A FAST PARALLEL HUFFMAN DECODER FOR FPGA IMPLEMENTATION

Laurentiu ACASANDREI Marius NEAG
Silicon Systems Transylvania SRL, Tel: +40258775181, acasandreilaurentiu@yahoo.com
Technical University of Cluj-Napoca, Tel: +40264401470, Marius.Neag@bel.utcluj.ro

Abstract: The paper presents a novel algorithm and architecture for implementing a Huffman decoder. It starts with an overview of the basics, from the entropy coding and the way the Huffman coding is obtained, to the way a Huffman coder handles data and image components within the Jpeg standard. Then it briefly discusses the decoding procedures proposed by the ISO/IEC 10918-1(1993E) standard; due to their sequential nature, a decoder that simply implements these procedures requires several execution cycles to output one set of decoded data. A new decoding algorithm is then introduced, based on a parallel architecture that allows it to output a set of decoded data per each clock cycle. This approach was validated through actual implementation on an of-the-shelf FPGA; this not only demonstrates the proposed algorithm and architecture but also proves that it can operate at very high frequencies, up to 100MHz. A limitation of this implementation is the relatively large amount of hardware resources it requires.

Key words: *Jpeg Standard, Fast Huffman Decoder, Image Acquisition and Processing, Parallel Architecture, FPGA*

I. INTRODUCTION

The Huffman coding is an entropy algorithm used for lossless data compression. This algorithm is used in JPEG compression standard at the final encoder processing step as shown in *Figure 1*. This step achieves additional compression losslessly by encoding the quantized discrete cosine transform (DCT) coefficients more compactly based on their statistical characteristics. The JPEG standard specifies two entropy coding methods: Huffman coding and arithmetic coding.

The baseline sequential codec uses Huffman coding. It is useful to consider the entropy coder as a 2-step process: the first step converts the zig-zag sequence of quantized coefficients into an intermediate sequence of symbols; the second step converts the symbols to a data stream in which the symbols no longer have identifiable boundaries. The form and definition of the intermediate symbols depend on both the Jpeg mode of operation and the entropy coding method.

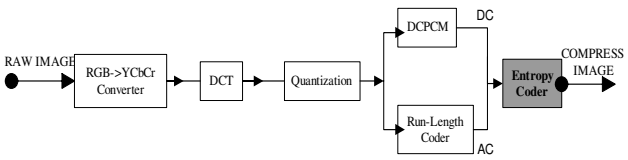

Figure 1.Sequential Jpeg Coder

The Jpeg standard requires that one or more sets of Huffman code tables are specified by the application. The code tables used to compress an image are needed to decompress it. Huffman tables may be predefined and used in an application as defaults, or they can be computed specifically for a given image in an initial statistic gathering pass prior to compression. Such choices are application-specific as the Jpeg standard does not specifically require Huffman tables [1]. But it does require that for every symbol, the minimum size in bits that can keep the symbol value is stored in the data stream, as detailed in *Table 1*.

Table 1. Category(Cat) and symbol representation*

Values	Cat*	Bits for the value
0	0	-
-1,1	1	0,1
-3,-2,2,3	2	00,01,10,11
-7,-6,-5,-4,4,5,6,7	3	000,001,010,011,100, 101, 110, 111
-15,..,-8,8,..,15	4	0000,..,0111, 1000,..,1111
-31,..,-16,16,..,31	5	00000,..,01111, 10000,..,11111
-63,..,-32,32,..,63	6	…
-127,..,-64,64,..,127	7	…
-255,..,-128,128,..,255	8	…
-511,..,-256,256,..,511	9	…
-1023,..,-512,512,.., 1023	10	…
-2047,..,-1024,1024,..,2047	11	…
-4095,..,-2048,2048,..,4095	12	…
-8191,,-4096,4096,..,8191	13	…
-16383,..,-8192,8192,..,16383	14	…
-32767,..,-16384,16384,..,32767	15	…

The most popular procedure for Jpeg coding is the baseline encoding for 8-bit sample precision. The encoder

Manuscript received November 15, 2007; revised April 17, 2008

may employ up to two DC and two AC Huffman tables: a DCT coefficient for which the frequency is zero in both dimensions is called DC, while for an AC coefficient the frequency is not zero in at least one dimension. Huffman tables are specified in terms of a 16-byte list giving the number of codes for each code length from 1 to 16. This is followed by a list of 8-bit symbol values, each of which is assigned a Huffman code. The symbol values are placed in the list in order of increasing code length. Code lengths greater than 16 bits are not allowed. In addition, the codes must be generated such that the all-1 code words are reserved as a prefix for longer code words.

The root of a Huffman code is placed toward the MSB (most-significant-bit) of the byte, and successive bits are placed in the direction MSB to LSB (least-significant-bit) of the byte. Remaining bits, if any, go into the next byte following the same rules.

Integers associated with Huffman codes are appended with the MSB adjacent to the LSB of the preceding Huffman code.

The DC code table consists of a set of Huffman codes (maximum length 16 bits) and appended additional bits (in most cases) which can code any possible value of the difference between the current DC coefficient and the prediction. The Huffman codes for the difference categories are generated in such a way that no code consists entirely of 1-bit (X'FF' prefix marker code avoided).

The two's complement difference magnitudes are grouped into the first 12 categories detailed in *Table 1* and a Huffman code is created for each of the 12 difference magnitude categories. For each category, except 0, an additional bit field is appended to the code word to uniquely identify which difference in that category actually occurred.

The encoding procedure for DC coefficients is based on a set of extended tables, which contain the complete set of Huffman codes and sizes for all possible difference values.

The extended Huffman code table and the table of sizes are generated from the encoder by appending to the Huffman codes for each difference category the additional bits that completely define the difference. By definition, the extended Huffman code table and table sizes have entries for each possible difference value. The extended Huffman code table contains the concatenated bit pattern of the Huffman code and the additional bits field; the extended Huffman length contains the total length in bits of this concatenated bit pattern. Both are indexed by the difference between the DC coefficient and the prediction.

Each non-zero AC coefficient is described by a composite 8-bit value with the form: the 4 least significant bits define a category for the amplitude of the next non-zero coefficient in zig-zag, and the 4 most significant bits give the position of the coefficient in zig-zag relative to the previous non-zero coefficient (i.e. the run-length of zero coefficients between non-zero coefficients). Since the run length of zero coefficients may exceed 15, the value X'F0' is defined to represent a run length of 15 - this can be interpreted as a run length of 16 zero coefficients.

In addition, a special value '00000000' is used to code the end-of-block (EOB), when all remaining coefficients in the block are zero. The composite value is Huffman coded and each Huffman code is followed by additional bits which specify the sign and exact amplitude of the coefficient.

The AC code table consists of one Huffman code (maximum length 16 bits, not including additional bits) for each possible composite value. The Huffman codes for the 8-bit composite values are generated in such a way that no code consists entirely of 1-bits. The format for the additional bits is the same as in the coding of the DC coefficients.
If the last coefficient ($K = 63$) is not zero, the EOB code is bypassed [2].

II. IMAGE DATA ENCODING AND MULTIPLE IMAGE COMPONENT CONTROL

For DCT based encoders the data unit is formed out of 8x8 blocks of samples. The minimum code unit (MCU) is the smallest group of data units that is coded.
The scan header of a Jpeg file specifies the order by which source image data units shall be encoded and placed within the compressed image data. For a given scan, if the scan header parameter *number of components in scan* equals 1, then data from only one source component shall be present within the scan. This data is non-interleaved by definition.
If *number of components in scan* is greater than 1, then data from the components shall be present within the scan. These data will always be interleaved. The order of components in a scan shall be according to the order specified in the frame header. The ordering of data units and the construction of minimum coded units (MCU) is defined as follows:

– For non-interleaved data the MCU is one data unit.

– For interleaved data the MCU is the sequence of data units defined by the sampling factors of the components in the scan.

When *number of components in scan* equals 1 the order of data units within a scan shall be left-to-right and top-to-bottom, as shown in *Figure 2.*

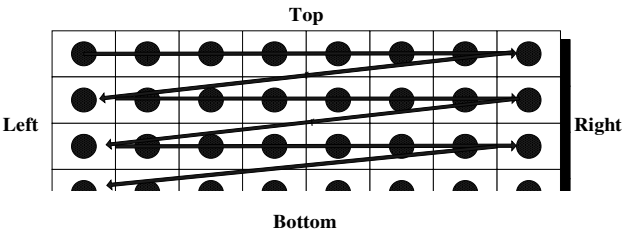

Figure 2. Non-interleaved data ordering

When *number of components in scan* is larger than 1, each scan component is partitioned into small rectangular arrays of horizontal data units by vertical data units. Within each rectangular array the data units are ordered from left-to-right and top-to-bottom; in turn, the arrays are ordered from left-to-right and top-to-bottom within each component.

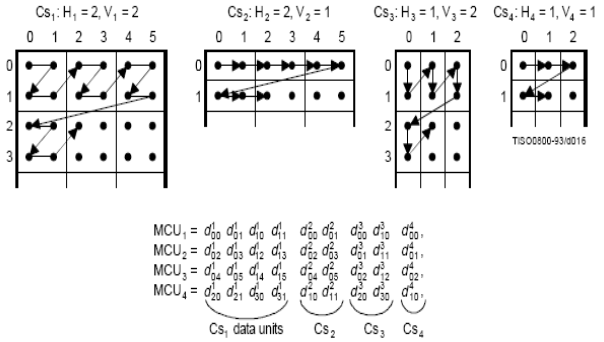

Figure 3. Example of interleaved data ordering

An example is presented in *Figure 3*, for which the *number of components in scan* = 4, and MCU1 consists of data units taken first from the top-left-most region of Cs1, followed by data units from the corresponding region of Cs2, then from Cs3 and then from Cs4. MCU2 follows the same ordering for data taken from the next region to the right of the four components.

Another major part of the coding process is the procedure which controls the order in which the image data from multiple components are processed in order to create the compressed data, which ensures that the proper set of table data is applied to the corresponding data units in the image. *Figure 4* shows an example of how the encoding process selects between multiple source image components as well as between multiple sets of table data. The source image in this example consists of three components - A, B and C - and there are two sets of table specifications.

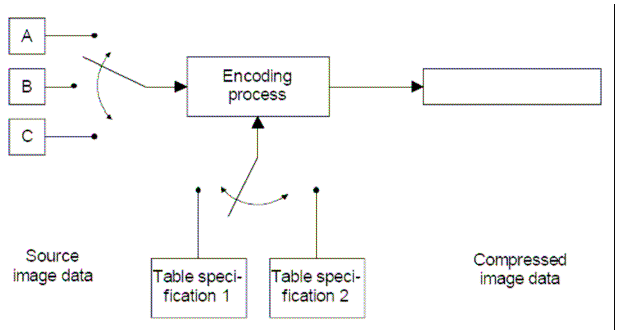

Figure 4. Component interleave and table-switching control

In sequential mode, encoding is *non-interleaved* if the encoder compresses all image data units in component A before beginning with component B, then compressing all of B before start compressing C. Encoding is *interleaved* if the encoder compresses a data unit from A, a data unit from B, a data unit from C, then back to A, etc. These alternatives are illustrated in *Figure 5*, which shows a case in which all three image components have identical dimensions: X columns by Y lines, for a total of n data units each.

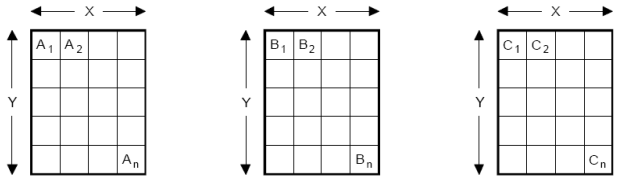

Figure 5. Three image components with same dimensions

In the non-interleaved mode the scans are organized as follows :

Scan 1=$A_1,A_2,..,A_n$; *Scan 2*=$B_1,B_2,..,B_n$;*Scan 3*=$C_1,C_2,..,C_n$

In the interleaved mode the scans are organized as follows:

Scan 1=$A_1, B_1, C_1, A_2, B_2, C_2,\ldots, A_n, B_n, C_n$

These control procedures are also able to handle cases in which the source image components have different dimensions. *Figure 6* shows a case in which two of the components, B and C, have half the number of horizontal samples compared to component A. In this case, two data units from A are interleaved with one each from B and C.

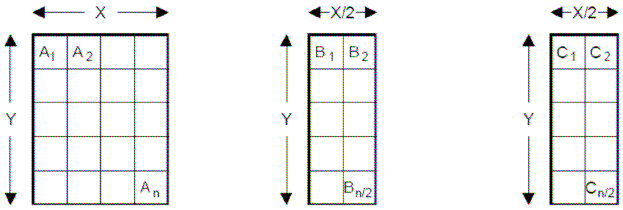

Figure 6. Three image components with different dimensions

For the interleaved order the scans is organized as follows:

Scan 1=$A_1,A_2,B_1,C_1,A_3,A_4,B_2,C_2,\ldots,A_{n-1},A_n,B_{n/2},C_{n/2}$

III. ISO/IEC 10918-1 BASELINE HUFFMAN DECODER PROCEDURES

The baseline decoding procedure is designed for 8-bit sample precision. The decoder must be capable of using up to two DC and two AC Huffman tables within one scan. The decoding procedure for the DC difference, DIFF, is [2]:

T = *DECODE*

DIFF = *Decode_ZZ(K)*

The *DECODE* procedure decodes an 8-bit value which, for the DC coefficient, determines the difference magnitude category. For the AC coefficient this 8-bit value determines the zero run length and non-zero coefficient category. This particular implementation of *DECODE* makes use of the ordering of the Huffman codes in a list according to both value and code size. Many other implementations of DECODE are possible.

The implementation of *DECODE* described in *Figure 7* uses three tables, MINCODE, MAXCODE and VALPTR, to decode a pointer to the list of values assigned to each Huffman code (HUFFVAL).

MINCODE, MAXCODE and VALPTR each have 16 entries, one for each possible code size. MINCODE(I) contains the smallest code value for a given length I, MAXCODE(I) contains the largest code value for a given

length I, and VALPTR(I) contains the index to the start of the list of values assigned to each Huffman code which are decoded by code words of length I. The values in MINCODE and MAXCODE are signed 16-bit integers; therefore, a value of –1 sets all of the bits.

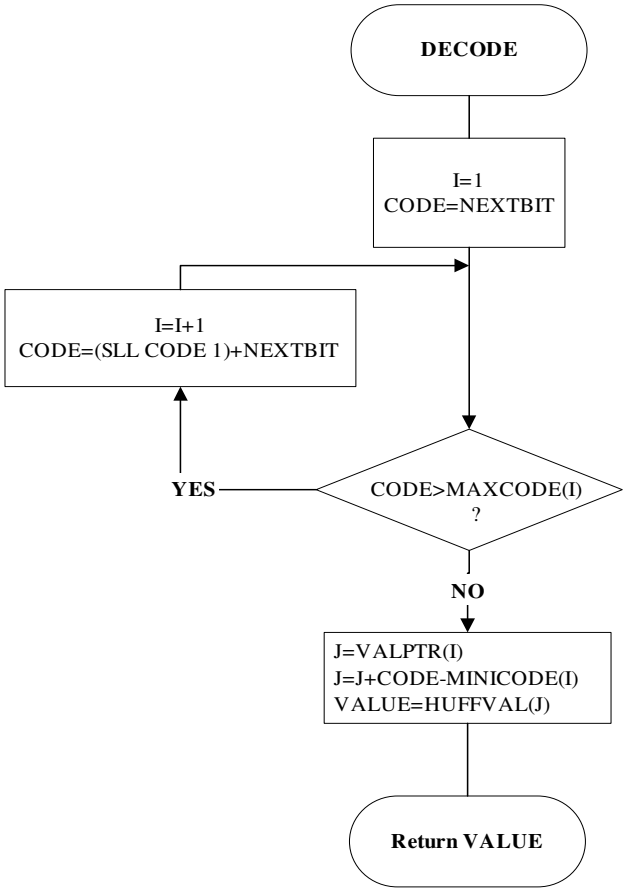

Figure 7. Procedure for DECODE

The *NEXTBIT* procedure reads the next bit of compressed data and passes it to higher level routines. It also intercepts and removes stuff bytes and detects markers. *NEXTBIT* reads the bits of a byte starting with the MSB (see *Figure 8*).Before starting the decoding of a scan, and after processing a RST marker, CNT is cleared. The compressed data are read one byte at a time, using the procedure *NEXTBYTE*. Each time a byte, B, is read, CNT is set to 8. The only valid marker which may occur within the Huffman coded data is the RST marker. Other than the EOI or markers which may occur at or before the start of a scan, the only marker which can occur at the end of the scan is the DNL (define-number-of-lines).

Normally, the decoder will terminate the decoding at the end of the final restart interval before the terminating marker is intercepted. If the DNL marker is encountered, the current line count is set to the value specified by that marker. Since the DNL marker can only be used at the end of the first scan, the scan decode procedure must be terminated when it is encountered [2].

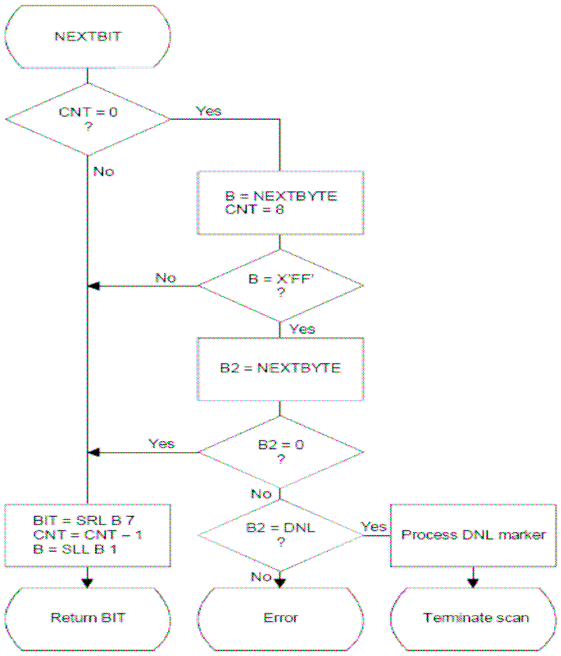

Figure 8. Procedure for fetching the next bit of compressed data

The decoding of the amplitude and sign of the non-zero coefficient is done in the procedure “Decode_ZZ(K)”, shown in *Figure 9*.

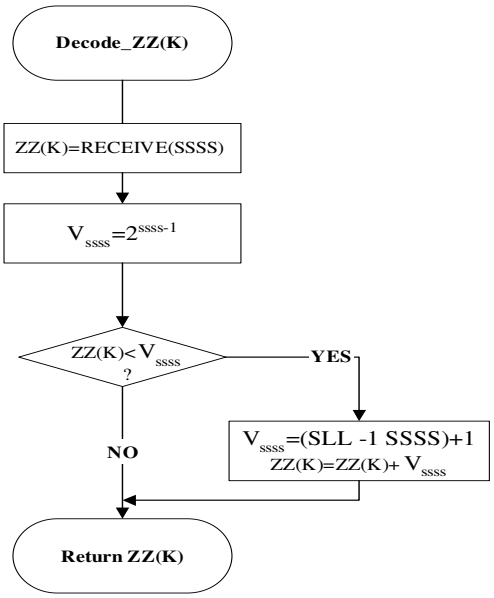

Figure 9. Decoding a non-zero coefficient.

The *RECEIVE* procedure is a procedure which places the next category bits of the entropy-coded segment into the low order bits of DIFF, MSB first. It calls *NEXTBIT* and it returns the value of DIFF to the calling procedure, as shown in *Figure 10*.

Figure 11 illustrates the decoding procedure for AC coefficients.

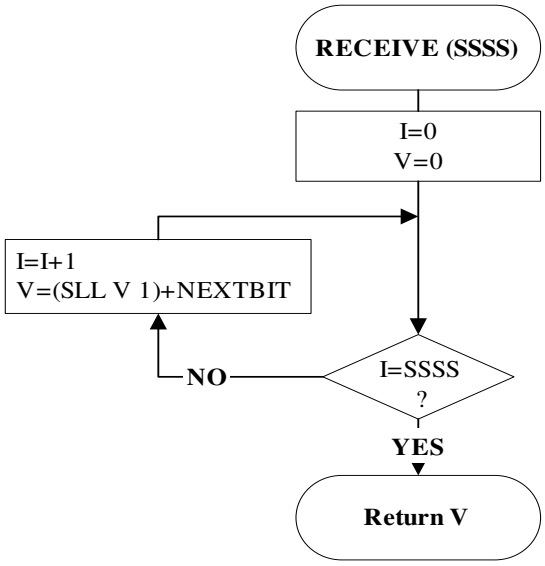

Figure 10. Procedure for RECEIVE(SSSS)

The decoding is done in a loop process, by extracting the number of zero coefficients from the last non zero value and the dimension of the actual AC coefficient. The resulting coefficients are stored in a 63-bit long vector.

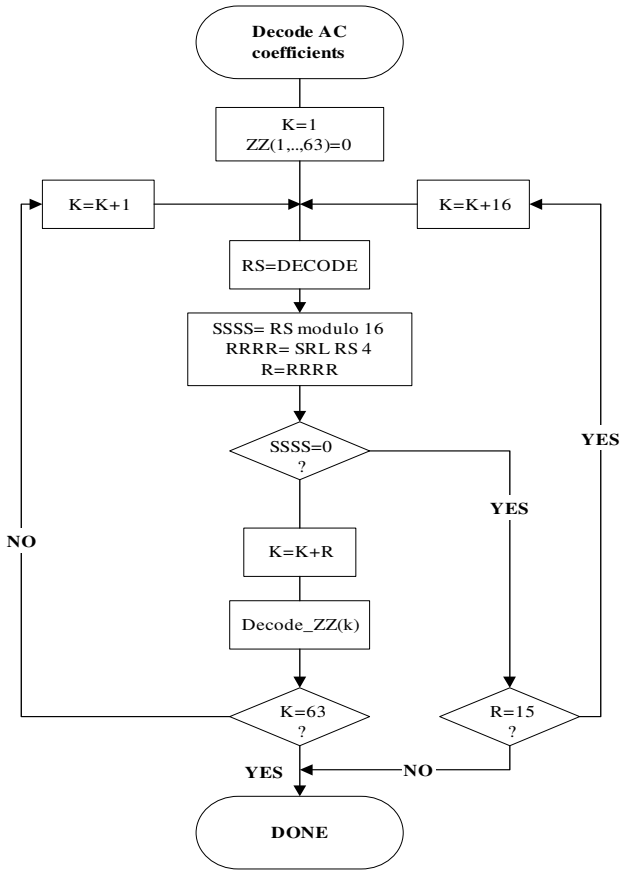

Figure 11. Huffman decoding for AC coefficients

IV. HUFFMAN DECODING PROPOSED PARALLEL ALGORITHM AND ARHITECTURE

The procedures for Huffman decoding proposed by the ISO/IEC 10918-1 standard have been widely used in the software applications. From the point of view of the processor time decoding an MCU takes a considerable amount of execution cycles in order to compute one decoded data value, due to the numerous mathematical operations required. This is a general drawback of sequential decoders, which basically implement a state machine that traverses the code-tree until a symbol is found. Several techniques have been proposed for reducing the execution time, for example by using lookup tables stored in a memory [3]. However, such a decoder still needs several clock cycles for decoding each element, limiting its use in high-speed applications.

In order to overcome this drawback we propose the parallel Huffman decoding algorithm presented in *Figure 12*

The main idea of the proposed algorithm is briefly described here. Two extended Huffman table are used: one is for the code words and second is the category or length corresponding to the code words. The extended tables are supposed to be previously determined from the Huffman tables. A part of the scans is compared at the same time with all the Huffman code words. If we have a match (and due to the nature of Huffman codes we can have only one match) then the useful data is extracted from scans. After the data are extracted, they are processed for sign transformation.

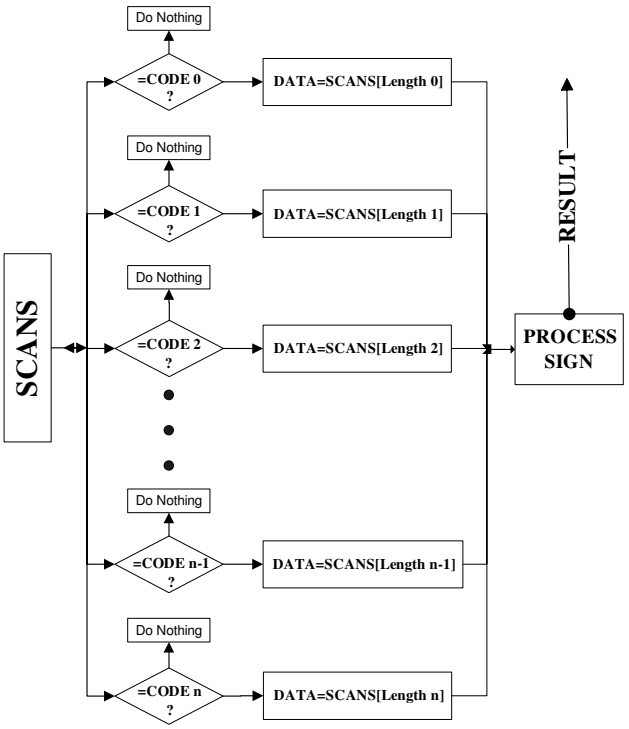

Figure 12. Parallel algorithm for n Huffman code words

The proposed algorithm can be used as a Huffman decoder component in a Jpeg file decoder. The block diagram of the hardware implementation of such a decoder

is shown in *Figure 13*. It was coded in Verilog, resulting in a decoder fully compatible with the specifications for Huffman decoder as per standard ISO/IEC 10918-1

This implementation uses 2 DC extended code word tables, 2 DC length tables, 2 AC extended code word tables, and 2 AC run length tables. The tables are previously determined from the Huffman application tables. The tables are capable of storing 256 records of information each (256 is the maximum number of code words allowed by the standard for Huffman coding/decoding).

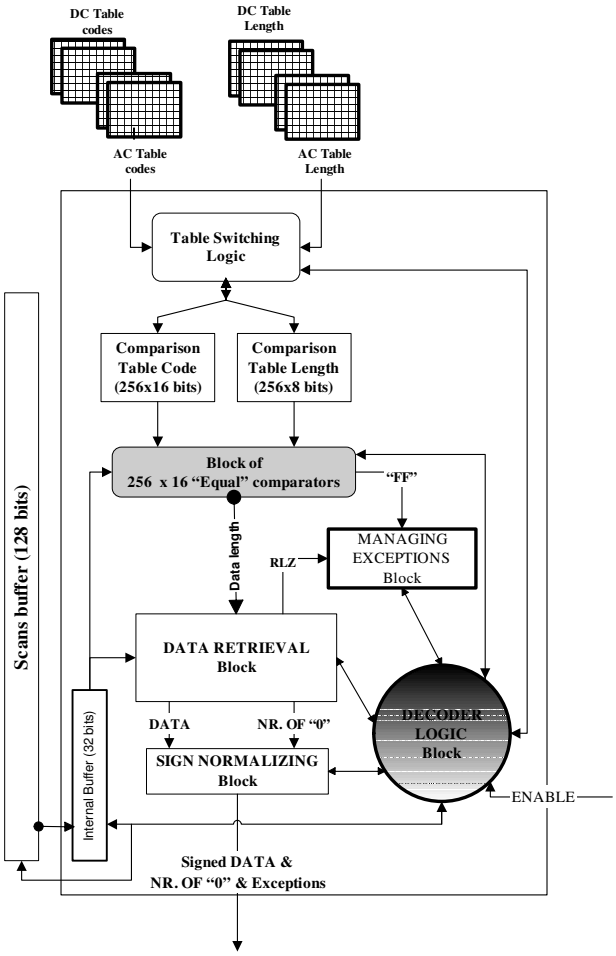

Figure 13. Functional block diagram of the parallel Huffman decoder

The *Internal buffer* (32 bits wide) is used to store de 32 MSB of the *Scans buffer*. The value stored in the internal buffer is used in the combinational *block of 256 comparators* were it is compared with each of the 256 Huffman code words, in parallel. If the MSB 16 bits of *Internal buffer* corresponds to the code word j ($1 \leq j \leq 256$) then the length corresponding to the code word j is transmitted to the *DATA RETRIEVAL* block. The comparators block is also responsible for detecting segment markers.

The *DATA RETRIEVAL* block is a sequential block composed of 16 left shift units. The value *Data length* activates the corresponding left shift unit (i.e for length k, where $0 \leq k \leq 15$, it activates the left shift unit on k bits), which determines the *Internal buffer* to be shifted and the data to be retrieved. The data is passed to the *SIGN NORMALIZATION* block were the MSB bit is verified, and if it is 0 then number is negative so the corresponding adjustment is done. The signed processed data and the *Nr. Of "0"* signal are outputted together with the exceptions flags.

The *Nr. Of "0"* is a 4 bit wide signal which is always 0 for the DC data element and it contains the number of zero bytes form the last non-zero element for the AC data.

The *Table Switching* block is responsible for loading the corresponding DC or AC table to the *comparison table code* and the *comparison table length*, under direct supervision of the *DECODER LOGIC* block.

The *DECODER LOGIC* block is a finite state machine (FSM) based control unit which manages the entire process of decoding. It receives information from the other blocks and controls their behaviour. It is responsible for correct multiple-component decoding, exceptions and data handling, *Scans* and *Internal buffer* data loading and shifting

V. Experimental Results – Comparison with the standard sequential Huffman decoder

FPGA implementation

The proposed Huffman decoding architecture was synthesized using the Xilinx IseWEbpack 8.2i for the target device xc3s1000-4ft256. Extensive simulations and lab tests proved that this FPGA implementation was able to generate complete decoded data on a single clock cycle.

The FPGA synthesis report shown that the implementation used the following resources: 80% of the total amount of slices (13824 from the total of 17288) and 8% of the total RAM (34.56k from 432k) provided by the Xilinx xc3s1000-4ft256. More importantly the same report indicated a reasonably large value for the max clock frequency, just under 100MHz (97.63MHz).

The decoder has been tested using randomly-generated and specifically-aimed coded sequences. An example of the later is presented in this Section; the decoding of the test sequence by using the standard sequential Huffman software is also described, in order to allow for a fair comparison between the two solutions.

Test Setup The following coded sequence was sent to both decoders: (101110011100 1111100010101011000). Let us assume that the standard sequential decoder runs on the Microblaze 32 bit processor [4] – a very popular choice. For each operation (shift left, shift right), addition or substitution one clock cycle is used. For multiplication, division operations, Boolean conditions (IF-THEN-ELSE) or loop conditions two clock cycles are used. Also, the parallel hardware Huffman is supposed to be attached as an IP on the OPB (On-Chip Peripheral Bus) [5], thus working on the processor clock frequency. The coded sequence is stored in the internal Scans buffer (as shown in *Figure 13*).

Pseudo code for standard sequential software decoder

In order to analyze the results from the standard sequential software decoder we have transformed the algorithms described in *Figures* 7 to 11 into pseudo-code, then indicated the number of clock cycles used by each operation. The most called for function is NEXTBIT (see *Figure 8*). The corresponding pseudo-code is as follows:

```
FUNTION NEXTBIT
BEGIN
IF CNT=0 THEN                                   ;2clocks
        B= the next 8 bit from sequence         ;1clock
        IF B=xFF THEN                           ;2clocks
        B2= the next 8 bit from sequence        ;1clocks
        ENDIF;
                IF B2<>0 THEN                   ;2clocks
                IF B2=DNL marker THEN;2clocks
                 Process DNL marker
                ELSE
                 Signalize Error
                 ENDIF
        ENDIF
ENDIF
BIT=Shift Right with 7 position of B            ;1clock
CNT=CNT-1                                       ;1clock
B=Shift Left with 1 position of B               ;1clock
RETURN BIT
END NEXBIT
```

It results that the minimum number of cycles required by this routine is five – this is the case when no DNL marker is present and there are no errors.

The NEXBIT function calls in the DECODE function – see *Figure 7*.The corresponding pseudo-code and the number of clock cycles required by each operation are:

```
FUNCTION DECODE
BEGIN
I=1                             ;1clock
CNT=8                           ;1clock
CODE= call NEXTBIT              ;5 clocks
WHILE CODE>MAXCODE(I)           ;2 clocks
I=I+1                           ;1 clock
CODE=Shift Left with 1 position of CODE ;1 clock
CODE=CODE+ call  NEXTBIT        ;6 clocks
END WHILE
J=VALPTR(I)                     ;1 clock
J=J+CODE-MINICODE(I)            ;3 clocks
VALUE=HUFFVAL(J)                ;1 clock
RETURN VALUE
END DECODE
```

Test sequence applied to the standard sequential decoder

The first element to be decoded is the DC element.

Dc= call DECODE

Step1: First bit is retrieved form the test sequence.

The *call DECODE* uses 17 clocks cycles, but this is not an actual DC Huffman code.

Step2: The sequence “10” is analyzed. *call DECODE* uses 17 clocks cycles, but this is not a DC Huffman code.

Step3: The sequence “101” is analyzed; *call DECODE* uses 17 clocks cycles and finds a valid DC Huffman code. It uses 4 more clock cycles to return the length in bit of the data.

It results that for a Huffman code with the code length of m(where m[1..15]) the Decode function uses 17*m+4 clock cycles.

The next step of the DC decoding is the retrieval of the data and sign decoding. This is done using the function Decode_ZZ(k) – see *Figure 9* - where k is the length of data. The pseudo-code of this function and the corresponding number of clock cycles are detailed bellow:

```
FUNCTION Decode_ZZ(k)
BEGIN
I=0                                     ;1clock
ZZ=0                                    ;1clock
WHILE  I<k                              ;2clocks
        I=I+1                           ;1clock
        ZZ=Shift Left with 1 position of ZZ ; 1clock
        ZZ=ZZ+call NEXTBIT              ; 6 clocks
END WHILE
V_sign=Shift left with (k-1) positions of 2     ; 1clock
IF ZZ<V_sign THEN                       ; 2clocks
        V_sign=Shift Left with k positions of -1; 1clock
         V_sign= V_sign+1               ;1clock
        ZZ=ZZ+ V_sign                   ; 1clock
END IF
RETURN ZZ
END Decode_ZZ(k)
```

It results that the function Decode_ZZ(k) uses 5 +k*10 clock cycles for a positive number and 5 +k*10+3 clock cycles for a negative number.

After these steps the DC coefficient was decoded. The Huffman code was “101” (m=3), the data “1100” (k=4), and because the data was a positive number, the total number of clock cycles used was 17*m+4+5 +k*10=100 clock cycles.

The next bits in the scans represent the AC elements. The algorithm described in *Figure 11* for AC coefficient decoding has the following pseudo code:

```
Decode AC coefficient routine
K=1                                     ;1 clock
ZZ(1,...,63)=0
WHILE K<63                              ;2 clocks
        RS=call DECODE                  ;17*m+4
        SSSS=RS modulo 16               ;2 clocks
        RRRR=Shift Right with 4 positions of RS :1clock
        R=RRRR                          ;1 clock
        IF SSSS=0 THEN                  ;2 clocks
                IF R=15 THEN            ;2 clocks
                K=K+16                  ;1 clock
                END IF
        ELSE

                K=K+R                   ;1 clock
                Call Decode_ZZ(k);5 +k*10 clocks
                K=K+1                   ;1 clock
        END IF
END WHILE
```

The first decoded coefficient had the AC Huffman code "11100" so m=5 and the data '1' length k=1. The routine required 109 clock cycles. The second decoded coefficient had the AC Huffman code "1111000" so m=7 and the data '101010' length was k=6. The routine required 198 clock cycles. The third and last decoded coefficient had the AC Huffman code "1100" so m=4 and the data '0' length k=1. The data was negative so the routine used 100 clock cycles.

To conclude, for the chosen test string, the standard software Huffman decoder required 100 clock cycles for the DC coefficient and 407 clock cycles for the AC coefficients, an overall total of 507 clock cycles.

Test sequence applied to the proposed parallel decoder

For the parallel Huffman architecture – see *Figure 13* - the first 16 bits of the test sequence are compared in parallel with every element for the DC table. Only one of the parallel comparisons is true and the 32 bit Internal Buffer is shifted with the corresponding length and the data is retrieved. The data are processed in the Sign Normalizing block and outputted in a single clock cycle. The AC coefficients are retrieved in the same manner but using the corresponding AC table. The entire string was decoded in only 4 clock cycles.

This compares very favorably with the standard software decoder, a ratio of 507 clock cycles to 4 – that is, the proposed parallel decoder was 127 faster than the standard software approach in this case.

VI. Comparison with other parallel decoders

There are numerous papers on Huffman decoders in general, and parallel implementations in particular. Only some of them are going from proposing an algorithm all the way to hardware implementation. Most of the proposed solutions involve pipeline architectures [6], optimized for particular ASIC implementations [7],[8] in order to obtain a high frequency of operation and/or low power consumption. It is difficult to compare against such solutions, as their speed rely largely on process performances, but it should be noted that they usually require more than one clock cycle to output decoded data. Another drawback of these implementations is their need for substantial hardware resources: for example, the decoders proposed in [7] use computation look-ahead and decomposition techniques to realise efficient parallel architectures; this results in a rather complex implementation for a Huffman decoder.

A representative example of the (very few) papers on FPGA implementation of parallel Huffman decoders is [9]: it proposes a decoder able to output decoded data on one clock cycle, that was fully implemented on an Altera FPGA FLEX 10K20RC240-3. There the max clock frequency was 11.54MHz and the total logic utilization was 99% [9]. Both the algorithm and architecture proposed in this paper are different from the ones in [9]; moreover, the solution here is far faster – the max clock frequency in up to nine times higher – and uses a smaller portion of the available hardware

VII. CONCLUSIONS

This paper presents a novel algorithm and parallel architecture for Huffman decoding. A Jpeg file decoder was then developed and implemented using an off-the-shelf FPGA. Extensive simulations and lab tests have not only demonstrated the proposed algorithm and architecture but have also proven that the new Jpeg decoder can operate at very high clock frequencies, up to 100MHz

Main features that differentiate the proposed decoder from other solutions proposed in the literature are its ability to output decoded data on one clock cycle combined with a relatively simple implementation, which allows high clock rate and reduces hardware requirements.

A detailed comparison with the standard sequential Huffman software decoder shown that the proposed decoder is up to 127 times faster; compared to a parallel decoder implemented on a similar FPGA, the solution here has a max clock frequency almost nine times larger, while using a far smaller percentage of the available hardware resources.

From the integration point of view, this architecture can be attached as a peripheral to one of the embedded software processors like Microblaze or Nios II, were it can be used as a Huffman hardware decoder accelerator for the image processing applications. Another possible application is to combine it with the one of the fast hardware IDCT IP cores available on the market, in order to create an ultra-fast hardware Jpeg decoder.

REFERENCES

1. G. Wallace, The JPEG Still Picture Compression Standard. April 1991, Vol. 34, No. 4, Communication of the ACM
2. CCITT Rec. T.81 (1992 E), Digital compression and coding of Continuous-Tone Still Images – Requirements and Guidelines (ISO/IEC 10918-1 : 1993(E), 1992, pp. 18-21, 50-53,.88-93,103-113
3. S. Ho, P. Law – Efficient Hardware Decoding Method for Modified Huffman Code, Electronics Letters, vol 27, No 10, May 1991, pp. 855-856
4. Microblaze Processor Reference Guide http://www.xilinx.com/support/documentation/ sw_manuals/edk92i_mb_ref_guide.pdf, 2007, pages 9-63
5. On-Chip Peripheral Bus V2.0 with OPB Arbiter, http://www.xilinx.com/support/documentation/ ip_documentation/opb_v20.pdf, 2005. pages 1-30
6. M.K. Rudberg, L. Wanhammar - High speed pipelined parallel Huffman decoding, Proceeding of IEEE ISCAS'97, vol 3, June 1997, pp 2080-2083
7. K.K. Parhi - High-speed VLSI architectures for Huffman and Viterbi decoders, IEEE Tran. On Circuits and Systems II, vol 39/6, June 1999, pp 385-391
8. S. T. Klein, Y. Wiseman, "Parallel Huffman Decoding," Data Compression Conference 2000, pp. 383
9. Z. Aspar, Z. Mohd Yusof, I. Suleiman - Parallel Huffman decoder with an optimized look up table option on FPGA, Proceedings of TENCON 2000. pp.73-76.