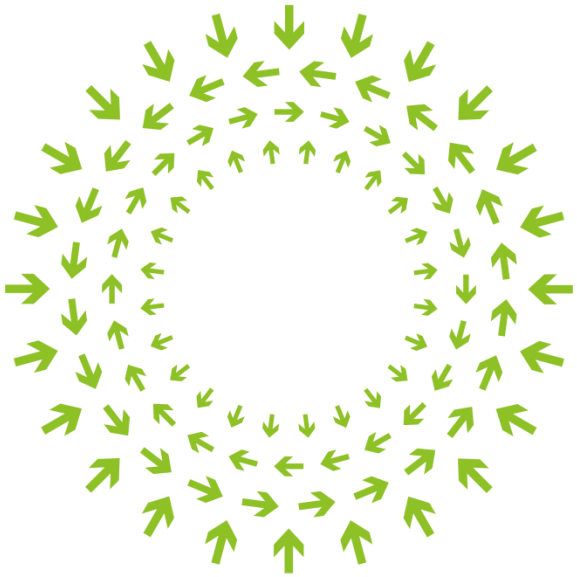


OPEN

Compute Project

Project Zipline – Huffman Encoder Micro Architecture Specification

Authors:
Microsoft Corporation
Broadcom Corporation

Revision History

Date	Description
03/11/2019	Version 1.0
	-

License

Contents

Table of Figures

Table of Tables

1 Huffman Encoder Top Level

1.1 Overview

The Huffman Encoder is responsible for the Huffman encoding of the LZ77 output. The encoder will break the incoming data frames into blocks (referred to as Huffman blocks throughout) and encode them in either the XP or DEFLATE format as specified in the Compression Header TLV. A pass-through mode is also supported. Depending on the encode format specified as well as other directives in the Compression Header TLV, the encoder will evaluate several encoding options (e.g. Retrospective, Predetermined, RAW) and select the option which results in the smallest overall output stream size.

The Verilog RTL being contributed refers to XP10 instead of Project Zipline. To remain consistent with the source code, XP10 will be referred to in this document.

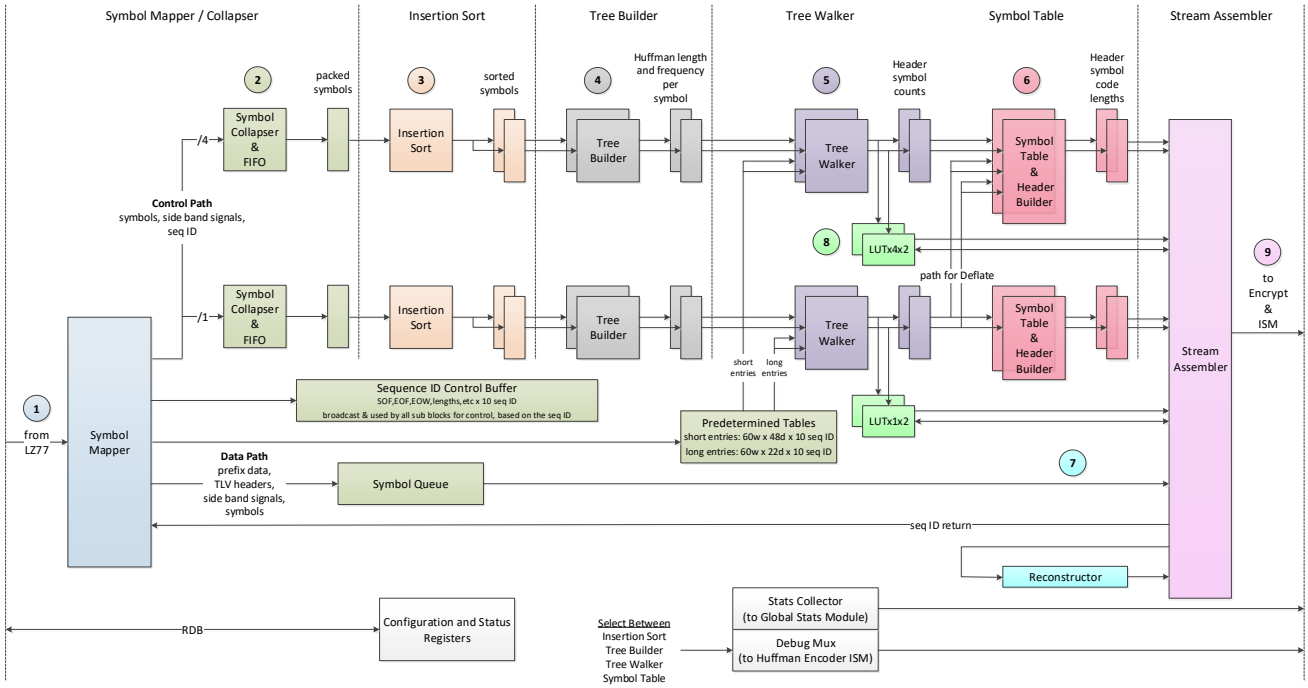


Figure 1 : Huffman Encoder Block Diagram

1.2 Interface Description

Name	I/O	Description
Clocks, resets and test		
clk	I	800MHz clock
rst_n	I	Active low reset
scan_rst_n	I	Scan test reset
scan_mode	I	Scan test mode
scan_en	I	Scan test shift enable
ovstb	I	Memory test over-strobe
mlvm	I	Memory test signal
lvm	I	Memory test signal
Interrupts		
huf_comp_int.tlvp_err	O	Internal protocol error
huf_comp_int.uncor_ecc_err	O	Uncorrectable ECC error
huf_comp_int.bimc_int	O	Memory controller interrupt
Top level, from LZ77 Encoder		
huf_comp_ib_in.tvalid	I	Bus Qualifier • 0 – not valid • 1 – valid
huf_comp_ib_in.tlast	I	End of frame signaling for pass through mode, where lz77_henc_tlv_data[63:0] will be packed with 8-bytes
huf_comp_ib_in.tstrb[7:0]	I	End of frame byte valids for pass through mode
huf_comp_ib_in.tuser[7:0]	I	Multi-purpose user bits, used for signaling start and end of frames and sub-frames.
huf_comp_ib_in.tid	I	Transaction ID
huf_comp_ib_in.tdata[63:0]	I	TLV Header Data, depends on the Token Type Note, the “LZ77 Payload” TLV header is described in Table 4.
huf_comp_ib_out.tready	O	Huffman Encoder block cannot accept any more data from the LZ77 Encoder
Top level, to Encryption Block		
huf_comp_ob_in.tready	I	The Encryption Block cannot accept any more data from the Huffman Encoder
huf_comp_ob_out.tvalid	O	Bus Qualifier • 0 – not valid • 1 – valid
huf_comp_ob_out.tdata[63:0]	O	TLV Header Data, depends on the Token Type
huf_comp_ob_out.tstrb[7:0]	O	Byte valids

huf_comp_ob_out.tuser[7:0]	O	Multi-purpose user bits, used for signaling start and end of frames and sub-frames.
huf_comp_ob_out.tid	O	Frame ID
huf_comp_ob_out.last	O	Last beat of frame
Top level, to Scheduler		
su_ready	I	Scheduler interface ready
huf_comp_sch_update.valid	O	Scheduler update valid
huf_comp_sch_update.rqe_sched_handle[15:0]	O	Identifier from the RQE TLV
huf_comp_sch_update.last	O	Last update in a frame
huf_comp_sch_update.tlv_frame_number[10:0]	O	Frame number from the RQE TLV
huf_comp_sch_update.tlv_eng_id[3:0]	O	Engine ID from the RQE TLV
huf_comp_sch_update.tlv_seq_num[7:0]	O	Sequence from the RQE TLV
huf_comp_sch_update.byte_in[23:0]	O	Frame or block input byte count
huf_comp_sch_update.byte_out[23:0]	O	Frame or block output byte count
huf_comp_sch_update.basis[23:0]	O	Frame or block input raw byte count
RDB Interface		
cfg_start_addr[19:0]	I	Hardwired start of address space for Huffman Encoder block
cfg_end_addr[19:0]	I	Hardwired end of address space for Huffman Encoder block
rbus_ring_i.addr[19:0]	I	RDB ring address input
rbus_ring_i.wr_strb	I	RDB ring write strobe input
rbus_ring_i.wr_data[31:0]	I	RDB ring write data input
rbus_ring_i.rd_strb	I	RDB ring read strobe input
rbus_ring_i.rd_data[31:0]	I	RDB ring read data input
rbus_ring_i.ack	I	RDB ring ack input
rbus_ring_i.err_ack	I	RDB ring error input
rbus_ring_o.addr[19:0]	O	RDB ring address output
rbus_ring_o.wr_strb	O	RDB ring write strobe output
rbus_ring_o.wr_data[31:0]	O	RDB ring write data output
rbus_ring_o.rd_strb	O	RDB ring read strobe output
rbus_ring_o.rd_data[31:0]	O	RDB ring read data output
rbus_ring_o.ack	O	RDB ring ack output
rbus_ring_o.err_ack	O	RDB ring error output

Stats Accumulator Output		
huf_comp_stat_events[63:0]	O	Statistic strobes sent to the global Statistics Accumulator block. See Table 35 for details.
huf_comp_XP10_decomp_lz77d_stat_events[63:0]	O	Reconstructor specific statistic strobes sent to the global Statistics Accumulator block. See Decompressor document for details.
Huffman Internal ISM		
ism_consumed_huf[1:0]	I	Huff Comp inter-stage monitor diagnostic ready signal from CPU
ism_consumed_he_st_sh[1:0]	I	Huffman Encoder Engine inter-stage monitor diagnostic ready signal from CPU
ism_consumed_he_st_lng[1:0]	I	Huffman Encoder Engine inter-stage monitor diagnostic ready signal from CPU
ism_consumed_he_sh[1:0]	I	Huffman Encoder Engine inter-stage monitor diagnostic ready signal from CPU
ism_consumed_he_lng[1:0]	I	Huffman Encoder Engine inter-stage monitor diagnostic ready signal from CPU
ism_available_huf[1:0]	O	Huff Comp inter-stage monitor diagnostic ready signal to CPU
ism_available_he_st_sh[1:0]	O	Huffman Encoder Engine inter-stage monitor diagnostic ready signal to CPU
ism_available_he_st_lng[1:0]	O	Huffman Encoder Engine inter-stage monitor diagnostic ready signal to CPU
ism_available_he_sh[1:0]	O	Huffman Encoder Engine inter-stage monitor diagnostic ready signal to CPU
ism_available_he_lng[1:0]	O	Huffman Encoder Engine inter-stage monitor diagnostic ready signal to CPU

1.3 Processing Flow

The following section lists the flow through the Huffman Encoder for a frame. The numbering corresponds to the color coded numbers in Figure 1.

1. When a frame enters the Huffman encoder (always beginning with a Compression Header TLV received on the input interface), the Compression Header TLV is sent to the Symbol Mapper for parsing. The Symbol Mapper will use this information to break each frame into blocks (if necessary), and then guide each block through the pipe and ensure each block has the correct control signals it needs to properly process the block.

2. Prefix data may follow the Compression Header TLV. Prefix data is used in upstream units to seed the history buffer and enable possible matches at the start of a frame. The encoder writes any prefix data into the Symbol Queue. When the output data is assembled, the stream assembler will send the prefix data to downstream units immediately after the Compression Header TLV is sent. Additionally, in cases where sending RAW (unencoded) data is determined to be the best option, the prefix data will be preloaded into the reconstructor unit's history buffer to properly reconstruct the RAW data stream from the incoming symbols.

Following the prefix data (or after the Compression Header TLV if no prefix data is present) the data stream begins. The data stream itself is written into the symbol queue for retrieval by the Stream Assembler after the Huffman codes are available and all decisions regarding the type of stream output have been made.

The extra offset and length bits in the data stream (bits that are not a part of the symbols which are Huffman encoded…see DEFLATE and XP specs for details) are also counted for each Huffman block. These counts are used in calculations of the encoded data stream size for making final encoding decisions (e.g. retrospective Huffman encoding vs. RAW). These counters are maintained in the Symbol Mapper.

The symbols themselves are also divided by type and sent down one of two pipes. For XP, the pipes are divided between short and long symbols. For DEFLATE, the pipes are divided between literal/length and distance symbols. In the rest of this document, the short symbol (XP) and literal/length symbol (DEFLATE) pipe will be referred to as the short pipe. The long symbol (XP) and distance symbol (DEFLATE) pipe will be referred to as the long pipe. The first step of each symbol pipe is the Collapser unit. This unit combines any non-unique symbol values in preparation for sorting in the next stage. In addition, it contains a FIFO to smooth out stalls from downstream units.

The Compression Header TLV and other side band information are stored in an 8 deep, Sequence ID Control Buffer. A corresponding 3 bit sequence ID will be passed with each Symbol Set through the Control Path. If Predetermined Table data is present in the Prefix TLV, the data will be written to one of 8 Predetermined Table entries to be used by the Tree Walker. Note, the Compression TLV must be first in order to determine how to process the subsequent TLVs.

3. After the Collapser block, symbols enter the insertion sort unit, where they are accumulated and sorted based on their frequency within the Huffman block. Once a Huffman window trigger point is reached (based on the entry count in the symbol buffer, receiving an EOF, or the total size of the uncompressed data), the sorted symbol list and frequency counts are forwarded to the Tree Builder stage.

   Note that starting with the tree builder stage, there are two copies of each unit until reaching the stream assembler. Two copies are needed in order to keep throughput on 2kB frames. For 2kB frames, a new frame could be received every 2048/4 = 512 cycles. The latency of some steps may exceed 576 cycles. With the exception of pathological cases, each stage will need to have a latency of less than 1024 cycles in order to maintain throughput for 2kB frames. For each of the duplicated stages (Huffman Tree Builder, Huffman Tree Walker/Symbol Table Builder), the data can switch between the two sub-pipes within the same symbol pipe based on availability as long as they enter the stream assembly unit in the correct order. With a single exception, there are no dependencies between the two symbol pipes, the exception is DEFLATE

mode, where the encoding of the symbol tables is done with a single encoding across both the literal/length and distance symbols.

4. The Tree Builder will use the sorted output of the Insertion Sort block to create a tree structure, with symbols having the highest frequency nearest the top of the tree and symbols with the lowest frequency near the bottom of the tree.

5. The Tree Walker block will walk the Huffman tree created by the previous block and generate canonical codes for both predetermined and retrospective encoding methods. The canonical codes are loaded into memory based look up tables for eventual use by the Stream Assembler. The Huffman code lengths are also passed to the Symbol Table generator for the retrospective method.

6. The Symbol Table block takes in the header symbols (compressed symbol table) from the Huffman Tree Walker and generates the Huffman encoded symbol table for the stream assembler. This is a multi-step process that will reuse the Insertion Sort, Tree Builder and Tree Walker blocks to encode the compressed Huffman symbol (code lengths) table. The final Huffman Header will be constructed for use by the Stream Assembler.

7. When both the Short and Long symbol control paths are ready for a given Sequence ID, the Stream Assembler will begin reading data from Symbol Queue. TLV type data will be passed as is through to the Encrypt block. When payload data is reached, data will begin to be processed as well as sent through the Reconstructor block.

8. The Stream Assembler will decode and process short and long symbols through the LUT. Up to four lookups will be performed to the short LUT per clock cycle as well as one lookup to the long LUT.

9. The Stream Assembler will wait for the Header Builder in each symbol path to finish before starting the stream assembly process. The Stream Assembler will read data out of the Symbol Queue and will process the TLV encoded data. All of the TLV types not associated with compression will be passed through to the Encryption interface unmodified. The only TLV types of interest to the Stream Assembler block are the Compression and Payload TLV types. The Stream Assembler will modify the XP10_uncompressible_data field in the Compression TLV if the payload data is sent raw instead of compressed. When the Stream Assembler has completed all processing for a frame, the Sequence ID is returned to the Symbol Mapper.

1.4 XP10 Processing Overview

XP10 Processing mode refers to both “normal” and CFH mode frames (via compound commands) submitted that request XP10 headers to be built. The following options and frame handling are specified below:

1.4.1 XP10 General

1. When XP10 frames are generated, the encoder will make a block-by-block determination of the encoding method, always choosing the smallest from the available options, which are:
   a) Predetermined Huffman Encoding (using the trees provided from the prefix selector)
   b) Retrospective Huffman Encoding (using all combinations of the SHORT/LONG and Simplified encoding schemes)
   c) Raw (uncompressed data)
2. The Encoder will always minimize the unoccupied symbol locations in the Retrospective Huffman trees by using the largest number of zero table fill commands possible.
3. The Encoder will only transmit the MTF fields upon a transition from a RAW block to a Compressed Coding block. Back to Back Coding blocks will not have the MTF fields populated on the second (or subsequent sequential coding block), nor will RAW frames ever carry the MTF fields. The first coding block of a frame will never have the MTF fields populated. This will minimize the size of the XP10 block header.
4. Coding Block boundaries will be dictated by a programmable watermark (or end of frame) which is based on the number of symbols (not bytes) in the compressed coding block. This shall be at least 8K symbols, where a symbol is a PTR/MTF/Literal.

1.4.2 XP10 CFH Mode Handling

1. Compact Frame Header (CFH) mode frames will be limited to either 4KB or 8KB of Raw data. This check will be enforced at the hardware level and there will be an error indicator propagated in the metadata that accompanies the frame. Frame headers populated by software will specify that this mode is enabled for the frames.

2. CFH Frames will always only be encoded as a single Huffman Coding block that handles all the frame data. The CFH Frames have several header options that are described in Table 1.

CFH Mode	Block Header	Frame Header	CRC Present/Type
CFH XP10	CFH Block Header	XP10 Frame Header	CRC32
			No CRC

Table 1 : CFH Reduced Header Support

Note: CFH XP10 Reduced Mode headers are not parsable and rely on having metadata with the frame in order to understand what the header stack up is (unlike XP10).

3. CFH Frames have additional handling requirements which XP10 Normal mode does not have, the following code points exist and apply to only CFH Frames:

CFH Frame Size	Threshold	Usage
4KB	4096 Bytes	Compressible Threshold is CFH Frame Size (4KB or 16KB)
	4080 Bytes	This mode reserves 16B at the end of the buffer for SW use. - Minus 16 byte mode
	Infinite	This forces a Huffman Encoding – either Retrospective, Simple or Predetermined to be used so that SW doesn't need to check if the frame is RAW
8KB	8192 Bytes	Compressible Threshold is CFH Frame Size (4KB or 16KB)
	8178 Bytes	This mode reserves 16B at the end of the buffer for SW use. - Minus 16 byte mode
	Infinite	This forces a Huffman Encoding – either Retrospective, Simple or Predetermined to be used so that SW doesn't need to check if the frame is RAW

Table 2 : XP10 CFH Options

4. When a CFH Frame has been deemed to be uncompressible (aka the compressed frame with headers is larger than the raw frame), the frame shall be output without block coding or CFH (XP10/CFH Reduced 0 or 1 headers).
   - In this case, the UNCOMPRESSIBLE bit shall be set in the TLV Header
   - There exists a mode (Infiinte) that forces CFH frames to always be compressed even if the frame is inflated post compression.

1.4.2.1 XP10 CFH Header

This header is defined as follows:

1) This reduced CFH Header has 3 fields:
2) OUTPUT SIZE (16 bits), measured in units of bits is like the XP10 OUTPUT_SIZE field. This field indicates how many bits in size the compressed CFH frame is. The size is computed as the sum of the entire GREEN area below
3) USER_PREFIX (6-bits), prefix selector which shall be interpreted as in Table 3:
4) RAW (1 bit), value of 1 means raw, value of 0 means compressed mode
5) CHF_MODE, 1 means 8K, 0 means 4K

Value	Usage
0	No Prefix and Coding is Retrospective

1-63	Prefix # = Value and Coding may be Retrospective or Predetermined. This selection is from the SHORT_SYMBOL_ENCODE_TYPE field as defined by XP10.

Table 3 : XP10 CFH Prefix

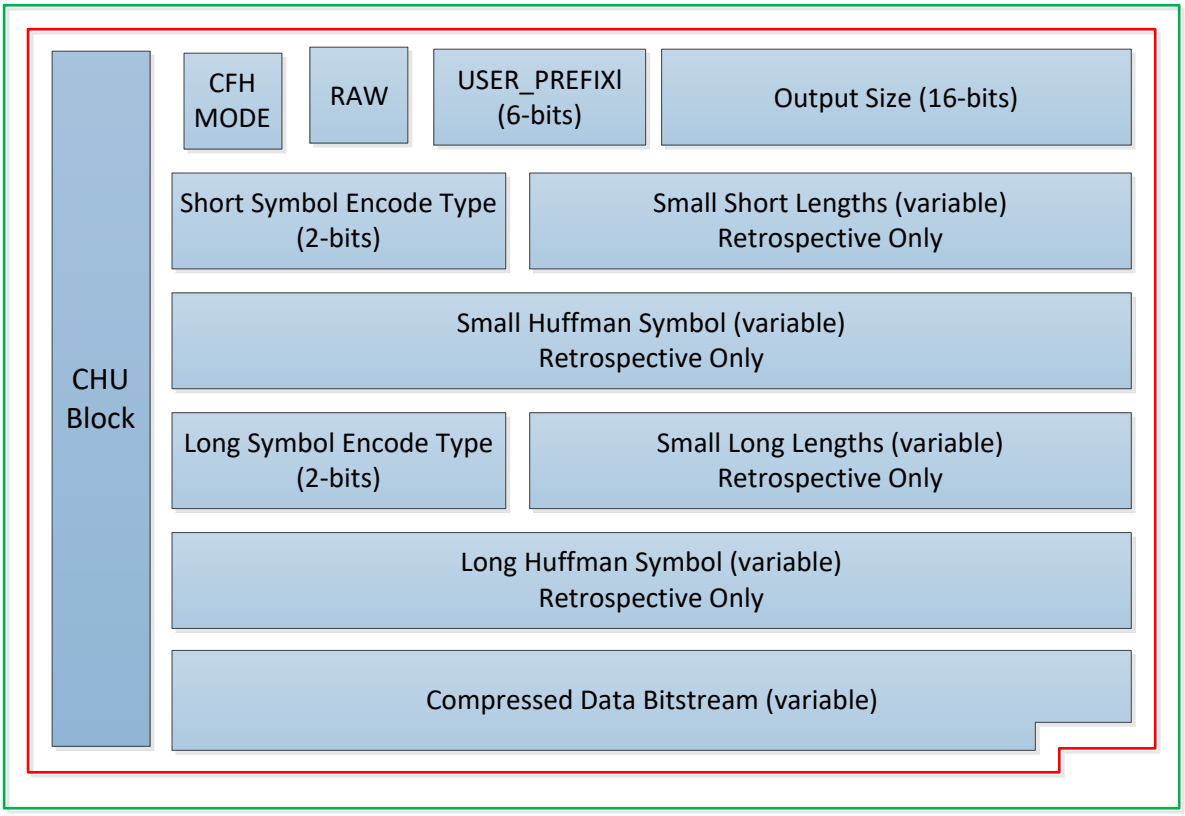


Figure 2 : CFH Reduced Mode 1 Block Header

1.4.3 CFH Append Mode

CFH Append mode is similar to the XP10 CFH Mode with the following differences:

- For each frame of the compound command the LZ77 Compressor and Decompressor Window will **not** be reset.
- The MTF cache **will be reset** in between frames; this implies that each coding block emitted by the Huffman Encoder will **NOT** contain the MTF table for the compound command.
- Only the first frame in the compound command will be run through he prefix engine and have the predetermined prefix attached to it.
- All the frames within the compound command may use the Predetermined Prefix that was determined by the 1st frame in the compound command, but **will not** have the ability to use a different predetermined prefix.

- Coding Blocks will be delineated on the compound command frame boundary; data from one frame will never be mingled with data from another frame in the same Huffman Coding Block.

2 Symbol Mapper

lz77_henc_tlv_data	LZ77 Payload TLV Header Type	Description
63:60	lz77_framing[3:0]	Framing information: • [0,4] – 0 through 4 lanes valid • [15] – EOF only, no data associated with transfer
59:52 51:44 43:36 35:28	lz77_data0[7:0] lz77_data1[7:0] lz77_data2[7:0] lz77_data3[7:0]	Literal data or lower bits of back reference offset • For PTR – ptr_offset[7:0] • For MTF – mtf_num[1:0]
27	lz77_backref_vld	Bus Qualifier • 0 – back reference is not present • 1 – back reference is present
26	lz77_backref_type	Back reference type • 0 – PTR • 1 – MTF
25:24	lz77_backref_lane[1:0]	Lane number containing the back reference • 0 – Lane 0 • 1 – Lane 1 • 2 – Lane 2 • 3 – Lane 3
23:16	lz77_offset_msb[7:0]	Upper bits of PTR offset
15:0	lz77_length[15:0]	Back reference length

Table 4 : LZ77 Payload TLV Header Type

2.1 Symbol Mapper Overview

The Symbol Mapper will translate literal and LZ77 length and offset data from the LZ77 Encoder block into symbols for Deflate and the specific XP10 formats. The Symbol Mapper will also be responsible for breaking up large frames into smaller Huffman blocks as set by software configuration registers. The outputs of the Symbol Mapper are symbols for the Symbol Collapser and Symbol Queue, and various control information written to the Sequence ID memory. The Symbol Mapper will only process inputs from the LZ77 Encoder if there is room in the Symbol Mapper and Symbol Queue blocks and there is a free Sequence ID returned from the Stream Assembler. Otherwise, the Huffman encoder will assert a stall signal to the LZ77 Encoder. The Sequence ID is used to track Huffman windows through the Control

and Data paths. The Stream Assembler will only start the Huffman block evaluation and output process when the Sequence ID is present on the short and long symbol control paths.

The functionality of the Symbol Mapper is shown in Figure 3.

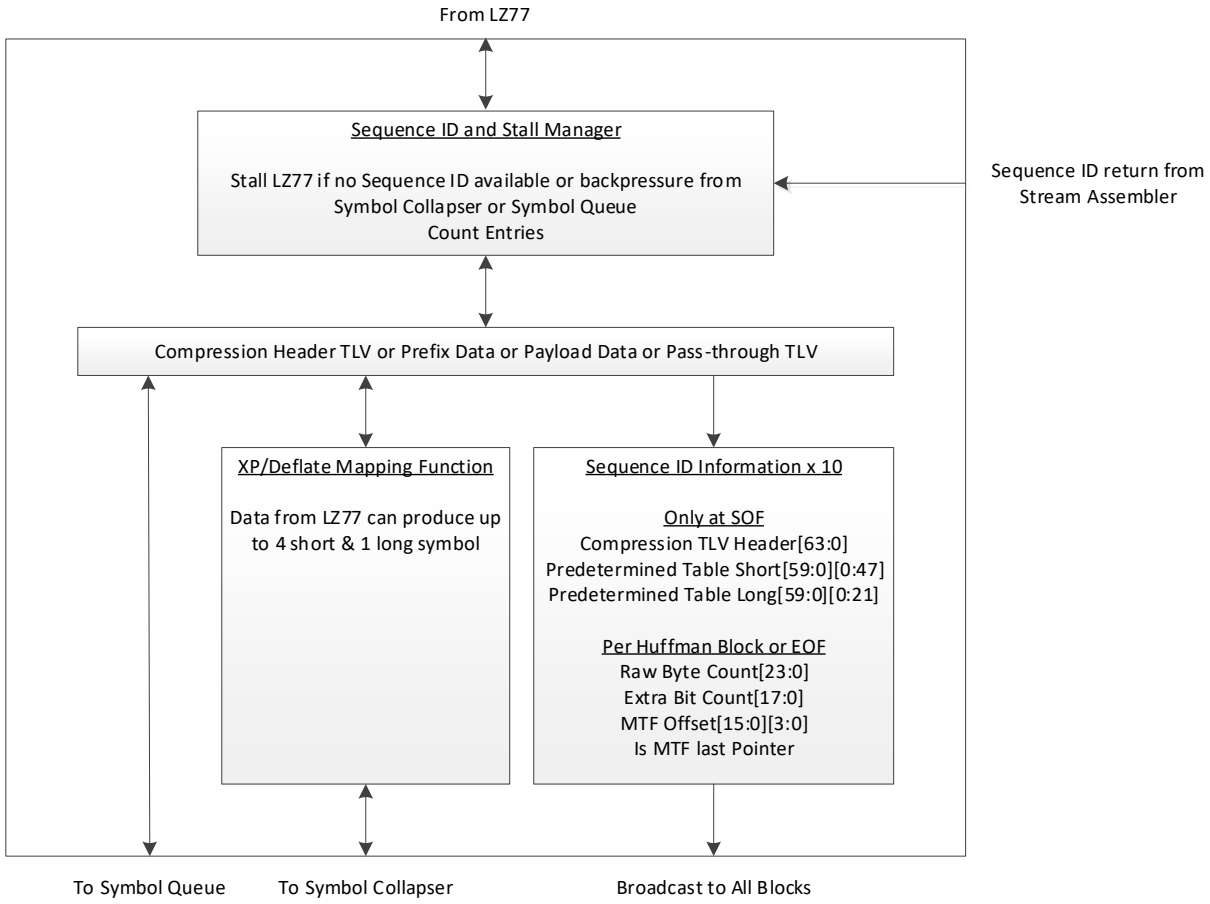


Figure 3: Symbol Mapper Functionality

The Huffman Encoder will determine Huffman block sizes as follows, using software configuration registers:

symbol_entry_count >= HENC_HUFF_WIN_SIZE_IN_ENTRIES

2.1.1 XP10 Symbol Mapping

The Symbol Mapper will process map LZ77 data into XP10 long and short symbols as shown in Figure 4.

L: Length
m: min_length

Intermediate Representation																
XP9 Short Symbol																Additional Data
X: 0=Latest, 1=2nd Latest, 2=3rd Latest, 3=4th latest																
Y: floor(log2(offset))																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
LIT	0	0	0	0	0	0	0	0	Literal Value								
MTF X (L - m < 15)	0	0	0	0	0	0	0	1	0	0	X		L - m				
MTF X (L - m >= 15)	0	0	0	0	0	0	0	1	0	0	X		1	1	1	1	Long Symbol
OFFSET Y (L - m < 15; Y == 0)	0	0	0	0	0	0	6'd20 + Y						L - m				
OFFSET Y (L - m >= 15; Y == 0)	0	0	0	0	0	0	6'd20 + Y						1	1	1	1	Long Symbol
OFFSET Y (L - m < 15; Y > 0)	0	0	0	0	0	0	6'd20 + Y						L - m				Offset
OFFSET Y (L - m >= 15; Y > 0)	0	0	0	0	0	0	6'd20 + Y						1	1	1	1	Offset, Long Symbol
XP9 Long Symbol																	
Z: floor(log2(L - m - 246))																	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
L - m <= 246	0	0	0	0	0	0	0	0	L - m - d15								
L - m > 246	0	0	0	0	0	0	0	0	Z + d232								Length

Figure 4 : XP10 Symbol Mapping

The pseudo code for mapping the short and long symbols for the Symbol Collapser and calculating the raw byte count and extra length and offset bit counts is described below.

```
// XP10 LZ77 to Huffman Encoder Symbol Mapping

// At the start of Payload TLV clear the symbol and raw byte counts

if (Payload TLV) {

        raw_byte_cnt[23:0]    = 0;

        sym_entry_cnt[12:0]  = 0;

        extra_bits_len[17:0]   = 0;

        extra_bits_ofs[17:0]   = 0;

        extra_bits[17:0]         = 0;

        for (int i=0; i<4; i++) {

                mtf_offset[i] = 0;

        }

}

if (lz77_henc_tlv_vld) {
```

```
        sym_entry_cnt = sym_entry_cnt + 1;
if ((lz77_framing == 1 || lz77_framing == 6 || lz77_framing == 11) && !(lz77_backref_vld && lz77_backref_lane == 0)) {
                raw_byte_cnt++;
                sm_sc_shrt_vld[0]     = 1;
                sm_sc_shrt0                     = lz77_framing_data0;
        }
        if ((lz77_framing == 2 || lz77_framing == 7 || lz77_framing == 12) && !(lz77_backref_vld && lz77_backref_lane == 1)) {
                raw_byte_cnt++;
                sm_sc_shrt_vld[1]     = 1;
                sm_sc_shrt1                     = lz77_framing_data1;
        }
        if ((lz77_framing == 3 || lz77_framing == 8 || lz77_framing == 13) && !(lz77_backref_vld && lz77_backref_lane == 2)) {
                raw_byte_cnt++;
                sm_sc_shrt_vld[2]     = 1;
                sm_sc_shrt2                     = lz77_framing_data2;
        }
        if ((lz77_framing == 4 || lz77_framing == 9 || lz77_framing == 14) && !(lz77_backref_vld && lz77_backref_lane == 3)) {
                raw_byte_cnt++;
                sm_sc_shrt_vld[3]     = 1;
                sm_sc_shrt3                     = lz77_framing_data3;
        }
// MTF or PTR
if (lz77_backref_vld) {
    // common stuff for MTF or PTR
    raw_byte_cnt = raw_byte_cnt + lz77_length;
                // load MTF number, offset and length fields
                case (lz77_backref_lane) {
                        2'b00 : {
            lz77_mtf_num[1:0]          =  lz77_framing_data0;
```

```
lz77_offset[15:0]                   = { lz77_offset_msb, lz77_framing_data0};
sm_sc_shrt_vld[0]                   = 1;
                }
                2'b01 : {
lz77_mtf_num[1:0]       =  lz77_framing_data1;
lz77_offset[15:0]                   = { lz77_offset_msb, lz77_framing_data1};
sm_sc_shrt_vld[1]                   = 1;
                }
                2'b10 : {
lz77_mtf_num[1:0]       =  lz77_framing_data2;
lz77_offset[15:0]                   = { lz77_offset_msb, lz77_framing_data2};
sm_sc_shrt_vld[2]                   = 1;
                }
                2'b11 : {
lz77_mtf_num[1:0]       =  lz77_framing_data3;
lz77_offset[15:0]                   = { lz77_offset_msb, lz77_framing_data3};
sm_sc_shrt_vld[3]                   = 1;
                }
    }
    sm_length_vld                   = lz77_length > 246+m;
    sm_length                       = lz77_length & {(floor(log(lz77_length-m-246))+1){1}};
    extra_bits_len                  = extra_bits_len + floor(log(lz77_length-m-246))+1;
    raw_byte_cnt                    = raw_byte_cnt + lz77_length;
    // PTR specific
    if (lz77_symbol_type == PTR) {
                sm_offset_vld       = floor(log(lz77_offset)) > 0;
                sm_offset                       = lz77_offset & {floor(log(lz77_offset)){1}};
                extra_bits_ofs                  = extra_bits_ofs + floor(log(lz77_offset));
                is_mtf_last_ptr     = 0;
                if          (lz77_length < 15+m) {
```

```
                sm_sc_shrt_temp      = 320 + floor(log(lz77_offset))*16+lz77_length-m;
        } else {
                sm_sc_shrt_temp      = 320 + floor(log(lz77_offset))*16+15;
                sm_sc_long_vld       = 1'b1;
                if (lz77_length <= 246+m) {
                        sm_sc_long            =  lz77_length-m;
                } else
                        sm_sc_long            = 232 + {floor(log(lz77_length-m-246)){1}};
                }
        }
        // Now update MTF Offset Table
        offset_mtch = 0;
        // check for match in table
        for (int i=1; i<4; i++) {
                if ((offset_mtch == 0) && (lz77_offset == mtf_offset[i])) {
                        offset_mtch = 1;
                        // shift older entries down one location
                        for (int j = 0; j < i; j++) {
                                mtf_offset[j+1] = mtf_offset[j];
                        }
                }
        }
        // put newest entry at top
        mtf_offset[0] = lz77_offset;
        }
}
// MTF Specific
} else {
        is_mtf_last_ptr = 1;
```

```
                        if          (lz77_length < 15+m) {
                                    sm_sc_shrt_temp     = 256 + lz77_mtf_num*16+lz77_length-m;
                        } else {
                                    sm_sc_shrt_temp     = 256 + lz77_mtf_num*16+15;
                                    sm_sc_long_vld      = 1'b1;
                                    if (lz77_length <= 246+m) {
                                                sm_sc_long          =  lz77_length-m;
                                    } else {
                                                sm_sc_long                      = 232 + {floor(log(lz77_length-m-246)){1}};
                                    }
                        }
            }
            // Load short symbol to correct lane
            case (lz77_backref_lane) {
                        2'b00 :     sm_sc_shrt0 = sm_sc_shrt_temp;
                        2'b01 :     sm_sc_shrt1 = sm_sc_shrt_temp;
      2'b10 :   sm_sc_shrt2 = sm_sc_shrt_temp;
      2'b11 :   sm_sc_shrt3 = sm_sc_shrt_temp;
                        }
}
      // Now check for Huffman block boundaries
if ( (sym_entry_cnt >= HENC_HUFF_WIN_SIZE)) {
            if (lz77_framing >= 10) {
                        sm_sc_eob == 2'b11:
            } else {
                        sm_sc_eob = 2'b10;
            }
            seq_id                  = seq_id + 1;
            // Now load control information and counts into the Seq ID buffer and clear the counters
            extra_bits  = extra_bits_len + extra_bits_ofs;
```

```
                raw_byte_cnt[23:0]    = 0;

                sym_entry_cnt[12:0]   = 0;

                extra_bits_len[17:0]  = 0;

                extra_bits_ofs[17:0]  = 0;

        }

}
```

2.1.2 Deflate Symbol Mapping

The Symbol Mapper will process map LZ77 data into Deflate long and short symbols as shown in Figure 5.

	Intermediate Representation																	
	DEFLATE Literal/Length Symbol																	
Max Length: 258																		
L: Length																		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Additional Data	Formula
Literal	0	0	0	0	0	0	0	0	Literal Value									
End of Block	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0		
3 <= Length <= 10	0	0	0	0	0	0	0	1	0	0	0	L - 2						
11 <= Length <= 18	0	0	0	0	0	0	0	1	0	0	0	(L-11)/2 + 9					1 bit length	(L-11)%2
19 <= Length <= 34	0	0	0	0	0	0	0	1	0	0	0	(L-19)/4 + 13					2 bit length	(L-19)%4
35 <= Length <= 66	0	0	0	0	0	0	0	1	0	0	0	(L-35)/8 + 17					3 bit length	(L-35)%8
67 <= Length <= 130	0	0	0	0	0	0	0	1	0	0	0	(L-67)/16 + 21					4 bit length	(L-67)%16
131 <= Length <= 257	0	0	0	0	0	0	0	1	0	0	0	(L-131)/32 + 25					5 bit length	(L-131)%32
Length == 258	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	1		
	DEFLATE Distance																	
D: Distance																		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1 <= Distance <= 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D - 1			Formula	*
5 <= Distance <= 8	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	*	1 bit distance	(L-5)%2	(L-5)/2
9 <= Distance <= 16	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	*	2 bit distance	(L-9)%4	(L-9)/4
17 <= Distance <= 32	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	*	3 bit distance	(L-17)%8	(L-17)/8
33 <= Distance <= 64	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	*	4 bit distance	(L-33)%16	(L-33)/16
65 <= Distance <= 128	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	*	5 bit distance	(L-65)%32	(L-65)/32
129 <= Distance <= 256	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	*	6 bit distance	(L-129)%64	(L-129)/64
257 <= Distance <= 512	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	*	7 bit distance	(L-257)%128	(L-257)/128
513 <= Distance <= 1024	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	*	8 bit distance	(L-513)%256	(L-513)/256
1025 <= Distance <= 2048	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	*	9 bit distance	(L-1025)%512	(L-1025)/512
2049 <= Distance <=4096	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	*	10 bit distance	(L-2049)%1024	(L-2049)/1024
4097 <= Distance <= 8192	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	*	11 bit distance	(L-4097)%2048	(L-4097)/2048
8193 <= Distance <= 16384	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	*	12 bit distance	(L-8193)%4096	(L-8193)/4096
16385 <= Distance <= 32768	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	*	13 bit distance	(L-16385)%8192	(L-16385)/8192

Figure 5 : Deflate Symbol Mapping

The pseudo code for mapping the short and long symbols for the Symbol Collapser and calculating the raw byte count and extra length and offset bit counts is described below.

```
// Deflate LZ77 to Huffman Encoder Symbol Mapping

// At the start of Payload TLV clear the symbol and raw byte counts

if (Payload TLV) {

        raw_byte_cnt[23:0]   = 0;

        sym_entry_cnt[12:0]  = 0;

    extra_bits_ofs[17:0]     = 0;

        extra_bits_len[17:0]  = 0;

        extra_bits                          = 0;

}

// Literals

if (lz77_henc_tlv_vld) {

        sym_entry_cnt = sym_entry_cnt + 1;

    if ((lz77_framing == 1 || lz77_framing == 6 || lz77_framing == 11) && !(lz77_backref_vld && lz77_backref_lane == 0)) {

                raw_byte_cnt++;

                sm_sc_shrt_vld[0]     = 1;

                sm_sc_shrt_temp             = lz77_framing_data0;

        }

        if ((lz77_framing == 2 || lz77_framing == 7 || lz77_framing == 12) && !(lz77_backref_vld && lz77_backref_lane == 1)) {

                raw_byte_cnt++;

                sm_sc_shrt_vld[1]     = 1;

                sm_sc_shrt1                 = lz77_framing_data1;

        }

        if ((lz77_framing == 3 || lz77_framing == 8 || lz77_framing == 13) && !(lz77_backref_vld && lz77_backref_lane == 2)) {

                raw_byte_cnt++;

                sm_sc_shrt_vld[2]     = 1;

                sm_sc_shrt2                 = lz77_framing_data2;

        }

        if ((lz77_framing == 4 || lz77_framing == 9 || lz77_framing == 14) && !(lz77_backref_vld && lz77_backref_lane == 3)) {
```

```
                raw_byte_cnt++;
                sm_sc_shrt_vld[3]     = 1;
                sm_sc_shrt3                   = lz77_framing_data3;
        }
    // Length & Distance (offset)
    if (lz77_backref_vld) {
        case (lz77_backref_lane) {
                        2'b00 : {
            lz77_offset[15:0]                 = { lz77_offset_msb, lz77_framing_data0};
            sm_sc_shrt_vld[0]         = 1;
                        }
                        2'b01 : {
            lz77_offset[15:0]                 = { lz77_offset_msb, lz77_framing_data1};
            sm_sc_shrt_vld[1]         = 1;
                        }
                        2'b10 : {
            lz77_offset[15:0]                 = { lz77_offset_msb, lz77_framing_data2};
            sm_sc_shrt_vld[2]                 = 1;
                        }
                        2'b11 : {
            lz77_offset[15:0]                 = { lz77_offset_msb, lz77_framing_data3};
          sm_sc_shrt_vld[3]           = 1;
                        }
                }
        raw_byte_cnt                  = raw_byte_cnt + lz77_length;
                // Process length
                if (lz77_length < 11) {
                        sm_sb_shrt_temp     = 256 + lz77_length-2;
                } else if (lz77_length < 19) {
```

```
                    sm_sb_shrt_temp     = 256 + ((lz77_length-11)/11);
                    sm_sb_length                = (lz77_length-11)%2;
                    extra_bits_len              = extra_bits_len + 1;
                } else if (lz77_length < 35) {
                    sm_sb_shrt_temp     = 256 + ((lz77_length-19)/17);
                    sm_sb_length                = (lz77_length-19)%4;
                    extra_bits_len              = extra_bits_len + 2;
        } else if (lz77_length < 67) {
                    sm_sb_shrt_temp     = 256 + ((lz77_length-35)/25);
                    sm_sb_length                = (lz77_length-35)%8;
                    extra_bits_len              = extra_bits_len + 3;
                } else if (lz77_length < 131) {
                    sm_sb_shrt_temp     = 256 + ((lz77_length-67)/37);
                    sm_sb_length                = (lz77_length-67)%16;
                    extra_bits_len              = extra_bits_len + 5;
                } else if (lz77_length < 258) {
                    sm_sb_shrt_temp     = 256 + ((lz77_length-131)/57);
                    sm_sb_length                = (lz77_length-131)%32;
                    extra_bits_len              = extra_bits_len + 5;
                } else {
                    sm_sb_shrt_temp     = 256 + ((lz77_length-11)/11);
                }
                // Process Offset (distance)
                if (lz77_offset <= 4)    {
                    sm_sb_long                  =       lz77_offset – 1;
                } else (
                    for (i=3; i < 16; i++)    {
                        if ((lz77_offset <= 1<<i) && (lz77_offset > 1<<(i-1))) {
                            sm_sb_long          =       i+1+ (((lz77_offset-((1<<(i-1))+1))
/ (1<<(i-2)));
```

```
                                        extra_bits_ofs          = extra_bits_ofs + i - 2;
                                        sm_sb_offset            = ((lz77_offset-((1<<(i-1))+1)) % (1<<(i-2));
                                }
                        }
                }
                // Load short symbol to correct lane
                case (lz77_backref_lane) {
                        2'b00 :     sm_sc_shrt0 = sm_sc_shrt_temp;
                        2'b01 :     sm_sc_shrt1 = sm_sc_shrt_temp;
    2'b10 :   sm_sc_shrt2 = sm_sc_shrt_temp;
    2'b11 :   sm_sc_shrt3 = sm_sc_shrt_temp;
                }
        }
        // Now check for Huffman block boundaries
        if ((sym_entry_cnt >= HENC_HUFF_WIN_SIZE) || (lz77_framing >= 10)) {
                if (lz77_framing >= 10) {
                        sm_sc_eob == 2'b11:
                } else {
                        sm_sc_eob = 2'b10;
                }
                seq_id                  = seq_id + 1;

                extra_bits  = extra_bits_len + extra_bits_ofs;
                raw_byte_cnt[23:0]    = 0;
                sym_entry_cnt[12:0]   = 0;
                extra_bits_len[17:0]    = 0;
                extra_bits_ofs[17:0]    = 0;
        }
}
```

3 Symbol Collapser

3.1 Overview

The Symbol Collapser is the interface between the input data stream and the Insertion Sort block. There is a separate collapser unit for the A symbol set pipe and the B symbol set pipe. The collapser takes the symbols presented on the input interface each cycle and reduces them to a set of unique symbol values. Each collapser also has a FIFO which can absorb additional input symbols in case of a stall downstream. If the FIFO fills, then a stall must be asserted upstream. When the FIFO depth is greater than 2 entries, additional combining between successive cycles of symbols on the input interface is attempted through a combine register.

The table below describes the differences between the short symbol and long symbol collapsers.

Symbol Pipe	I/O max symbols/cycle	Max count per symbol	Size of FIFO entry (symbols)
short (A)	4	4	4
long (B)	1	1	1

Table 5 : Symbol Collapser Details

The description that follows is for the collapser in the short symbol pipe. The long symbol pipe is a straightforward simplification of the short symbol pipe case.

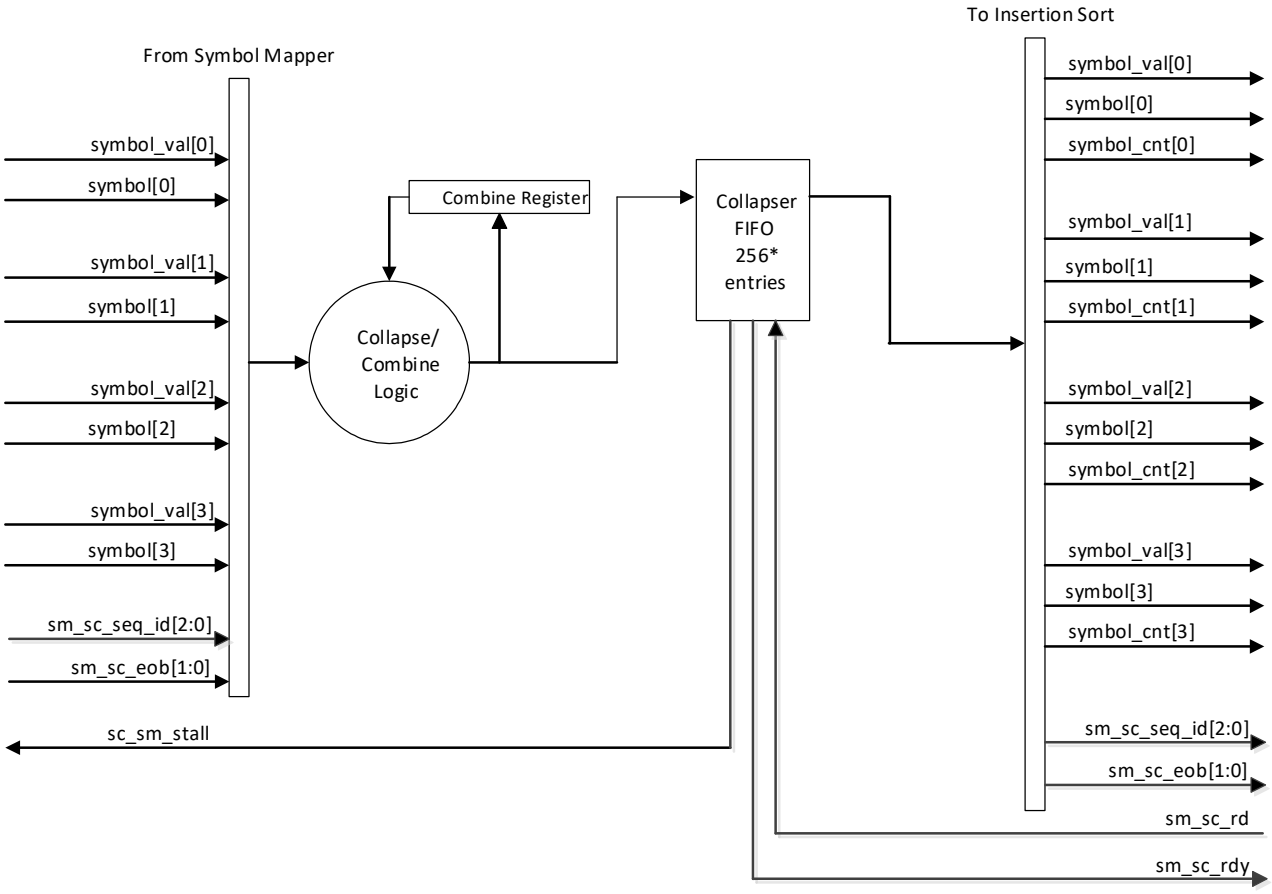


Figure 6 : Symbol Collapser Block Diagram

The general idea is to pass the symbol data straight through, using only one location in the FIFO. Once the FIFO starts to back up, the collapse/combine logic (along with the combine register) will aggressively attempt to condense symbols as much as possible. Note that when combining symbols across multiple cycles, the crossing of a block boundary is not allowed.

If the FIFO is less than 2 entries deep and the combine register is not valid, then the FIFO input data is collapsed according to the following rules:

- Valid duplicate symbols will be removed.
  - The valid of the symbol with the larger index will be invalidated.
  - The count of the symbol with the lower index will be incremented by 1.
- The holes left by invalidated symbols will be filled by shifting down valid symbols from higher indexes to lower indexes.

For example, if symbol[0] = 2, symbol[1] = 2, symbol[2] = 5, and symbol[3] = 2, the resulting output would be symbol[0] = 2, symbol_cnt[0] = 3, symbol[1] = 5, symbol_cnt[1] = 1. symbol_val[2] = symbol_val[3] = 0.

If the FIFO is greater than 2 entries deep, then the actions taken depend on whether or not the contents of the combine register are valid and the end of block signal. Note that if the combine register valid bit is set and the combine register end_of_block bit is signaled, action will still be taken on the FIFO input for that cycle even if there are no valid input symbols.

- If Combine register valid == 0
  - Collapse the input as if the collapser were in bypass mode (If no valid input symbols, do nothing and skip the rest of the steps below).
  - If eob == 0
    - Write the combine register with the results and mark the combine register as valid (do not write the FIFO or update the FIFO write pointer).
  - If eob == 1
    - Write the output of the collapser logic directly into the FIFO and update the collapser FIFO write pointer. (do not write the combine register or mark it valid)
    - Write a 1 to the EOB bit for the FIFO entry.
- If Combine register valid == 1
  - If combine register end_of_block == 0
    - Combine the contents of the FIFO combine register with the current inputs as described in the Multi Cycle Combining Rules below.
    - If internal eob == 1
      - if there are still valid input symbols after collapsing/combining:
        - Set the combine register valid and end_of_block bits
      - If there are not still valid input symbols after collapsing/combining:
        - Set the FIFO entry EOB bit and clear the combine register valid bit
  - If combine register end_of_block == 1
    - Collapse the inputs as if in bypass mode (if valid)
    - Write the current contents of the combine register into the FIFO and update the FIFO write pointer. Set the EOB bit in the entry
    - Clear the combine register end_of_block bit
    - if there are valid input symbols this cycle:
      - Set the combine register valid bit.
    - If there are no valid input symbols this cycle
      - Clear the combine register valid bit

Multi Cycle Combining Rules: When combining the current contents of the combination register with the current symbol inputs, use the following rules:

- Starting with the input symbol at index zero (and sweeping through all 4 symbols)

    - If there is a match with a symbol in the combine register AND the count for the symbol in the combine register is < 4
        - invalidate the input symbol.
        - Add 1 to the matching symbol count from the combine register
- Combine the remaining valid input symbols as described above for bypass mode.
- If there are any unused (invalid) symbols currently in the updated combine register, promote the lowest valid symbol index (and count) from the remaining collapsed input symbols to the lowest invalid index in the current combine register contents as long as the input symbol does not match any symbol currently in the combine register. Mark the promoted input symbol as invalid and the new combine register symbol as valid. Repeat as long as there are invalid combine register symbols and valid input collapsed input symbols.
- If there are still valid input symbols after combining:
    - Write the updated contents of the combine register into the FIFO and update the write pointer.
    - Write the remaining input symbols into the combine register and set the combine register valid bit.
- If there are not any remaining valid input symbols after combining:
    - Write the updated contents of the combine register back into the combine register and keep the combine register valid bit set.
    - Do not write the FIFO

The sc_sm_stall output signal is asserted when the FIFO is full and the collapser can no longer accept input data, otherwise de-asserted.

The structure of a collapser FIFO entry (Short Symbol (A) Pipe is):

Bits	Value
9:0	Symbol 0
11:10	Symbol 0 Count (2'b00 = 4)
12	Symbol 0 Valid
22:13	Symbol 1
24:23	Symbol 1 Count (2'b00 = 4)
25	Symbol 1 Valid
35:26	Symbol 2
37:36	Symbol 2 Count (2'b00 = 4)
38	Symbol 2 Valid
48:39	Symbol 3
50:49	Symbol 3 Count (2'b00 = 4)
51	Symbol 3 Valid
53:52	EOB
56:54	Seq_ID

Table 6 : Symbol Collapser Short Symbol FIFO Format

The structure of a collapser FIFO entry (Long Symbol (B) Pipe is):

Bits	Value
7:0	Symbol 0
8	Symbol 0 Valid
10:9	EOB
13:11	Seq_ID

Table 7 : Symbol Collapser Long Symbol FIFO Format

FIFO sizes (assuming implemented with 1R/1W port memories or register arrays):

Symbol Pipe	Size
short	COLLAPSER_FIFO_SIZE x 57
long	COLLAPSER_FIFO_SIZE x 14

Table 8 : Symbol Collapser FIFO Sizing

4 Insertion Sort

4.1 Overview

The Insertion Sort hardware block will sort symbols based on the count of each unique symbol in a given Huffman block. The Insertion Sort function is important to overall latency through the Control path, in that it allows the Huffman Tree Builder stage to have a significantly shorter critical timing path.

4.2 Block Diagram

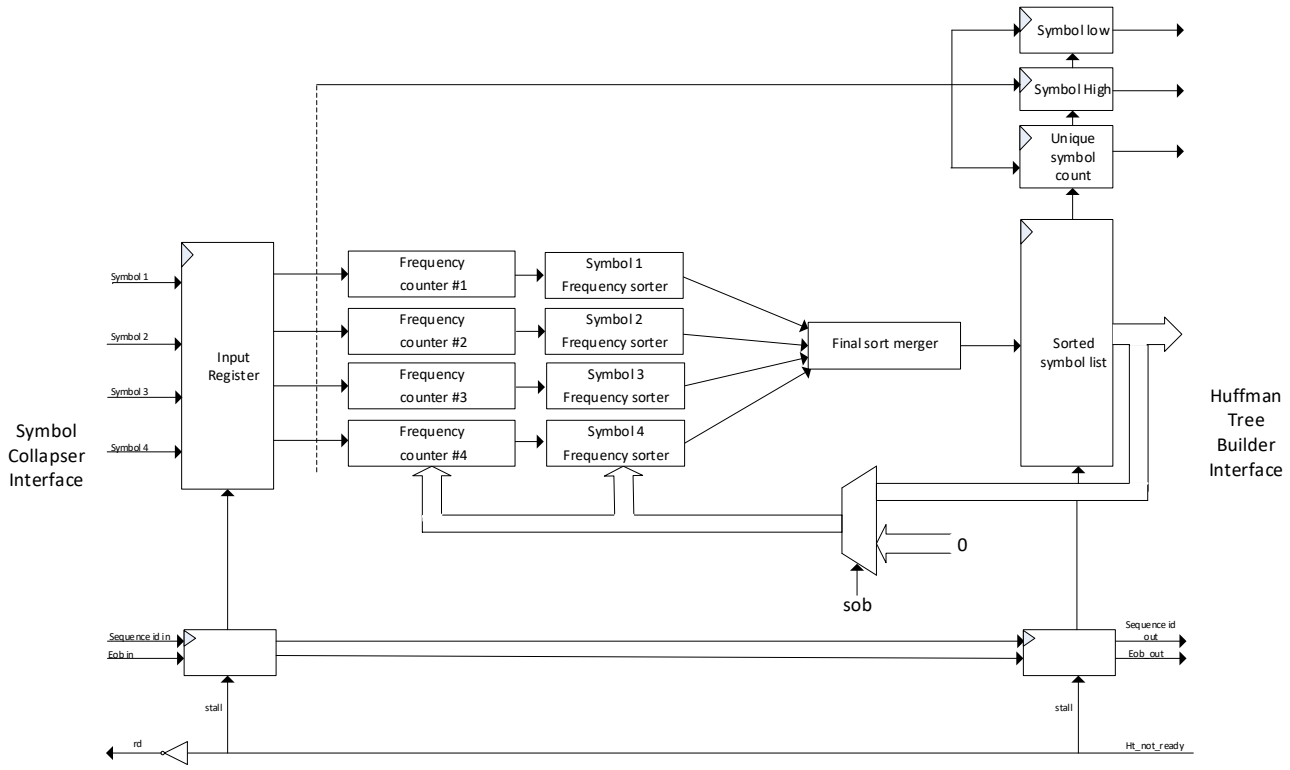


Figure 7 : Insertion Sort Block Diagram

The symbols are read from the collapser interface with one transaction per “rd” strobe. Read requests are returned with the entire symbol valids de-asserted if the collapser FIFO is empty. Control information like sequence id and eob code are also read along with the symbols. Sequence id remains static throughout the Huffman block. These inputs are shown in the interface section and are passed along the pipe to the downstream hardware blocks.

The start of block (sob) and xp/deflate information are read from the context memories external to this hardware block and are indexed with sequence id input. The latency for this context fetch should be 0 cycle.

The input symbols are compared and sorted against the last sorted list.

At sob, the inputs are compared against zeroed sort list. Zeroed sort list is the symbol order list with 0 frequencies.

NOTE: For the DEFLATE literal/length insertion sort unit, symbol 256 (EOB) starts with a value of 1. This symbol will never show up on the input interface to the Huffman encoder, but is inserted into the encoded bit stream once at the end of the block to indicate the termination of the Huffman block.

The symbols are sorted as per the functions described below.

The symbol frequency-of-occurrences sort order depicted in the diagrams below is top-to-bottom lowest-to-highest. Sorting is done,

1) First, based on the frequency-of-occurrence of the symbols, and
2) Second, in case of ties, based on the symbol order.

To illustrate this, assume that there are 10 possible symbols, and we receive only one symbol each cycle. Initially, the frequency count for each symbol (red box) will be zero, since no symbols have been received yet. The array of symbol frequencies will be still be sorted, however, based on the symbol order (blue box) which, in the example below, happens to match the values assigned to the symbols (green box).

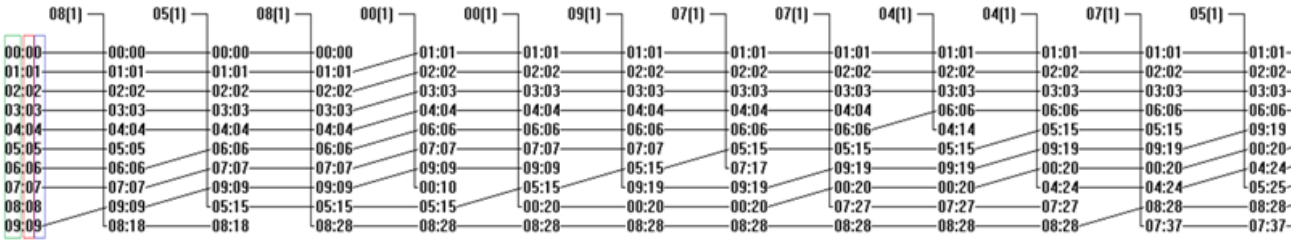


Figure 8: Insertion sort with one symbol inserted per cycle.

To make these drawings more human-readable in decimal notation, the total number of symbols was set to 10 and the TieBreakerValue (blue box) was set to be the same value as the Symbol (green box). Notation shown here is “Symbol:Frequency*10+TieBreakerValue” and “Symbol (NumInserted)”.

When the first symbol arrives at the input (in this case we receive one instance of the symbol “08”), it is matched the frequency counter that currently stores the value “08”. That frequency counter will increase by one. As a result of the increase, the order must be re-sorted.

We want to perform the re-sort in a single cycle, so the logic must determine that the frequency counter that stores symbol “09” must move up by one, and the newly incremented symbol “08” must be inserted into the bottom of the array.

The frequency counters become more complex with an insertion sort that can accept multiple symbols in parallel.

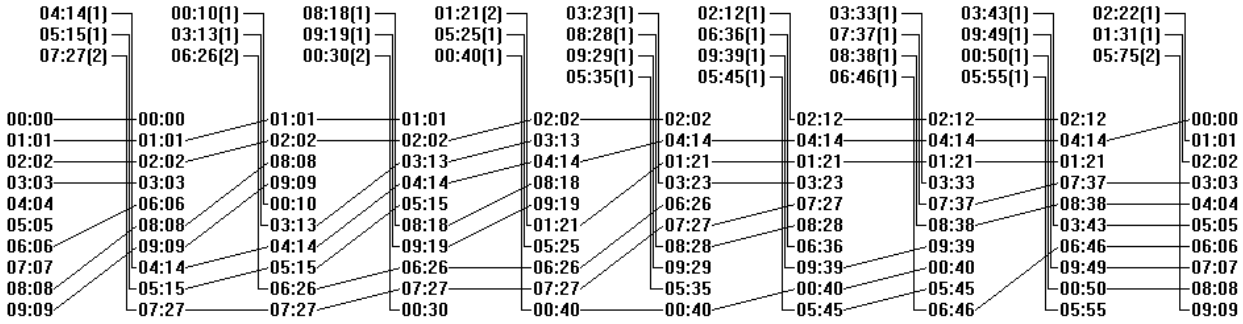


Figure 9: Insertion Sort with up to four symbols per cycle.

To make these drawings more human-readable in decimal notation, the total number of symbols was set to 10 and the TieBreakerValue was set to be the same value as the Symbol. Notation shown here is "Symbol:Frequency*10+TieBreakerValue" and "Symbol:IncrementedFrequency*10+TieBreakerValue (NumInserted)".

To achieve this goal, each frequency counter bus must provide the following information:

a) The symbol,
b) The original frequency associated with the symbol,
c) The new frequency for the symbol
d) A "valid" flag.

Each sort engine first creates an array of four two-bit values that capture the relationship between each frequency count in the sort list and the original and new frequency counts of the symbol on the frequency counter bus. The code below illustrates how this is done (covers all four input symbols). This code doesn't illustrate order based tie breaker.

```
for (s = 0; s<num_list_symbols; s++) {
   for (i = 0; i<num_input_symbols; i++) {
      if (list_freq[s] < in_sym[i].orgfrq) {
         array[s][i] = 0;
      }
      else if ((s+1 < num_list_symbols) && (list_freq[s+1] <= in_sym[i].newfrq)) {
         array[s][i] = 1;
      }
      else if (list_freq[s] > in_sym[i].newfrq) {
         array[s][i] = -2; // binary '10'
      }
```

```
      else {
         array[s][i] = -1; // binary '11'
      }
   }
}
```

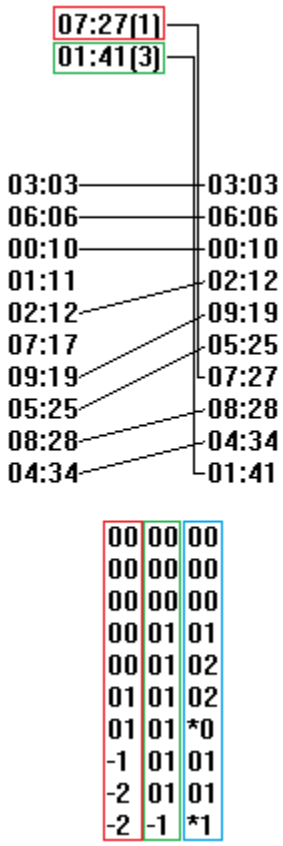


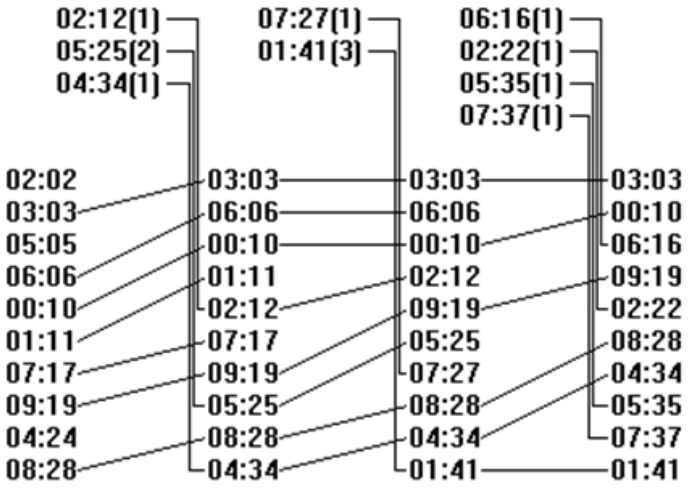
Figure 10: Insertion of two new symbols, and creation of reorder tables for each symbol and for the pair of symbols.

Referring to Figure 10, the single instance of the symbol ‘07’ (within the top red box) causes the creation of another corresponding reorder table (within bottom red box). Likewise, three instances of the symbol ‘01’ (within the top green box) cause the creation of a corresponding reorder table (within bottom green box). Each of these two reorder tables would be useful for indicating how to shuffle values around to correctly insert the new symbols *individually* while maintaining the sort order of all the symbols. The values ‘00’ and ‘-2’ indicate that the previous value should be retained. The value ‘01’ indicates that the next value down should be shifted up and used. Finally, the value ‘-1’ indicates that that the new value should be inserted.

However, because in this example we are inserting two symbols *at the same time*, we need the reorder table contained within the blue box. In this reorder table, the value '00' indicates no change to the data set in the sort list. The values '01' through '04' represent that a data set down in line should be shifted up and in. For example, '01' means first data set after this value in the list should be shifted up and in, '02' means second data set after this value in the list should be shifted up and in and so on. The "pointer" values '*0' through '*3' indicate that a frequency count on one of the frequency counter busses should be selected and brought in.

The algorithm for creating each element of the final reorder table (blue) takes as input the nearby values in the per-symbol reorder tables (red and green).

Figure 11 illustrates how the algorithm for creating the final reorder table from the per-symbol reorder tables is supported in the cases of two, three and four new symbols per clock.



01 00 00 01	00 00 00	00 00 00 00 00
01 00 00 02	00 00 00	01 00 00 00 01
01 01 00 02	00 00 00	01 00 00 00 *0
01 01 00 02	00 01 01	-1 01 00 00 01
01 01 00 *0	00 01 02	-2 -1 00 00 *1
-1 01 00 01	01 01 02	-2 -2 01 00 02
-2 01 00 01	01 01 *0	-2 -2 01 01 02
-2 01 00 *1	-1 01 01	-2 -2 01 01 *2
-2 -1 01 01	-2 01 01	-2 -2 -1 -1 *3
-2 -2 -1 *2	-2 -1 *1	-2 -2 -2 -2 00

Figure 11: Per-symbol reorder table elements used in the creation of the final reorder table.

4.2.1 Pointer to Lowest Non-Zero Entry

There is also a need to track the number of unique symbols used (sym_unique) in each Huffman block.

One way to maintain this count is by maintaining a pointer to the lowest non-zero frequency, which it will pass on to the next stage along with all of the sorted symbols and their frequencies at the end of the Huffman Window. This pointer needs to be designed to handle the initial state where all of the frequency counts are zero. For example, it could initialize to 'N' if there were a maximum of 'N' possible symbols, since only values of 0 through 'N-1' would be valid. In the small example table below, where there are only 4 symbols supported, N would start out being 4 because all frequencies are zero. It would be become 3 after the first symbol was inserted, and gradually move "up" as more symbols are inserted and their respective frequency counts increase.

Row	Pntr	Freq		Pntr	Freq		Pntr	Freq		Pntr	Freq		Pntr	Freq
0		0			0			0			0		->	2
1		0			0			0		->	3			4
2		0			0		->	1			7			19
3		0		->	1			1			44			125
4	->													

In case of no downstream stalls, the insertion sort unit is able to perform sort on each transaction of input symbols in one clock cycle.

In addition to performing the insertion sort, this hardware block is also responsible for tracking the lowest numbered and highest numbered symbol used within a Huffman block. To do this, 2 registers are implemented: sym_low and sym_high. Note: So far there is no need for the sym_low tracking other than debug. Each cycle, the lowest and highest symbol value should be calculated across all input symbols. These values are then compared against the sym_low/sym_high registers and the registers updated accordingly. These values will be passed to the Tree Builder unit along with the sorted frequency data.

The sorted list is read by Huffman tree builder at eob.

If the tree builder is not ready at eob on sort-tree interface, it asserts not_ready. Not_ready stalls the insertion sort pipeline and hence also de-asserts “rd” strobe to the collapser interface.

In case of pass through mode, sob is don’t care and the tree builder will read the control information without waiting for the eob.

4.3 Resources

Flops:

Short path

Input register	53
Sort list	14405
Total	14458

Long path

Input register	13
Sort list	5213
Total	5226

4.4 Parameters

Parameter	Symbol Pipe A	Symbol Pipe B	Symbol Table Builder	Comment
IS_SORT_ROWS	576	248	33	Symbol table depth
IS_SORT_IPC	4	1	2	Insertions per cycle
IS_SORT_MAX_FREQ_BITS	15	13	10	Max frequency of symbols

Table 9 : Insertion Sort Parameters

5 Tree Builder

5.1 Overview

The Tree Builder will use the sorted output of the Insertion Sort block to create a tree structure, with symbols having the highest frequency nearest the top of the tree and symbols with the lowest frequency near the bottom of the tree.

Huffman tree builder receives a pointer to the lowest non-zero frequency (sym_unique) from the insertion sort engine. This allows the Huffman Tree Builder to start building the tree immediately without spending cycles to find the lowest non-zero frequency.

The tree is created in approximately the same number of clock cycles as there are symbols with non-zero frequency counts in the table.

Data structure:

For each row of the table received from insertion sort, the following data structure is assigned.

Top Node fields are initialized to “invalid”, and “Depth in Tree” fields are initialized to zero.

Lowest non-zero	Name of What’s at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0
	Symb6 Freq	*Symb6	0 (Freq1)	-	0
...is here	Symb3 Freq	*Symb3	3 (Freq2)	-	0
	Symb1 Freq	*Symb1	4 (Freq3)	-	0
	Symb0 Freq	*Symb0	5 (Freq4)	-	0
	Symb2 Freq	*Symb2	6 (Freq5)	-	0
	Symb5 Freq	*Symb5	8 (Freq6)	-	0
	Symb4 Freq	*Symb4	9 (Freq7)	-	0

Table 10: Tree Builder Initial Data Structure

Each cycle, the two least frequently occurring symbols are converted into a “Node” with two “Leaves”. It is not necessary to retain the frequencies for the individual leaves; however, their node will receive a frequency value that is the sum of their individual frequencies. Therefore, the bits that were used to store a symbol’s frequency can be reclaimed as tree creation progresses, and used to store pointers to other symbols or nodes instead. Blue cells are changed. Note that the “Top Node” field will be assigned the ID of the Node that points to it, and the “Depth In Tree” field will be set to one.

Lowest non-zero	Name of What's at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0
	Symb6 Freq	*Symb6	0 (Freq1)	-	0
	Node0 Ptrs	*Symb3	*Symb1	Node0	1
...is here?	Node0 Freq	*Node0 Ptrs	7 (Freq2 + Freq3)	-	0
	Symb0 Freq	*Symb0	5 (Freq4)	-	0
	Symb2 Freq	*Symb2	6 (Freq5)	-	0
	Symb5 Freq	*Symb5	8 (Freq6)	-	0
	Symb4 Freq	*Symb4	9 (Freq7)	-	0

However, the above table is potentially *not* in the correct sort order. In order to be able to go into the next clock cycle properly prepared, we need to finish off the current clock cycle with an ordered list, as shown below.

Lowest non-zero	Name of What's at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0
	Symb6 Freq	*Symb6	0 (Freq1)	-	0
	Node0 Ptrs	*Symb3	*Symb1	Node0	1
...is here	Symb0 Freq	*Symb0	5 (Freq4)	-	0
	Symb2 Freq	*Symb2	6 (Freq5)	-	0
	Node0 Freq	*Node0 Ptrs	7 (Freq2 + Freq3)	-	0
	Symb5 Freq	*Symb5	8 (Freq6)	-	0
	Symb4 Freq	*Symb4	9 (Freq7)	-	0

Therefore, during its creation, the row labeled "Node0 Freq" really needs to be pulled out and re-inserted into the list at the correct location. In case of ties, a newly inserted entry will be treated as if it were larger than the values that it tied with. Note that this behavior differs from the technique used in the prior "Insertion Sort" block, as in this case the Symbol's value is not used for breaking ties. Insertion will cause rows below the insertion point in the table to be shifted down. Now the table is ready for the next clock cycle…

Lowest non-zero	Name of What's at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0

	Symb6 Freq	*Symb6	0 (Freq1)	-	0
	Node0 Ptrs	*Symb3	*Symb1	Node0	1
	Node1 Ptrs	*Symb0	*Symb2	Node1	1
...is here	Node0 Freq	*Node0 Ptrs	7 (Freq2 + Freq3)	-	0
	Symb5 Freq	*Symb5	8 (Freq6)	-	0
	Symb4 Freq	*Symb4	9 (Freq7)	-	0
	Node1 Freq	*Node1 Ptrs	11 (Freq4 + Freq5)	-	0

Note the pattern here. The blue cell in the "Stored Value B" column always receives its value from the Stored Value A cell of the row below *in the previous table*. The beige cells are receiving their values from the cells directly below *in the previous table*. The yellow cells represent the newly inserted rows. The "Stored Value A" part of the newly inserted row is the sum of the two lowest two frequencies from the previous table. The Stored Value B part of the newly inserted row is a pointer that happens to increment by one every time a new row is inserted.

Repeating the procedure again, we produce…

Lowest non-zero	Name of What's at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0
	Symb6 Freq	*Symb6	0 (Freq1)	-	0
	Node0 Ptrs	*Symb3	*Symb1	Node0	1
	Node1 Ptrs	*Symb0	*Symb2	Node1	1
	Node2 Ptrs	*Node0 Ptrs	*Symb5	Node2	1
...is here	Symb4 Freq	*Symb4	9 (Freq7)	-	0
	Node1 Freq	*Node1 Ptrs	11 (Freq4 + Freq5)	-	0
	Node 2 Freq	*Node2 Ptrs	15 (Freq6+ Freq2+ Freq3)	-	0

In this case something new happened. We made a row out of a Symbol and a Node (*Node0 Ptrs) as opposed making a row out of two Symbols. Let this node, shown above in red, be referred to as a "Subsumed Node". When a node is subsumed, its ID is broadcast to all the other nodes along with the ID of the node that is subsuming it ("*Node2 Ptrs" in this case). All rows will receive the broadcast. Any row whose current Top Node matches the ID if a subsumed node should: a) increment its "DepthInTree" value by one, and b) replace its current Top Node with the ID of the new top node that is subsuming the previous one. Note that it is possible for IDs from up to two subsumed nodes to be broadcast, but both of these will share the same new top node. Therefore, all rows must check their current Top Node against up to two subsumed node IDs.

After this additional rule is applied, the Table looks like this (note that the rule affected the green cells)…

Lowest non-zero	Name of What's at this Address	Stored Value A	Stored Value B	Top Node	Depth In Tree
	Symb7 Freq	*Symb7	0 (Freq0)	-	0
	Symb6 Freq	*Symb6	0 (Freq1)	-	0
	Node0 Ptrs	*Symb3	*Symb1	Node2	2
	Node1 Ptrs	*Symb0	*Symb2	Node1	1
	Node2 Ptrs	*Node0 Ptrs	*Symb5	Node2	1
...is here	Symb4 Freq	*Symb4	9 (Freq7)	-	0
	Node1 Freq	*Node1 Ptrs	11 (Freq4 + Freq5)	-	0
	Node 2 Freq	*Node2 Ptrs	15 (Freq6+ Freq2+ Freq3)	-	0

5.2 Block Diagram

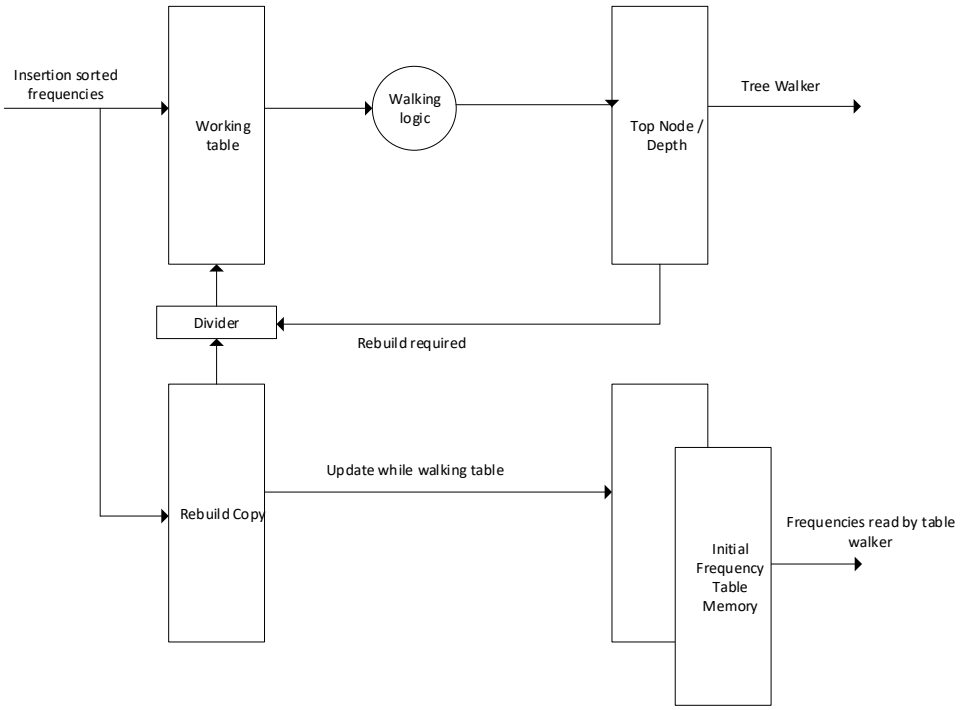


Figure 12: Huffman Tree Builder Block Diagram

Huffman Tree builder is initially loaded directly from the insertion sort stage by loading the symbol values into the a_value registers and the frequencies into the b_value registers. They are loaded on end of window (eow) from insertion sort engine. Additionally, the depth registers should be initialized to 0 on the load cycle. The top_node registers are a don't care (they will be filled in as the table_row counter traverses the table). A table_row counter will be initialized to sym_unique in order to skip processing for unused symbols (sym_unique is an output of the insertion sort stage) as unused rows will remain in the lowest index rows of the structure. A successful tree build will then complete in 576-sym_unique cycles for symbol set A and 256-sym_unique cycles for symbol set B. If no rebuild is required, then ht_hw_eob = 2'b01/2'b10 is asserted at this time and the build is complete.

The a_value registers will contain either symbols or node pointers. To indicate whether or not the value is a symbol or a node, the symbol values are extended by 1b, and the msb will be set to 1'b1 if it is a node, and remain 1'b0 if it is a symbol. Each a_value register is then 11b wide for symbol set A, and 9b wide for symbol set B.

The top_node and depth registers are kept in a separate structure from the a_value and b_value registers. This structure consists of entries in symbol row order (e.g. entry N will always contain the top_node and depth registers for symbol N). The top_node and depth registers are updated each cycle as shown below for entry N. Note that it will be common for multiple entries in this structure to be updated as many symbols will share the same top node.

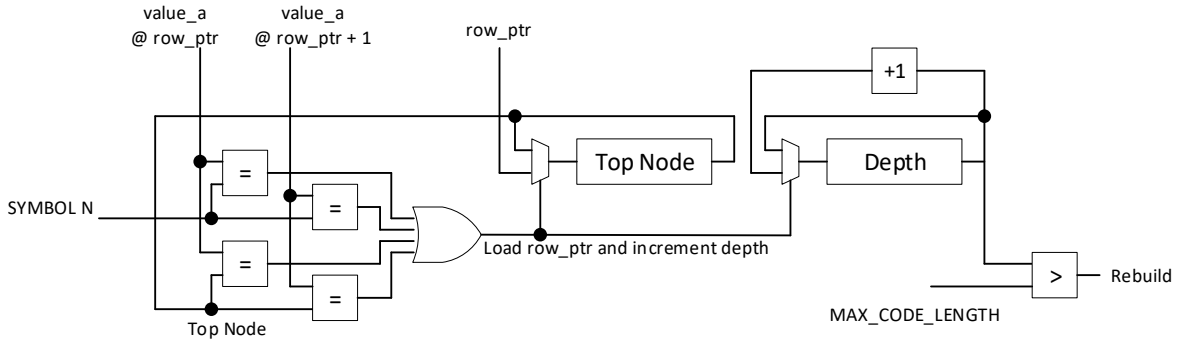


Figure 13 : Tree Builder Logic

If at any time, any row's depth value is > MAX_CODE_LENGTH for the symbol set, then the tree build is restarted and the starting frequencies are halved (specifically, MAX (1, freq >> div_reg)). div_reg is initialized to 1 and increments by one each time a tree build is terminated due to MAX_CODE_LENGTH being exceeded. Note that this means a copy of the original frequencies and symbols in sorted order

from the insertion sort stage needs to be retained in order to reload the table on a restart. A copy of the sym_unique value will also need to be retained.

This iterative process will repeat until the tree is valid, and the process may potentially cause the pipeline to stall in the event that it takes many iterations to produce a valid tree.

MAX_CODE_LENGTH is defined as follows:

Context	MAX_CODE_LENGTH
XP short/long symbol	27
XP table symbol	8
DEFLATE literal/length/symbol/distance	15
DEFLATE table	7

Table 11 : Tree Builder Max Code Lengths

Symbol set A: XP short or DEFLATE literal/length

Symbol set B: XP long or DEFLATE distance

While the table is being generated, the initial frequency values (before any division for rebuilds) are copied into an output array for retrieval by the tree walker for final encode-size calculations. Writes of this memory occur during the first table build for the block only (not during rebuilds). As the tree table is undergoing its initial build, the symbol value (a_value) is used to generate the write location, and the frequency value (b_value) is the frequency to be written to the memory. Since the code length will be 0 for unused symbols and these values are ultimately multiplied by their code length, there is no need to invalidate entries that are not used in each block.

There are two set of memories for the frequency values. An on-line memory which is written by the tree builder and the other, off-line memory which is read by the tree walker. The memories can switch between being on-line and off-line based on which one is free and which one is filled. If both the memories are filled, they backpressure the tree builder and the upstream blocks.

Once the tree is built, it is passed to the corresponding Huffman tree walker (HTW) stage. If the HTW is busy working on previous data, it will backpressure HTB by asserting not_ready. This can lead to back pressuring of Insertion sort engine if both the HTBs are not free.

In case of pass through mode, the tree builder will pass the control information (sequence id and eob) immediately to the downstream block and will not attempt tree build.

Two copies of Huffman Tree Builder(HTB) logic will be instantiated in order to meet the throughput requirements for small independent data streams (i.e. maintain throughput for 4K files, so a minimum of one small (<=4K) file can be processed every 512 clocks). A free instance will start working on building the tree when the Huffman Window ends, and it will be tagged as a busy instance until it has completed its work, at which time it will be marked free once again. If there is no free instance, then the Huffman Tree Builder stage will exert backpressure to Insertion sort engine, potentially causing upstream stages of the pipeline to stall.

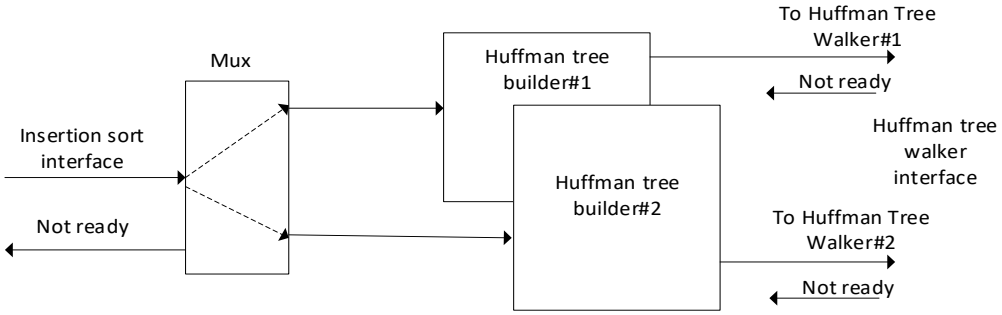


Figure 14 : Two instances of Tree Builder

5.3 Error Handling

If the tree builder fails to build the tree even after the maximum number of rebuilds (See SW accessible registers), it will set error signal ht_hw_build_error to the next stage along with the valid eob to end the build.

5.4 Resources

Flops:

Short path	
Freq_table	14400
Copy	14400
Output code table	2880
Other flops	45

Total	31725

Long path

Freq_table	5376
Copy	5376
Output code table	1280
Other flops	37
Total	12069

RAM:

Short path

Output frequency table	2x288x30

Long path

Output frequency table	2x128x26

5.5 Parameters

Parameter	Symbol Pipe A	Symbol Pipe B	Symbol Table Builder	Comment
IS_SORT_ROWS	576	248	33	Symbol table depth
TW_MAX_DEPTH	27	27	8	Maximum bit length for any format

Table 12 : Tree Builder Parameters

6 Tree Walker

6.1 Overview

This block will walk the Huffman tree created by the previous block and generate canonical codes for both predetermined and retrospective encoding methods. The canonical codes are loaded into memory based look up tables for use by the Output Control block stream assembler.

The Huffman code lengths are also passed to the Huffman header generator for the retrospective method.

6.2 Block Diagram

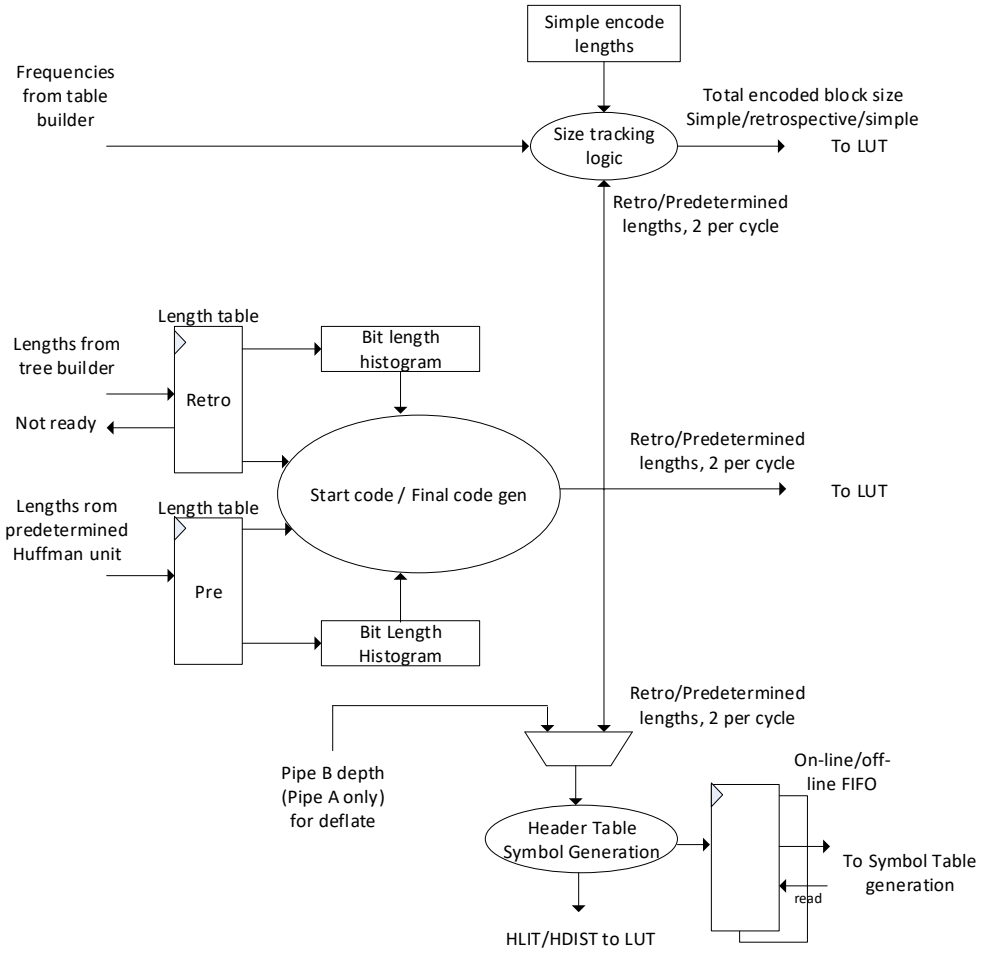


Figure 15 : Tree Walker Block Diagram

The Tree Walker and canonical code generation stage is a 3 step procedure which is kicked off by eob from Huffman tree builder, performed in the following sequence:

1. Sum Code lengths: Each row in the top node/depth table generated in the tree builder step broadcasts a 1b count for each of the 27 possible non-zero depths. It broadcasts a value of 1'b1 for the depth contained in its depth register, and a value of 1'b0 for all others. For each potential Huffman code length (1 to 27), these values are summed across all rows. The accumulation for each code length is done in parallel. Multiple cycles may be needed to complete the addition across the entire table structure. The end result is a histogram of the frequency of each of the possible 27 Huffman code lengths.

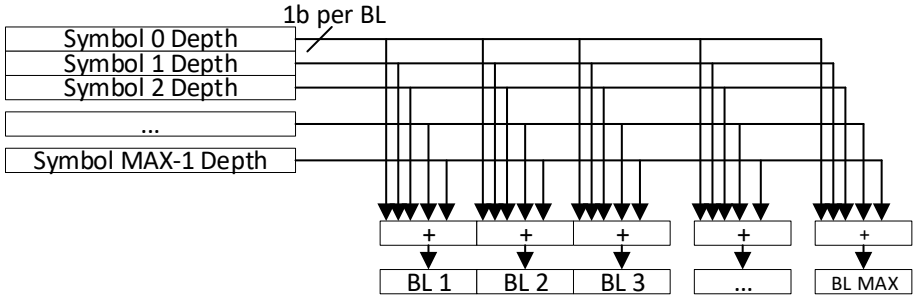


Figure 16 : Tree Walker Bit Length Bins

2. Code length Starting Codes: Each bin in the Code Length histogram computes the starting Huffman code for its code length (algorithm below). Since each bin has a dependency on the previous bin for determining its value, this is done 1 bin per cycle. The process can be stopped once the maximum code length has been reached, for a maximum total latency of 27 cycles.

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_CODE_LENGTH; bits++) {
code = (code + bl_count[bits-1]) << 1;
next_code[bits] = code;
}
```

3. Final Code Gen: The symbols are swept through in symbol order, 2 per cycle (for a total latency of MAX_SYMBOLS_TABLE / 2). The Final Code Gen stage always start with symbol 0 and proceeds all the way to MAX_SYMBOLS_TABLE regardless of which symbols are actually used in order to support the first step of the Symbol Table build. The canonical code assigned to each data symbol is determined by the current value of Huffman code for the code length. Each time a code length is assigned, the bin code for that bit-length is incremented by 1. If both symbols chosen this cycle (symbol n and symbol n+1) have the

same code length, then Symbol n gets the current Huffman code value for that bin and Symbol n+1 gets value code+1 and the current code for that bit length is incremented by 2. In parallel to the symbol code generation, the sequence of *symbol table symbols* for implementing the symbol table is generated. Note: If in DEFLATE mode, this step for the Distance Codes symbols (symbol B pipe) cannot be entered until the Literal/Length codes (symbol A pipe) have completed this step since the symbol table is generated across both symbol sets in sequence in DEFLATE.

4. MAX_SYMBOLS_USED/MAX_SYMBOLS_TABLE is format dependent and defined as follows:

Symbol Set	Search Window Size	MAX_SYMBOLS_USED	MAX_SYMBOLS_TABLE
XP10 short	64kB	576	576
XP10 long	64kB	248	248
XP10 short	16kB	544	544
XP10 long	16kB	246	246
XP10 short	8kB	528	528
XP10 long	8kB	245	245
XP10 short	4kB	512	512
XP10 long	4kB	244	244
XP table	Any	33	33
DEFLATE literal/length	32kB	286	286
DEFLATE distance	32kB	30	30
DEFLATE table	32kB	19	19

Table 13 : Tree Walker Max Symbols

In the table above, MAX_SYMBOLS_USED used represents the maximum number of symbols that could be used for processing the block. MAX_SYMBOLS_TABLE represents the maximum number of symbols that need to be defined in the symbol tables when inserted into the output stream. For XP10, MAX_SYMBOLS_TABLE symbols will always be defined. In DEFLATE, the actual number defined in the table may be less (As specified in the HCLEN/HLIT/HDIST fields of the header).

MAX_CODE_LENGTH is defined as follows:

Context	MAX_CODE_LENGTH
XP short/long symbol	27
XP table symbol	8
DEFLATE literal/length/symbol/distance	15
DEFLATE table	7

Table 14 : Tree Walker Max Code Lengths

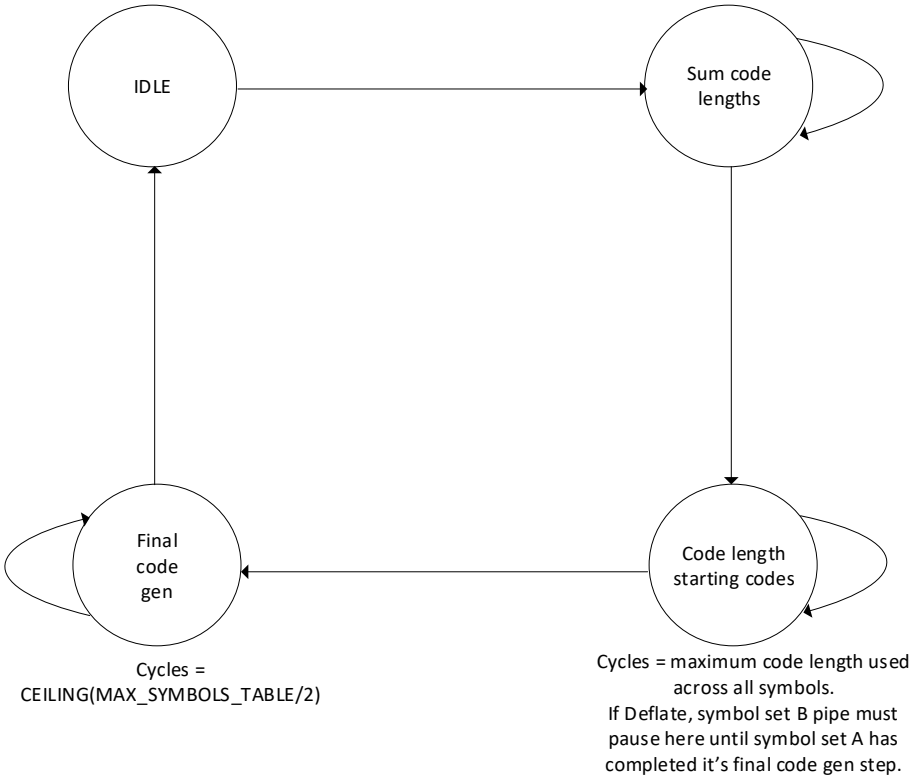


Figure 17 : Tree Walker State Transitions

As the Canonical codes are generated in the Final Code Gen stage (2 per cycle), the codes are written into the LUT via the LUT interface. Note that a set of LUTs must be available prior to starting the Final Code Gen stage otherwise the pipe is stalled. In addition to the bit-code, the bit length of the code will also be written into the LUT for each symbol. LUT entries are not written for symbols that are > MAX_SYMBOLS_USED – 1.

At every start of block, SOF fields are read out of a set of eight SOF contexts residing external to this block which are indexed by the sequence id. These fields provide various control information.

If predetermined Huffman is found to be enabled for the block being processed, the canonical code for the predetermined set of code lengths is generated in parallel. To accomplish this, a parallel table of code lengths will be loaded with the values from the local Predetermined Huffman buffer based on the sequence id.

For XP10 formats where the symbol space is reduced, it is expected that the predetermined Huffman table delivered will still be the max size predetermined table that Corisca supports (a XP10 64k window) with the trimmed symbols zeroed out.

The steps for the Canonical Huffman code generation for the predetermined case are then the same as the retrospective case and are controlled by the same state machine as the retrospective case and calculated in parallel.

In addition to the generation of the canonical codes for each symbol in the Final Code Gen stage, the total size of the encoded data symbols for both retrospective, predetermined Huffman, and simple encode mode is calculated. As the symbols are stepped through 2 at a time, the predetermined code lengths, retrospective code lengths, and simple encode code lengths are multiplied by the frequency count for each symbol. These values are read from the symbol frequency memory in the Tree Builder. A separate count for retrospective, predetermined, and simple encode is maintained. Each cycle, the calculated total for each symbol (up to two) is added to the total count. The absolute worst case sum (which is impossible to actually achieve) would be 27*32768=884736 bits, so a 20 bit counter for each will suffice (For symbol pipe B, an 18 bit counter will suffice). The final value of the three counts is written to the LUT unit when the canonical code generation is completed.

The bit lengths for simple encode are defined as follows (Simple encode mode is only a feature of XP, not DEFLATE):

Symbols	Bit-length
XP10 64k short	
0-447	9
448-575	10
XP10 64k long	
0-7	7
8-247	8
XP10 16k short	
0-479	9
480-543	10
XP10 16k long	
0-9	7
10-245	8
XP10 8k short	
0-495	9
496-527	10
XP10 8k long	
0-10	7
11-244	8
XP10 4k short	
0-511	9
XP10 4k long	
0-11	7
12-243	8

Table 15 : Tree Walker Max Bit Lengths per Encode Method

As the Symbol Lengths are walked through in symbol order in the Final Code Gen stage, the lengths will also be mapped into Huffman table symbols two at a time using the mapping algorithms in the subsections below. The mapping is dependent on which output format is being used (DEFLATE or XP). Note that in XP mode, this step is run in parallel for each symbol table. However, in DEFLATE, the Literal/Length symbols will be stepped through first, followed by the Distance symbols.

In DEFLATE, the symbol table build will only continue in the symbol A pipe (literal/length pipe). A mux control unit will coordinate the coupling of the Tree Walker unit in the B pipe to the proper A pipe instance of the Symbol Table Builder based on the sequence id. Once pipe A's Tree Walker has completed its Final Code Gen step, the Final Code Gen for symbol tree B may start its Final Code Gen step. The symbol lengths (depths) (2 per cycle) will be presented to stage A for header table symbol generation as well as used in stage B for data symbol LUT generation and total data bit count.

The symbol table will be written into a FIFO. This FIFO module is a combination of two FIFOs where the *on-line* FIFO is written by the tree walker while the off-line FIFO is being processed by the next hardware block (Symbol table builder).

The tree walker will be stalled if none of the symbol table FIFOs or the LUTs are free for the next Huffman block. The Huffman tree walker (HTW) block will backpressure Huffman Tree Builder (HTB) by asserting not_ready if it is busy processing the previous Huffman block.

The outputs to header table generator will be zero if the symbol table FIFOs are empty.

In case of pass through mode, the tree walker will pass the control information (sequence id and eob) immediately to the downstream block and will not attempt data processing.

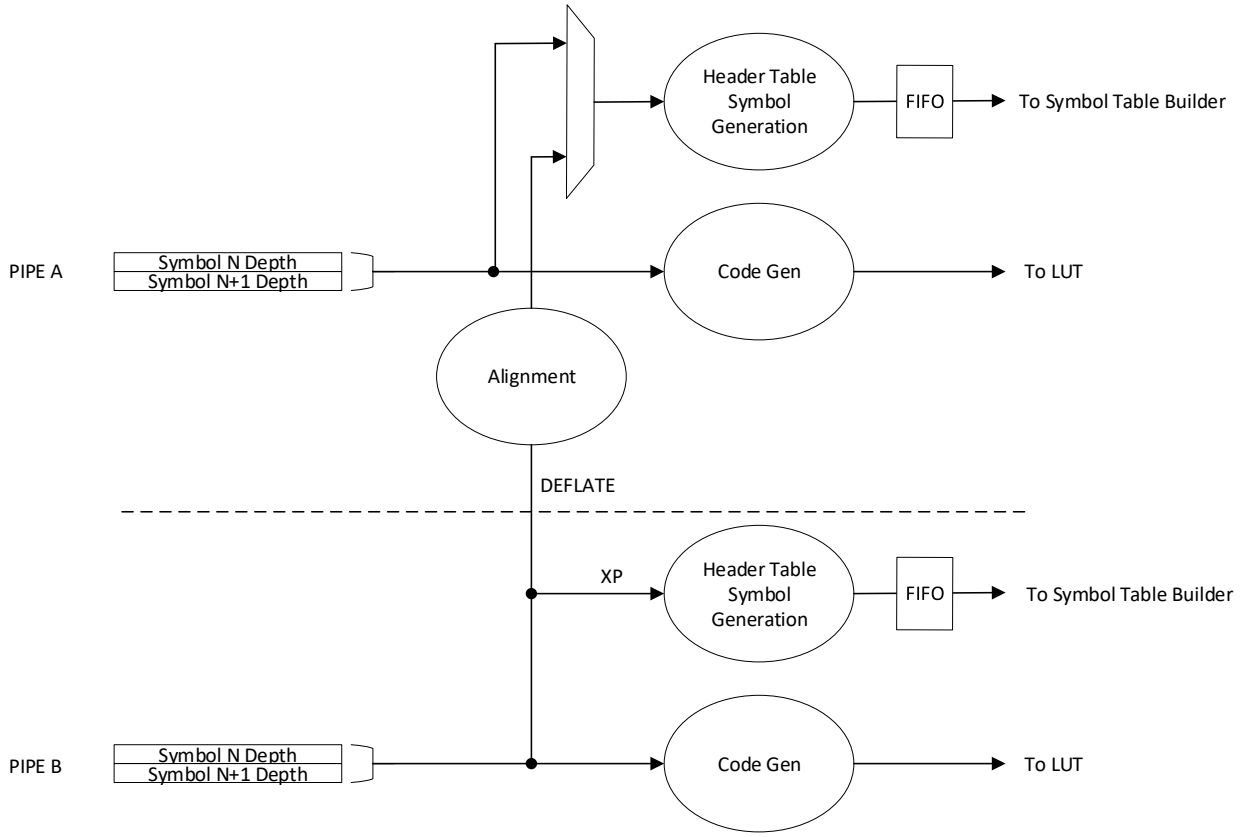


Figure 18 : Tree Walker Output

6.2.1 Header Symbol Generation Algorithms

The number of symbols that the symbol table covers (#symbols in the algorithm below) varies by standard and other parameters. The table below defines #symbols for all cases of interest:

Standard	Window	Symbol Set	#symbols
XP10	64k	Short	576
XP10	64k	Long	248
XP10	16k	Short	544
XP10	16k	Long	246
XP10	8k	Short	528
XP10	8k	Long	245
XP10	4k	Short	512
XP10	4k	Long	244
DEFLATE	N/A	Literal/Length*	hb_hw_sym_high

DEFLATE	N/A	Distance*	MAX(1,hb_hw_sym_high)#
DEFLATE	N/A	Total	Literal/Length + Distance

Table 16 : Tree Walker Symbol Set

* Literal/Length and Distance symbols are combined into 1 table, reflected in the Symbol Set == Total row.

if there are zero Distance codes used, one distance code of zero must be added to the end of the table. This case is communicated to the symbol A pipe through the hwb_hwa_no_code input signal.

The worst case number of symbols that may be produced on a cycle is 6.

This worst case occurs in if the code lengths processed this cycle are X and Y, neither are zero, and previously there had been a run of 4 0s that did not cross a fill boundary. On this cycle 4 zero-length symbols, and a symbol for X and Y may be generated for a total of 6.

In DEFLATE mode, HLIT, HDIST are calculated based on the sym_high inputs are sent to the LUT unit on the same cycle as the final encoded size data.

6.2.1.1 XP10 Header Symbol Generation

Definitions of the header symbols are located in the XP10 spec.

A counter to track extra bits that will be encoded within the header symbol is needed to be sent to the symbol table builder.

6.2.1.2 DEFLATE Header Symbol Generation

Definitions of the header symbols are located in the DEFLATE spec (rfc1951-version 1.3; Section 3.2.7-Compression with dynamic Huffman codes (BTYPE=10)).

A counter to track extra bits that will be inserted into the header symbol table is needed to be sent to the symbol table builder. Note that alignment logic is necessary if the size of (Encoded Literal/Length Symbols) is odd. An odd number of Literal/Length symbols mean that the last literal/length symbol that a symbol table code has generated cannot be processed until the distance codes arrive. (In other words,

the last literal/length code would be processed with the first distance code, and thereafter the second code length received on the previous cycle from the symbol B pipe would be paired with the first code length of symbol A pipe received on the current cycle).

6.3 Error Handling

build_error input from the tree builder stage doesn't affect predetermined processing. The error information bypasses retrospective processing and passes the eob and sequence id along the symbol table builder pipe as if it is in bypass mode to the stream assembler.

In case of xp, it waits for the eob for the predetermined canonical codes to be written into the LUT before passing the build_error to the symbol table builder.

6.4 Resources

Flops:

Short path	
Input code length table	2880
Predetermined data store	60
Step 1 adder	486
Step 2 starting codes	31104
Other flops	84
Total	34614
Long path	
Input code length table	1280
Predetermined data store	60
Step 1 adder	486
Step 2 starting codes	13824
Other flops	84
Total	15734

RAM:

Short path

Symbol table interface	
FIFO	2X352x32

Long path

Symbol table interface	
FIFO	2X128x32

7 Huffman Encoder Look Up Table

7.1 Overview

This block has LUTs and store all the size information and tables for Huffman encoder. It receives the data from tree walker and symbol table builder. Stream assembler consumes the data from this block.

The Huffman Look up Table (LUT) is double buffered so that one buffer can be updated while the other buffer is in use. The buffer that is actively being used to translate Huffman Symbols into Huffman Codes is called the “On-line Buffer”, and the other buffer, that is potentially being updated, is called the “Off-line Buffer”.

Each buffer is able to translate multiple Huffman Symbols into Huffman Codes every cycle. Huffman#1 supports up to four transactions per cycle and Huffman#2 supports up to two transactions per cycle.

7.2 Block Diagram

The LUT block for each pipe manages shared data for 2 blocks (Tree Walker and Symbol table builder):

- Symbol pipe A only: 4 instances of the data LUT (supports 4 simultaneous lookups per cycle)
- Symbol pipe B only: 1 instance of the data LUT
- The encoded Symbol table symbol code lengths for the header (stcl)
- The encoded Symbol table (st)
- Any other per block data needed for the stream assembler/control unit:
    - Symbol table code length size (in bits)
    - Symbol table size (in bits)
    - HLIT
    - HDIST
    - HCLEN
    - Retrospective Huffman encoded data size (in bits)
    - Predefined Huffman encoded data size (in bits)
    - Simple encoded data size (in bits)

When writing the data symbol LUT, all instances within a set are updated at the same time (red LUTs in the diagram below), two symbols at a time. When reading the data symbol LUT (shown in green in the diagram below), up to 4 symbols (1 symbol for the B pipe) may be read each cycle by the Stream Assembler unit. In order to save the latency of having to calculate predetermined codes to replace the retrospective codes, both are stored for each symbol during the tree walk.

The diagram below shows the A pipe data symbol LUTs:

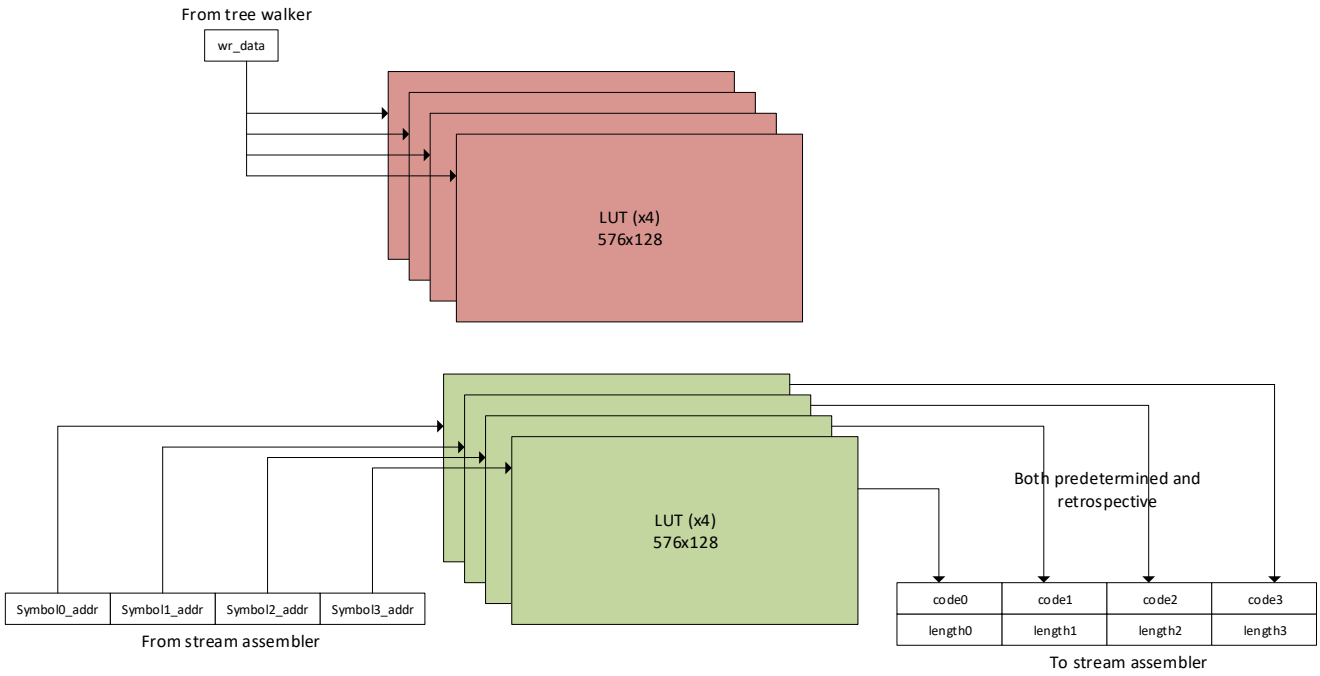


Figure 19 : LUT Block Diagram Short Symbols

The diagram below shows the shared XP long symbol and DEFLATE distance LUTs:

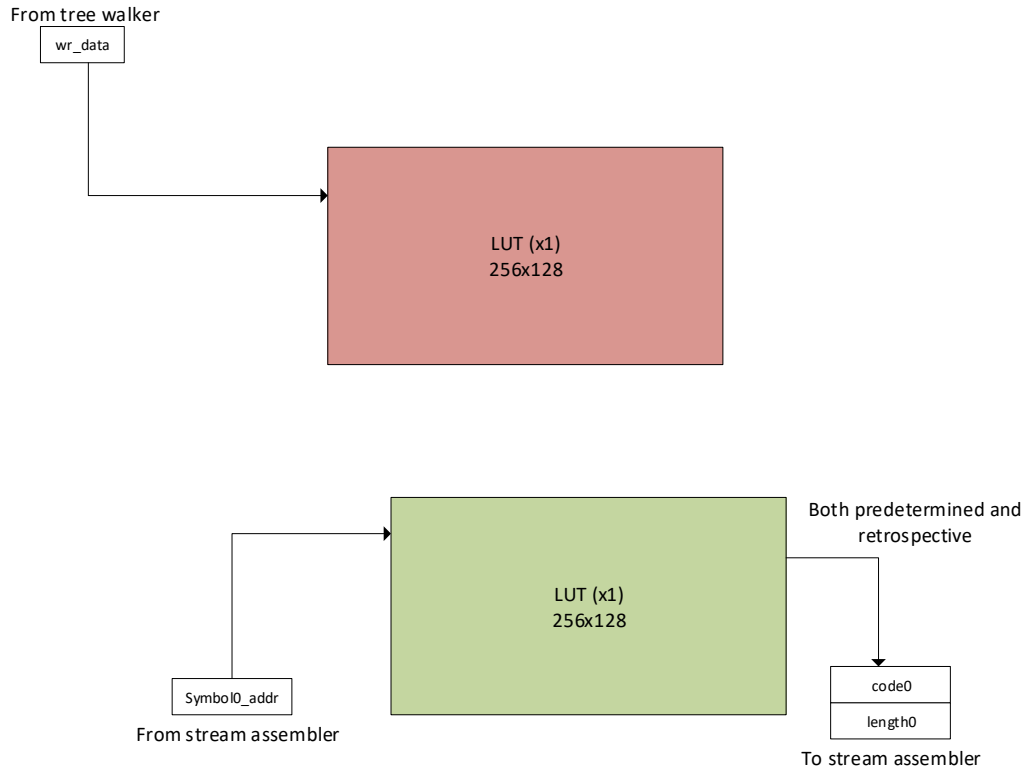


Figure 20 : LUT Block Diagram Long Symbols

Each memory entry is the same regardless of symbol table. Each address holds the codes and code lengths for 2 symbols.

Bit	Field
127:101	Symbol A Retrospective Code
100:96	Symbol A Retrospective Code Length
95:69	Symbol A Predetermined Code
68:54	Symbol A Predetermined Code Length
63:37	Symbol A+1 Retrospective Code
36:32	Symbol A+1 Retrospective Code Length
31:5	Symbol A+1 Predetermined Code
4:0	Symbol A+1 Predetermined Code Length

Table 17 : LUT Entry Format

If stream assembler requests LUT read for an unknown id or for a pass through block, the valids for the data bus are deasserted.

In pass through mode, all the size information to the stream assembler is zero.

7.3 Resources

Flops:

Short path	
ST code lenth table	132
Size variables	138
Total	270
Long path	
ST code lenth table	132
Size variables	124
Total	256

RAM:

Short path	
Huffman code	8x288x64
Symbol table(ST)	88x64
Long path	

Huffman code	8x128x64
Symbol table(ST)	39x64

7.4 Parameters

Parameter	Symbol Pipe A	Symbol Pipe B	Symbol Table Builder	Comment
IS_SORT_ROWS	576	248	33	Symbol table depth
TW_MAX_DEPTH	27	27	8	Maximum bit length for any format

Table 18 : LUT Parameters

8 Symbol Table

8.1 Overview

This block takes in the header symbols (compressed symbol table) from the Huffman tree walker and generate Huffman encoded symbol table for the stream assembler. This is a multi-step process that will reuse the Insertion Sort, Tree Builder and Tree Walker blocks to encode the compressed Huffman symbol (code lengths) table.

The final Huffman Header will be constructed for use by the Output Control block stream assembler.

8.2 Block Diagram

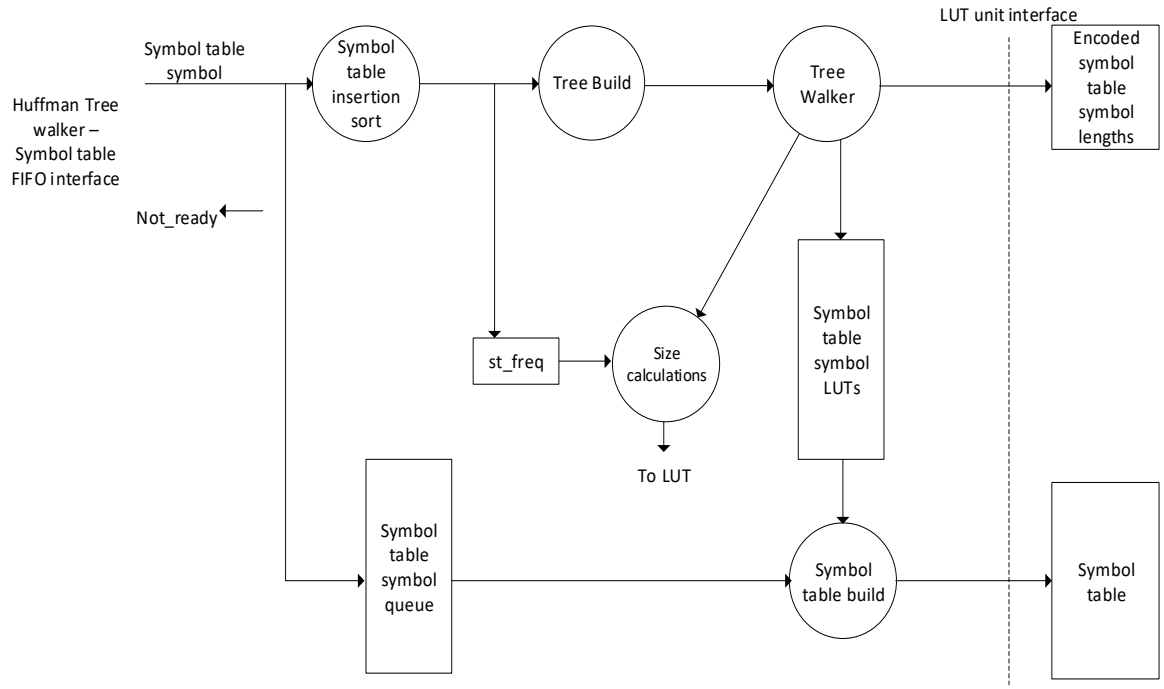


Figure 21 : Symbol Table Block Diagram

At every start of block, SOF fields are read out of a set of eight SOF contexts residing external to this block which are indexed by the sequence id. These fields provide various control information.

There are 4 steps performed by the Header Tree Builder and Encoder stage (Only first 3 if a non-retrospective mode is chosen), done in the following sequence (Note that before Tree build is started, the Symbol Table Builder is receiving symbol table symbols from the Tree Walker):

1. Insertion Sort: As the Length codes are converted into header table symbols by the tree walker stage, those symbol table symbols (up to 2 per cycle) are the inputs of an insertion

sort stage in the Symbol Table Builder. This block can re-use the generic insertion sort module, parameterized as follows:

Number of insertions per cycle: 2
Max Number of Symbols: 33
Max Frequency: 10b count (576 symbols max)

2. Tree Build: The Huffman Tree build of the symbol table (33 cycles of latency for XP, 19 cycles of latency for DEFLATE. There is always potential for a rebuild due to the max code length being exceeded).
3. Tree Walker: Canonical start codes for each bit length (1-8) are calculated. Translation of the bit lengths calculated in step 2 to the final encoded form of the *symbol table symbol code length* values that are sent in the header. At this point, all data to make the decision on RAW vs. simple vs. predetermined vs. Retrospective is available. Additionally, the canonical codes that will be used if the symbol tables are actually encoded, are generated.
4. Symbol Table Build: Build the actual Symbol table if retrospective was chosen as the best encoding format. Store it in 74(32 for pipe B) x64b entries in the LUT unit.

In addition to being sent to the insertion sort unit, the generated symbol table symbols need to be stored into a small array for the Symbol Table Build step (the symbol table symbol queue). To support this, a 352x26 array for the Symbol A pipe and a 128x26 array for the Symbol B pipe is used (this supports read of two at a time during the symbol table build). See the following table for the entry description. Symbols should be packed so there are no empty symbol slots. The first two symbols will be at address 0. A count of total symbols will be kept externally in order to know where the last entry is. In addition, the total count of actual extra bits (encoded length of zero repeats or symbol repeats in case of deflate) received from the header symbol generator of tree walker, needs to be stored for the final encoded data stream size calculation (st_extra_bits).

Bits	Description	Content
25:19	Extra bits for Symbol 1	XP: if (symbol 0 - 29), takes on values of 5-15 decimal. To be translated into actual encoded extra bits during Final Encode Stage of The symbol table build. DEFLATE: Represents the final extra bits for the following symbols. 16/17: repeat value – 3 18: repeat value - 11
18:13	Symbol Table Symbol 1	0-33
12:6	Extra bits for Symbol 0	XP: if (symbol 0 - 29), takes on values of 5-15 decimal. To be translated into actual encoded extra bits during Final Encode Stage of The symbol table build.

		DEFLATE: Represents the final extra bits for the following symbols. 16/17: repeat value – 3 18: repeat value - 11
5:0	Symbol Table Symbol 0	0-33

Table 19 : Header Table Symbol Queue Entry

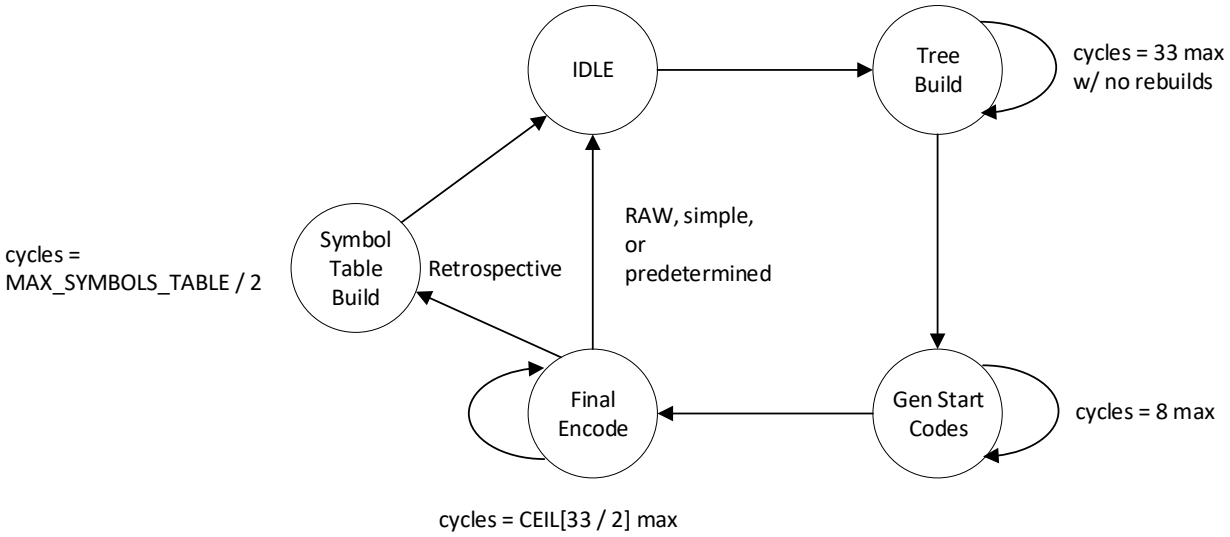


Figure 22 : Symbol Table Tree Builder State Transitions

Tree Build: The module for creating the Huffman table tree on this histogram is the same as the Huffman Tree Build from earlier in the pipe, with the following changes:

- Max Frequency: 10b count
- XP max rows in table is 33, DEFLATE max rows in table is 19
- Rebuilds are triggered once depth exceeds 8 in XP, 7 in DEFLATE

Note that the original frequencies will need to be maintained for size calculations once the final tree build is complete. Similar to the tree build process for the data symbols, as each row is processed in the sorted histogram, the original count for each symbol is stored into a 33XP10 structure. The address for each write is the symbol value, and the data written is the frequency of that symbol (st_freq).

The histogram for the symbol table symbols is combined across Literal/Length and Distance symbols in DEFLATE. For XP, the histograms and Huffman encodings are performed separately in each pipe. Therefore, for DEFLATE, the symbol table generation will be done using the resources in the literal/length pipe (symbol set A pipe) only.

The end result of this step is a code length for each symbol in the Huffman symbol table alphabet, which is needed for the final table encoding and symbol table LUT generation.

Tree Walk: The engines from the generic tree walker are reused.

Gen Start Codes: Similar to the Code Length Starting Codes step in the Huffman Tree Walker. The Initial value for each the canonical code for each bit length is calculated. For XP, this process takes 8 cycles, for DEFLATE 7 cycles.

Final Encode: The next step is to translate the symbol table symbol code lengths into their final encoded formats as described in section 1.1.1. The final encoding is packed and written to the lut 64b at a time. In parallel, the canonical symbol table symbol codes to encode the symbol table symbols will be generated and stored in a symbol table symbol LUT for the symbol table build step (if needed).

In addition, the size (in bits) of the final encoded symbol table symbol code lengths is calculated (st_stcl_size). Also, the size of the final encoded symbol table is calculated by multiplying the frequency from the st_freq structure (built during the Tree Build stage) by the bit code length for each symbol table symbol. This count is maintained in a st_size_sym register. At the end of the final encode step, the total size of the encoded symbol table is sent to the LUT as st_size_sym + st_extra_bits.

Symbol Table Build: This step is only performed if the retrospective encoding mode was chosen. To generate the symbol table, a counter is initialized to zero when entering this step and increments by 2 every cycle until the end of the symbol table symbols is reached (MAX_SYMBOLS_TABLE-1). Each cycle 2 symbol table symbols are translated into their encoded value, and along with the extra bits, assembled into the header table for later reading by the stream assembly unit. In case of XP, the extra bits are also encoded in this step. The data is written to the LUT unit 64b at a time.

The rules for packing into bytes are listed in section 3.1.1 of the Deflate spec (RFC 1951).

The rules for XP are similar and are located in section 4.1 of the XP10 spec.

The following table illustrates how the extra bits are translated from how they were stored in the symbol table symbol queue for the HUFFMAN_ENCODED_TABLE_ZERO_RPT symbol.

Consecutive 0s	Non-Encoded bits that follow the symbol
5	2'b00
6	2'b01
7	2'b10
8	5'b11000
9	5'b11001
10	5'b11010
11	5'b11011
12	5'b11100

13	5'b11101
14	5'b11110
15	8'b11111000

Table 20 : Symbol Table Extra Bit Encoding

The final symbol table is stored in the LUT unit for reading by the stream assembler. Worst case size for the symbol A pipe is 584 symbols (576 + 8 table fills) * 8 bit length max in addition to the extra bits = 4672 bits + st_extra_bits. A 78x64 memory will suffice. Worst case size for the symbol B pipe is 249 symbols (248 + 1 table fill) * 8 bit length max in addition to the extra bits = 1992 bits + st_extra_bits. A 34x64 memory will suffice.

8.2.1 Final Encode

8.2.1.1 XP

For XP, the symbol table code lengths will be stepped through 2 at a time starting with symbol 0, for a max latency of 17 cycles.

Each cycle, the final encode for two symbols will be calculated based on the algorithm below.

What follows next is the Huffman encoded table code lengths (0-32), encoded using the following algorithm:

Previous symbol length prev = 4

For each symbol

K= length of the the symbol

If (k == prev) Write 1'b0

Else

Write 1'b1.

If (k > prev) write 3'b(K-1) else write 3'bk

Prev = k;

End foreach

The final encode values are assembled and stored in the LUT unit, sent 64b at a time. A max of 33*4=132 bits is needed for this storage.

The canonical values are also stored in a 33x8 LUT, using the same algorithm as used in the Huffman Tree Walker.

8.2.1.2 DEFLATE

For DEFLATE, the symbol table code lengths will be stepped through 2 at a time. However, they are not stepped through in symbol order. Instead, they are stepped through in the order specified in the DEFLATE spec. The table below lays out the cycle by cycle symbol order walk (Symbol A is sequentially earlier than Symbol B in the overall sequence):

Cycle	Symbol A	Symbol B
0	16	17
1	18	0
2	8	7
3	9	6
4	10	5
5	11	4
6	12	3
7	13	2
8	14	1
9	15	N/A

Table 21 : Symbol Table Walk Order

Each cycle, the following occurs:

- Symbol A and Symbol B code lengths are generated. They are concatenated as 3b lengths (so 6b total between the two symbols) as described in the DEFLATE spec for the HCLEN field. The total size cannot exceed 57b (19 *3) and is transferred to the LUT on completion of the final encode step.
- The last non-zero value observed should be tracked to determine the value of HCLEN. It is tracked in a 5b register as follows.

```
last_nonzero = 0
for (cycles = 0; cycles < 10; cycles++) {
 if (sym_b_length > 0)
   last_nonzero = cycles*2 + 1
 else if (sym_a_length > 0 )
   last_nonzero = cycles*2
}
```

Note that by definition the first 4 codes (16, 17, 18, 0) will always need to be sent.

Once the final encode is completed, the value of HCLEN is calculated and sent to the LUT unit:

HCLEN = MIN(4,last_nonzero+1)

The value of stcl_size is HCLEN * 3 and is also sent to the LUT unit.

In parallel, the symbols are stepped through in symbol order (2 at a time) in order to generate the canonical values to store in a 33x8 LUT (same algorithm as used to generate the canonical codes as the Huffman Tree Walker step). The st_size calculation is also accumulated at this time.

In case of pass through mode the symbol table builder will pass the control information (sequence id and eob) immediately to the stream assembler and will not attempt data processing.

8.3 Error Handling

In case of build_error (due to either hw_st_build_error or symbol table tree build error), the symbol table builder will pass the control information (sequence id and eob) immediately to the stream assembler and will not attempt data processing.

If st_sa_size_rdy is asserted with st_sa_build_error, then only the information related to predetermined and simple encodings are valid. Stream assembler may not expect st_sa_table_rdy in such case.

8.4 Resources

Flops:

Short path	
Insertion sort	545
Tree Build	990
Tree Walker	454
Internal LUT	264
Symbol queue	9152
Counters	21
Total	11426
Long path	

Insertion sort	545
Tree Build	990
Tree Walker	454
Internal LUT	264
Symbol queue	3328
Counters	21
Total	5602

8.5 Parameters

Parameter	Symbol Pipe A	Symbol Pipe B	Symbol Table Builder	Comment
IS_SORT_ROWS	576	248	33	Symbol table depth
TW_MAX_DEPTH	27	27	8	Maximum bit length for any format

Table 22 : Symbol Table Parameters

9 Symbol Queue

9.1 Overview

The Symbol Queue is a FIFO buffer to hold entries from the LZ77 interface. Side band signals will be provided by the Symbol Mapper that will mark boundaries between frames, blocks and TLV headers. The Symbol Queue will be sized to hold all TLV headers, including prefix data, as well as 8K payload entries from the LZ77 Compressor.

The Huffman Window Size, measured in entries, is 8K. This is roughly equivalent to 32Kbytes of uncompressible data.

The maximum prefix size in entries is roughly 64K/(8 bytes per entry), although it may be slightly larger due to packing inefficiency. As a result, 8K entries will be reserved for prefix data.

Another 2K entries will be reserved for latency through the Huffman Control path stages. An additional 2K entries will be reserved for additional TLV headers that are passed through to downstream blocks.

To meet the 64Kbyte Window size requirement for the CCEIP64 design, the Symbol Queue will be sized to store 8K+8K+2K+2K=20K entries.

10 Reconstructor

10.1 Overview

The Stream Assembler will configure the Reconstructor, then send prefix and compressed data to the Reconstructor as shown in Figure 23. Note, the Reconstructor MTF and LZ77 sub-blocks are reused from the Decompression block. Please refer to the Decompression micro architecture document for details.

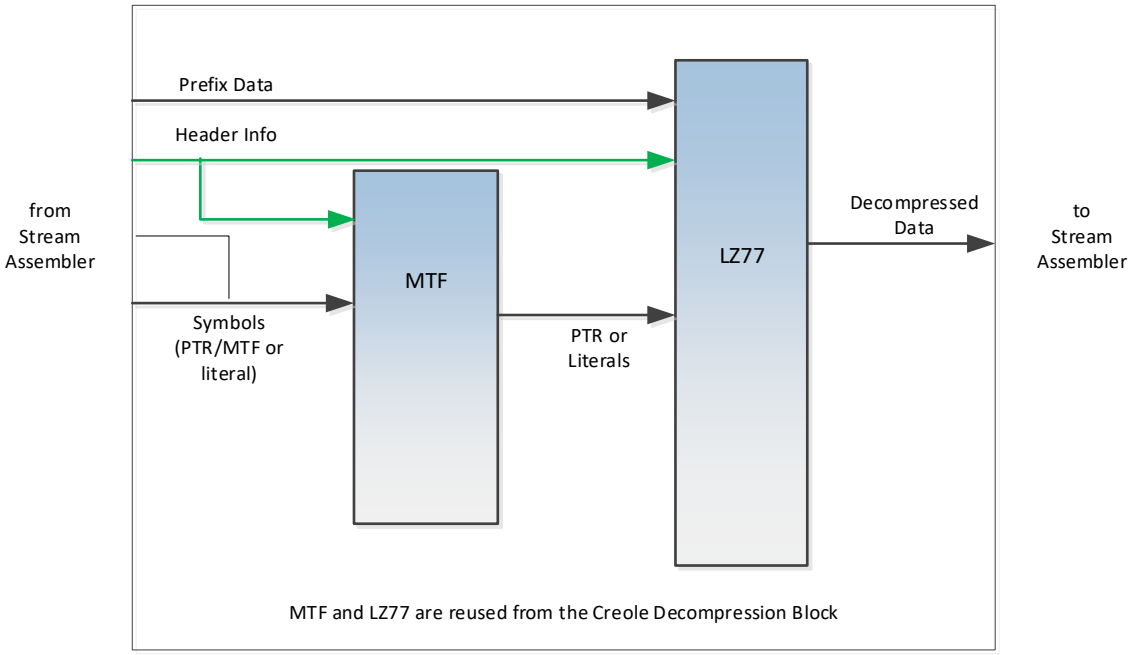


Figure 23 : Reconstructor

11 Stream Assembler

11.1 Overview

The Stream Assembler will wait for the Header Builder in each symbol path to finish before starting the stream assembly process. The Stream Assembler will read data out of the Symbol Queue and will process the TLV encoded data. All of the TLV types not associated with compression will be passed through to the Encryption interface unmodified. The only TLV types of interest to the Stream Assembler block are the Compression and Payload TLV types. The Stream Assembler will modify the XP10_uncompressible_data field in the Compression TLV if the payload data is sent raw instead of compressed.

The Stream Assembler block will perform similar mapping functions as detailed in sections XP10 Symbol Mapping and Deflate Symbol Mapping.

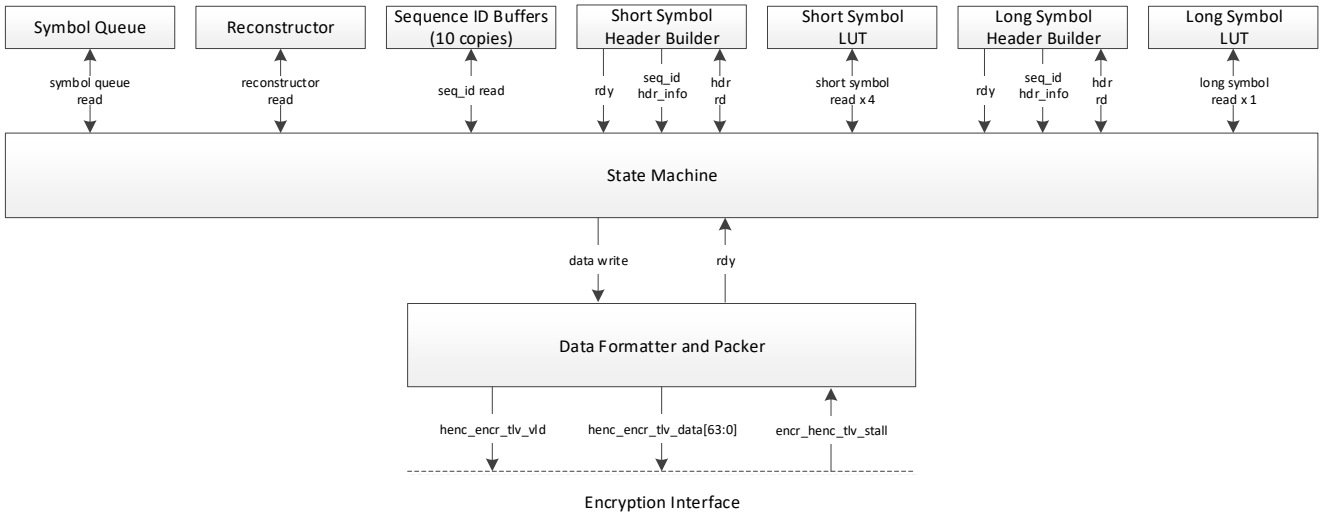


Figure 24 : Stream Assembler Overview

11.2 Final Encoding Decision

Once the Header Builder has completed the Final Encode step (for both the short and long symbol pipes in XP, short pipe only for DEFLATE), the final decision of the output format to use is made. If the retrospective mode is chosen, then the Symbol Table builder will generate the final symbol table for insertion into the block header. Note that in XP mode, the decision is independent for each symbol pipe

(e.g. long symbols could be encoded with simple encode while the short symbols are encoded in retrospective mode).

The following sections will describe the necessary comparisons for each format and mode to determine the final encoding decision.

11.2.1 Deflate GZIP/ZLIB

For each block, a decision will be made as to whether the block will be sent out encoded or raw. For the comparison there will be up to 7 unused bits in the raw stream between BTYPE and the LEN field. The BFINAL/BTYPE bits themselves are common to either format and are ignored in the comparison.

deflate_size_raw

The raw size in bits is the sum of:

- up to 7 (unused bits between BTYPE and LEN field)
- 32 (LEN/NLEN fields)
- block_size_raw (from the Symbol Mapper via the Sequence ID Control Buffer) * 8

deflate_size_cmp

The compressed size in bits is the sum of:

- 14 (sizeof (HLIT + HDIST+HCLEN))
- lut_sa_shrt_stcl_size[12:0] size in bits of the encoded symbol table code lengths, from the pipe short LUT data set
- lut_sa_shrt_st_size[12:0] size in bits of the encoded symbol table symbols, from the pipe short LUT data set
- lut_sa_shrt_ret_size[19:0] size in bits of the encoded symbols, from both the short and long LUT data set
- The extra data bit count for the block (from the Offset Length Bit Counter for the block)

Raw encoding will be used if: **deflate_size_raw <= deflate_size_cmp**

11.2.2 XP10

XP10 will use the 8K symbol entry boundary for all Huffman blocks. XP10 will also add a predetermined Huffman table option and may also select raw encoding for each Huffman block. There are 2 main modes of XP10, basic and CFH 4K/8K.

11.2.2.1 XP10 Basic

A comparison is made between the raw data and the compressed data. The smaller result is used as output. The decision process is described below.

XP10_long_size_ret

For the long symbol pipe, retrospective is the sum of:

- lut_sa_long_ret_stcl_size[12:0]
- lut_sa_long_ret_st_size[12:0]
- lut_sa_long_ret_size[19:0]

XP10_shrt_size_ret

For the short symbol pipe, retrospective is the sum of:

- lut_sa_shrt_ret_stcl_size[12:0]
- lut_sa_shrt_ret_st_size[12:0]
- lut_sa_shrt_ret_size[19:0]

XP10_long_size_sim

For the long symbol pipe, simple encode is the sum of:

- lut_sa_long_sim_size[19:0]

XP10_shrt_size_sim

For the short symbol pipe, simple encode is the sum of:

- lut_sa_shrt_sim_size[19:0]

XP10_long_size_pre

For the long symbol pipe, simple encode is the sum of:

- lut_sa_long_pre_size[19:0]

XP10_shrt_size_pre

For the short symbol pipe, simple encode is the sum of:

- lut_sa_shrt_pre_size[19:0]

XP10_size_cmp

The total bit count for compressed is then the sum of:

- MIN (**XP10_long_size_ret, XP10_long_size_sim, XP10_long_size_pre**)
- MIN (**XP10_shrt_size_ret, XP10_shrt_size_sim, XP10_shrt_size_pre**)
- Extra data bit count for the block (from the Offset & Length Bit Counter for the block)
- 48 or 64 (size of the XP10 block header) If HENC_XP10_FLG_EXTRA = 1, then insert HENC_XP10_FLG[15:0]
- 32 (size of the XP10 fixed block header)
- Variable length MTF block header, present if the current block is not the 1st block of the frame and the previous block was sent raw.
- 6 (additional bits to specify retrospective vs simple, 3 bits for short, 3 bits for long)

XP10_size_raw

The raw size in bits is:

- block_size_raw (from the Symbol Mapper via the Sequence ID Control Buffer) * 8

Raw encoding for the frame should be used if: **XP10_size_raw** <= **XP10_size_cmp**

11.2.2.2 XP10 CFH 4K/8K

There are a few changes to the "XP10_size_cmp" calculation of XP10 CFH 4K/8K modes, depending on configuration bits in the SOF token:

- CFH_reduce_max_comp_frm_size = 1 Subtract 16 from **XP10_size_raw** before making the comparison
- CFH_uncompressed_data_bypass = 0 Always select XP10_size_cmp, regardless of XP10_size_raw
- CFH_frame_header_disable = 1 Do not send the XP10 frame header, save 48 or 64 bits
- CFH_block_header_format = 0 Always insert an XP10 block header
- CFH_block_header_format = 1 Use reduced XP10 block header, only include:

output_size[17:0]

- CFH_block_header_format = 2 include: Use reduced XP10 block header, only output_size[17:0] and prefix[5:0]

11.3 State Machine

In Idle, the State Machine will monitor the ready status signals from the Short and Long Symbol Header Builders. When both are ready with the same sequence ID, the State machine will start processing the TLV types stored in the Symbol Queue. The State Machine will simply pass all TLV types, unmodified to the Encryption interface. The only exception is the Compression TLV. If an XP frame is to be sent raw, the XP10_uncompressible_data field will be set and the CRC regenerated.

When the Payload TLV is found, the data will either be sent raw or compressed, as determined by the final encoding decision as described in section 11.2. If data is to be sent compressed, the frame and block headers and footers will be generated as described in a later section.

The Stream Assembler controls the output data interface and is responsible for assembling the output data stream. The start of a transfer will be initialized when the Long and Short Header Builders are done. All non-Compression TLVs are transmitted unmodified to the Encryption block. When the Compression TLV is encountered, a state machine inside of the stream assembler will be triggered that progresses through the stages below.

The sequence through the state machine is described below.

1. Send the Compression TLV
2. Send Prefix TLV (if required)
3. Insert the Frame Header (if required)
4. Repeat as needed to complete the frame
    a. Insert the Block Header (if required)
    b. Insert the MTF offsets header (if required)
    c. Insert the Symbol tables (if required)
    d. Insert the Data
    e. Insert the Block Footer (if required)
5. Insert the Frame Footer (if required)

11.4 XP10 Framing

11.4.1 XP10 Frame Header

Bit(s)	Field	Source	Notes

31:0	XPRESS10_ID	Hardcoded	HENC_XP10_ID default: 32'hC039E510
34:32	WINDOW_SIZE_SEL	Compression TLV	Compression TLV
35	MIN_MATCH_LEN_SEL	Compression TLV	Compression TLV (Characters per n-gram). 1'b0: 3 1'b1: 4
37:36	MODE	Compression TLV	Compression TLV (Prefix Mode)
43:38	PREDEF_SEL	Compression TLV	Compression TLV (Prefix Selector)
45:44	RSVD	Hardcoded	2'b0
46	CRC_OPTION	Compression TLV	Compression TLV
47	FLG_EXTRA	Hardcoded	HENC_FLG_EXTRA default: 0
63:48	FLG_EXTENSION	Hardcoded	HENC_FLG_EXTENSION default: 0

Table 23 : XP10 Frame Header

11.4.2 XP10 Block Header

Bit(s)	Field	Source	Notes
27:0	OUTPUT_SIZE	Calculated by Stream Assembler	Output size of the blocks in bits. Includes all headers but not the padding bits between blocks.
28	RSVD	Hardcoded	1'b0
29	BLK_TYPE	Stream Assembler	Uncompressed: 1'b0 Compressed: 1'b1
30	MTF_HDR_PRESENT	Stream Assembler	Inserted on encoded blocks that follow unencoded blocks within the same frame. Never asserted on the 1st block of a frame.
31	LAST_BLK	Stream Assembler	Last Block in Frame: 1'b1 Not Last Block in Frame: 1'b0

Table 24 : XP10 Block Header

The XP10 MTF Header will need to be inserted when MTF_HDR_PRESENT == 1'b1.

Fields are listed in order from least significant bit to most significant bit:

Field	1st block	subsequent blocks based on MTF snapshot

MTF_OFFSET_EXP0	5'b00000	FLOOR(LOG2(MTF_OFFSET_0))
MTF_OFFSET_LSB0	N/A	MTF_OFFSET_0 – 2^(MTF_OFFSET_EXP)
MTF_OFFSET_EXP1	5'b00000	FLOOR(LOG2(MTF_OFFSET_1))
MTF_OFFSET_LSB1	N/A	MTF_OFFSET_1 – 2^(MTF_OFFSET_EXP)
MTF_OFFSET_EXP2	5'b00000	FLOOR(LOG2(MTF_OFFSET_2))
MTF_OFFSET_LSB2	N/A	MTF_OFFSET_2 – 2^(MTF_OFFSET_EXP)
MTF_OFFSET_EXP3	5'b00000	FLOOR(LOG2(MTF_OFFSET_3))
MTF_OFFSET_LSB3	N/A	MTF_OFFSET_3 – 2^(MTF_OFFSET_EXP)

Table 25 : XP10 MTF Header

The LSB fields are variable length and will be represented in MTF_OFFSET_EXP0/1/2/3 bits.

The MTF snapshot is generated by the Symbol Mapper and will be updated at the end of every Huffman block (for the header of the next Huffman Block).

Following the MTF fields, the symbol tables are written out (If BLK_TYPE == Compressed). The long symbol table is sent first, followed by the short symbol table. The contents of the table are broken down as follows:

# Bits	Field	Notes
2	[LONG/SHORT]_ENCODE_TYPE	3'b00: Simple Encode 3'b01: Predefined 3'b10: Retrospective 3'b11: Reserved
stcl_size	Encoded Huffman Code Length Symbols	Include only if Encoder Select = 2'b10
st_size	Encoded Huffman Code Table	Include only if Encoder Select = 3'b10

Table 26 : XP10 Symbol Code Length and Code Table

11.4.3 XP10 Frame Footer

Field	Source	Notes
CRC64	EOF Token	Insert in Frame Footer TLV

Table 27 : XP10 Frame Footer

11.5 GZIP Framing

11.5.1 GZIP Frame Header

Bit(s)	Field	Source	Notes
7:0	ID1	Hardcoded	8'h1F (GZIP Identifier)
15:8	ID2	Hardcoded	8'h8B (GZIP Identifier)
23:16	CM	Hardcoded	8'h08 (DEFLATE)
24	FLG.FTEXT	Hardcoded	1'b0
25	FLG.FHCRC	Hardcoded	1'b0
26	FLG.FEXTRA	Hardcoded	1'b0
27	FLG.FNAME	Hardcoded	1'b0
28	FLG.FCOMMENT	Hardcoded	1'b0
31:29	FLG.reserved	Hardcoded	3'd0
63:32	MTIME	Hardcoded	32'd0
71:64	XFL	Hardcoded	8'h02
79:65	OS	Hardcoded	8'hFF
Compressed Blocks in DEFLATE format			

Table 28 : GZIP Frame Header

11.5.2 GZIP Frame Footer

Bit(s)	Field	Source	Notes
31:0	CRC32	Frame Footer	Word 8 [63:32]
63:32	ISIZE	Frame Footer	Word 8 [31:0]

Table 29 : GZIP Frame Footer

11.6 ZLIB Framing

11.6.1 ZLIB Frame Header

Bit(s)	Field	Source	Notes
3:0	CMF.CM	Hardcoded	4'h8 (DEFLATE)
7:4	CMF.CINFO	Hardcoded	4'h7 (32K window)
12:8	FLG.FCHECK	Hardcoded	5'h1A

13	FLG.FDICT	Hardcoded	1' b0
15:14	FLG.FLEVEL	Hardcoded	2'h3
Compressed Blocks in DEFLATE format			

Table 30 : ZLIB Frame Header

11.6.2 ZLIB Frame Footer

Bit(s)	Field	Source	Notes
31:0	ADLER32	Frame Footer	Word 8 [63:32]

Table 31 : ZLIB Frame Footer

11.7 Deflate Framing

Deflate block framing, common to GZIP and ZLIB.

11.7.1 Deflate Block Header

Bit(s)	Field	Source	Notes
0	BFINAL	Stream Assembler	Set if this is the final Huffman block of the frame.
2:1	BTYPE	Stream Assembler	2'b00: No compression (Encoded data + header expanded the block) 2'b01: Reserved 2'b10: Compressed with dynamic Huffman codes 2'b11: Reserved
<16>	LEN	Stream Assembler	Number of bytes (Max 65535). Only if BTYPE = 2'b00
<16>	NLEN	Stream Assembler	1's complement of LEN. Only if BTYPE = 2'b00
<LEN*8>	data	Reconstructor	Raw Data . Only if BTYPE = 2'b00
<5>	HLIT	Header Builder	# of Literal/Length codes - 257
<5>	HDIST	Header Builder	# of Distance codes - 1
<4>	HCLEN	Header Builder	# of Code Length codes - 4
<HCLEN*3>	Code Length Codes, See DEFLATE spec for order		
<Variable>	Literal/Length Codes		
<Variable>	Distance Codes		
<Variable>	Compressed Data		
<Variable>	Literal/Length symbol 256 (End of Data)		

Table 32 : Deflate Block Header

11.7.2 Deflate Block Footer

If encoding in DEFLATE mode and BTYPE != 2'b00, then the distance/length symbol 256 (end of block symbol) is inserted at the end of the data stream.

12 Debug and Configuration

12.1 Overview

Programmable registers, software based table reads, statistics counters and interrupt registers along with iner stage monitors(ISM) provide debug facility to the Huffman encoder.

12.2 SW accessible registers

Parameter	Bits	Default	Comment
HENC_FORCE_REBUILD_EN	1	0	By selecting this mode, it forces the tree builder to undergo the selected programmed number of rebuilds, regardless of if they were needed or not. Used to test the rebuild and frequency division hardware. Note, the Force Rebuild feature is applicable to XP10 and Deflate.
HENC_FORCE_REBUILD_ SHORT	5	0	Number of rebuilds forced for the short/literal(In case of deflate) Huffman tree builder
HENC_FORCE_REBUILD_ LONG	5	0	Number of rebuilds forced for the long/distance(In case of deflate) Huffman tree builder
HENC_FORCE_REBUILD_ ST_SHORT	4	0	Number of rebuilds forced for the short symbol table's tree builder. This is also used for deflate's symbol table tree builder.
HENC_FORCE_REBUILD_ ST_LONG	4	0	Number of rebuilds forced for the long symbol table's tree builder. This is not used for deflate's symbol table tree builder.
HENC_XP_SHORT_MAX_CODE_LEN	5	27	Max code length for XP10 short symbols. If exceeded, a rebuild is started.
HENC_DEFLATE_SHORT_MAX_CODE_LEN	5	15	Max code length for Deflate short symbols. If exceeded, a rebuild is started.
HENC_XP_LONG_MAX_CODE_LEN	5	27	Max code length for XP10 long symbols. If exceeded, a rebuild is started.
HENC_DEFLATE_LONG_MAX_CODE_LEN	5	15	Max code length for Deflate long symbols. If exceeded, a rebuild is started.
HENC_XP_ST_MAX_CODE_LEN	4	8	Max code length for XP10 short symbol table's tree builder. If exceeded, a rebuild is started.
HENC_DEFLATE_ST_MAX_CODE_LEN	4	8	Max code length for Deflate short symbol table's tree builder. If exceeded, a rebuild is started.
HENC_XP_REBUILD_THRESHOLD_SHORT	8	255	Maximum number of rebuilds on XP10 SHORT/LITERALS pipe when the tree build is aborted with a retrospective encoding invalid being asserted.
HENC_DEFLATE_REBUILD_THRESHOLD_SHORT	8	255	Maximum number of rebuilds on Deflate SHORT/LITERALS pipe when the tree build is aborted with a retrospective encoding invalid being asserted.
HENC_XP_REBUILD_THRESHOLD_LONG	8	255	Maximum number of rebuilds on XP10 LONG/DISTANCE pipe when the tree build is aborted with a retrospective encoding error being asserted.
HENC_DEFLATE_REBUILD_THRESHOLD_LONG	8	255	Maximum number of rebuilds on Deflate LONG/DISTANCE pipe when the tree build is

			aborted with a retrospective encoding error being asserted.
HENC_XP_REBUILD_THRESHOLD_SHORT_ST	8	255	Maximum number of rebuilds on XP10 SHORT symbol table pipe when the tree build is aborted with a retrospective encoding error being asserted.
HENC_DEFLATE_REBUILD_THRESHOLD_SHORT_ST	8	255	Maximum number of rebuilds on DEFLATE SHORT symbol table pipe when the tree build is aborted with a retrospective encoding error being asserted.
HENC_XP_REBUILD_THRESHOLD_LONG_ST	8	255	Maximum number of rebuilds on XP10 LONG symbol table pipe when the tree build is aborted with a retrospective encoding error being asserted.
HENC_HUFF_WIN_SIZE_IN_ENTRIES[13:0]	14	8192	Huffman window size in Symbol queue entries.
HENC_XP10_FLG_EXTRA	1	0	Enables 16 extra flag bits to be included in the XP10 frame header
HENC_XP10_FLG	16	0	Extra bits included in the XP10 frame header if HENC_XP10_FLG_EXTRA is set.
HENC_XP10_DISABLE_MODES	8	0	Disables particular encoding modes for XP10. Multiple modes may be disabled. If there is no valid mode, the encoder will default to retrospective and set an error. Bit mapping: 0: Disable simple encode mode 1: Disable retrospective mode 2: Disable raw mode 3: Disable Predetermined Huffman Mode Others: Reserved
HENC_DEFLATE_DISABLE_MODES[7:0]	8	0	Disables particular encoding modes for GZIP/ZLIB. Multiple modes may be disabled. If there is no valid mode, the encoder will default to retrospective and set an error. Bit mapping: 0: Reserved 1: Disable retrospective mode 2: Disable raw mode Others: Reserved
HENC_FORCE_BLOCK_STALL	8	0	Force a stall between internal blocks by forcing ready to a 0. 0: Symbol Mapper to LZ77 1: Symbol Collapser to Symbol Mapper 2: Insertion Sort to Symbol Collapser 3: Tree Builder to Insertion Sort 4: Tree Walker to Tree Bulder 5: Symbol Table to Tree Walker 6: Stream Assembler to Symbol Table 7: Symbol Queue to Symbol Mapper
HENC_DISABLE_SUB_PIPE	1	0	Disable the second sub-pipe for the Tree Builder, Tree Walker and Symbol Table blocks.

Table 33 : Configuration Registers

12.3 SW accessible table reads

All the dynamic tables (Hardware updatable) including the LUTs are readable by the software.

12.4 Local Statistics Counters and Registers (clear on read, Non roll over)

SHORT_rebuild_threshold	Number of times the SHORT/LITERAL Huffman code rebuild errored out by hitting the max threshold.
LONG_rebuild_threshold	Number of times the LONG/DISTANCE Huffman code rebuild errored out by hitting the max threshold.
SHORT_symbol_table_rebuild_threshold	Number of times the SHORT/DEFLATE symbol table Huffman code rebuild errored out by hitting the max threshold.
LONG_ symbol_table_rebuild_threshold	Number of times the LONG symbol table Huffman code rebuild errored out by hitting the max threshold.
Number_SHORT_rebuilds	Number of times the SHORT/LITERAL Huffman code rebuilds occurred.
Number_LONG_rebuilds	Number of times the LONG/DISTANCE Huffman code rebuilds occurred.
Number_SHORT_symbol_table_rebuilds	Number of times the SHORT/DEFLATE symbol table Huffman code rebuilds occurred.
Number_LONG_ symbol_table_rebuilds	Number of times the LONG symbol table Huffman code rebuilds occurred.
Deflate_short_max_symbol_width	Max symbol width
Deflate_long_max_symbol_width	Max symbol width
XP10_short_max_symbol_width	Max symbol width
XP10_long_max_symbol_width	Max symbol width
Symbol_Queue_high_watermark	Max Symbol Queue Depth
Number_Short_Symbol_Out_of_Range	Number of short symbol out of range errors
Number_Long_Symbol_Out_of_Range	Number of long symbol out of range errors
Block Ready Status Live	32-bit register with live status of the ready signals between blocks
Block Ready Status Sticky	32-bit clear-on-read status of the ready signals between blocks

Table 34 : Local Stats Counters and Registers

12.5 Global Statistics

Strobes to be sent to the Global Statistics Accumulator will be generated for the events listed in Table 35

Global Counters	Bit
Short Symbol Out of Range	0
Long Symbol Out of Range	1
XP10 Encoded Block	9
XP10 Raw Block	10
XP10 Short Simple Block	11
XP10 Long Simple Block	12
XP10 Short Retrospective Block	13
XP10 Long Retrospective Block	14
XP10 Short Predetermined Block	15
XP10 Long Predetermined Block	16
XP10 Frame	17
CFH8K Encoded Block	18
CFH8K Raw Block	19
CFH8K Short Simple Block	20
CFH8K Long Simple Block	21
CFH8K Short Retrospective Block	22
CFH8K Long Retrospective Block	23
CFH8K Short Predetermined Block	24
CFH8K Long Predetermined Block	25
CFH8K Frame	26
CFH4K Encoded Block	27
CFH4K Raw Block	28

CFH4K Short Simple Block	29
CFH4K Long Simple Block	30
CFH4K Short Retrospective Block	31
CFH4K Long Retrospective Block	32
CFH4K Short Predetermined Block	33
CFH4K Long Predetermined Block	34
CFH4K Frame	35
Deflate Encoded Block	36
Deflate Raw Block	37
Deflate Short Simple Block	38
Deflate Long Simple Block	39
Deflate Short Retrospective Block	40
Deflate Long Retrospective Block	41
Deflate Frame	42
No-compress Frame	43
Byte Lane 0	44
Byte Lane 1	45
Byte Lane 2	46
Byte Lane 3	47
Byte Lane 4	48
Byte Lane 5	49
Byte Lane 6	50
Byte Lane 7	51
Encrypt Downstream Stall	52

LZ77 Upstream Idle	53

Table 35 : Global Stats Counters

12.6 Interrupts

1. Uncorrectable ECC Interrupt, software will determine how to recover from an uncorrectable ECC interrupt.
2. Memory Controller Interrupt, for debug and testing purposes.
3. TLV Error, occurs when a malformed TLV is detected. Unlikely to occur in normal operation but may occur during software development.

12.7 Debug Inter Stage Monitors (ISM)

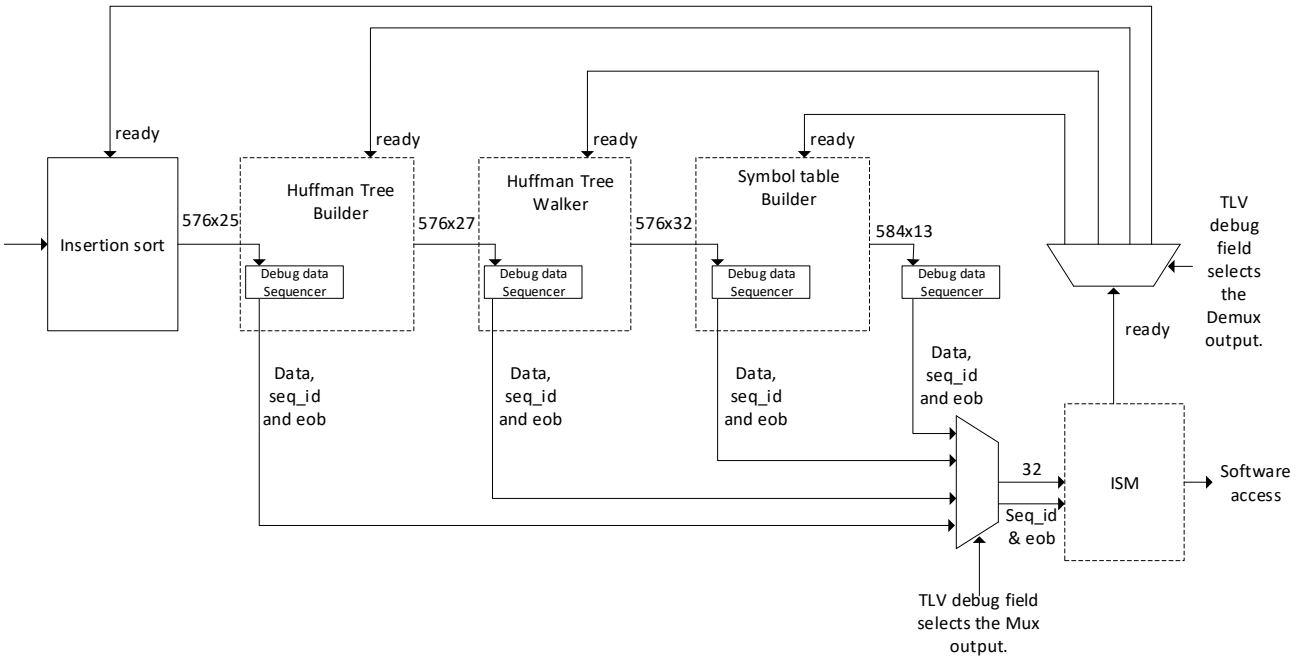


Figure 25 : Bus monitors inside Huffman encoder

The outputs of Insertion sort, Tree builder, Tree walker and Symbol table builder can be monitored for debug by muxing them to the software accessible registers via a block called Inter stage monitor(ISM).

When the output of a selected hardware block is ready with its output, the output data is transferred to the ISM if the ISM is ready. The hardware block being debugged cannot start processing a new Huffman block until the ISM ready is high. So in effect ISM ready can stall the upstream pipeline.

The muxing of data and ready is done by TLV fields for debug. (TBD)

Data write to the ISM is completed with *eob* asserted. Sequence id which is a unique incremental number given to every Huffman block inside the encoder is also available to the ISM to monitor.

Since the buses between the encoder blocks are two dimensional arrays, a block called as data sequencer sits between each block that converts the array into one entry per transfer to the ISM.

13 Resource Estimates

The flop and memory estimates for each section of the Huffman Encoder are included in Table 36.

Block	Inst	Per Instance		Total		Notes
		Flops	Mem (kB)	Flops	Mem (kB)	
Symbol Mapper	1	300	0	300	0	Register inputs, pipeline stages for mapping function
Symbol Collapser Short	1	150	1.8	150	1.8	Includes 256x54 FIFO
Symbol Collapser Long	1	50	0.4	50	0.3	Includes 256x13 FIFO
Symbol Queue	1	300	180	300	180	
Sequence ID Control Buffer	8	175	0	1400	0	Compression header TLV, raw byte count, extra bit count, MTF header in flops for access by all pipeline blocks
Insertion Sort Short	1	14458	0	14458	0	Pipeline and short list
Insertion Sort Long	1	5226	0	5226	0	Pipeline and long list
Tree Builder Short	2	31725	2.2	63450	4.4	
Tree Builder Long	2	12069	0.8	24138	1.6	
Tree Walker Short	2	34614	2.8	69228	5.6	
Tree Walker Long	2	15734	1	47202	2	
Symbol Table Short	2	11426	0	22852	0	
Symbol Table Long	2	5602	0	11204	0	
LUT Short	2	270	19.2	540	38.4	
LUT Long	2	256	8.6	512	17.2	
Predetermined Tables	8	12	0.525	96	4.2	Few flops for mapping

Stream Assembler	1	750	0	750	0	Header builder, pipe line staging between packing functions, state machine
Reconstructor	1	4532	1064	4532	64	
Config and local Debug	1	1324	0	1324	0	
Sub total				267712	319.5	

Table 36 : Resource Estimates