Prefix data, LZ77-SYM and Data TLVs are consumed in this block.

The RQE, CMD, FRMD, STATS, CQE and any other TLV is passed through to the Back End block. The status field from the FRMD TLV is extracted and sent to the LFA. This field indicates if the data frame is compressed or raw and the compression format used if it is compressed.

For a detailed description of various TLVs and the fields in them, please refer to 2.

The pre-determined Huffman data is extracted and forwarded to the Huffman Table Formatter block. The prefix data is extracted and forwarded to the LZ77 block.



Figure 3 Frame Format To/From Decompressor

##### Predetermined Huffman TLV

If the Project Zipline frame indicates the presence of the Pre-determined Huffman header, the frame header parser extracts the header table and sends it to the Huffman Table Formatter. The CRC64 at the end of the table data is checked.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pre-determined Huffman** | | | |
| **Field** | **Size** | **Word** | **Notes** |
| TLV Type | 8 | 0 | See Project Zipline TLV Type Reference Table  Type = 3 |
| TLV Length | 24 | 0 | Length = 536 |
| TLV CRC | 4 | 0 | Project Zipline TLV CRC-4 |
| TLV Engine ID | 8 | 0 | 0-3 CCEIP, 8-11 CDDIP |
| TLV Sequence Num | 8 | 0 | Increments per RQE/Command |
| Short Symbol Lengths | 2880 | 1-46 | 576 Short Symbol Lengths |
| Long Symbol Lengths | 1020 | 47-65 | 204 Long Symbol Lengths |
| CRC-64 | 64 | 66 | CRC-32 covering Predetermined Huffman data (includes 4 bits of upper nibble pad). |

For more information on the pre-determined Huffman data, please refer to section 3.2.

##### Prefix Data TLV

If the Project Zipline header indicates the presence of prefix data, the frame header parser extracts the prefix data and sends it to the LZ77 block through a side band bus. In the case of pre-defined prefix, the LZ77 block will be able to accept the prefix data continuously as it has two dedicated 1K buffers in the event that one is being used by the current frame. The prefix data will be pre-loaded into the history buffer in the case of user-defined prefix data. If the LZ77 block is in the middle of processing a frame, upto 1K of the user-defined data will be pre-loaded into the 1K prefix buffer and the frame header parser will be paused until the current frame processing is over before the rest of the data can be accepted. Note that the CRC is part of the prefix data. The CRC will be checked, but will also be forwarded to the LZ77 block.

The CRC64 is checked for the prefix data in the Frame Parser block.

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Size** | **Word** | **Notes** |
| TLV Type | 8 | 0 | See Project Zipline TLV Type Reference Table  Type = 4 |
| TLV Length | 24 | 0 | Length = 8 + Length of Prefix data in bytes |
| TLV CRC | 4 | 0 | Project Zipline TLV CRC-4 |
| TLV Engine ID | 8 | 0 | 0-3 CCEIP, 8-11 CDDIP |
| TLV Sequence Num | 8 | 0 | Increments per RQE/Command |
| Prefix Data | 1KB-64KB | 128-8192 | Supports 1KB-64KB prefixes in 1KB increments.   1. The last 8 bytes of the prefix data is a CRC-64 using the XP10 polynomial |

##### DATA TLV

The Data TLV is consumed by the FHP block. The frame data that is extracted is forwarded to the Latency FIFO/Data Aligner block. (LFA).

##### LZ-SYM TLV

There is a special debug mode TLV in which the LZ77 block’s input data is inserted as a TLV in the Project Zipline frame.

The frame header parser after sending the Project Zipline header to the backend will send the data in the LZ-SYM TLV to the LZ77 block. The frame footer is sent to the back end after the debug data is sent.

##### Frame Footer TLVs

This includes the FOOTER, STATS, CQE TLVs. All these TLVs are passed through asis to the Back End block.

#### Latency FIFO/Data Aligner

The Latency FIFO performs the following functions:

* Latency compensation for the header processing.
* Buffers up input data to feed into the Symbol Data Decoder (SDD) while the block header is being interpreted by the Huffman Table Formatter (HTF).
* Aligns the Symbol table data for the Block Header Parser to send to the HTF.
* Filters out header data (header + symbol table) from the block data to send to the SDD.
* Aligns the compressed data to the full input width (64 bits) of the downstream blocks.
* Provides replay functionality for cases when too much data that was not header data was sent to the Block Header Parser or when header data was sent to the Symbol Data Decoder.
* Tracks frame boundary locations within the FIFO by maintaining up-to 4 SOF pointers in a separate FIFO. If the SOF FIFO is full, the FHP block is back pressured thereby, pausing the data transaction interface.



Figure 4 Latency FIFO/Data Aligner

* The Latency FIFO is sized to hold 8k Bytes of data, which can be fine-tuned based on performance test cases.
* The Data Aligner is capable of repacking data for the output bus on arbitrary boundaries on a 64-bit granularity.

##### Dataflow

The Latency FIFO accepts data from the input data transaction bus and stores the data in its 8K FIFO. The contents of the data with respect to block boundaries may not be known as data enters the FIFO and it may contain data corresponding to multiple blocks within the same frame at any time.

The control flow for this block is as follows:

* Read the Header Info from FHP.
* Accept data on the input from the FHP if available.
* Send the header info and data from the Latency FIFO to the Block Header Parser until it de-asserts ready. After the MTF offsets are extracted by the BHP (or ZLIB, GZIP, Deflate headers are parsed), it will acknowledge the total header bits consumed so far to the LFA. LFA will then realign the rest of the header data which is the symbol table and forward to the BHP until ready is de-asserted.
* The BHP block, based on feedback from the Huffman Table Formatter will acknowledge the total bits used from the data it received for the symbol tables and the total bits of compressed data to send to the SDD.
* The LFA based on the bits consumed by the HTF, re-aligns the compressed data to send 64-bit aligned data words to the SDD. If some of the compressed data bits were incorrectly sent to the BHP, they need to be resent to the SDD instead.
* The SDD for every 64 bits of data it receives from the LFA, will acknowledge the number of bits consumed. The SDD can acknowledge up-to a maximum of 115 bits (31 bits taking up literals and one long symbol of the maximum size 84 bits) in one clock.
* For XP9 and XP10, the LFA knows exactly how many compressed bits to send to the SDD. The number of bits in the last EOB word will be set by the LFA. EOB will be set for the last word of the block and EOF will also be set if it is the last block of the frame. If there are any errors for the current frame, the ERR signal will also be asserted along with the EOF.
  + XP9 and XP10 have the total compressed bits in the block header that is used to calculate the number of bits to send to the SDD.
* For Deflate, LFA will send data to SDD until it receives an EOB/ERR ack back. The last block is treated differently than the rest of the blocks in the frame. If LFA gets to the end of the last block, it will send all the bits it has and assert EOB, indicating to the SDD that it has no more data to send. SDD will then decode the EOB and acknowledge back with either an EOB or ERR. LFA will then send the EOF to SDD. ERR will also be asserted if there any errors for the frame.
  + With the EOB and the bits consumed information from SDD, LFA can re-align the header data for the next block. In this case, if the next block’s header data was incorrectly sent to the SDD, it needs to be re-played into the BHP.
* The Latency FIFO will also be aware of the end of block constraints for the various formats and adjust accordingly (if requires) to find the start of the next block. For instance, if the block format has ends padded to byte boundaries (like XP9), then the replay FIFO needs to round the end of block location it receives up to the nearest byte boundary.
* For all frame formats, XP9, XP10 and Deflate, SDD always sends the EOF ack indicator. So, if LFA has already started processing the next frame after sending an EOF, it will apply any error indicators from SDD that are sent before the EOF ack to the previous frame and any that occur after the EOF ack to the current frame.

##### Pointer Management to help in re-alignment and replay of data

For each block that is being processed there are two pairs of pointers, one for the header data and the other for the compressed data. Each pair has a head and a tail pointer.

On the start of transmission of a frame, the header data’s head and the tail pointers will point to the first FIFO location for the frame, offset 0. As 64 bit words are sent to the BHP, the tail pointer will be incremented. When the BHP first acknowledges the bits consumed, the head and the tail pointers will be moved to the location/offset that points to the original head pointer + bits consumed.

Now the LFA sends the rest of the header bits (symbol table) to BHP. The head pointer will remain at the same location, but the tail pointer will move as 64 bit words are forwarded. After the receipt of header bits consumed by the HTF, the head tail/pointers will point to the previous head + bits consumed. The data head/tail pointer will be set to this location and the header data’s head/tail pointers are available to be used by the next block if the data end is known for the current block.

As the compressed data is forwarded to the SDD, the data tail pointer will increment by bits sent. On the receipt of bits consumed acknowledgement from SDD, the data head pointer will move by bits acknowledged. Bits acknowledged by the SDD or the HTF will never have to be replayed.

Data between the head/tail pointers will not be available to be over-written at any time. As a result, that data can be replayed to either the BHP or the SDD.

##### Header and Data boundaries for different frame/block formats

Based on the format of the frame/block, the header processing of the second block/frame can be done while the data of the first frame is being decompressed.



Here are two examples that illustrate the operation of the LFA block when the end of data is known and unknown.

**Known Data End**

In this case, the SDD is still processing a data block while the symbol table for the next block is being decoded by the HTF.

1. The header data is initially sent from the Latency FIFO to the Block Header Processor. The bandwidth of the input data to BHP is greater than the input bandwidth of the header processing in the HTF. Since there is some buffering internal to the HTF, it is possible that more than just the header data was sent to the HTF. It could be data for the current block or some of the Header Data for the next block (in the case of a 0 sized data payload).
2. When the Latency FIFO receives the symbol table bits consumed, the data’s tail and head pointers need to be adjusted to the start of the data payload – the pointer may need to be adjusted either forward or backward in time (relative to what has been sent into the BHP)
3. Once the pointers are adjusted and the output data word aligner configured, data is streamed to the SDD (if there is data).
4. Once the location of the start of the next header (after the end of the current data block) is in the FIFO, the header’s head/tail pointers may jump to that point and send data to the BHP and then to the HTF for the next block.
5. Since the SDD internally only operates on 32 bits at a time, it will not assert ready continuously. The Latency FIFO is read during those free cycles to send data to the BHP for the next block if it is ready to send the next block’s header data.

**Unknown Data End**

In this case, the end of the data is unknown until the last symbol in the data block is fully resolved. This means that the header processing for the next block cannot begin until the current block has been fully decoded.

1-3. This is same as the Known End case.

1. In this case, the start of the next header is unknown, so the Latency FIFO needs to wait on the feedback interface from the SDD for the bit location of the EOB.
2. Once the EOB location is received, the header’s head/tail pointer will be moved to this location and the header data for the next block is streamed to the BHP.

#### Block Header Parser

The Block Header Parser does the block/frame header parsing in two steps.

* Parse the frame/block header and extract the relevant information to send to the HTF. Based on the format of the frame/block, it first requests n-data words that are required to parse until the MTF offset headers (or the ZLIB/GZIP/Deflate/CHU headers). Since the MTF offset header is a variable header, BHP might request more than needed. The number of bits consumed until the MTF header offsets is sent to the LFA so that the rest of the header data can be re-aligned and forwarded.
* The second phase of the header parsing will be to forward the symbol tables to the HTF. Since the format of the block is known, the BHP will request worst case symbol table (except for XP9 where it knows the exact count) bits from the LFA. BHP will forward the aligned data to the HTF and wait for an acknowledgement from the HTF that that symbol decode is complete with the exact count of bits consumed. This is then used to calculate the compressed data bits to be forwarded to the SDD (from the Output bits in the XP9/XP10/CHU headers). The BHP forwards both the symbol table consumed bits and the data bits information to the LFA.

##### Compression format specific frame parsing

**XP9 Block header**

* The XP9 format identifier is checked.
* Extract the input size to send to the back end block to check if this matches the decompressed data from LZ77.
* Extract the output size. This includes the 4 word (4, 8 byte) data header, MTF entries, symbol table and compressed data. This is used to send the exact number of bits to the Huffman Table Formatter and the Symbol Data Decoder.
* Check the block index to make sure the sequence is correct.
* Check the crc32 of the header data.
* Extract the MTF header offsets.
* Send the bits consumed ack to LFA.
* Request re-aligned rest of the header which now has the symbol tables and send that to HTF.
* Wait for the bits consumed ack from HTF and forward that to LFA. Also, calculate the compressed data bits to forward to the SDD and send that information as well to LFA.
* LFA sends compressed data to SDD.
* LFA asserts EOB for end of block and EOF for end of frame and ERR if any errors are present.

**XP10 Frame + block headers**

* The XP10 format identifier is checked.
* The first 32 bits are ignored. This is the same as what’s in the Project Zipline header about LZ77 prefix and pre-determined Huffman.
* Extract the output bits from the Block header. Includes the header/data except padding bits.
* Extract compressed/uncompressed bit to send to HTF.
* Extract last block information to set EOF.
* Extract the MTF header offsets if present.
* Send the bits consumed ack to LFA along with last block information.
* Request re-aligned rest of the header which now has the symbol tables if the data is compressed. This is sent to the HTF.
* Wait for the bits consumed ack from HTF and forward that to LFA. Also, calculate the compressed data bits to forward to the SDD and send that information as well to LFA.
* LFA sends the compressed data to SDD.
* LFA asserts EOB for end of block and EOF for end of frame and ERR if any errors are present.

**Deflate**

* Extracts EOF from BFINAL.
* Extracts Compression format from BTYPE and sets an error if BTYPE is 1.
* For uncompressed blocks, sends the LEN information to the LFA block.
* Send the bits consumed ack to LFA along with last block information.
* Request re-aligned rest of the header which now has the symbol tables.
* Wait for the bits consumed ack from HTF and forward that to LFA.
* LFA sends re-aligned compressed data to SDD until EOB is asserted from SDD.
* If it is the last block and LFA has reached the end of the frame data, EOB is asserted to SDD indicating there is no more data to send. LFA asserts EOF after it receives EOB/ERR ack from SDD.

**ZLIB**

* Compression Info is extracted from the frame. Only compression method of 8 for a Window size of 32K is supported.
* The rest of the steps are the same as the Deflate case.
* However, it has to wait for the EOB ack with the number of bits consumed to correctly extract the Adler32. Send the Adler32 to the backend block.

**GZIP**

* Process the header fields. Extract ID1/ID2 and Compression Info.
* The other fields are examined so that we can get to the start of the symbol table.
* CRC16 checked if it is present.
* The rest of the steps are the same as the Deflate case.
* However, it has to wait for the EOB ack with the number of bits consumed to correctly extract the CRC32 and the input size. Send the CRC32 and input size of the uncompressed data to the backend block.

**CHU Mode**

* The Frame Header Parser on parsing the FRMD TLV will be able to determine the following –
  + Size of the uncompressed data (4K or 8K)
  + If the frame is compressed, the CHU block header that has the compressed size in bits and prefix information.
* If the frame is compressed, based on the data size, the worst symbol table bits will be forwarded to the Huffman Table Formatter.
* The compressed data will be forwarded to SDD.
* If the frame is uncompressed, the raw bytes will be sent to the SDD along with indication to the HTF that this is a raw frame.
* The CHU frames do not have a CRC appended at the end. So, a value of 0 will be sent to the backend.

##### MTF Header

Some formats like the XP9 and XP10 have the MTF offset header after the block header.

For XP10, the MTF\_HDR\_PRESENT bit in the header will indicate the presence of the MTF header.

For XP9, the MTF header is always present.

The minimum length of the MTF offset header is 21. The LSB bit 0 (for XP9 only) indicates if the last match is a pointer or MTF. The next 5 bits are the MTF EXP0. This is followed by MTF LSB0. The size of the MTF LSB0 is dependent on the MTF EXP0 value. The MTF EXP1/2/3 and MTF LSB1/2/3 are processed the same way as MTF EXP0/LSB0.

The MTF EXP0/1/2/3 and OFFSET0/1/2/3 are forwarded to the MTF block.

### Errors

Errors in the Front End block will be handled in the following way –

* Any errors in the Frame Header Parser, Block Header Parser or Symbol table decode (HTF) will result in the rest of the frame being terminated. An SOB, EOB, EOF, ERR will be asserted to the SDD block to indicate an empty terminated frame. The header head/tail pointers and the data head/tail pointers will advance to offset that point to the next SOF.
* Any error in the Data decode part (SDD or LFA) will result in the rest of the frame terminated with an EOB, EOF and ERR. The data head/tail pointers will advance to the offset that point to the next SOF.

The “Table of Errors in the Decompression Block” section 3.8 has a list of Errors in the Front End Parser block.

## Huffman Table Formatter

### Overview

This block builds the Huffman decoder tables from the symbol table information that it received from the Block Header Processor. These Huffman decoder tables are then used by the Symbol Data Decoder block to decode the Huffman-encoded LZ77 symbols in the compressed data stream.

The Huffman Table Formatter may receive pre-determined Huffman bit length sets from the Frame Header Parser (XP10 only), or compressed symbol tables from the Block Header Parser (XP9/XP10, or DEFLATE). A pre-determined Huffman bit length set may be used for multiple XP10 blocks over the course of an entire XP10 frame and is not de-allocated until after the last block in a frame no longer needs it.

For each block, the Table Formatter either reads from the buffered compressed symbol table (received from the Block Header Parser), or it reads out the buffered pre-determined Huffman bit length set, and then builds the formatted decoder tables.

XP9/XP10 simple encoding, DEFLATE fixed Huffman coding, or RAW literal/byte mode, are special cases where “pre-canned” values are loaded into the Huffman decoder tables used by the Symbol Data Decoder.



Figure 5 Huffman Table Formatter High-Level Block Diagram

After beginning to consume an encoded symbol table for Block *N*, the Table Formatter Logic must be ready to begin consuming the encoded symbol table for Block *N+1*no more than 512 clock cycles later. This is necessary to keep up with the throughput requirement of 25Gbps for 2KB blocks (each block can use a different Huffman code). Accordingly, the Symbol Data Decoder will store 2 copies of the formatted decoder tables: one is the “working” copy for the Table Formatter Logic (i.e. actively being populated), and the other is the copy currently being used by the Symbol Data Decoder parallel Huffman decoder pipeline. When a new block starts, the “working” copy is used by the Symbol Data Decoder, and the previously used copy because the new “working” copy (i.e. the one written to by the Huffman Table Formatter).

### Interfaces

The Huffman Table Formatter has the following interfaces:

* Frame Header Parser Interface
  + Pre-determined Huffman Bit Length Data:  
    When an XP10 frame contains a pre-determined Huffman bit length TLV, the data is received on this interface.
* Block Header Parser Interface
  + Symbol Table Data:   
    This is a 64-bit wide bus receives an XP9, XP10, or DEFLATE symbol table header from a compressed data block.
  + Header Info:   
    This receives messages which describe the format of the next block: XP9, XP10, DEFLATE fixed, DEFLATE dynamic, or RAW. For DEFLATE. If DEFLATE fixed or RAW is indicated, there will be no associated data on the Symbol Table Data bus. The message includes the window size, which the Huffman Table Formatter needs for XP10 in order to know how many bit lengths are in the symbol table. For XP9, the bit length data is checked to be consistent with the window size. Additionally, when an XP10 frame contains a pre-determined Huffman bit length TLV, a signal on this interface indicates that associated bit length set can be de-allocated after processing the symbol table of the associated block.
  + Symbol Table Processing Status:   
    In response to every header info message, an indication of the total size of the associated symbol table is sent back to the Block Header Processor, once enough data has been processed to determine its size. This is used by the Header Processor to identify where the beginning of the compressed data is in the block. This interface also indicates if errors have occurred during processing of the symbol table.
* Symbol Data Decoder Interface
  + BCT and SAT write interface:   
    This interface is used to transmit BCT and SAT entries to the Symbol Data Decoder for both symbol alphabets. A complete set of tables is transmitted as a sequence of single-cycle transfers. No addresses are needed because the entries are transmitted in order, starting with the entries corresponding to a bit length of 1. A ***last*** signal is asserted when the entries associated with the longest Huffman codeword are transmitted (this may be less than the maximum possible codeword size for the Huffman code).
  + SLT write interfaces:   
    These interfaces are used to transmit SLT entries to the Symbol Data Decoder. Each of the following interfaces includes address and data:
    - Short SLT even entry write interface
    - Short SLT odd entry write interface
    - Long SLT even entry write interface
    - Long SLT odd entry write interface
  + Table completion interface:  
    This interface consists of a ***done*** signal to the Symbol Data Decoder and a ***credit*** signal from the Symbol Data Decoder. Out of reset, the Huffman Table Formatter has “credit” to write 2 complete decoder table sets to the Symbol Data Decoder. It asserts the ***done*** bit and decrements its credit when the last write for a decoder table set has been completed. If there are no credits, a new table set cannot be written. When the Symbol Data Decoder is finished using a decoder table set, it asserts the ***credit*** bit, and the Huffman Table Formatter increments its credit. The compression format associated with the decoder tables (i.e. either XP9/XP10 or DEFLATE) is indicated coincident with the ***done*** signal.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **Input Interface** | | | |
| fhp\_htf\_bl\_valid | I | 1 | Indicates if the data input is valid |
| htf\_fhp\_bl\_ready | O | 1 | Indicates if the HTF is ready to receive bit length data. |
| fhp\_htf\_bl\_bus.data | I | 64 | Contains pre-determined Huffman bit lengths (from an XP10 TLV). Data is direct from the Decompressor DTB. |
| fhp\_htf\_bl\_bus.last | I | 1 | Indicates the end of a bit length set. |
| fhp\_htf\_bl\_bus.trace\_bit | I | 1 | Trace bit indicator. |
| **Block Header Parser to Huffman Table Formatter Header Data Interface** | | | |
| bhp\_htf\_hdr\_dp\_valid | I | 1 | Indicates if the data input is valid. |
| htf\_bhp\_hdr\_dp\_ready | O | 1 | Indicates if the HTF is ready to receive symbol table data. |
| bhp\_htf\_hdr\_dp\_bus.data | I | 64 | Contains block-level symbol table. Data is direct from the Decompressor DTB. |
| bhp\_htf\_hdr\_dp\_bus.last | I | 1 | Indicates the end of a symbol table. |
| bhp\_htf\_hdr\_dp\_bus.trace\_bit | I | 1 | Indicates if the statistics should be counted for this frame. |
| **Block Header Parser to Huffman Table Formatter Header Info Interface** | | | |
| bhp\_htf\_hdrinfo\_valid | I | 1 | Indicates if the header info input is valid. |
| htf\_bhp\_hdrinfo\_ready | O | 1 | Indicates if the HTF is ready to receive header info. |
| bhp\_htf\_hdrinfo\_bus.fmt | I | 3 | Format of the data on the symbol table data bus:  0 = XP9 block  1 = XP10 block  2 = DEFLATE block dynamic  3\* = DEFLATE block fixed  4\* = RAW block  \*3 and 4 will have no associated data on the symbol table data bus. |
| bhp\_htf\_hdrinfo\_bus.sub\_fmt | O | 2 | 0= GZIP  1= CHU4k/ZLIB  2= CHU8k |
| bhp\_htf\_hdrinfo\_bus.winsize | I | 3 | The number of bit lengths encoded in the symbol table is variable for XP10 and depends on the window size. Window sizes less than 4 are only supported. |
| bhp\_htf\_hdrinfo\_bus.min\_ptr\_len | I | 1 | Minimum match length for XP9/XP10 PTR:  0 = 3  1 = 4 |
| bhp\_htf\_hdrinfo\_bus.mtf\_len | I | 1 | Minimum match length for XP9 MTF:  0 = 2  1 = 3 |
| bhp\_htf\_hdrinfo\_bus.predef\_last | I | 1 | Indicates that this is the last XP10 block in a frame containing a pre-determined Huffman bit length set. |
| bhp\_htf\_hdrinfo\_bus.phd\_present | I | 1 | Indicates if the pre-determined Huffman table was loaded as part of this frame. |
| bhp\_htf\_hdrinfo\_bus.trace\_bit | I | 1 | Trace bit indicator |
| bhp\_htf\_hdrinfo\_bus.sof\_blk | I | 1 | Indicates if this is the first block of the frame. |
| bhp\_htf\_hdrinfo\_bus.hdr\_bits\_in | I | 23 | Total number of frame/block header bits consumed for this block by the BHP. |
| bhp\_htf\_hdrinfo\_bus.rqe\_sched\_handle | I | 16 | Scheduler handle information passed from RQE tlv to the scheduler block |
| bhp\_htf\_hdrinfo\_bus.tlv\_frame\_num | I | 11 | Frame number from the DATA TLV |
| bhp\_htf\_hdrinfo\_bus.tlv\_eng\_id | I | 4 | Engine ID from the DATA TLV |
| bhp\_htf\_hdrinfo\_bus.tlv\_seq\_num | I | 8 | TLV Sequence number from the DATA TLV. |
| **Huffman Table Formatter to Header Parser Status Interface** | | | |
| htf\_bhp\_status\_valid | O | 1 | Indicates the symbol table size information is valid. This message is only sent in response to formats 0, 1, and 2 (see bhp\_htf\_hdrinfo\_fmt above). |
| bhp\_htf\_status\_ready | I | 1 | Block ready to accept data |
| htf\_bhp\_status\_bus.size | O | 13 | Size of the received symbol table in bits (i.e. total number of bits consumed before start of compressed data). |
| htf\_bhp\_status\_bus.error | O | 8 | Error status of symbol table processing: |
| **Huffman Table Formatter to Symbol Data Decoder BCT/SAT Write Interface** | | | |
| htf\_sdd\_bct\_sat\_wen | O | 1 | Write enable for BCT and SAT. |
| htf\_sdd\_bct\_sat\_type | O | 1 | 0 = short table (also DEFLATE literal/length)  1 = long table (also DEFLATE distance) |
| htf\_sdd\_bct\_data | O | 26 | BCT entry value. LSB is implicitly 0. |
| htf\_sdd\_sat\_data | O | 10 | SAT entry value. |
| htf\_sdd\_bct\_sat\_addr | O | 5 | The address of the BCT/SAT tables being written (i.e. the corresponding Huffman codeword bit length). |
| htf\_sdd\_bct\_valid | O | 1 | Indicates that there are valid codewords fo the corresponding length (i.e. addr). Note that all BCT entries default to invalid (i.e. they remain invalid if not indexed by htf\_sdd\_bct\_sat\_addr before htf\_sdd\_complete\_valid is asserted). |
| **Huffman Table Formatter to Symbol Data Decoder SLT Write Interface** | | | |
| htf\_sdd\_{ss,ll}\_slt\_wen[[1]](#footnote-2) | O | 2 | Write enable for SLT:  [0] = {short,long} SLT write port 0  [1] = {short,long} SLT write port 1 |
| htf\_sdd\_{ss,ll}\_slt\_addr | O | [2][10] | SLT entry address. |
| htf\_sdd\_{ss,ll}\_slt\_data | O | [4][10] | SLT entry data. |
| **Huffman Table Formatter to Symbol Data Decoder Table Completion Interface** | | | |
| htf\_sdd\_complete\_valid | O | 1 | Asserted for 1 cycle coincident with the final write for a set of decoder tables. Qualfies the other htf\_sdd\_complete\_\* signals. |
| htf\_sdd\_complete\_error | O | 1 | Indicates that an error was encountered while decoding the symbol tables. This is important to keep to Front-End, Huffman Table Formatter, and Symbol Data Decoder in sync in the event of mid-frame errors. |
| htf\_sdd\_complete\_fmt | O | 2 | 0 = XP9  1 = XP10  2 = DEFLATE DYNAMIC  3 = DEFLATE FIXED  4 = RAW |
| htf\_sdd\_complete\_min\_ptr\_len | O | 1 | Minimum match length for XP9/XP10 PTR:  0 = 3  1 = 4 |
| htf\_sdd\_complete\_min\_mtf\_len | O | 1 | Minimum match length for XP9 MTF:  0 = 2  1 = 3  Ignored for XP10. |
| htf\_sdd\_complete\_sched\_info | O | 63 | This is actually a struct that contains the hdr\_bits\_in, rqe\_sched\_handle, tlv\_frame\_num, tlv\_eng\_id, and tlv\_seq\_num fields as described above for bhp\_htf\_hdrinfo\_bus. All of the field values are passed through from bhp\_htf\_hdrinfo\_bus except for hdr\_bits\_in, to which the HTF adds the total number of decoded symbol tables bits to the total received by BHP. |
| sdd\_htf\_busy | I | 1 | Asserted when the SDD can accept no more decode tables. The SDD can store 2 complete sets of decode tables: one set being used to actively decode while the next is populated via the htf\_sdd interface. If this busy signal is asserted, it cannot be de-asserted until the block currently being decoded by the SDD is completed. Backpressure on the scheduler update interface can also cause this signal to be asserted. |

Table 1 Huffman Table Formatter Interfaces

A symbol table describes a Canonical Huffman tree in terms of a sequence of bit lengths. The bit lengths themselves arrive in an encoded format which differs between XP9/XP10 and DEFLATE. For both XP9/XP10 and DEFLATE, there are actually 3 separate Huffman symbol alphabets:

* XP9/XP10 has a short symbol alphabet (up to 576 symbols) and a long symbol alphabet (up to 248 symbols) that are used to encode LZ77 literals and matches. There is also a small symbol alphabet (33 symbols) which is used within the symbol table itself in order to compress the bit lengths of the short and long symbol alphabets. The XP9/XP10 symbol table is actually divided into 2 distinct symbol tables, one containing the bits lengths of the short symbol alphabet, and the other containing the bit lengths of the long symbol alphabet. Each has an associated bit length table for its own associated small symbol alphabet.
* DEFLATE has a literal/length symbol alphabet (285 symbols) and a distance symbol alphabet (30 symbols) that are used to encode LZ77 literals and matches. There is also a code-length symbol alphabet (19 symbols) which is used within the symbol table itself in order to compress the bit lengths of the literal/length and distance symbol alphabets. DEFLATE concatenates the bit lengths for the literal/length and distance symbol alphabets into a single continuous sequence of bit lengths which itself is encoded by the singular code-length symbol alphabet.

In order to format XP9/XP10 Huffman decode tables, the Table Formatter Logic must, for both the short and long symbol alphabets:

1. From the symbol table header, identify the symbol table encoding type (see XP9/XP10 spec) from the symbol table header as either simple, pre-defined Huffman, or retrospective Huffman.
2. For retrospective Huffman, use delta-decompression (see XP9/XP10 spec) to decode the bit lengths of the small symbol alphabet from the symbol table.
3. Build Huffman decoder tables for the small symbol alphabet. For pre-defined Huffman, since there is no small table, the same Huffman decoder tables default to a configuration that will interpret the pre-defined Huffman bit lengths as 5-bit codewords.
4. Apply the decoder tables built in step 3 to decode the bit lengths of the short/long symbol alphabet from the symbol table. If the Block Header Parser has indicated RAW or the symbol table header has indicated simple encode, the Huffman decoder tables are populated with the appropriate configuration (e.g. 8-bit identify function for RAW).
5. Build Huffman decoder tables for the short/long symbol alphabet. For simple encoding, the same Huffman decoder tables are populated with the appropriate configuration for the Symbol Data Decoder to decode simple-encoded symbols.

In order to format DEFLATE Huffman decode tables, the Table Formatter Logic must:

1. For dynamic Huffman codes (i.e. what XP9/XP10 refers to as retrospective), from the symbol table header, identify the number of literal/length codes, distance codes, and code-length codes contained within the symbol table.
2. Extract the bit lengths of the code-length symbol alphabet from the symbol table.
3. Build the Huffman decoder tables for the code-length symbol alphabet.
4. Apply the decoder tables built in step 3 to decode the bit lengths of both the literal/length and distance symbol alphabet.
5. Build Huffman decoder tables for both the literal/length and distance symbol alphabet. If the Block Header Parser has indicated RAW or fixed Huffman, the Huffman decoder tables are populated with the appropriate configuration, as is the case for XP9/XP10.

For RAW encoding (i.e. literals only), which is indicated directly by the Block Header Parser, the Huffman decoder tables are populated with the appropriate configuration for the Symbol Data Decoder to decode RAW literals. XP9/XP10 and DEFLATE also differ in that for XP9/XP10, simple encoding is indicated in the symbol table header, with separate indications for the short and long symbol tables. For DEFLATE, the BTYPE (which is parsed by the Block Header Parser) indicates the analogous “fixed” Huffman coding.

Much of the core table formatter logic will be shared for XP9/XP10 and DEFLATE because it operates in terms of a generic Canonical Huffman code described by bit lengths. Obviously, internal resources will be sized to accommodate the XP9/XP10 tables, of which DEFLATE will only use a fraction because its analogous symbol alphabets are smaller.

### Detailed Description

#### Decoder Table Format

The Huffman Table Formatter delivers Huffman decoder tables to the Symbol Data Decoder block where they are used to decode the compressed bit stream. The Table Formatter Logic itself builds an internal set of tables with the same generic format (although much smaller) which it uses to decode the compressed XP9/XP10 or DEFLATE Huffman symbol bit lengths. The following generic description applies to both of these use cases.



Figure 6 Decoder Table Formats

In order to decode a Huffman-encoded bit stream, 3 tables are used:

* Base Code Table (BCT)
* Start Address Table (SAT)
* Symbol Lookup Table (SLT)

The Symbol Data Decoder block specification describes precisely how these are used to decode a Huffman-encoded bit stream.

In order to help illustrate how these tables represent a Huffman code, consider the following simple example alphabet with 8 symbols:

|  |  |
| --- | --- |
| Huffman Code | Symbol |
| 0 | 5 |
| 100 | 1 |
| 101 | 6 |
| 1100 | 0 |
| 1101 | 3 |
| 1110 | 7 |
| 111100 | 2 |
| 111101 | 4 |

Table 3 Example Huffman Alphabet Sorted by Codeword

|  |  |
| --- | --- |
| Symbol | Huffman Code |
| 0 | 1100 |
| 1 | 100 |
| 2 | 111100 |
| 3 | 1101 |
| 4 | 111101 |
| 5 | 0 |
| 6 | 101 |
| 7 | 1110 |

Table 4 Example Huffman Alphabet Sorted by Symbol

##### Base Code Table (BCT)

This table has 1 entry per bit length of the associated alphabet (i.e. one for each Huffman tree level). For example, the XP9/XP10 small symbol alphabet BCT has 8 entries (of which 7 entries are used for the DEFLATE code-length alphabet), and the XP9/XP10 long and short symbol BCTs each have 27 entries (of which 15 entries are used for the DEFLATE literal/length and distance alphabets).

Each entry contains the lowest-value Huffman codeword of the group of all codewords with the same bit length. Any possible BCT can ONLY have 0 as the 1-bit base code, so it is not actually necessary to store this entry in the table. Also, the LSB of every base code must be 0, and thus doesn’t need to be stored. For example, the BCT for the XP9/XP10 short and long symbol alphabets stores base codes for lengths 2 through 27, totaling 364 bits.

The BCT for the Huffman code shown above would be:

|  |  |
| --- | --- |
| Base Code Table (BCT) | |
| Index | Value |
| 1 | 0 |
| 2 | 10 |
| 3 | 100 |
| 4 | 1100 |
| 5 | 11110 |
| 6 | 111100 |

Table 5 Base Code Table Example

Note that for the greyed out entries, there are no valid codewords of the associated length. The values shown here are populated as a side-effect of the algorithm which computes the BCT entries. Also, the LSB of each of these entries is not actually stored.

##### Start Address Table (SAT)

Like the BCT, this table also has 1 entry per bit length of the associated alphabet (i.e. one for each Huffman tree level).

If you imagine all Huffman codewords in the alphabet arranged in a list sorted by codeword value (e.g. Table 3), the SAT stores, for each bit length in the Huffman code, the index of the item in that list containing the lowest-value codeword of that bit length. The entry associated with bit length 1 can ONLY be 0, and thus doesn’t need to be stored. For example, the SAT for the XP9/XP10 short symbol alphabet has storage for entries 2 through 27, with 10 bits per entry (i.e. in order to index 576 codewords), totaling 260 bits.

The SAT for the Huffman code shown above would be:

|  |  |
| --- | --- |
| Start Address Table (SAT) | |
| Index | Value |
| 1 | 0 (implicit) |
| 2 | 1 |
| 3 | 1 |
| 4 | 3 |
| 5 | 6 |
| 6 | 6 |

Table 6 Start Address Table Example

It is important to note that the SAT is essentially a cumulative histogram of bit lengths, such that the value at a given index represents the total number of codewords with a bit length LESS than the index.

##### Symbol Lookup Table (SLT)

This table has a number of entries equal to the number of Huffman codewords (i.e. the number of symbols in the alphabet), and each entry contains the actual symbol value for the associated codeword. For example the SLT for the XP9/XP10 short symbol alphabet has storage for 576 entries, with 10 bits per entry, totaling 5760 bits.

The SLT for the Huffman code shown above would be:

|  |  |
| --- | --- |
| Symbol Lookup Table (SLT) | |
| Index | Value |
| 0 | 5 |
| 1 | 1 |
| 2 | 6 |
| 3 | 0 |
| 4 | 3 |
| 5 | 7 |
| 6 | 2 |
| 7 | 4 |

Table 7 Symbol Lookup Table Example

This can also be thought of as mapping from codeword index to symbol.

#### Symbol Table Buffering

The Frame Header Parser will extract the pre-determined Huffman bit length set from the frame header (if the associated TLV exists) and send the data to the Huffman Table Formatter. The TLV has 65x64-bit words of payload formatted as follows:

* 45 words containing 576x5-bit short symbol lengths
* 20 words containing 248x5-bit long symbol lengths

Packing of the 5-bit lengths follows the XP9/XP10/DEFLATE bit-packing format as if each length is a 5-bit Huffman codeword (i.e. bit-reversed order). For example, the most-significant bit of the 5-bit length for short symbol 0 is packed into the least-significant bit (i.e. bit 0) of the 1st byte of the TLV payload. The least significant bit of the 5-bit length for short symbol 0 is packed into bit 4 of the 1st byte of the TLV payload. Bytes within a 64-bit TLV word are packed in little-endian format.

This data is kept in a Bit Feeder FIFO (BFF) and will be used to build Huffman decoder tables when an XP10 symbol encode type indicates pre-determined Huffman. Either short or long or both pre-determined bit lengths may be used. The pre-determined decoder tables are rebuilt when the previous block uses simple encode or retrospective Huffman. The size of the pre-determined Huffman BFF will be 130x64-bit words (8320 bits). This is enough to accommodate 2 complete sets of 576x5-bit short symbol bit lengths and 248x5-bit long symbol bit lengths. It can be parameterized to be smaller if the maximum XP10 window size is smaller (e.g. 16K vs. 64K). The BFF is logically split into 2 separate banks, one for each bit length set. After buffering 2 bit length sets, it cannot accept a new bit length set until the last block header in a frame is parsed enough to know that the associated pre-determined bit lengths are no longer needed.

The Block Header Parser recognizes the beginning of the symbol table and begins to send the symbol table data to the Huffman Table Formatter, where it is stored in a separate BFF. For cases where the end of the symbol table is unknown, the Block Header Parser sends enough data from the block to encompass the worst case size of the encoded symbol table, in order to avoid upstream head-of-line blocking. The Block Header Symbol Table BFF will be sized to accommodate this. As soon as the end of the encoded symbol table is identified, the Huffman Table Formatter will discard the unused data and indicate the number of bits consumed (i.e. the size of the symbol table) back to the Block Header Parser. This size of the block header BFF will be 125x64-bit words (8000 bits). This is enough to accommodate the following:

* 3-bit short symbol table header
* 33x4-bit small table bit lengths (for encoding short table bit lengths)
* 704x8-bit short table bit lengths
* 3-bit long symbol table header
* 33x4-bit small table bit lengths (for encoding long table bit lengths)
* 256x8-bit long table bit lengths

Since the XP9 symbol table includes bit lengths for all 704 short symbols and all 256 long symbols, regardless of the supported window size, this will NOT be smaller for an instance that only supports smaller LZ77 windows (e.g. 16K vs. 64K).

#### Table Formatter Logic

**Terms:**

SS: short symbol table

SmallSS: small symbol table used to encode bit lengths of SS table

LL: long symbol table

SmallLL: small symbol table used to encode bit lengths of LL table

Histogram: A table with 1 entry per bit length that records the total number of codewords of each bit length

The Table Formatter Logic must be capable of generating a complete set of Huffman decoder tables every 512 clocks. The latency measured from consuming the 1st bit of the encoded symbol table to populating the last associated SLT entry may actually greater than 512 clocks, as long as work on the next encoded symbol table can begin within 512 clocks. The process can be pipelined into the following 4 stages:

1. Bit Length Decode  
   This stage is broken up into several sub-steps that must be serialized because each step depends on the previous step having been completed.
   1. SmallSS/SmallLL Bit Length Decode  
      This step reads the encoded SmallSS/SmallLL bit length information from the block header symbol table BFF into the small BLT and accumulates a histogram of bit length values. For XP9/XP10, delta-decompression (see section 3.2.3.3.1) is used to decode SmallSS/SmallLL bit lengths, and the resulting Huffman decoder tables are used to decode the SS/LL bit lengths. For DEFLATE, code-length codes are stored as sequence of 3-bit values (see RFC1951 for details), and the resulting Huffman decoder tables are used to decode the literal/length and distance bit lengths. This step also needs to spend a cycle at the beginning of each symbol table decoding the associated header (i.e. 2-bit/3-bit for XP9/XP10 and a variable number of bits for DEFLATE, starting with HLIT).

Another difference between XP9/XP10 and DEFLATE is that XP9/XP10 specifies the encoding type (i.e. simple, pre-determined, or retrospective) in the symbol table header, while DEFLATE specifies this in the BTYPE (which is parsed by the Block Header Parser).

This step will populate the small Bit Length Table (BLT) at a rate of 2 bit lengths per cycle, consuming a total of about 17 cycles (i.e. for 33 bit lengths).  
This step can be skipped for pre-determined Huffman because there is no small table.

* 1. SmallSS/SmallLL SAT and BCT Populate  
     This step populates the SAT and BCT tables which are then used to decode the “main” symbol bit lengths from the symbol table bit stream. This is very similar for XP9/XP10 and DEFLATE code-length codes, only differing in the MAX\_BIT\_LENGTH (i.e. 8 for XP9/XP10 and 7 for DEFLATE). The following pseudocode shows how the SAT and BCT are populated:

prev\_SAT = SAT[1] = POINTER[1] = 0;  
prev\_BCT = BCT[1] = 0;  
for i from 2 to MAX\_BIT\_LENGTH {  
 prev\_SAT = SAT[i] = POINTER[i] = prev\_SAT + HISTOGRAM[i-1];  
 prev\_BCT = BCT[i] = (prev\_BCT + HISTOGRAM[i-1]) << 1;  
}

Note that the POINTER table contains the same information as the SAT, but will be modified as the SLT is populated in the next step. Therefore, a local copy is needed that is physically separate from the SAT.

This step populates 1 entry per cycle, consuming a total of about 6 cycles.  
This step can be skipped for pre-determined Huffman because there is no small table.

* 1. SmallSS/SmallLL SLT Populate  
     This step populates the small SLT table which is then used to decode the “main” symbol bit lengths from the symbol table bit stream. This is very similar for XP9/XP10 and DEFLATE, only differing in the number of symbols and bit lengths. The following psuedocode shows how the SLT is populated:

for i from 0 to NUM\_SYMBOLS-1 {

SLT[POINTER[BLT[i]] = i;  
 POINTER[BLT[i]]++;  
}

NUM\_SYMBOLS is 33 for XP9/XP10 and up to 19 for DEFLATE (see RFC1951).  
This step populates 2 entries per cycle, consuming a total of about 17 cycles.  
This step can be skipped for pre-determined Huffman because there is no small table.

* 1. SS/LL Bit Length Decode  
     When the preceding 3 sub-steps are complete, the Huffman-encoded SS/LL bit lengths can be decoded from the symbol table. This can be done at a rate of 2 bit-lengths per cycle by consuming 2 small Huffman codewords per cycle (see section 3.2.3.3.2). As the bit lengths are read in, the BLT is populated and a bit length histogram is accumulated. This step will populate the BLT in 288 cycles for SS bit lengths, 124 cycles for LL bit lengths, 143 cycles for DEFLATE literal/length bit lengths, and 15 cycles for DEFLATE distance bit lengths.

For XP9/XP10, once the SS bit lengths have been fully decoded, these 4 sub-steps are repeated for the long symbol table.

For DEFLATE, the bit lengths for both symbol alphabets are read in one continuous sequence.

Once this step is completed for both symbol alphabets, this stage will be ready to start reading the symbol table for the following block.

For XP9, even though only the last 128 short symbols are not used (i.e. associated symbol bit lengths are 0), these 0-length symbols are still encoded in the short table.

Since they don’t need to be written to the BLT, and assuming that they are encoded with TABLE\_FILL SmallSS symbols, there should only be an addition 8 SmallSS symbols in the symbol table, which will take only 4 additional cycles to consume.

1. SS/LL SAT and BCT Populate  
   This stage can overlap execution with the prior stage and begin as soon as all the bit lengths for a single symbol alphabet have been read. The algorithm is identical to step 1.2 from above, except that the XP9/XP10 codes have bit lengths up to 27 and DEFLATE codes have bit lengths up to 15. Some of the hardware can be reused for both symbol alphabets, although the POINTER table must be separate for SS and LL because their associated SLTs will be populated concurrently.
2. SS SLT Populate  
   This stage can overlap with the execution of stage 1. It can also overlap with stage 2 when stage 2 is populating the SAT/BCT for the LL table. The algorithm is identical to step 1.2 from above, except that there are MANY more SS symbols (i.e. 576) compared to SmallSS/SmallLL symbols. Given so many symbols, it is necessary to execute this stage at a rate of 2 SLT entries per cycle. Incrementing 2 POINTER entries per cycle (which can mean incrementing the SAME entry twice in some case) can be tricky, but is still quite doable given that the POINTER table has a maximum of only 26 entries.
3. LL SLT Populate  
   This stage can overlap with the execution of the previous stage because the LL bit lengths will be completely read, and the associated SAT/BCT completely populated, before the SS SLT is fully populated. This algorithm is identical to the previous step, but only needs to generate 248 SLT entries at a rate of 2 per clock.

The following figure shows how these stages are overlapped in time and illustrates how a rate of 1 full symbol table per 512 clocks can be maintained. Note the small gap in between “SS Bit Length Decode” and “SmallLL Bit Length Decode”: this represents the few extra cycles that are needed to consume TABLE\_FILL SmallSS symbols associated with the unused short symbols (i.e. short symbols 576 to 703 are not used when the XP9 window size is limited to 64KB).



Figure 7 Table Formatter Logic Pipelining

Note that a single large 576x10 entry BLT can be used to store both the bit lengths of the SS and LL tables. This is because as stage 3 reads BLT entries for the SS table in sequence in order to populate the associated SLT, it is effectively “freeing” entries that can then be populated by LL table bit lengths from stage 1. Following this, once the LL table bit lengths have been fully read in, stage 3 is still iterating through the upper half of the BLT in order to populate the SS table SLT. Stage 4 can begin populating the LL table SLT, reading entries out of the same physical BLT from different addresses than the stage 3 logic. As mentioned earlier, the associated POINTER tables must be physically separate.

##### Delta Decompression Details

The delta decompression block simply decodes the codes lengths for each of the small table symbols (there are 33 for XP9/XP10). The logic looks at the output of the BFF and determines the bit length to send to the small table. The delta codes are interpreted according to the following pseudocode:

previous\_symbol\_length = 4;

for i from 0 to 32 {

next\_bit = get\_bits(1);

if (next\_bit == 0)

small\_BLT[i] = previous\_symbol\_length;

else {

k = get\_bits(3);

if (k < previous\_symbol\_length)

previous\_symbol\_length = k;

else

previous\_symbol\_length = k+1;

small\_BLT[i] = previous\_symbol\_length;

}

}

The hardware processor accomplishes the above algorithm by looking at the most recent 4-bit window in the BFF output and issues a variable shift command based on what it sees. If the next bit is 0, it writes the next symbol and acks 1 BFF bit, if the next bit is 1, the next 3 bits are examined and the symbol determined, then 4 bits are acknowledged to the BFF. It does this effectively 2 times per cycle, consuming as many as 8 bits from the BFF, in order to populate 2 BLT entries per cycle.

##### Huffman Decoder Details

A Huffman Decoder is used to determine the BLT entries for both the long and short symbol tables. The Huffman Decoder hardware is the same circuit instantiated multiple times in the Symbol Data Decoder, and is described in that section. However, in this instance, the maximum Huffman code size for the encoded bit lengths is 8 bits (7 bits for DEFLATE), and the underlying hardware operations are scaled accordingly.

Unlike the heavily parallel data decoder, the Huffman Decoder in the Header Processor does not need the same parallel performance and is only working on data encoded by a single Huffman tree (and no raw data interleaved), so its processing can occur in serialized fashion – a symbol is decoded and it’s length is determined and the input data is shifted by the corresponding number of bits for the next symbol. This process can determine the number of bits consumed by the next 2 Huffman codewords, and shift by the appropriate amount for the next clock cycle. There is a feedback path from the determination of the symbol size to shifting the bits for the next symbol decode, but the depth of the logic is minimized because the Huffman Decoder resolves a symbol’s size as early as possible (this is Symbol\_Size\_o in Figure 10).

Both XP9/XP10 and DEFLATE have special cases where certain symbols can be followed by extra bits in order to effectively run-length-encode a sequence of bit lengths. To handle this, dedicated logic will recognize the specific codeword (symbol 29 for SmallSS/SmallLL, and symbols 16, 17, and 18 for DEFLATE code-length codes), and “switch” the processing state machine to recognize the number of extra bits and shift accordingly. This can take multiple cycles in the case of SmallSS/SmallLL symbol 29.

#### Formatted Decoder Tables

The formatted decoder tables are double buffered in ping-pong fashion so that one set of tables can be used by the parallel Symbol Data Decoder while the next set of tables is being built. These tables are physical stored in the Symbol Data Decoder, but the Huffman Table Formatter block has an interface to the Symbol Data Decoder which is used to write the entries of these tables

For the SAT and BCT, only a single write interface is needed over which the entries are transmitted (one entry from each table per cycle). This is done separately for the SS and LL decoder tables using the same interface, and does not need to be done concurrently.

For the SLT, multiple write interfaces are needed. This is because the SS and LL decoder SLTs must be populated concurrently, AND because each needs to be populated at a rate of 2 entries per cycle. Accordingly, there are 2 pairs of write interfaces, one for the SS SLT, and one for the LL SLT.

### Errors

The Huffman Table Formatter may encounter the following errors:

* E\_ILLEGAL\_SYMBOL\_TABLE\_SHORT:  
  The XP9 short symbol table contains symbols with a non-zero bit length that are incompatible with the window size. For example, short symbols 576 to 703 should ALWAYS have a bit-length of 0.
* E\_ILLEGAL\_SYMBOL\_TABLE\_LONG:  
  Similar to above, but for the XP9 long symbol table.
* E\_INVALID\_SYMBOL\_TABLE\_ENCODING:  
  Occurs when the XP9/XP10 symbol table header uses a reserved codepoint, or when the XP10 symbol table headers specifies pre-determined Huffman when no pre-determined Huffman bit-lengths were received.
* E\_BAD\_HUFFMAN\_CODE:  
  Huffman decoding of the symbol table encountered an invalid bit sequence.

## Symbol Data Decoder

### Overview

The Symbol Data Decoder takes the incoming bit stream from the Latency FIFO and Data Aligner (LFA) block and performs a parallel decode operation for encoded Huffman symbols and fully resolves multiple symbols per cycle. The parallel search is complicated due to the inherent serialization of the data within the bit-stream: the start of a format symbol is not known until the previous format symbol has been fully resolved. And each format symbol may include a variable number of both Huffman codewords (1-2) and raw data fields (0-2), where each sub-field is also variable length that is unknown until the previous sub-field is resolved.

At a high level, the design achieves high-throughput decoding of the compressed bit stream through the following process:

1. Speculatively decode all possible aggregate symbols in a given window of input data, one starting from each possible bit offset within the window. Each aggregate symbol includes 1 to 2 Huffman codewords and the associated 0 to 2 raw data fields.
2. After finding all possible symbols in a window, and their associated lengths, chain valid symbols together from a starting bit offset.
3. After a variable number of valid symbols are found, the symbols are collected and packed into a bus format compatible with the processing capabilities of downstream processing units.

This process is mapped to hardware functions as seen in the following top level block diagram of the Symbol Data Decoder.



Figure 8 Symbol Data Decoder Top Level Block Diagram

An overview of the high level operation of the hardware will be given in this section and the details of each of the sub-blocks are described in separate sections.

There are three primary inputs to the Symbol Data Decoder:

1. Configuration information regarding the format of the compressed data, which is passed from the Block Header Parser.
2. Writes from the Huffman Table Formatter containing decoder table entries.
3. The compressed data bit stream (the block header is removed upstream).

The Window feeder collects the input data stream and sends a window that is sized to hold the largest possible aggregate symbol for the configured format to every decode lane.

Each decode lane each takes a full window of the compressed data stream, presented by the window feeder, and fully resolves the contents (if possible) for the corresponding symbol format.

The output of the decode lanes are a set of candidate symbols to the symbol selector every cycle. The symbol selector then determines which symbols presented by the decode lanes are actually valid symbols. The symbol selector may backpressure if the window contained more symbols than it can find/output per cycle.

The validated symbols are then sent to the symbol packer, which assembles symbols into an output bus that is compatible with downstream blocks.

### Interfaces

The Symbol Data Decoder has the following interfaces:

* Latency FIFO and Data Aligner (LFA) Interface
  + Bit Stream Data Bus:  
    The LFA can send 64 bits per cycle of compressed data, with associated SOB, EOB, and EOF flags. It also includes the number of valid bits (up to 64) for the transfer.
  + Bit Stream Acknowledge:  
    The Symbol Data Decoder (SDD) uses the interface to acknowledge that bits have been consumed (i.e. symbols decoded) from the bit stream. It indicates when EOB is reached, which is used by the LFA in order to “rewind” the compressed data stream and find the location of the next DEFLATE block header. Finally, it includes an error indicator, after which the SDD will not send another acknowledge message until AFTER it sees an EOF.
* Move to Front Interface
  + LZ77 Symbol Data Bus:  
    This interface is used to send up to 4 symbols to the Move to Front block, one of which can be a PTR/MTF back-reference. A ***framing*** signal specifies the number of valid symbols, and whether or not they are the final symbols within a compressed block or frame. EOF with errors is a special case indicated by a unique ***framing*** signal codepoint. No other symbols are transmitted during an EOF error transfer, and the remaining bits on the bus are used to signal error conditions.
* Huffman Table Formatter (HTF) Interface  
  See a detailed description of this interface in section 3.2.2.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **LFA to SDD Bit Stream Data Interface** | | | |
| lfa\_sdd\_dp\_valid | I | 1 | Indicates if the data input is valid. |
| sdd\_lfa\_dp\_ready | O | 1 | Indicates if the SDD is ready to receive data. |
| lfa\_sdd\_dp\_bus.data | I | 64 | Compressed data to the SDD. |
| lfa\_sdd\_dp\_bus.numbits | I | 7 | Number of valid bits on the bus. |
| lfa\_sdd\_dp\_bus.sob | I | 1 | Start of Block |
| lfa\_sdd\_dp\_bus.eob | I | 1 | End of Block for XP9/XP10. End of available data for DEFLATE. |
| lfa\_sdd\_dp\_bus.eof | I | 1 | End of Frame. Asserted with EOB for XP9/XP10 frames. Asserted AFTER EOB for DEFLATE frames. |
| lfa\_sdd\_dp\_bus.error | I | 8 | Error code is qualified by EOF. Non-zero value means that an error has been detected upstream of the SDD. |
| lfa\_sdd\_dp\_bus.trace\_bit | I | 1 | Trace bit indicator |
| lfa\_sdd\_dp\_bus.frame\_bytes\_in | I | 28 | Total number of bytes received in this frame. |
| lfa\_sdd\_dp\_bus.last\_frame | I | 1 | Last frame indicator for compound commands |
| **SDD to LFA Bit Stream Acknowledge Interface** | | | |
| sdd\_lfa\_ack\_valid | O | 1 | Indicates a valid acknowledge message. |
| sdd\_lfa\_ack\_bus.numbits | O | 7 | The total number of bits consumed from the bit stream. The LFA uses this to advance its tail pointer. |
| sdd\_lfa\_ack\_bus.eob | O | 1 | For XP9/XP10, indicates that final symbol in block was decoded. For DEFLATE, indicates that EOB symbol was decoded. |
| sdd\_lfa\_ack\_bus.err | O | 1 | Indicates that a decode error or missing EOB has occurred. numbits will be 0. |
| **SDD to MTF Symbol Data Interface** | | | |
| sdd\_mtf\_valid | O | 1 | Indicates that the symbol data is valid. |
| mtf\_sdd\_ready | I | 1 | Indicates that the MTF is ready to receive symbol data. |
| sdd\_mtf\_framing | O | 4 | Framing information:  [0,4] = 0 to 4 lanes valid  [5,9] = 0 to 4 lanes valid with EOB  15 = EOF. If there is an error, the 32 bit data carries the error code. |
| sdd\_mtf\_data0 | O | 8 | Literal data or lower bits of back-reference offset:  for PTR: ptr\_offset[7:0]  For EOF with Error: 32 bits of error data |
| sdd\_mtf\_data1 | O | 8 |
| sdd\_mtf\_data2 | O | 8 |
| sdd\_mtf\_data3 | O | 8 |
| sdd\_mtf\_backref | O | 1 | Indicates that a back-reference is present |
| sdd\_mtf\_backref\_lane | O | 2 | Lane number containing the back-reference |
| sdd\_mtf\_backref\_type | O | 1 | Type of the back-reference:  0 = PTR  1 = MTF |
| sdd\_mtf\_offset\_msb | O | 8 | Upper bits of pointer offset |
| sdd\_mtf\_length | O | 16 | Back-reference length |
| **HTF to SDD BCT/SAT Write Interface** | | | |
| htf\_sdd\_bct\_sat\_wen | I | 1 | Write enable for BCT and SAT. |
| htf\_sdd\_bct\_sat\_type | I | 1 | 0 = short table (also DEFLATE literal/length)  1 = long table (also DEFLATE distance) |
| htf\_sdd\_bct\_data | I | 26 | BCT entry value. LSB is implicitly 0. |
| htf\_sdd\_sat\_data | I | 10 | SAT entry value. |
| htf\_sdd\_bct\_sat\_addr | I | 5 | The address of the BCT/SAT tables being written (i.e. the corresponding Huffman codeword bit length). |
| htf\_sdd\_bct\_valid | I | 1 | Indicates that there are valid codewords fo the corresponding length (i.e. addr). Note that all BCT entries default to invalid (i.e. they remain invalid if not indexed by htf\_sdd\_bct\_sat\_addr before htf\_sdd\_complete\_valid is asserted). |
| **HTF to SDD SLT Write Interface** | | | |
| htf\_sdd\_{ss,ll}\_slt\_wen[[2]](#footnote-3) | I | 2 | Write enable for SLT:  [0] = {short,long} SLT write port 0  [1] = {short,long} SLT write port 1 |
| htf\_sdd\_{ss,ll}\_slt\_addr | I | [2][10] | SLT entry address. |
| htf\_sdd\_{ss,ll}\_slt\_data | I | [4][10] | SLT entry data. |
| **HTF to SDD Table Completion Interface** | | | |
| htf\_sdd\_complete\_valid | I | 1 | Asserted for 1 cycle coincident with the final write for a set of decoder tables. Qualfies the other htf\_sdd\_complete\_\* signals. |
| htf\_sdd\_complete\_error | I | 1 | Indicates that an error was encountered while decoding the symbol tables. This is important to keep to Front-End, Huffman Table Formatter, and Symbol Data Decoder in sync in the event of mid-frame errors. |
| htf\_sdd\_complete\_fmt | I | 2 | 0 = XP9  1 = XP10  2 = DEFLATE DYNAMIC  3 = DEFLATE FIXED  4 = RAW |
| htf\_sdd\_complete\_min\_ptr\_len | I | 1 | Minimum match length for XP9/XP10 PTR:  0 = 3  1 = 4 |
| htf\_sdd\_complete\_min\_mtf\_len | I | 1 | Minimum match length for XP9 MTF:  0 = 2  1 = 3  Ignored for XP10. |
| htf\_sdd\_complete\_sched\_info | I | 63 | This is actually a struct that contains the hdr\_bits\_in, rqe\_sched\_handle, tlv\_frame\_num, tlv\_eng\_id, and tlv\_seq\_num fields as described above for bhp\_htf\_hdrinfo\_bus. All of the field values are passed through from bhp\_htf\_hdrinfo\_bus except for hdr\_bits\_in, to which the HTF adds the total number of decoded symbol tables bits to the total received by BHP. |
| sdd\_htf\_busy | O | 1 | Asserted when the SDD can accept no more decode tables. The SDD can store 2 complete sets of decode tables: one set being used to actively decode while the next is populated via the htf\_sdd interface. If this busy signal is asserted, it cannot be de-asserted until the block currently being decoded by the SDD is completed. Backpressure on the scheduler update interface can also cause this signal to be asserted. |
| **Scheduler Update Interface** | | | |
| xp10\_decomp\_sch\_update.valid | O | 1 |  |
| xp10\_decomp\_sch\_update.rqe\_sched\_handle | O | 16 | Scheduler handle information passed from RQE tlv to the scheduler block |
| xp10\_decomp\_sch\_update.last | O | 1 | Indicates the last scheduler update message for a frame. |
| xp10\_decomp\_sch\_update.tlv\_frame\_num | O | 11 | Frame number from the DATA TLV |
| xp10\_decomp\_sch\_update.tlv\_eng\_id | O | 4 | Engine ID from the DATA TLV |
| xp10\_decomp\_sch\_update.tlv\_seq\_num | O | 8 | TLV Sequence number from the DATA TLV. |
| xp10\_decomp\_sch\_update.bytes\_in | O | 24 | Bytes received for the current frame. |
| xp10\_decomp\_sch\_update.basis | O | 24 | Same as basis (included to conform with interaface signal definition. bytes\_in and basis may differ for other engines/functions). |
| xp10\_decomp\_sch\_update.bytes\_out | O | 24 | Bytes transmitted for the current frame (accounts for size post-LZ77-decompression). |
| su\_afull\_n | I | 1 | Scheduler interface messages canot be sent when su\_afull\_n=0. SDD pipeline can be stalled as a result. |

### Detailed Description

#### Huffman Decoder Tables

The contents of the Huffman Decoder Tables are received from the Huffman Table Formatter (HTF) block. The Symbol Data Decoder maintains two “banks” of formatted tables, one to which table writes are directed (i.e. for decoding the next block), and the other which is used to decode the symbol data currently in flight. Out of reset, writes will go to bank A. The Huffman Table Formatter asserts the ***complete\_valid*** signal when a table bank has been completely written. Only then, will the next block begin to be decoded. A counter keeps track of the number of occupied banks and asserts a ***busy*** signal when both are occupied. The ***busy*** signal won’t be de-asserted until End-Of-Block (EOB) is detected and the in-use set of decoder tables are no longer needed, at which point the counter is decremented. This allows the Huffman Table Formatter to begin writing the NEXT table set to the freed bank, and so on.

Coincident with the ***complete\_valid*** signal, the HTF also indicates if the associated tables are for an XP9/XP10 compressed block, a DEFLATE (fixed or dynamic) compressed block, or a RAW uncompressed block. This information is used by the decode lanes in order to interpret the Huffman symbols.

Section 3.2.3.4 has more details regarding the interface used to write these table entries.

The Symbol Data Decoder also keeps track of the number of which BCT entries are valid. This highest valid BCT index can also be thought of as the longest possible bit length for a Huffman codeword within the associated block.

#### Window Feeder

The Window feeder is a specialized Bit Feeder FIFO structure that collects the input bit stream and presents it to the decode lanes efficiently. The window feeder adapts the input bus to build a bit window set for the decode lanes. In order to maintain flexibility on both the input width and the number of lanes, the design of the window feeder balances flexibility and efficiency. Note that for certain specific ratios of the input bus to the number of decoder lanes, it is possible to remove the Bit Feeder FIFO and replace with simpler logic, however it is important to maintain flexibility as the Number of Decoder lanes is a big factor in the overall throughput.

The Bit Feeder FIFO’s purpose is to adapt the input bus width (i.e. 64-bit) to the internal bandwidth of the Symbol Decoder logic (i.e. 32-bit). Bit Feeder FIFO’s internal width is configured to the width of the bus into the Symbol Data Decoder (also the width of the top level data bus) and it outputs the number of bits for the configured number of lanes.

Each incoming 64-bit transfer includes the number of bits valid and may include an EOB or EOF flag. The window feeder must also ensure that the output EOF or EOB flag is associated with the correct output 32-bit transfer.

In order to consume the compressed bit stream at a rate of 25Gbps, 32 bits per clock need to be processed. For XP9/XP10, the maximum size of an aggregate symbol is 86 bits (2x27-bit Huffman codewords and 2x15-bit raw data fields). In order to keep up, the Window Feeder needs to present 32x84-bit slices downstream, one 84-bit slice for each decode lane.



Figure 9 Window Feeder

For XP9/XP10, the window feeder counts the number of compressed bits fed into the decode lanes and compares this to the total number of compressed bits in the block. When it detects that an XP9/XP10 block has been completed, it can immediately assert the ***credit*** signal to the Huffman Table Formatter and swap decoder table banks for the next block.

#### Decode Lanes

The decode lanes are a set of parallel symbol decoders, where each decoder lane has the capability to fully decode a literal, back reference, or any other type, per the supported compression format. Each literal, back reference, etc., may consist of multiple Huffman encoded symbols (1 to 2) and multiple raw data elements (0 to 2).

Each decode lane can consume one 86-bit slice per cycle, where the processing for each slice is fully pipelined. And since each decode lane looks at directly adjacent slices, the number of decode lanes determines the number of bits consumed per cycle (without backpressure). The decode lanes may be stalled by downstream blocks not being ready to consume additional data.

A block diagram of each decode lane is shown below. This diagram shows an example the data flow in the lane when configured for the XP9/XP10 specification. The operations are re-ordered for DEFLATE, where the primary hardware elements (i.e. the symbol decoders) are re-used, but re-organized via multiplexors.



Figure 10 XP9/XP10 Decode Lane

The decode lane carries the input slice through each stage, building up an output LZ77 symbol in the process as the slice is interpreted. The Huffman encoded symbols are interpreted differently per each compression standard (i.e. XP9/XP10 and DEFLATE), but in general the interpretation of the symbols both contributes to building the output symbol and determines if additional data in the slice needs to be interpreted to build the final output LZ77 symbol. Each Huffman Decoder (the Short/Long Symbol decoders in the previous diagram) does an ultra-low latency and efficient decode of a candidate Huffman symbol.

##### Huffman Symbol Decoder

The Huffman symbol decoder takes a slice of bits up to the maximum length of a Huffman codeword and determines the symbol and its length/size, as follows (refer to Figure 9):

1. For each bit length *n* from 1 to the 27 (i.e. the maximum), extract the 1st *n* bits from the slice and check if the value is greater than or equal to the associated *n-*bit base code. All compares are done in parallel. The compare for bit length 1 is an implicit success since the base code must always be 0. Compares for “invalid” lengths (i.e. no codewords of that length) shorter than the maximum length may also succeed, but the nature of the prefix code guarantees that compares for even longer valid lengths will also succeed. Compares for lengths greater than the maximum length (see section 3.3.3.1) are disqualified.
2. Combining the results of each compare into an 27*-*bit vector, perform an efficient priority encode, selecting the index (i.e. bit length) of the result associated with the longest slice which is greater than or equal to its base code. This effectively identifies the bit length of the codeword.
3. For the selected codeword length *n*, subtract the associated *n-*bit base code from the *n*-bit slice. This tells us the relative offset of the codeword within the set of codewords of the same length. Note that this has already been computed if the compares from step 1 are implemented with a subtraction (i.e. the >= compare succeeds if the sign bit of the subtraction result is 0). In that case, just select the result from the associated subtractor.
4. This relative offset is added to the *n*th SAT entry, which results in an index into the SLT. In order to reduce levels logic at the expense of more area, this can be pre-computed for each bit length by using carry-save arithmetic: slt\_index[*n*] = slice[26:27-*n*] – BCT[*n*] + SAT[*n*].
5. The SLT entry at this index contains the actual symbol (e.g. 0 to 575 for the XP9/XP10 short symbol alphabet). **If there are no valid Huffman codewords, this will manifest as an out-of range index into the SLT.**



Figure 11 Huffman Symbol Decoder

There are 2 instances of this logic in the pipeline: one for decoding XP9/XP10 short symbols or DEFLATE literal/length symbols, and the other for decoding XP9/XP10 long symbols or DEFLATE distance symbols.

Note that the BCT and SAT are small tables, but the SLT can be considerably larger. All must be implemented in flops because the tables need to be accessed by 32 independent decode lanes in parallel. If the layout of the associated logic has problems with congestion, multi-cycle-path constraints can be applied from the outputs of the BCT, SAT, and SLT storage flops because the data is guaranteed to remain stable for the duration of an entire Huffman block.

##### Short Symbol Interpret

This stage is responsible for determining how many (if any) extra bits there are associated with the 1st symbol, and for shifting the original bit slice by the number of bits consumed in preparation for the long symbol decode in the next stage.

For XP9/XP10, this stage determines whether or not there is a long symbol and/or extra bits to fully define the offset. However, since the encoder always packs the long symbol codeword after the short symbol codeword, the shift only accounts for the number of bits consumed by the short symbol, the extra bits needed to fully define the offset will appear AFTER the long symbol and additional length bits.

For DEFLATE, extra bits used to fully define the match length appear after the literal/length symbol, BUT before the distance symbol (DEFLATE ALWAYS has a distance symbol). In this case, the shift accounts both for the bit consumed by the short symbol, AND for the extra length bits (if any). The extra length bits are extracted and passed down stream.

##### Long Symbol Interpret

This stage is responsible for determining how many (if any) extra bits there are associated with the 2nd symbol, and for shifting the bit slice by the number of bits consumed.

For XP9/XP10, the decoded long symbol may be followed by extra length bits that are consumed in this stage.

For DEFLATE, the extra length bits have been passed down from the short symbol interpret stage.

##### Extra Offset/Distance

This stage consumes the extra bits used to fully define the offset/distance.

For both XP9/XP10 and DEFLATE, these bits are the last bits consumed before the LZ77 symbol is fully resolved.

#### Symbol Selector

The symbol selector takes all of the candidate symbols from the decode lanes and determines which symbols are valid symbols. In addition, the symbol selector reports the number of bits consumed from the stream and whether the last symbol from a block has been decoded, either due to an explicit EOB symbol or a known EOB position in the stream. A block diagram of the symbol selector is shown below.



Figure 12 Symbol Selector

The symbol selector receives the output of each of the 32 decoder lanes described above. This information includes: whether or not there is a valid symbol starting from that lane (since the decoder lane can fail to produce a valid symbol when the starting offset is invalid), the length of the symbol (includes 1 or 2 Huffman codewords and 0 to 2 extra bits fields), and the symbol value itself (including any extra offset and length bits).

Next, for each lane in parallel, the ending location of the decoded symbol (i.e. the starting offset of the next symbol) is computed by adding the lane number (i.e. its starting bit offset modulo 32) to the symbol length. The result of this computation is buffered one extra cycle such that the following stage can look across the decode results from 2 consecutive 32-bit slices at once. This is necessary to handle the case when the starting offsets of the symbols within the next set of 4 selected straddles 2 different 32-bit slices.

In the next stage, the symbol selector maintains a current bit offset relative to the next 32-bit slice. This is the starting bit offset of the symbol walker. The symbol walker uses the current bit offset as a mux select in order to select the decode lane associated with the next valid symbol. The ending location from that decode lane is used as the offset of the next valid symbol. This muxing logic cascades up to 4 times (or less if there are no more bits available to consume), in order to generate up to 4 symbols per clock cycle. The total number of bits consumed is added to the current bit offset. If this sum exceeds 31, this means that the current 32-bit slice has been totally consumed and can be “popped” from the head of the pipeline (i.e. assume all of the pipeline stages are fully interlocked).

When a slice is “popped”, 32 is subtracted from the current bit offset, such that the new offset will be relative to the next 32-bit slice. If a large number of bits are consumed (e.g. more than 64) in a single cycle, such that the updated current bit offset exceeds 31, the following 32-bit slices will effectively pass through without any symbols being generated, until the current bit offset falls back below 32.

For each 32-bit slice popped by the symbol walker, the total number of bits consumed is signaled back to the LFA. This can be 0 bits if a long symbol started in the previous slice straddled the boundary and completely consumed the current slice. This message will be accompanied by an EOB flag when a DEFLATE EOB symbol is detected, or when a 32-bit slice with an explicit EOB (i.e. for XP9/XP10) is encountered.

After an EOB symbol is encountered, the pipeline will switch into flush mode (i.e. popping incoming data without generating symbols) until an explicit SOB flag is encountered in the data. For XP9/XP10, the SOB should follow immediately after EOB or EOF, but for DEFLATE the pipeline may need to drain first (except when the LFA finishes a DEFLATE frame and know where the next frame starts).

Both EOB and EOF are propagated downstream to the Symbol Packer (and the MTF block after that).



Figure 13 Symbol Walker

#### Symbol Packer

The Symbol Packer takes in 1 to 4 Valid symbols and outputs up to 4 valid symbols. The output symbols are packed according to the constraints of the downstream blocks (MTF and LZ).

The Symbol Packer uses a Symbol Feeder-FIFO, which is the same design as the Bit Feeder FIFO but with symbols instead of bits. The Symbol Feeder FIFO presents a window to the packer logic and the packer logic examines the type fields of the symbols and determines how many symbols to pack into the output bus. The packer must maintain symbol ordering. A high-level block diagram of the Symbol Packer is given below.



Figure 14 Symbol Packer

The downstream blocks from the packer can handle up to 4 symbols per cycle, but only 1 of them can be a pointer. The packer logic optimally packs pointers and literals under these constraints. It simply packs symbols into the output bus until a second pointer is seen or the output bus is filled. An example is given below with a 6 symbol window, and 4 output symbols (N=4), considering the a data block with 6 symbols.



Figure 15 Symbol Packer Example

#### Scheduler Update Messages

The external logic responsible for scheduling traffic through the decompressor needs to do so without knowing how much output data will be generated by compressed input data at the time the data is sent. Therefore the scheduler logic needs to be notified about the size of the decompressed output data for each frame. This data is provided over the scheduler update interface described in the table in section 3.3.2. This interface also provides the amount of input data received for the frame so that the scheduler logic doesn’t have to maintain as much state in order to keep track of which frames the messages are associated with.

Each block within a frame results in a single scheduler update message (it’s preferable to provided more granular updates to the scheduler than simply waiting till the end of a potentially very long frame). Generally, there is an additional message at the end of each frame in order to account for partially consumed bytes after the last block (the internal counters keep track of bits consumed by the message is in terms of bytes), or mismatches between the number of decoded bits vs. the actual number of its in the frame which arise because of errors during the decode process. The EOF message may be omitted if there is no useful information (i.e. no remaining unaccounted for input/output bytes AND where the frame is NOT the last frame in a compound command).

Scheduler update messages are generated in the symbol packer after symbols are fully decoded. While a block within a frame is decoded, counters keeps track of the number of bits consumed from the input and the number bytes that will be generated on the output when the symbols are finally decoded by the LZ77 decompressor. When EOB is reached, a consumed bit count provided by the Front End Block is added to the number of bits consumed in order to account for any header data in the block which is not decoded by the HTF or the SDD. For the 1st block in a frame, this may include frame header bytes prior to but not explicitly belonging to the 1st block (including a user prefix). After the message is sent, where the amount of data is recorded in terms of bytes, the associated number of bits is subtracted from the counters keeping track of the number of bits consumed and generated. If the number of bits consumed is not a multiple of 8, the leftover bits are added to the total of the next block or rounded up to an additional byte at the end of the frame. When EOF is reached, the Front End Block sends the final number of input bytes for the entire frame (including all frame headers AND footers). If this value differs from the total number of bits counted inside the Symbol Data Decoder, the difference is accounted for by ***bytes\_in/basis*** in the final EOF scheduler update message. In the case where a decoder error causes the Symbol Data Decoder to consume MORE bits than are actually present in the frame, the difference is instead accounted for by ***bytes\_out*** in the final EOF scheduler update message.

### Errors

The Symbol Data Decoder may encounter the following errors:

* **E\_INVALID\_SYMBOL**  
  An invalid symbol is found at the next expected starting position. When a decode lane is fed a slice that does NOT start on a Huffman codeword boundary, it may or may not resolve a valid symbol. A failure to find a valid symbol will manifest as an out-of-range index into the SLT during the Huffman decode step. If, by chance, such a decode lane DOES resolve a valid symbol, either an invalid symbol will be encountered later in the block, or the E\_END\_MISMATCH error will occur.
* **E\_END\_MISMATCH**  
  For XP9/XP10, valid symbols go past the end of the expected EOB. The symbol selector knows exactly which bit the block ends on because the LFA signals the number of valid compressed data block bits in each transfer.
* **E\_MISSING\_EOB\_SYM**For DEFLATE, this occurs when the end of available data is reached (signaled by ***lfa\_sdd\_sob***) without decoding an EOB symbol.

#### EOB/EOF handling and related error scenarios

XP9/XP10 normal case  
At the end of each block, LFA raises EOB and signals remaining number of valid bits in the block. It can send SOB (with data 64-bit aligned) immediately on the next cycle. SDD should decode a symbol that ends EXACTLY on EOB boundary, or it will generate an error. SDD will also signal when it reaches EOF, which LFA uses in order to know which frames ack messages refer to. For XP9, there is no explicit last block indicator, so the LFA will always attempt to send additional bits following EOB to the block header parser. For XP10, data following CRC32/CRC64 after the final block is just considered to be additional padding and may or may not be flushed through the pipe to the SDD prior to EOF.

XP9/XP10 decode error  
If something goes wrong during decode in the SDD, the error flag in the ack message will be set. LFA uses this to know that the NEXT ack message from the SDD will be for the next frame. Since the error was first detected by the SDD rather than the LFA, the LFA will flush out the packet (either skipping to the end immediately or pushing all the data through) and signal EOF without an error, allow the SDD to fill in its own error code on the output to MTF. After encountering an error, SDD effectively ignores all incoming data until it sees an EOF flag, after which it encodes an EOF with the appropriate error code on its output bus.

DEFLATE normal case  
For non-final blocks, the LFA uses EOB signal from the SDD ack message in order to find the start of the next DEFLATE block header. For the final block, the LFA uses the EOB signal from the SDD ack message in order to find the start of the ZLIB Adler-32 or GZIP CRC32. The EOF is sent down the pipe to the SDD only after receiving acknowledgement for the final decoded EOB, and may or may not be preceded by additional padding data (which the SDD will ignore). The LFA sends the EOB flag to the SDD in order to indicate that there is no more data left in the frame. The SDD uses this to “truncate” the final parallel decode window instead of waiting for additional data.

DEFLATE missing EOB  
LFA will not send EOF to the SDD until it receives acknowledgement for the final EOB. If the final EOB is missing, the LFA will see that the SDD has consumed all of the available data without decoding an EOB, and thus will know to generate an error.

## Move to Front Replacer

### Overview

The Move to Front Replacer (MTF) in the decompressor operates on XP9/XP10 frames from the SDD, replacing MTF back-references with PTR back-references, using a cache the 4 most recently used LZ77 back-reference offsets. When a back-reference (PTR or MTF) is encountered, the associated offset is moved to the front of the cache, shifting the other entries down.

### Interfaces

Move to Front Replacer has the following interfaces:

* Block Header Parser Interface
  + MTF Header Info:  
    The Block Header Parser extracts the MTF offsets header from the XP9/XP10 block header and sends it to the MTF block.
* Symbol Data Decoder Interface
  + LZ77 Symbol Data Bus:  
    This interface is used to receive up to 4 symbols per clock from the SDD block, one of which can be a PTR/MTF back-reference. A ***framing*** signal specifies the number of valid symbols, and whether or not they are the final symbols within a compressed block or frame. EOF with errors is a special case indicated by a unique ***framing*** signal codepoint. No other symbols are transmitted during an EOF error transfer, and the remaining bits on the bus are used to signal error conditions.
* LZ77 Decompressor Interface
  + LZ77 Symbol Data Bus:  
    This interface has the same format as the SDD interface. The MTF block will only send PTR back-references downstream (i.e. never MTF back-references).

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **BHP to MTF Header Info Interface** | | | |
| bhp\_mtf\_valid | I | 1 | Indicates that the header info is ready. |
| mtf\_bhp\_ready | O | 1 | Indicates that the MTF block is ready to receive header info. |
| bhp\_mtf\_hdr\_bus.format | I | 1 | 0 = XP9  1 = XP10 |
| bhp\_mtf\_hdr\_bus.ptr\_last | I | 1 | This bit is set if the previous match is PTR or MTF. This is used only in Xpress9. |
| bhp\_mtf\_hdr\_bus.present | I | 1 | Indicates that MTF cache should be updated with new header info. Otherwise, retain the current cache state. |
| bhp\_mtf\_hdr\_bus.exp0 | I | 5 | MTF exponent0 |
| bhp\_mtf\_hdr\_bus.offset0 | I | 16 | MTF offset 0 |
| bhp\_mtf\_hdr\_bus.exp1 | I | 5 | MTF exponent1 |
| bhp\_mtf\_hdr\_bus.offset1 | I | 16 | MTF offset 1 |
| bhp\_mtf\_hdr\_bus.exp2 | I | 5 | MTF exponent2 |
| bhp\_mtf\_hdr\_bus.offset2 | I | 16 | MTF offset 2 |
| bhp\_mtf\_hdr\_bus.exp3 | I | 5 | MTF exponent3 |
| bhp\_mtf\_hdr\_bus.offset3 | I | 16 | MTF offset 3 |
| **SDD to MTF Symbol Data Interface** | | | |
| sdd\_mtf\_valid | I | 1 | Indicates that the symbol data is valid. |
| mtf\_sdd\_ready | O | 1 | Indicates that the MTF is ready to receive symbol data. |
| sdd\_mtf\_framing | I | 4 | Framing information:  [0,4] = 0 to 4 lanes valid  [5,9] = 0 to 4 lanes valid with EOB  15 = EOF (If there is an error, the 32-bit data carries error code instead of data) |
| sdd\_mtf\_data0 | I | 8 | Literal data or lower bits of back-reference offset:  for PTR: ptr\_offset[7:0]  For EOF with Error: 32 bits of error data |
| sdd\_mtf\_data1 | I | 8 |
| sdd\_mtf\_data2 | I | 8 |
| sdd\_mtf\_data3 | I | 8 |
| sdd\_mtf\_backref | I | 1 | Indicates that a back-reference is present |
| sdd\_mtf\_backref\_lane | I | 2 | Lane number containing the back-reference |
| sdd\_mtf\_backref\_type | I | 1 | Type of the back-reference:  0 = PTR  1 = MTF |
| sdd\_mtf\_offset\_msb | I | 8 | Upper bits of pointer offset |
| sdd\_mtf\_length | I | 16 | Back-reference length |
| **MTF to LZ77 Symbol Data Interface** | | | |
| mtf\_lz\_dp\_valid | I | 1 | Indicates that the symbol data is valid. |
| lz\_mtf\_dp\_ready | O | 1 | Indicates that the LZ77 is ready to receive symbol data. |
| mtf\_lz\_dp\_bus.framing | I | 4 | Framing information:  [0,4] = 0 to 4 lanes valid  [5,9] = 0 to 4 lanes valid with EOB  [10,14] = 0 to 4 lanes valid with EOF  15 = EOF with Error (32-bit data carries error code instead of data) |
| mtf\_lz\_dp\_bus.data0 | I | 8 | Literal data or lower bits of back-reference offset:  for PTR: ptr\_offset[7:0]  For EOF with Error: 32 bits of error data |
| mtf\_lz\_dp\_bus.data1 | I | 8 |
| mtf\_lz\_dp\_bus.data2 | I | 8 |
| mtf\_lz\_dp\_bus.data3 | I | 8 |
| mtf\_lz\_dp\_bus.backref | I | 1 | Indicates that a back-reference is present |
| mtf\_lz\_dp\_bus.backref\_lane | I | 2 | Lane number containing the back-reference |
| mtf\_lz\_dp\_bus.backref\_type | I | 1 | Type of the back-reference:  0 = PTR  1 = MTF |
| mtf\_lz\_dp\_bus.offset\_msb | I | 8 | Upper bits of pointer offset |
| mtf\_lz\_dp\_bus.length | I | 16 | Back-reference length |

### Detailed Description

#### Move to Front Algorithm

The LZ77 decompression block supports one back-reference lookup per clock, and this constraint also applies to the MTF logic. The input bus can have up to 4 symbols, only one of which can be a back-reference (MTF or PTR). When an MTF back-reference is presented on a symbol lane, the associated 2-bit index references the cache in order to obtain the PTR offset.

Both MTF and PTR back-references cause the cache to be updated, moving the associated offset to the front, and shifting all other entries down by 1. For MTF back-references, this update happens AFTER the cache is indexed to find the offset.

Doing a table update on an MTF back-reference means that an entry that already exists in the table is being pushed to the “front” of the table (meaning into MTF[0]). This may mean that some or all entries are not disturbed. For instance, a hit on MTF[0] means the associated offset is already at the front of the table and no updates are needed. A hit on MTF[1] would swap MTF0] and MTF[1], but the remaining entries would be undisturbed. A hit on MTF[2] would move the offset from MTF[2] to MTF[0], and would shift previous values in MTF[0] and MTF[1] down by one entry, but would not modify MTF[3]. A hit on MTF[3] would modify all entries as the offset at MTF[3] moves to the front of the table and shifts all other entries.



Figure 16 Example MTF update after MTF miss



Figure 17 Example MTF update after MTF hit

#### MTF Offsets Header

The front end of the decompressor extracts an MTF offsets header from each XP9/XP10 compressed data block and presents it to the MTF block. The MTF block always waits for this header information before processing the next block (either out-of-reset or after receiving EOB for the previous block). The MTF offsets header contains a 16-bit offset for each of the 4 entries of the cache. If it is not provided, the cache will retain its current state, which means that offset values from a previous block can be referenced.

It is expected that a new frame will always populate the table with PTR types and will not reference any “stale” entries from a previous frame. Note that since there is no way to represent an invalid entry in the MTF offsets header, there is no way to check if a stale entry is incorrectly referenced.

#### XP9 handling of MTF after back-reference

In XP9, special handling is required if an MTF symbol follows immediately after a back-reference. (e.g. lane 0 has an MTF and the highest valid lane during the previous transfer was either a PTR or MTF). In this case the lookup accesses one entry further back in the table (i.e. the entry associated with the immediately preceding back-reference is inaccessible), AND an MTF index of 3 is not allowed (i.e. an index of 3 causes an error). For example, an MTF index of 0 actually refers to MTF[1]. Similarly, an index of 1 actually refers to MTF[2], and an index of 2 actually refers to MTF[3].

The table update behavior is unchanged. This special handling does not apply to XP10.

## LZ77 Decompression

### Overview

There are two main components to LZ77 decompression, the incoming compressed data and the already decompressed history buffer.

* The incoming data to be decompressed can either be a literal or a pointer. Literals do not need any further processing. Pointers are back references that are used to represent repeating patterns from the already decompressed data. A pointer is an {offset, length} pair that indicates the number of bytes equal to the length to copy over from the offset in the history buffer.
* The already decompressed data is maintained in the history buffer. The history buffer has a sliding window of size 64KB. As new incoming data is decompressed, the window slides to include the newer decompressed data such that the configured size of the history buffer is maintained throughout. At the beginning of each frame, the history buffer will be empty.

The literals are a byte long and the pointers are 4 byte symbols made up of 2, 16 bit fields (offset and length). The maximum sliding window size is 64KB, so 16 bit length and offsets are necessary to represent all valid values. Literals are copied over as is to the history buffer and sent out the output streaming interface. On receipt of the pointer, the history buffer will be read from the location indicated by the offset until the number of bytes read is equal to the length. The corresponding bytes will be copied over to the history buffer for future back references and also sent out on the output streaming interface.

### Interface Signals

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **MTF <-> LZ77** | | | |
| mtf\_lz\_dp\_valid | I | 1 | Indicates that the symbol data is valid. |
| mtf\_lz\_dp\_bus.framing | I | 4 | Framing information:  [0,4]: 0 through 4 lanes valid  [15] EOF with error codes. Error code will be in the mtf\_lz\_data0/1/2/3. |
| mtf\_lz\_dp\_bus.data0 | I | 8 | Literal data or lower bits of back-reference offset  For PTR : ptr\_offset[7:0]  For EOF with Error, 32 bits of data carries error code instead of data. |
| mtf\_lz\_dp\_bus.data1 | I | 8 |
| mtf\_lz\_dp\_bus.data2 | I | 8 |
| mtf\_lz\_dp\_bus.data3 | I | 8 |
| mtf\_lz\_dp\_bus.backref | I | 1 | Indicates that a back-reference is present. |
| mtf\_lz\_dp\_bus.backref\_lane | I | 2 | Lane number containing the back-reference. |
| mtf\_lz\_dp\_bus.offset\_msb | I | 8 | Upper bits of pointer offset. |
| mtf\_lz\_dp\_bus.length | I | 16 | Back-reference length |
| lz\_mtf\_dp\_ready | O | 1 | Transaction level ready  1 : LZ77 block is ready to accept data  0 : Backpressure to the MTF block to pause sending data. |
| **Front End Frame Parser -> LZ77** | | | |
| fhp\_lz\_prefix\_hdr\_valid | I | 1 | Valid Indicator |
| fhp\_lz\_prefix\_hdr\_bus.data\_sz | I | 7 | Size in 1K chunks |
| fhp\_lz\_prefix\_hdr\_bus.prefix\_type | I | 1 | 0: pre-defined  1: user defined |
| fhp\_lz\_prefix\_hdr\_bus.trace\_bit | I | 1 | Trace bit indicator |
| lz\_fhp\_prefix\_hdr\_ready | O | 1 | LZ77 block is ready to accept prefix information on SOF. If all three the 1K prefix buffers are in use, ready will be de-asserted until at least one of them frees up. |
| **Frame Parser <-> LZ77** | | | |
| fhp\_lz\_prefix\_valid | I | 1 | Frame Parser to LZ77 prefix data valid |
| fhp\_lz\_prefix\_dp\_bus.data | I | 64 | Frame Parser to LZ77 data |
| fhp\_lz\_prefix\_dp\_bus.sof | I | 1 | Start of prefix data. |
| fhp\_lz\_prefix\_dp\_bus.last | I | 1 | Last beat of prefix data. |
| fhp\_lz\_prefix\_dp\_bus.prefix\_type | I | 1 | 0: pre-defined  1: user defined |
| lz\_fhp\_pre\_prefix\_ready | O | 1 | LZ77 is ready to accept predetermined prefix data |
| lz\_fhp\_usr\_prefix\_ready | O | 1 | LZ77 is ready to accept user prefix data. |
| **Frame Header Parser to LZ77 Debug Path** | | | |
| fhp\_lz\_dbg\_data\_valid | I | 1 | Debug data path from FHP to LZ is valid |
| fhp\_lz\_dbg\_data | I | 64 | 64 bit data. |
| lz\_fhp\_dbg\_dp\_ready | O | 1 | LZ is ready to accept debug data |
| **Decompressed Data Output** | | | |
| lz\_be\_dp\_bus.data | O | 64 | Decompressed data out |
| lz\_be\_dp\_bus.bytes\_valid | O | 8 | Bytes valid in the decompressed output. Indicates if the corresponding byte data is valid.  lz77\_be\_bytes\_vld[0] if 1, indicates lz77\_be\_data[7:0] is valid,  lz77\_be\_bytes\_vld[1] if 1, indicates lz77\_be\_data[15:8] is valid and so on. |
| lz\_be\_dp\_bus.type | O | 2 | Type of Data out  2’b00 : Invalid  2’b01 : Data Only  2’b10 : EOF with Data  2’b11 : EOF with Error |
| lz\_be\_dp\_valid | O | 1 | Valid indicator for the data out. |
| be\_lz\_dp\_ready | I | 1 | 1 : Backend is ready to receive data from LZ77  0 : Backpressure to LZ77 to stop sending data. |

### LZ77 Decompression Block Diagram



Figure 18 LZ77 Decompression Block

#### Overview of the LZ77 Decompression block

The LZ77 decompression block can receive up-to 4 entries to be decompressed per clock. Each entry can either be a literal or a pointer. There is a limitation however, that there can only be one pointer per clock. This is enforced via the Symbol Data Decoder’s output to the MTF block. As entries are received by the LZ77 decompression block, they are written into the Input FIFO. The input FIFO is read when the Entry processing block is able to process new entries.

The Entry processing block is the main engine of the decompression block. If the entry processed is a literal, it is sent directly to the Byte Merge block. If the entry processed is a pointer, the offset and length in the pointer is used to read the appropriate bytes from the history buffer. This block will generate as many reads to the history buffer as necessary depending on the length obtained from the pointer. As a read command is issued to the history buffer, it is also responsible for calculating the start offset and end offset in the history buffer word to copy over to the output.

This block also has visibility into the number of decompressed bytes that are not yet written into the history buffer since we only write the history buffer in 16 byte increments.

The pointer generation block maintains head (start) and tail (end) pointers to the history buffer. These are circular pointers such that the 64KB sliding window is maintained at all times. When a new entry is added, the tail pointer increments and when the window size exceeds the configured limit during a write, the head pointer will be incremented.

The Byte Merge block will merge the literal data, bytes from the history buffer or the latest word and write out to the Decompressed output block in 8 byte words.

The Decompressed Output block has two odd and even FIFOs for the 8-byte words from the Byte Merge block. The 8 byte words are ping-ponged between the two FIFOs, starting with the odd, then even and back to odd. The odd/even fifos are alternatively read and the 8 bytes are sent out of the output bus. In addition to the odd/even FIFOs, there is another 8 byte wide FIFO to store the partial 8 byte word until 16 bytes are available to be written to the history buffer.

Certain compression formats like XP10 allow including prefix data that is available as a history buffer before the start of LZ77 decompression of a frame. The prefix data can either be user-defined or pre-defined. In either case, the prefix data is sent as part of the Project Zipline frame. The prefix data is extracted from the frame and sent to the LZ77 block by the Front end frame parser. There are three 1K prefix buffers in the LZ77 decompression block, since upto 3 frames can be in transit in the decompression block. In the case of pre-defined prefix, the prefix data will reside in the 1K buffer. In the case of user-defined prefix, the first 1K of user data will be loaded in the 1K buffer and the rest will be loaded in the history buffer after the current frame processing is over. Since it can only pre-load upto 1K, the Front End block will be paused until the current frame processing is over before rest of the prefix data can be accepted.

There is more information on how the prefix and history buffers are used for lookup in section ??.

#### Prefix Buffer and Control

On receipt of a frame, the front end frame parser indicates to the LZ77 block if the corresponding frame has prefix data associated with it. The prefix data will be referenced from the 1K dedicated prefix buffer in the case of pre-defined prefix and from the prefix and history buffer in the case of user-defined prefix.

On receipt of a frame, the front end parser sends the prefix information to the LZ77 prefix control block. There are three 1 entry FIFOs in the Prefix Control block, one for the Input Header, one for holding the next entry and other for the Data control. The presence of an entry in the Input Header FIFO indicates that the prefix data is being processed by the LZ77 block. Once the prefix data is processed, the entry is moved to the Hold fifo if that is empty. If the Data fifo is empty, the Hold fifo entry is moved to the Data fifo. The presence of an entry in the Data Header fifo indicates that the LZ77 is currently decompressing a frame.

For every frame received by the Front End block, the prefix information is written into the Input Header FIFO. Once all the prefix data has been accepted, if the hold fifo is empty, the header fifo entry is moved to the hold fifo. If the Data Header fifo is empty, the entry from the hold fifo is moved to the Data fifo. On the receipt of an SOF in the data stream, the data from the Data Header FIFO is applied to the frame. The data indicates if the frame has prefix data and if present the type of prefix data as well as the size of the prefix data.

On the receipt of an EOF in the data stream, the Data FIFO is popped and the next entry from the hold FIFO is ready to be moved to the Data FIFO.



Figure 19 Prefix Control and Prefix/History Buffer

##### Pre-defined prefix

* The Input Header FIFO is written with the information about the prefix data, whether it is present and if so, which of the three prefix buffers it is written into.
* Once all the 1K prefix data is loaded into the prefix buffer, the Data Header FIFO’s state is monitored and when it is empty, the hold FIFO entry is moved to the Data FIFO.
* On the receipt of an SOF, the data from the Data Header FIFO entry is applied to the frame. The prefix data length (always 1K) is sent to the Entry Processing block and the Address generation block so that the Current position pointer and head/tail pointers can be adjusted.
* On receipt of an EOF in the data stream at the output of the Entry Processing block, the Data Header FIFO entry is popped.
* The pre-defined prefix data is used as part of the history buffer. As the window size grows, the prefix data is rolled out of the window first.
* While a frame with prefix data is being processed, the prefix processor is free to process any prefix data for the next frame as long as the Input Header FIFO is empty.

##### User defined prefix

If the format is user defined prefix, the size of the data can be anywhere from 1 KB to 64KB (up-to the size of the window). The user defined data is always in multiples of 1KB. The user defined data is always copied to the history buffer, before it can start processing the frame associated with it.

* The Input Header FIFO is written with the prefix information.
* Since we have three copies of the prefix buffer, up-to 1K of the user-defined prefix data can be written into the prefix buffer. If the user defined data length exceeds 1K, the front end parser has to be paused until the current frame processing is complete. The rest of the prefix data beyond 1K is written into the history buffer.
* After all the user prefix data is written into the history buffer, the entry from the hold FIFO is moved to the Data FIFO along with the prefix information and the Current Position Pointer (POS). This is the current offset in the history buffer.
* This will start the LZ decompression of the current frame with the user defined data.

#### Input FIFO

The LZ77 decompression block receives 4 entries per clock to be decompressed. The entries are first written into the input FIFO. They are popped from the FIFO if the Entry processing block is ready to process the 4 entries.

Each entry can either be a literal or a pointer. If the entry is a literal, it is 1 byte wide. If it is a pointer, it can be up-to 4 bytes long. The entries are processed LSB -> MSB, i.e, lane 0 entry is fully processed first, followed by lane 1 and so on. Since pointer processing requires reading the history buffer to copy from previously written decompressed data, it might take more than a clock. Until all four entries per clock are fully processed, the input FIFO is not read.

If the input FIFO depth is beyond the configured pause threshold level, the ready signal to the MTF block is de-asserted there by pausing it from sending more data to be decompressed.

The Input FIFO is sized at 32 entries deep.

#### Entry processing

As the input FIFO is read, this block processes the four entries per read data. The entries are processed LSB first. Since a pointer might depend on the bytes decompressed from the previous entry, the next entry is processed only when the current entry is fully decompressed.

Literals don’t need any further processing. They are sent as-is to the Byte Merge block.

If the entry is a pointer, the history buffer or the prefix buffer has to be read multiple times depending on the length indicated in the pointer. The address and the bytes from each word in the history buffer (or prefix buffer) to copy over from will depend on the offset in the pointer. If the offset exceeds the number of bytes in the history buffer and if the prefix buffer is valid, the start address will be in the prefix buffer. Otherwise, it will be in the history buffer.

The frame header indicates if the prefix data has to be used for a particular frame or not.

This block maintains the current position pointer (POS), which is the number of bytes decompressed in the current frame. The offset indicated in the pointer is always with respect to this POS. In addition, it maintains the number of bytes copied over to the history buffer. As a result, it also has the number of bytes decompressed, but not yet written into the history buffer.



Figure 20 Lane Literal/Pointer Processing

##### Parameters maintained in the Entry Processing Block

* **Current Position Pointer (POS)**

The Current Position Pointer keeps track of the offset of the latest decompressed word. To start with, this will be 0 and as literals or bytes from the pointer decode are sent to the Byte Merge block, this will be incremented by the number of bytes sent. This includes the bytes written to the history buffer and the recently decompressed bytes not yet written to the history buffer. The Current Position Pointer will be decremented by 16 when the Pointer block indicates that the head pointer moved by one to maintain the sliding window size. Note, if prefix data is enabled for a frame, the POS will be initialized to the size of the prefix data at the start of frame processing.

* **History Buffer Offset (HBO)**

The History Buffer Offset keeps track of the number of bytes written into the history buffer memory. Every time a word is written to the history buffer, this will be incremented by 16.

On a write to the history buffer, if the head pointer is also incremented with the tail pointer to maintain the window size, the HBO is not incremented. Note, if prefix data is enabled for a frame, the HBO will be initialized to 1K at the start of frame processing.

* **Latest Word Length (LWL)**

This is the number of bytes yet to be committed into the text buffer. This will include the bytes that are still being merged to form a 16 byte word as well as the previous two words in the pipleline that are not yet written into the history buffer.

##### Parameters calculated during a pointer processing (History Buffer Read)

History buffer data is packed into MSB -> LSB, with the oldest bytes in the MSB. The latest word will be treated as if it was the last word of the history buffer. POS will include the prefix buffer if prefix data is included for the frame.

* **Bytes to read from history buffer**  bytes\_offset = POS – pointer\_offset

if (bytes\_offset < HBO) begin

hb\_read = 1’b1;

bytes\_to\_read = HBO – bytes\_offset;

end

else begin

hb\_read = 1’b0;

bytes\_to\_read = pointer\_offset;

//everything we need is in the latest word that is not yet written into the history buffer.

// See section 3.5.3.4.3

end

* **Start Offset in history buffer word to Read from (start\_offset)**

Depending on the pointer offset, all or part of the first history buffer word will be copied.

If (first read from history buffer after pointer decode)

start\_offset = (bytes\_to\_read mod 16) -1;

else

start\_offset = 15 //we are reading from MSB.

* **End Offset in history buffer word to Read to (end\_offset)**

If (curr\_length > (start\_offset+1)) //see below for curr\_length description

end\_offset = 0;

else

end\_offset = (start\_offset +1) – curr\_length);

* **Current Length (curr\_length)**

The number of bytes to copy over from the history buffer is indicated by the length in the pointer. As words are read from the history buffer and the appropriate number of bytes copied over from the history buffer, a curr\_length is maintained which represents the number of bytes still to copy over from the history buffer. A read command to the history buffer will be issued as long as the curr\_length is greater than 0 and we haven’t reached the end of the history buffer.

##### Parameters while copying from the latest word

Under the two conditions indicated below, the latest word will have the bytes to copy.

* All the bytes to be copied over from a pointer decode are from the latest word.
* If the pointer offset is less than the length, it indicates that the number of bytes equal to the offset should be copied over repeatedly until the number of bytes copied is equal to the pointer length. If the offset extends into the history buffer, the bytes from history buffer are copied over as explained before. After the history buffer read, if the pointer offset is less than the length of the latest word, the following parameters are calculated and sent to the Byte Merge block. The Byte Merge block will be responsible for repeatedly copying over the bytes from the latest word until the bytes copied is equal to the copy\_length.
  + **copy\_offset**

This is the same as the pointer offset. From the POS, go back copy\_offset bytes and start the copy.

* **copy\_length**

This is the number of bytes left to copy. Will repeat copying over from copy\_offset until bytes copied is equal to the copy\_length.

If the FIFOs in the Decompressed output buffer block are full, the entry processing will pause until there is space available for the decompressed entries. This might happen when there is a backpressure to the LZ77 Decompression block pausing reads from the FIFOs.

#### Address Generation

The history buffer is a sliding window of 64KB. This block maintains the head and tail pointers such that the sliding window size is maintained and the newer entries are included in the history buffer as old entries are excluded.

The head pointer points to the oldest word in the history buffer and the tail pointer points to the newest word in the history buffer. If the history buffer size is equal to the configured size, the head pointer will move down as new entries are added to maintain the window size. The oldest word will be excluded from the window to include the newest word.

The head and tail pointers are circular buffers and they roll over as they reach the maximum address in the physical memory.

Every time, the head pointer increments to maintain the sliding window size, the Entry processing block will be notified so that the current position pointer (POS) will be decremented by 16 and the history buffer offset is not incremented.

If a frame has prefix data associated with it, the prefix buffer is included in the window. So, the pre-defined prefix entries will roll out of the window as more pointers get decompressed.

The address to read from is calculated differently based on whether this is the first read to the history buffer for a particular pointer.

* **Base Addr (base\_addr)**

This is the address to start the read from based on the pointer offset. This is calculated for the first read to the history buffer for a pointer.

If ((tail\_ptr > head\_ptr) || ((tail\_ptr – words\_to\_read) >= 0 )

base\_addr = (tail\_ptr – words\_to\_read)

else

base\_addr = ((window\_sz + tail\_ptr+1) – (words\_to\_read))

* **Current Address (curr\_addr)**

This is the actual address to the history buffer to read from.

If (first read after pointer decode)

curr\_addr = base\_addr

else

curr\_addr = prev\_addr + 1 //curr\_addr rolls over if it reaches the window\_sz

#### Byte Merge

The Entry processing block will make sure that there is only one valid input at a time. The valid inputs can be either literals (up-to 4 bytes) or data (up-to 16 bytes) from the history buffer. The byte merge block packs the valid data it receives in 8 byte words and forwards it to the Decompressed output block.

Since each history buffer read can result in a maximum of 16 bytes, two 8 byte words can be sent to the Decompressed output block every clock. The offset of the bytes to read from each word will be sent by the Entry Processing block.

The byte merge block also has a copy of the latest word that is not yet written into the history buffer. Since the Entry Processing block maintains the current buffer offset and history buffer offset, it will indicate if the offset it sends corresponds to the history buffer or the latest word.

If the copy\_length from the Entry Processing block is greater than zero, the copy\_offset is used as the offset in the current word to copy from repeatedly until the length of bytes is equal to the copy\_length. Please see the Appendix 4 for an example.

Since the EOF control signal is part of the lane data, the Entry processing block will flush the latest word out on the EOF indicator.

#### Decompressed Output Buffer



There are two 16 deep odd and even FIFOs in this block. The 8 bytes of data from the Byte Merge block is ping-ponged to the two FIFOs. The odd FIFO is written first followed by the even FIFO and back to odd and so on.

When there is an entry in either one of the FIFOs, the FIFO is read and the entry is sent out on the output bus.

When both odd and even words from Byte Merge are available, the 16 byte word is sent to the history buffer. If the byte merge block sends an odd word in the first clock followed by the even word at a later time, the odd word will be saved in a separate 2 deep FIFO until the even word arrives and both can be sent out to the history buffer. Since the even word can be accompanied by the next odd word, the fifo is sized to be 2 deep to save the second odd word as we are reading out the previous odd word to be sent to the history buffer. The FIFO is completely decoupled from the odd/even FIFOs so that the history buffer can be quickly written as soon as 16 bytes are available.

The LZ77 block can be flow controlled by the back end processor. The odd/even FIFOs are drained continuously as entries are written in as long as the LZ77 is not flow controlled.

When the odd/even FIFO levels are beyond a certain configured limit, the Byte Merge and Entry processing blocks will be flow controlled. This will result in the input FIFO not being read. If the Byte Merge is in the middle of a copy, that will be paused and if the Entry processing block is in the middle of a pointer decode resulting in history buffer reads, that will be paused as well.

On the receipt of an EOF, the FIFOs will be flushed until all the data bytes corresponding to the frame are sent on the output bus. Since the history buffer is initialized for each frame on a SOF, the last EOF bytes, that are not a multiple of 16, will not be written to the history buffer, but will be sent out on the output bus.

### Local Statistic Counters

These counters will be maintained locally in the LZ77 decompression block. See section 3.7.4 for a more comprehensive list of statistics counters.

|  |  |  |
| --- | --- | --- |
| **Counter Name** | **Description** | |
| decomp\_byte\_cnt[63:0] | Number of bytes decompressed by this block | The decomp\_byte\_cnt should equal the mem\_wr\_cnt + current\_word\_cnt |
| mem\_wr\_cnt[63:0] | Number of 16 byte words written |
| curr\_word\_cnt[15:0] | Number of bytes decompressed but not yet written into the history buffer. |
| tail\_pointer | Points to where the tail pointer is pointing to in the history buffer | |
| head pointer | Points to where the head pointer is pointing to in the history buffer. | |

### Error Tokens to Back end

The error indicators are sent through the data bus along with the data type EOF if there are no other prior errors associated with the frame. Only the first errors are propagated to the back end processor. But, all the errors are captured in status registers.

See the Project Zipline headers document for more information on the breakdown of the fields in the token bus for error propagation.

|  |  |
| --- | --- |
| **Error** | **Description** |
| E\_HISTORY\_SOFT\_OVERFLOW | Back reference distance exceeds amount of history accumulated. |

### Debug Error Registers (Maskable Interrupts)

|  |  |
| --- | --- |
| **Signal** | **Description** |
| in\_fifo\_ovflow\_error | Input FIFO overflow. Indicates problem in lz77\_mtf\_bp generation. |
| odd\_fifo\_ovflow\_error | Decompressed Output odd FIFO overflow. Indicates problem in back pressure to the Entry Processing block. |
| even\_fifo\_ovflow\_error | Decompressed Output even FIFO overflow. Indicates problem in back pressure to the Entry Processing block. |
| output\_fifo\_ovflow\_error | Decompressed Output FIFO overflow. Indicates problem in back pressure to the Entry Processing block. |

## Back End Block

### Overview

The Decompressor Back End block combines the decompressed data from the LZ77 block with the header and footer data from the Front End block. The Front End’s Frame Header Parser Block, parses the Project Zipline frame and forwards the various Project Zipline TLVs that are not consumed by the Decompression block. There is a header FIFO that stores the header data and a footer FIFO that stores the footer data.

The Front End block extracts the CRC/Adler32 from the incoming compressed frame and forwards it to the Back End block. The CRC/Adler32 is calculated over the decompressed data and validated against the value received from the Front End Block.

The first error (if present) is propagated to the Back End with the EOF as the frame makes its way through various blocks. The first error is inserted in the frame footer before it is sent out of the Decompression Block.

Both input paths (from the LZ77 block and the Front End Parser) can be back-pressured with a ready/valid handshake.

### Interface Signals

|  |  |  |  |
| --- | --- | --- | --- |
| ***Signal*** | ***I/O*** | ***Width*** | ***Description*** |
| clk | I | 1 | Block Clock. 800 Mhz |
| rst\_n | I | 1 | Asynchronous block reset. Active Low |
| **Latency FIFO/Data Aligner -> Back End Block** | | | |
| lfa\_be\_crc\_valid | O | 1 | CRC/Adler valid |
| lfa\_be\_crc\_bus.frm\_format | O | 3 | 3’b000 : XP9  3’b001 : XP10\_CRC32  3’b010 : XP10\_CRC64  3’b011 : ZLIB  3’b100 : GZIP  3’b101 : Deflate  3’b110 : CHU with CRC  3’b111 : CHU without CRC |
| lfa\_be\_crc\_bus.input\_size | O | 32 | Indicates the original uncompressed data size. Valid for XP9 and ZLIB only. |
| lfa\_be\_crc\_bus.data | O | 64 | The frame format in the footer is used to determine if the data has the Adler or CRC.  [63:0] if CRC64. CRC32/CRC64 for XP10 frame indication is in the frame header.  [31:0] if CRC32  [31:0] if Adler32 |
| **Decompressed Data Output** | | | |
| lz\_be\_dp\_bus.data | I | 64 | Decompressed data |
| lz\_be\_dp\_bus.bytes\_valid | I | 8 | |  | | --- | | Bytes valid in the decompressed output. Indicates if the corresponding byte data is valid.  lz77\_be\_bytes\_vld[0] if 1, indicates lz77\_be\_data[7:0] is valid,  lz77\_be\_bytes\_vld[1] if 1, indicates lz77\_be\_data[15:8] is valid and so on. | |
| lz\_be\_dp\_bus.type | I | 2 | Type of Data out  2’b00 : Invalid  2’b01 : Data only  2’b10 : EOF with Data  2’b11 : EOF with Error |
| lz\_be\_dp\_valid | I | 1 | Valid indicator for the data out. |
| be\_lz\_dp\_ready | O | 1 | 1 : Backend is ready to receive data from LZ77  0 : Backpressure to LZ77 to stop sending data. |
| **Back End Output** | | | |
| xp10\_decomp\_ob\_out.tid | O | 1 | Block Identifier. Unique ID forwarded from the front-end. Will be sent out with each frame. |
| xp10\_decomp\_ob\_out.tvalid | O | 1 | Data valid |
| xp10\_decomp\_ob\_out.tlast | O | 1 | Last beat of the frame |
| xp10\_decomp\_ob\_out.tdata | O | 64 | 64 bit dataout |
| xp10\_decomp\_ob\_out.tsrtb | O | 8 | Data bytes valid on the bus  0: Bits [7:0]  1: Bits [15:8]  …  7: Bits [63:56] |
| xp10\_decomp\_ob\_out.tuser | O | 8 | 8 bits defined; undefined bits are reserved. OB\_TUSER[0] : End of block. |
| xp10\_decomp\_ob\_in.tready | I | 1 | 0 : Downstream block is not ready to receive decompressed data.  1 : Downstream block is ready to receive data. |

### Detailed Description

The Front End Block’s FHP (Frame Header Parser) forwards the pass through TLVs to the Back End Block. The pass through TLVs include the RQE, CMD, FRMD, FOOTER, STATS, CQE.

* The frame format, CRC/Adler and Input size are also forwarded to the Back End block for every frame being processed by the Front End’s LFA. If the frame format does not include CRC (XP9) or does not have the Input size, those fields will be don’t care.
* The decompressed data from the LZ77 block is forwarded to the Back End block as well. Any errors if present, will be forwarded by each block with the EOF. Only the first error will be propagated with the EOF.

The Back End block instantiates the TLV parser re-assembler and the AXI Master. The TLV parser re-assembler has two main interfaces :

* User TLV interface. Decompressed data from the LZ77 is sent via this interface.
* Pass Thru TLV interface. Pass thru TLVs are sent via this interface. The first error will be inserted in the FOOTER TLV before it is sent to the TLV parser re-assembler. The decompressed frame size will also be inserted in the FOOTER TLV before it is sent to the re-assembler.
* The re-assembler assembles the TLVs in order for a particular frame and forwards it to the AXI master which finally sends the frame out the AXI interface.

#### CRC/Adler checks

The Front End Parser when it parses the frame, extracts the CRC32/64/ or Adler32 and forwards it to the Back End block. Every frame will have a CRC word forwarded even if there is no valid CRC as part of the frame. This is to make sure that the correct CRC word aligns with the frame data.

As the LZ77 decompressed data is forwarded to the user TLV, the CRC/Adler is calculated and checked against the value received from the Front End Block.

#### Decompressed word size check

The XP9 header and GZIP trailer has the input size in bytes which is extracted by the Front End block and forwarded to the Back End block. As the decompressed data is forwarded to the user TLV parser interface the bytes forwarded is updated and finally checked against the value received from the Front End block.

### Errors

The following errors are checked and first error indicator inserted in the footer of the frame. Frames with errors will not be dropped, but forwarded with the first error information set in the footer.

* Frame CRC error – If the expected CRC/Adler32 does not match the value extracted from the incoming frame by the Front End Block.
* Output bytes exceeded the limit – If the size of decompressed bytes for a frame exceeds the configured limit. For example, if the OUTPUT\_SIZE is set to 8K and the decompressed bytes exceeds that value, this error is set.
* Decompressor bytes mismatch with the input size specified in the XP9 header or the GZIP footer – The Front End block sends the input size to the Back End block. If the decompressed size does match the value sent by the Front End block, this error is set.

## Debug Features

The decompressor provides the following debug features:

* An interface monitor will capture the output of the Symbol Data Decoder, allowing it to be read directly by software
* Table monitors will allow software to read the decoder table writes output by the Huffman Table Formatter.
* Raw LZ77 symbols, indicated by a special TLV, can be sent directly to the LZ77 decompresssor in order to facilitate independent testing of that block.
* A set of statistics and performance counters, each of which may be configured to count one of many different internal events.

### Interface Monitors

Interface monitors allow data exchanged between decompressor sub-blocks to be captured in a buffer and read by software. They can be configured to capture a single initial burst of data (i.e. fill the buffer once), to continuously overwrite the buffer with new data, or to halt the transfer of data over the connected interface when the buffer is full, allowing transfers to resume once software has read the data.

The trace buffer is viewed by software as an indirect access memory, with each memory entry storing a transfer on the interface. Transfers are stored at incrementing addresses. Availability of data is indicated by an interface monitor status register. An interface monitor control register is used by software to signal when buffered data can be overwritten with new data. See the interface monitor documentation for more details on the software interface.

The rest of this section will describe specifically where interface monitors are used in the decompressor (i.e. test/sample points).

#### Huffman Decoder and MTF Output Monitor

In order to facilitate testing and debug of the Decompressor pipeline from header parsing through to Huffman decoding and MTF replacement, an interface monitor is attached to the output of the MTF replacer. The data buffered this interface monitor is a stream of LZ77 symbols (i.e. LITERAL, PTR, or MTF), with up to 4 symbols per data transfer (see the associated interface description in section 3.4.2). Figure 1 shows a **DBG** block monitoring the output of MTF. The interface monitor buffers 64-bit per transfer on the interface. The size of the buffer is 512 data transfers.

#### Back End Output Monitor

In order to facilitate testing and debug of the Decompressor as a whole, an interface monitor is attached to the AXI4-Stream interface at the output of the Back End block. Figure 1 shows a **DBG** block monitoring the output of the Back End block. The interface monitor buffers 8 bytes per transfer and additional AXI4-Stream sideband signals on the interface. The size of the buffer is 512 entries.

### Table Monitors

In order to facilitate testing and debug of the Huffman Table Formatter, table monitors are used to allow software to read the contents of the BCT, SAT, and SLT decoder tables.

A table monitor has the same software interface as an interface monitor, but the internal implementation differs. Unlike an interface monitor, a table monitor doesn’t necessarily need any dedicated storage, because the hardware already requires the associated data to be stored for internal use. Conversely, an interface monitor requires a dedicated trace buffer because storage of the interface transfers is not otherwise required by the internal hardware. A table monitor is well-suited for monitoring the Huffman decoder tables.

A single table monitor software indirect access interface is used to monitor contents of the BCT, SAT, and SLT decoder tables for the SS (also DEFLATE literal/length), LL (also DEFLATE distance), SmallSS (also DEFLATE code-length), and SmallLL symbol alphabets generated by the Huffman Table Formatter. This is particularly well suited for the SS and LL decoder tables because their use is synchronized in time (i.e. by the Symbol Data Decoder). The use of the SmallSS and SmallLL tables is NOT time-synchronized, but they are small enough to be buffered and presented to software at the same time as the larger tables. All of the tables are concatenated together such that the software can view it as a single large table as follows:

* The first 27 entries contain BCT/SAT values such that the entry at index *n* contains the following values for bit-length *N*+1 (these are all packed into the same “word”):
  + LL\_valid[*n*]
  + LL\_SAT[*n*][7:0]
  + LL\_BCT[*n*][26:0]
  + SmallLL\_valid[*n*]
  + SmallLL\_SAT[*n*][5:0]
  + SmallLL\_BCT[*n*][7:0]
  + SS\_valid[*n*]
  + SS\_SAT[9:0]
  + SS\_BCT [*n*][26:0]
  + SmallSS\_valid[*n*]
  + SmallSS\_SAT[*n*][5:0]
  + SmallSS\_BCT[*n*][7:0]

SmallSS/SS/SmallLL/LL\_valid[*n*] indicate that that there is at least 1 symbol with the associated bit length. Small BCT/SAT never have any valid entries above index 7 (i.e. above bit-length 8).

* The next 576 entries contain SLT values such that the entry at index 27+*n* contains the SLT values at index *n*:
  + LL\_SLT[*n*][7:0]
  + SmallLL\_SLT[*n*][5:0]
  + SS\_SLT[*n*][9:0]
  + SmallSS\_SLT[*n*][5:0]

Note that entries 27+248=275 and above do not contain valid LL SLT values because the LL SLT table only has 248 entries. Entries 27+33=60 and above do not contain valid SmallSS/SmallLL values because the associated SLTs have only 33 entries.

Similarly, for DEFLATE, entries 27+29=56 and above do not contain valid distance alphabet SLT values. Entries 27+285=212 and above do not contain any valid length/literal or distance alphabet SLT values. Entries 27+19=56 and above do not contain valid code-length alphabet SLT values. Finally, SmallLL\_SLT[*n*] never contains valid values for DEFLATE because there is only 1 code-length alphabet, unlike XP9/10 which has both SmallSS and SmallLL.

### Huffman Decoder Bypass

Upon receipt of a frame containing a specific TLV (TBD), the Frame Header Parser will wait for any outstanding frames to flush through past the MTF replacer and then activate a mux select at the input to the LZ77 decompressor, interpreting the raw frame data as directly encoded LZ77 symbols, according to the format shown in section 3.5.2. This facilitates stand-alone testing of the LZ77 decompressor without Huffman decoding.

The front end will send the appropriate frame header and footer information to the back-end in order to guarantee correct operations (and no lockups), but it will not send any data to the block header parser, latency FIFO and data aligner, or Huffman table formatter.

### Performance and Statistics Events

The Decompressor outputs an event bus which will be routed to a global performance and statistic monitor unit. The global performance and statistics monitor has a limited set of counters, but can configure each one to capture one of many different possible events across the There is a bit on the bus for each one of these events, and the occurrence of the event causes the bit to be strobed for a single cycle. The list of events and associated bits on the event bus is as follows:

|  |  |
| --- | --- |
| **Originating Sub-Block** | **Event Description** |
| Frame Header Parser | XP9 Frame |
| Block Header Parser | XP9 Coding Block |
| Frame Header Parser | XP9 Uncompressed Frame |
| Frame Header Parser | XP10 Frame |
| Frame Header Parser | XP10 Frame with Prefix |
| Frame Header Parser | XP10 Frame with Predetermined Huffman |
| Block Header Parser | XP10 Coding Block |
| Block Header Parser | XP10 Uncompressed Block |
| Frame Header Parser | GZIP Frame |
| Block Header Parser | GZIP Coding Block |
| Block Header Parser | GZIP Uncompressed Block |
| Frame Header Parser | ZLIB Frame |
| Block Header Parser | ZLIB Coding Block |
| Block Header Parser | ZLIB Uncompressed Block |
| Block Header Parser | CHU 4K |
| Block Header Parser | CHU 8k |
| Block Header Parser | CHU 4K RAW |
| Block Header Parser | CHU 8K RAW |
| Frame Header Parser | Prefix CRC Error |
| Frame Header Parser | Pre-determined Huffman CRC Error |
| Block Header Parser | XP9 Header CRC Error |
| Frame Header Parser | Stalls Generated (i.e. backpressure to input interface) |
| Huffman Table Formatter | XP9 Simple Encoded Blocks |
| Huffman Table Formatter | XP9 Retrospective Huffman Encoded Blocks |
| Huffman Table Formatter | XP10 Simple Encoded Blocks |
| Huffman Table Formatter | XP10 Retrospective Huffman Encoded Blocks |
| Huffman Table Formatter | XP10 Predetermined Huffman Encoded Blocks |
| Huffman Table Formatter | DEFLATE Fixed Encoded Blocks |
| Huffman Table Formatter | DEFLATE Dynamic Huffman Encoded Blocks |
| Huffman Table Formatter | Stalls generated on predetermined Huffman interface |
| Huffman Table Formatter | Stalls generated on header data interface |
| Huffman Table Formatter | Stalls generated on header info interface |
| Latency FIFO/Data Aligner | Stalls generated on data input interface |
| Symbol Data Decoder | 1 literal transfers |
| Symbol Data Decoder | 2 literal transfers |
| Symbol Data Decoder | 3 literal transfers |
| Symbol Data Decoder | 4 literal transfers |
| Symbol Data Decoder | 1 MTF transfers |
| Symbol Data Decoder | 1 literal, 1 MTF transfers |
| Symbol Data Decoder | 2 literal, 1 MTF transfers |
| Symbol Data Decoder | 3 literal, 1 MTF transfers |
| Symbol Data Decoder | 1 PTR transfers |
| Symbol Data Decoder | 1 literal, 1 PTR transfers |
| Symbol Data Decoder | 2 literal, 1 PTR transfers |
| Symbol Data Decoder | 3 literal, 1 PTR transfers |
| Symbol Data Decoder | MTF 0 … 3 |
| Symbol Data Decoder | XP9/XP10 PTR bins 0 … 15 |
| Symbol Data Decoder | XPI/XP10 long symbols |
| Symbol Data Decoder | DEFLATE distance bins 0 through 13 extra bits |
| Symbol Data Decoder | Stalls generated on LFA interface |
| MTF | PTR offset hit in MTF cache |
| LZ77 Decompressor | Frames in |
| LZ77 Decompressor | Frames out |
| LZ77 Decompressor | Number of pointers received |
| LZ77 Decompressor | Number of Literals received in Lane 0 |
| LZ77 Decompressor | Number of Literals received in Lane 1 |
| LZ77 Decompressor | Number of Literals received in Lane 2 |
| LZ77 Decompressor | Number of Literals received in Lane 3 |
| LZ77 Decompressor | Match with length 3 |
| LZ77 Decompressor | Match with length 4 |
| LZ77 Decompressor | Match with length 5 |
| LZ77 Decompressor | Match with length 6 |
| LZ77 Decompressor | Match with length 7 |
| LZ77 Decompressor | Match with length 8 |
| LZ77 Decompressor | Match with length 9 |
| LZ77 Decompressor | Match with length 10 |
| LZ77 Decompressor | Match with 11 <= length < 32 |
| LZ77 Decompressor | Match with length 32 <= length < 64 |
| LZ77 Decompressor | Match with length 64 <= length < 128 |
| LZ77 Decompressor | Match with length 128 <= length < 256 |
| LZ77 Decompressor | Match with length 256 <= length |
| LZ77 Decompressor | Stalls generated on MTF inteface |

## Table of Errors in the Decompression Block

The following are the error conditions checked for in the Decompression block. The first error will be propagated through the data pipeline with the EOF. The first error code will also be inserted in the Project Zipline footer before it is sent out of the decompression block.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Error ID** | **Error Code** | **Description** | **Error Insertion Strategy** |  |
| **Common Errors in the Block** | | | | |
| 1 | HD\_MEM\_ECC | ECC error from the SRAMs in the design. | Inserted via BIMC |  |
| **Frame Header Parser** | | | | |
| 2 | HD\_FHP\_PFX\_CRC | CRC check failed for the prefix data. | Data XOR of data bytes in the prefix TLV |  |
| 3 | HD\_FHP\_PFX\_DATA\_ABSENT | Project Zipline header indicates the presence of prefix data, but the prefix TLV is not present. | Inserted via Testbench |  |
| 4 | HD\_FHP\_PHD\_CRC | CRC check failed for the pre-determined Huffman data. | Data XOR of data bytes in the pre-determined Huffman TLV. |  |
| 5 | HD\_FHP\_BAD\_FORMAT | If FRMD TLV indicates frame is parsable and the frame header parsing does not result in a valid frame format. | Inserted via Testbench |  |
| **Block Header Parser** | | | | |
| 6 | HD\_BHP\_INVALID\_WSIZE | Window size received does not match the accepted sizes. | Inserted via Testbench |  |
| 7 | HD\_BHP\_BLK\_CRC | Block-level header crc check failed | Data XOR of any header data byte/bytes. | CRC error |
| 8 | HD\_BHP\_HDR\_INVALID | Invalid block header | Inserted via Testbench |  |
| 9 | HD\_BHP\_XP9\_HDR\_SEQ | Block header sequence is incorrect in the XP9 header. | Inserted via testbench. |  |
| 10 | HD\_BHP\_XP10\_XTRA\_FLAG\_PRSNT | XP10 frame header received with the FLG\_EXTRA bit set | Inserted via Testbench |  |
| 11 | HD\_BHP\_ZLIB\_FDICT\_PRSNT | Zlib format file uses predefined dictioniary. | Inserted via Testbench |  |
| 12 | HD\_BHP\_GZ\_CM\_NOT\_DEFLATE | Gzip or Zlib file header has the compression mode marked as something other than “Deflate” | Inserted via Testbench |  |
| 13 | HD\_BHP\_ZLIB\_CINFO\_RANGE | Zlib file header has CINFO field > 7 | Inserted via Testbench |  |
| 14 | HD\_BHP\_ZLIB\_FCHECK | Fcheck of ZLIB header failed | Inserted via Testbench |  |
| 15 | HD\_BHP\_DFLATE\_LEN\_CHECK | Raw deflate block Length check failed | Inserted via Testbench |  |
| 29 | HD\_BHP\_STBL\_SIZE\_ERR | Symbol table size returned by HTF does match the size indicated in the XP9 header. |  |  |
| 37 | HD\_BHP\_ILLEGAL\_MTF | Illegal MTF offset received |  |  |
| **Latency FIFO/Data Aligner** | | | | |
| 16 | HD\_LFA\_REWIND\_FAIL | Requested rewind data no longer exists in the FIFO. Adequate rewind space should always be preserved. This should only happen if the communicated end is wrong. | Inserted via Testbench. No Adler32 or CRC32 present after the EOB resulting in no REWIND data to send after EOB ack from SDD. |  |
| 17 | HD\_LFA\_PREMATURE\_EOF | Compressed data size indicates there is more data in the block, but got an EOF indicator for the frame. | Data XOR of EOF footer that has the EOF code set.  . |  |
| 18 | HD\_LFA\_LATE\_EOF | Last blk indicator is set, reached end of compressed data based on compressed size in the frame, but there is more data in the frame. | Data XOR of the last block indicator in the penultimate block of the XP10 frame |  |
| 19 | HD\_LFA\_MISSING\_EOF | Last block indicator not set, but received a runt block instead of a proper block header. |  |  |
| **Huffman Table Formatter** | | |  |  |
| 20 | HD\_HTF\_XP9\_RESERVED\_SYMBOL\_TABLE\_ENCODING | Invalid encode mode for xp9 reserved encoding |  |  |
| 21 | HD\_HTF\_XP10\_RESERVED\_SYMBOL\_TABLE\_ENCODING | Invalid encode mode for xp10 reserved encoding |  |  |
| 22 | HD\_HTF\_XP10\_PREDEF\_SYMBOL\_TABLE\_ENCODING | Invalid encode mode for xp10 pre-determined Huffman encoding when frame header did not specify a predefined table. |  |  |
| 23 | HD\_HTF\_XP9\_ILLEGAL\_NONZERO\_BL | Found a non-zero bit length for short symbol between 576 and 703. |  |  |
| 24 | HD\_HTF\_RLE\_OVERRUN |  |  |  |
| 25 | HD\_HTF\_BAD\_HUFFMAN\_CODE | Huffman parser encountered an invalid bit sequence (this is possible in the case of incomplete codes) | Inserted via testbench by sending string of 1s for Huffman-encoded symbol table. Not possible if the Huffman tree is perfectly balanced with power-of-2 valid symbols. |  |
| 26 | HD\_HTF\_ILLEGAL\_SMALL\_HUFFTREE | Encoded small table is corrupt |  |  |
| 27 | HD\_HTF\_ILLEGAL\_HUFFTREE | Symbol table is corrupted. |  |  |
| 28 | HD\_HTF\_HDR\_UNDERRUN | Symbol table was truncated in the frame. |  |  |
| **Symbol Data Decoder** | | | | |
| 32 | HD\_SDD\_INVALID\_SYMBOL | An invalid symbol is found at the next expected starting position. When a decode lane is fed a slice that does NOT start on a Huffman codeword boundary, it may or may not resolve a valid symbol. A failure to find a valid symbol will manifest as an out-of-range index into the SLT during the Huffman decode step. If, by chance, such a decode lane DOES resolve a valid symbol, either an invalid symbol will be encountered later in the block, or the E\_END\_MISMATCH error will occur. | Inserted via testbench by sending string of 1s for Huffman-encoded compressed data. Not possible if the Huffman tree is perfectly balanced with power-of-2 valid symbols. |  |
| 33 | HD\_SDD\_END\_MISMATCH | For XP9/XP10, valid symbols go past the end of the expected EOB. The symbol selector knows exactly which bit the block ends on because the LFA signals the number of valid compressed data block bits in each transfer. | Inserted via testbench or via data XOR by corrupting OUTPUT\_SIZE in block header by +/- 1. |  |
| 34 | HD\_SDD\_MISSING\_EOB\_SYM | For DEFLATE, this occurs when the end of available data is reached (signaled by lfa\_sdd\_sob) without decoding an EOB symbol. | Inserted via testbench. |  |
| MTF | | | | |
| 35 | HD\_MTF\_XP9\_MTF3\_AFTER\_BACKREF | Found a encoded MTF3 after a valid pointer symbol. |  |  |
| 36 | HD\_MTF\_XP10\_MISSING\_MTF | MTF reference obtained when no MTF pointers are in the cache |  |  |
| **LZ77 Decompression** | | | | |
| 38 | HD\_LZ\_HBIF\_SOFT\_OFLOW | Back reference distance exceeds amount of history accumulated | Inserted via Testbench |  |
| **Back End Block** | | | | |
| 39 | HD\_BE\_FRM\_CRC | The CRC or ADLER sent with the frame did not match the expected value. | Data XOR of the CRC data. |  |
| 40 | HD\_BE\_OLIMIT | The number of decompressed bytes exceeds the configured limit. | Testbench configuration. | Abort frame when limit is exceeded. |
| 41 | HD\_BE\_SZ\_MISMATCH | For XP9 and ZLIB, the input byte count does not match the actual number of decompressed bytes. | Data XOR of the input size header bytes. |  |

### Obsolete Errors (removed from the original list)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Error ID** | **Error Code** | **Description** | **Error Insertion Strategy** |  |
|  | E\_HEADER\_SOF\_MISMATCH | Mismatch between the SOF indicator in the data transaction bus and the SOF header data. | Inserted via Testbench | Not valid anymore |
|  | E\_PREDET\_DATA\_ABSENT | Project Zipline header indicates the presence of pre-determined Huffman data, but the pre-determined TLV is not present. | Inserted via Testbench | HTF returns invalid symbol table encoding |
|  | E\_HISTORY\_BUFFER\_TOO\_SMALL | This block’s history buffer is too small to decode the file (based on info in the LZ77 window size field of the compression TLV). | Config register for history buffer compare. Needs to be part of RTL build parameters.    Testbench drives invalid window size that will not work with the RTL parameter. | Not valid  Any more. Will hit this in the HTF with out of range LZ pointers. |
|  | E\_FILE\_HEADER\_INVALID | File header cannot be parsed or is invalid (this will happen if the SOF compression type field is set incorrectly). | Inserted via Testbench | Not valid anymore; caught in the FHP |
|  | E\_HEADER\_MISSING | A header needed to decompress the stream is missing (possible if the XP10 Header/Footer control bit is set to not insert headers in the stream) | Inserted via Testbench | Will result in HTF decode errors & CRC error. Don’t need this |
|  | E\_BLOCK\_HEADER\_INVALID | Block header cannot be parsed or is invalid. | Inserted via Testbench | Will result in HTF decode error & CRC error. Don’t need this. |
|  | E\_HEADER\_FIELD\_NOT\_ZERO\_BLOCK | Non-zero data encountered in a header field that should be zero | Data XOR of reserved fields in the header. | No explicit checking is done.  Not valid |
|  | E\_FRAME\_SIZE\_ERROR | If the frame size indicated in the Project Zipline TLV does not match the actual frame size. | Data XOR of the data frame size TLV. | Data TLV size might always not be known. Not valid anymore. |
|  | E\_HISTORY\_HARD\_OVERFLOW | Back reference distance exceeds size of history buffer. If the history buffer size is 16K and the back reference distance exceeds 16K. | Inserted via Testbench | Not valid anymore. Caught in the HTF if the pointer size exceeds the window size config. |

## SRAM Estimate

|  |  |  |  |
| --- | --- | --- | --- |
| **Block** | **Memory Function** | **Size (DepthxWidth)** | **Num of Instances** |
| Front End Block | LFA (Latency FIFO/Data Alinger) | 1Kx68 | 1 |
| LZ77 | History buffer | 4Kx128 | 1 |
| LZ77 | Prefix Data Buffer | 64x128 | 2 |
| Huffman Table Formatter | Pre-determined bit-length buffering | 130x64 | 1 |
| Huffman Table Formatter | Symbol Table Buffer | 125x64 | 1 |
| MTF Output Interface Monitor | Data transfer buffer | 256x64 | 1 |
| Back-end Output Interface Monitor | Data transfer buffer | 256x64 | 1 |

## Flop Estimate

|  |  |  |  |
| --- | --- | --- | --- |
| **Block** | **Function** | **Flop Count** | **Num of Instances** |
| Front End Block | Frame Header Parser (input FIFO) | 8x66 | 1 |
| LFA/SOF fifo | 4x80 | 1 |
| LFA/Header Aligner | 8x70 | 1 |
| LFA/Data Aligner | 8x70 | 1 |
| BHP/Header fifo | 8x70 | 1 |
| BHP/HTF fifo | 8x66 | 1 |
| BHP/Hdrinfo fifo | 2x66 | 1 |
| Miscellaneous | 3000 | 1 |
|  | | | |
| LZ77 | Input FIFO | 32x64 | 1 |
| Input/Symbol fifos | 16x64 | 4 |
| Prefix hdr fifos | 1x11 | 3 |
| HB temp FIFO | 2x64 | 1 |
| Odd/Even | 16x69 | 2 |
| Miscellaneous | 3000 |  |
|  | | | |
| Back End Block | Header FIFO | 8x64 | 1 |
| Data FIFO | 32x69 | 1 |
| Passthru FIFO | 8x106 | 1 |
| Footer FIFO | 64x106 | 1 |
| CRC FIFO | 8x64 | 1 |
| Miscellaneous | 1000 |  |
|  | | | |
| Huffman Table Formatter | Small BLT | 33x4 | 1 |
| Small BCT | 28 | 1 |
| Small SAT/Pointers | 7x6 | 2 |
| Small SLT | 33x6 | 1 |
| Large BLT | 576x5 | 1 |
| SS Pointers | 26x10 | 1 |
| LL Pointers | 26x8 | 1 |
| SS SLT Write FIFOs | 14x20 | 2 |
| LL SLT Write FIFOs | 14x16 | 2 |
| Miscellaneous | 2000 | 1 |
|  | | | |
| Symbol Data Decoder | BCT | 377 | 4 |
| SS SAT | 26x10 | 2 |
| LL SAT | 26x8 | 2 |
| SS SLT | 576x10 | 2 |
| LL SLT | 248x8 | 2 |
| Small BCT (for monitor) | 28 | 2 |
| Small SAT (for monitor) | 7x6 | 2 |
| Small SLT (for monitor) | 33x6 | 2 |
| Symbol Packer | 4x64 | 1 |
| Lane Decoder | 500 (important to optimize this) | 32 |
| Window Feeder | 512 | 1 |
| Miscellaneous | 4000 | 1 |
|  | | | |
| MTF | Miscellaneous (from MSFT estimate) | 1000 | 1 |

# Appendix

## LZ77 Decompression example

In this example, the history buffer is empty and the input to the X10 LZ77 decompression block is abc{3,6}. “abc” are literals and {3,6} is a pointer. The format of the pointer is {offset, length}. So, {3,6} indicates, go back 3 bytes from current position and copy 6 bytes. Since we only have 3 bytes to copy, the remaining 3 bytes of the 6 bytes will be copied over from data that was just decompressed. This has the same effect of copying from the offset to the end of the current decompressed data and copying over again from the offset until the bytes copied has reached the length.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Byte Offset | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| First 3 literals copied as is | a | b | c |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pointer {3,6}. 1st word copy | a | b | c | a |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2nd word copy | a | b | c | a | b |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3rd word copy | a | b | c | a | b | c |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4th word copy | a | b | c | a | b | c | a |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5th word copy | a | b | c | a | b | c | a | b |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6th word copy | a | b | c | a | b | c | a | b | c |

The final decompressed result will be “abcabcabc”.

## Pointer processing Example 1

The following example illustrates how various parameters are calculated while processing a pointer entry. The example in this case is a pointer entry {35, 34}. The offset is 35 and the length is 34.

Assume 40 bytes have been decompressed so far in the current buffer. 32 bytes have been written to the history buffer. The history buffer looks like this before decoding/decompressing the pointer{35,34}.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **History Buffer** | | | | | | | | | | | | | | | | |
| Word 0 (head\_ptr) | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c | 2d | 2e | 2f |
| Word 1 (tail\_ptr) | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c | 3d | 3e | 3f |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 8 valid bytes** | | | | | | | | | | | | | | | |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |  |  |  |  |  |  |  |  |

Table 8 Initial Parameters on a pointer decode

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description/Comments** |
| Current Position Pointer (POS) | 40 | Current pointer offset indicates the number of bytes decompressed in the current buffer. Every time an entry is processed, this is incremented by the number of bytes decompressed. |
| History Buffer Offset (HBO) | 32 | History buffer offset indicates how many decompressed bytes are written into the history buffer. This is incremented by 16 on a write to the history buffer. |
| Latest Word Length (LWL) | 40 – 32 = 8 | This is the length in bytes of the temporary word that is not yet written into the history buffer since it is only written in 16 byte words |
| Byte\_end\_offset\_read\_from | 40 - 35 | 5 |
| Bytes\_to\_read (from TB) | 32 - 5 = 27 | HBO – byte\_end\_offset\_read\_from  Number of bytes to read from the history buffer. This will be used to calculate the base addr to start the reads from. |
| extra\_word\_to\_read | 27 mod 16 is greater than 0, so 1 |  |
| words\_to\_read | 27/16 + extra\_word\_to\_read = 1 + 1 =2 | This is the number of 16 byte words to read from memory. |
| current\_length | 34 |  |
| base\_addr | (0x01 – (2 -1)) = 0x00 | We are counting back words\_to\_read and counting up until the length is reached |
| curr\_addr | 0x00 |  |

Table 9 First Read from the history buffer

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description/Comments** |
| current\_length | 34 |  |
| curr\_addr | 0x00 |  |
| start\_offset | (27 mod 16) – 1 = 10 | (bytes\_to\_read mod 16) -1 |
| end\_offset | 0 | (current\_length > (start\_offset +1) ? 0  else (start\_offset+1) – current\_length |
| bytes\_read | (10 -0) + 1 = 11 | (start\_offset – end\_offset) +1 |
| current\_length | 34 – 11 = 23 | current\_length – bytes\_read |

This is how it will look like at the end of first read in processing the pointer {35, 34}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **History Buffer** | | | | | | | | | | | | | | | | |
| Word 0 (head) | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c | 2d | 2e | 2f |
| Word 1 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c | 3d | 3e | 3f |
| Word 2 (tail) | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Current Word (not yet written to the history buffer) 3 valid bytes** | | | | | | | | | | | | | | | | |
| 2d | 2e | 2f |  |  |  |  |  |  |  |  |  |  |  |  |  |

Table 10 Second Read from the history buffer

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description/Comments** |
| current\_length | 23 | 34 -11 = 23 |
| curr\_addr | 0x01 | curr\_addr = curr\_addr +1 |
| start\_offset | 15 | Subsequent reads is 15 |
| end\_offset | 0 | (current\_length > (start\_offset +1) ? 0  else (start\_offset+1) – current\_length |
| bytes\_read | (15 -0) + 1 = 16 | (start\_offset – end\_offset) +1 |
| current\_length | 23 - 16 = 7 | current\_length – bytes\_read |

This is how it will look like at the end of second history buffer read of processing the pointer {35, 34}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **History Buffer** | | | | | | | | | | | | | | | | |
| Word 0 (head) | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c | 2d | 2e | 2f |
| Word 1 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c | 3d | 3e | 3f |
| Word 2 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c |
| Word 3 (tail) | 2d | 2e | 2f | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Current Word (not yet written to the history buffer) 3 valid bytes** | | | | | | | | | | | | | | | |
| 3d | 3e | 3f |  |  |  |  |  |  |  |  |  |  |  |  |  |

Table 11 Third Read from the history buffer

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description/Comments** |
| current\_length | 7 | 23 – 15 = 7 |
| curr\_addr | 0x02 | curr\_addr = curr\_addr +1 |
| start\_offset | 15 | Subsequent reads is 15 |
| end\_offset | (15+1) – 7 = 9 | (current\_length > (start\_offset +1) ? 0  else (start\_offset+1) – current\_length |
| bytes\_read | (15 -9) + 1 = 7 | (start\_offset – end\_offset) +1 |
| current\_length | 7 – 7 = 0 | current\_length – bytes\_read. |

This is how it will look like at the end of third history buffer read of processing the pointer {35, 34}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **History Buffer** | | | | | | | | | | | | | | | | |
| Word 0 (head) | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c | 2d | 2e | 2f |
| Word 1 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c | 3d | 3e | 3f |
| Word 2 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 15 | 16 | 17 | 18 | 19 | 2a | 2b | 2c |
| Word 3 (tail) | 2d | 2e | 2f | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3a | 3b | 3c |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Current Word (not yet written to the history buffer) 10 valid bytes** | | | | | | | | | | | | | | | | |
| 3d | 3e | 3f | 30 | 31 | 32 | 33 | 34 | 35 | 36 |  |  |  |  |  |  |

The processing stops here since the current\_length is 0.

## Pointer Processing Example 2

In this example, the pointer that is being processed is {3, 9}. The offset is 3 and the length of bytes to copy is 9. For the sake of simplicity, let’s assume the LWL is 8.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 8 valid bytes** | | | | | | | | | | | | | | | |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |  |  |  |  |  |  |  |  |

First, set the offset to 3 back from the POS (Current Position Pointer). Copy until the end.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 11 valid bytes** | | | | | | | | | | | | | | | |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 35 | 36 | 37 |  |  |  |  |  |

The length of the bytes copied is 3.

We repeat the same thing as before until the length copied is equal to 9.

Go back 3 and copy until end.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 14 valid bytes** | | | | | | | | | | | | | | | |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 35 | 36 | 37 | 35 | 36 | 37 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 16 valid bytes** | | | | | | | | | | | | | | | |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 35 | 36 | 37 | 35 | 36 | 37 | 35 | 36 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Latest Word (not yet written to the history buffer) 1 valid byte** | | | | | | | | | | | | | | | |
| 37 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. htf\_sdd\_ss\_\* signals correspond to short table, and htf\_sdd\_ll\_\* correspond to long table. [↑](#footnote-ref-2)
2. htf\_sdd\_ss\_\* signals correspond to short table, and htf\_sdd\_ll\_\* correspond to long table. [↑](#footnote-ref-3)