FASTSERIES

USER'S MANUAL
JPEG 2000 Encoder

30002-00281

Document Name: ALRT RT SW Programmer's Guide & Reference User's Manual

Document Number: 30002-00281

Revision History: 1.0 December 12, 2001

1.1 September 3, 2003

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
Fax: 603-891-2745

Web Site:
http://www.alacron.com/

Email:
sales@alacron.com, or support@alacron.com

Table of Contents

JPEG2000 Software Architecture

1. Introduction

1.1 Purpose

The document is written for engineers and developers to provide a comprehensive architectural overview of the TriMedia oriented JPEG2000 encoder. It is intended to capture and convey the significant architectural decisions, algorithms and implementation aspects that have been made on the system.

1.2 Scope

This document describes software architecture of the TriMedia based JPEG2000 encoder. The main accents were made on the algorithmic part of the application and the corresponding implementation features.

1.3 Definitions, Acronyms and Abbreviations

J2K – TriMedia based JPEG2000 encoder

CBM – Coefficient Bit Modeling

DWT – Discrete Wavelet Transformation

1.4 References

ISO_IEC_15444-1 2000(E) – Information technology – JPEG 2000 image coding system. Part1: Core coding system.

ISO/IEC 14492-1, Lossy/lossless coding of bi-level images, 2000.

JPEG-2000 Phase 5 Development Summary, Alarity Corp., September, 2002.

2. Architectural Representation

Represented system consists of followed basic components:

- Input file format interpreter
- DWT processor
- Rate allocation unit
- Tier-1 encoder
- Tier-2 encoder
- Output component

Input file interpreter performs input file reading and its format interpretation (currently Portable BitMap -PBM).

DWT processor module performs irreversible (9-7) and reversible (5-3) discrete wavelet transformations according to ISO JPEG2000 standard.

Rate allocation unit controls compression quality / compression rate and provides optimal image quality for the preset target output file size.

Tier-1 encoder implements most of the coding functionality -- bit plane encoding and entropy arithmetic coding process.

Tier-2 encoder packages the tier-1 encoded data into data units (packets).

The output component performs output of the compressed image into output file.

3. General description

Image encoding process can be represented as shown in fig.1. It consists of five main stages: input image processing, wavelet transform, tier-1 and tier-2 processing and output file generation.

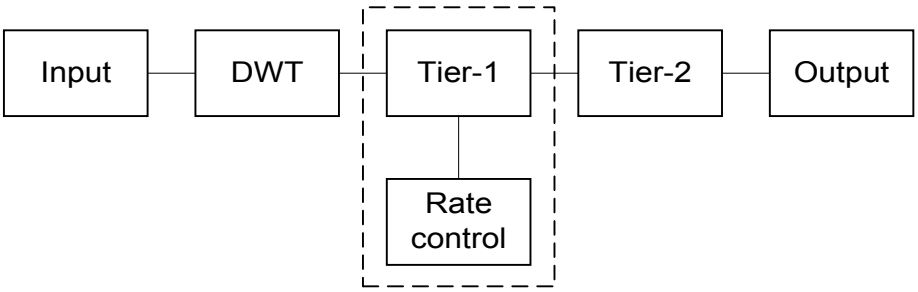


Fig.1 Image encoding process

The Input unit performs input image file reading and decoding. This module actually should be considered as an interface module and should not be treated as a part of JPEG2000 encoder. The only supported input format is 8 bits Portable BitMap (PBM). Any other input format may be supported, as long as it can be converted to 8 bits unsigned input data representation for DWT module input. This restriction comes from the finite precision arithmetic features used in DWT module optimized implementation. To improve performance, input data dynamic range zero centering (DC shifting procedure) was embedded into the first stage of wavelet transformation and does not have to be implemented in the Input module.

The Discrete wavelet transformation module supports both standard wavelet filters recommended by ISO JPEG2000 – reversible 5/3 and irreversible 9/7. All transformations are implemented in finite precision fixed-point arithmetic and in the case of 9/7 filter implementation output data values are scaled up by *nSamplingCoefficient* value (see *Algorithms and implementations* below).

The Tier-1 module provides encoding of resulting DWT coefficients with dynamic rate allocation control. Tier-1 uses original *Passes* allocation algorithm (see *Algorithms and implementations*) and implements single pass fidelity allocation method.

The Tier-module 2 performs zero-tree encoding and JPEG2000 final output stream generation.

The Output module provides the output interface and, as the input module, does not belong to the JPEG2000 encoding algorithm.

The current software provides support of tiles processing in accordance with the requirements of ISO/IEC 15444-1 standard. However, several restrictions were introduced for the performance optimization reasons. Each tile component must start at even row. Width of each tile must be the power of two.

In the present implementation image width, image height, tile width and tile height are defined as compilation-time constants (*Parameters.h*). This allows for the more efficient memory management (see 4.1 and 4.2 for details). The output image size (defining the compression ratio), the type of wavelet transformation filter and the type of rate allocation variant are specified for the encoder as external encoding parameters.

Two general modes are supported – internal tiles multiplexing (when division into tiles is performed inside the encoder) and external multiplexing (when tile data are passed into encoder from the outer source).

In the case when a whole image is passed for encoding, the *JPEG2000_Encode* function (*Encoder.c*) should be called. This function accepts as inputs:

- *pInputBuff* - pointer to the input buffer,
- *iOutputFileSize* - desired size of output file,
- *iTableVariant* - rate allocation variant (4.3),
- *iWavletTransform* - wavelet transformation mode (5/3 or 9/7) ,
- *pOutputBuff* - pointer on the output memory buffer (buffer for output image) .

After the function invocation, the size of the encoded image is stored in *pOutputDataSize* value. Compression ratio is defined implicitly by the output size of compressed image.

If necessary, division into tiles is performed inside the *JPEG2000_Encode* function. All tiles are processed sequentially and the output stream for JPEG2000 file is created.

The *pInputBuff* pointer can be initialized to NULL value. In this case the input image is assumed to be stored in the reserved input buffer (4.1). This feature was reserved for external to J2K compressor input data multiplexing mechanism. If *pInputBuffer* is not NULL the input data values are copied into the reserved buffer before the start of encoding.

External tile multiplexing is also supported. Individual tiles can be directly encoded by the *JPEG2000_EncodeTile* procedure. In this case, the input tile data should be pre-loaded into the reserved input buffer. *JPEG2000_EncodeTile* accepts the following arguments:

- *nTile* – current tile number,
- *pOutData* – pointer to output data buffer,
- *pOutDataSize* – pointer to value that will hold output stream size.

When the *nTile* is equal to 0 (processing of the first tile) *JPEG2000_EncodeTile* will perform necessary actions for JPEG2000 file header creation. These actions are performed by call to *EncodeMainHdr* (*Encoder.*c). *EncodeMainHdr* forms JPEG2000 file header in accordance with the requirements of standard, i.e. it outputs all the necessary description information to the output stream before the start of the output data stream for the first file. Similarly, if the *nTile* equals to the number of the last tile the end marker is output after the tile data stream.

In contrast to *JPEG2000_Encode* call such parameters as allocation variant, wavelet transformation type and output file size should be set before the first tile can be processed. These parameters may be set by the *InitDataStructures* function (*Encoder.c*).

The first stage of encoding process is the forward wavelet transformation (see 4.2 for details). Both transformation types are implemented in fixed point arithmetic and, in case of 9/7 wavelet filter, the output data are scaled up (see *nSamplingCoefficient*). This scaling effect is compensated in the code blocks analysis stage. The wavelet transformation also includes the constant offset data shifting (DC shifting).

4. Algorithms and Implementations

The present software uses the following important algorithmic features: static memory allocation, 2D wavelet transformation algorithm with lifting / convolution fusion scheme, passes-based rate control algorithm, tabulated code blocks / passes processing method and tabulated quantization scheme.

4.1 Static memory allocation

Static memory allocation is used to eliminate memory allocation / reallocation time during program execution. Some statically allocated data is re-used for different buffers during different execution stages. Allocated memory map can be represented as follows (see *Variables.h* for details):

Input Buffer – buffer for input image

Size: *ImageWidth * ImageHeight*.

Output Buffer – buffer for output wavelet coefficients, 16-bit words,

Size: *ImageWisth ImageWidth * ImageHeight * 2.*

DWT Temporary Buffer – buffer for temporary wavelet coefficients (used during wavelet transformation)

Size: *ImageWisth ImageWidth * ImageHeight * 2.*

Flags Temporary Buffer – buffer for arithmetic encoder flags

Size: *2 * CblksAmount * (CblkWidth + 2) * (CblkHeight + 2)*,

where *CblkWidth* is width in pixels of one code block, *CblkHeight* – height in pixels of one code block, *CblkAmount* – full number of code blocks.

Output Stream Buffer – buffer for output stream

Size: ceil(*TileWidth * TileHeight * 9 / 8).*

Tag Tree Buffer – buffer for tag trees data

Size: *2 * MaxTreeSize * SubbandsNumber * (sizeof (int) + sizeof (tagtreenode_t))*,

where *MaxTreeSize* is (2 + 2 * (2 * *CblkAmount* - 1) / 3), *SubbandsNumber* – number of wavelet subbands.

4.2 Discrete Wavelet Transformation

Some algorithm improvements were introduced into discrete wavelet transformation procedure. Most significant of them are parallel packed data processing, combined horizontal / vertical subbands transformation, fusion convolution-lifting algorithm for vertical wavelet decomposition.

Packed data processing reduces amount of data loads and allows using of TriMedia DSP commands such as *ifir8ui* (*ifir8ii*). Wavelet transformation module accepts data in unsigned bytes representation (not centered around zero level) and performs DC shifting simultaneously with the first stage of wavelet decomposition, saving on additional load/store operation otherwise necessary for DC shifting.

The current DWT implementation supports tiling adapted to work with image / tile width equal to power of two, while image and tile height could have arbitrary size. Also each vertical tile must start at even *y* coordinate. These restrictions were introduced for performance reasons. Power of two condition for image / tile width allows us to assume that for any decomposition level we have even number of elements in each row. This is important to ensure that for each decomposition level each row is always aligned on a 4-bytes boundary, which is required for efficient utilization of TriMedia memory I/O and loops unrolling.

For the 5/3 filter we use the recommended lifting scheme:

$$X(2n) = Y(2n) - \left\lfloor \frac{Y(2n-1) + Y(2n+1) + 2}{4} \right\rfloor,$$
$$X(2n+1) = Y(2n+1) + \left\lfloor \frac{X(2n) + X(2n+2)}{2} \right\rfloor$$

where Y – input vector, X – output vector.

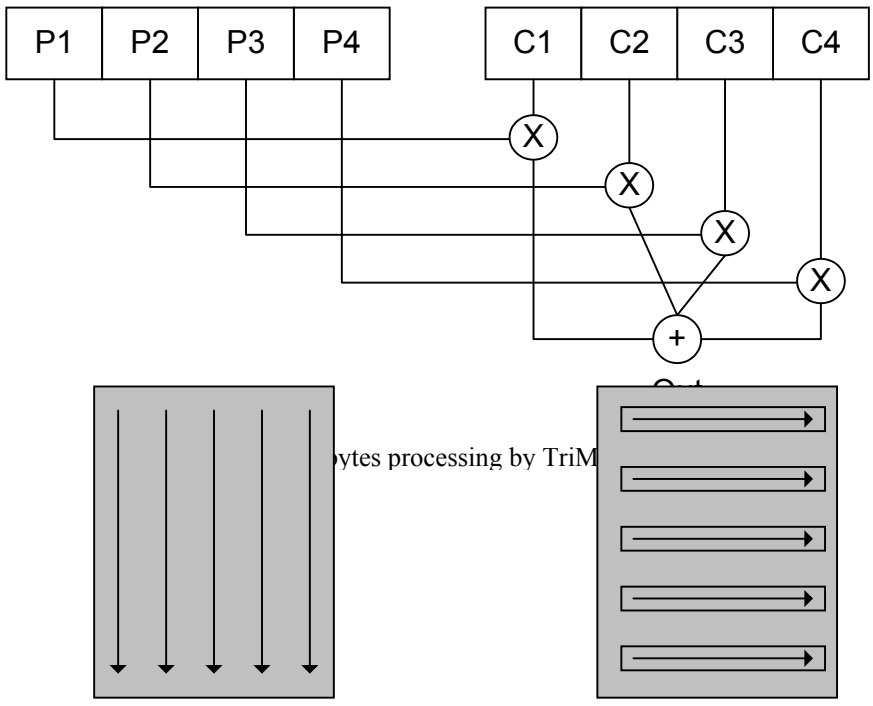
We use the lifting scheme for horizontal wavelet decomposition for the integer 5/3 filter and the convolution scheme for the real 9/7 filter.

To reduce memory I/O operations we do not use symmetric vector extension for boundary points. Instead, we process the first two and the last two points in a special way. For example, in case of lifting scheme for 5/3 filter the first two points are processed as

$$X(0) = Y(0) - \left\lfloor \frac{Y(1) + 1}{2} \right\rfloor,$$
$$X(1) = Y(1) + \left\lfloor \frac{X(0) + X(2)}{2} \right\rfloor.$$

To reduce number of memory load operations, all memory accesses are implemented as 32-bit reads. Individual values are then fetched using byte extraction operations.

In contrast to the 5/3 filter, we use a convolution scheme for real 9/7 wavelet transformation. This scheme was chosen to better utilize TriMedia parallel DSP instructions. These instructions (such as *ifir8ui* and *ifir8ii*) allow to perform four bytes convolution instead of a single 32-bit multiplication. Because we accept the unshifted input data (represented in unsigned bytes) we use *ifir8ui* instruction that accept unsigned values as first argument and signed as second one. For each pixel we need to perform convolution between either 9 or 7 bytes so each elementary convolution could take one or more execution of *ifir8ui.*



a) *b)*

Fig.3 Classical vertical decomposition – *a)*, vertical decomposition row-by-row – *b)*.

For the same reason, all computations are performed in fixed point. In the first iteration, lowpass and highpass wavelet transformation coefficients are scaled up to integer values by multiplication by *LMUL* and *HMUL* values (see *jpc\Wavelet\WConsts.h* for details*)*. *LMUL* and *HMUL* values were chosen to fit [–128, 127] one byte dynamic range.

The second stage of the first wavelet decomposition is performed with coefficients that convert our upscaled arithmetic back to the original scaling (all other decompositions are performed with the same transformation coefficients). This stage is equal to convolution with transformation coefficients modified to compensate the upscaling effect of the first stage (see *L2_*x and *H2_*x coefficients in *jpc\Wavelet\WConsts.h*). At the output of the first wavelet iteration all values are scaled up by four bits and all other iterations continue to operate on four bits upscaled fixed-point numbers.

A very significant aspect affecting algorithm performance is TriMedia caching mechanism. TriMedia processor has 16 Kb data cache and in case of frequent out-to-cache data processing, efficiency of any algorithm becomes very low. To avoid cache misses for vertical DWT filtering we propose a special horizontal-vertical decomposition method. Effectively, the filtering is carried out in the sliding window, moving from up to down (see picture below).

If data belonging to a sliding window fit into the cache, cache updates will happen only when a new row is introduced into processing. The amount of processing data can be estimated as

DataSize = *WindowHeight* * *WindowWidth* * *DataTypeSize*,

where *DataTypeSize* – number of bytes in one data element representation.

We use 16 bits data representation, so *DataTypeSize* is equal to 2. For example, in case of image width of 1024 pixels and 9/7 transform convolution scheme (where 9 elements are required for a single output coefficient calculation), we can estimate the required “working set” data size as 1024*9*2 = 18 Kb.

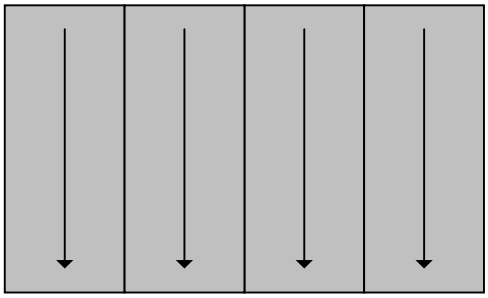


Fig.4 Separate vertical decompositions of image parts

If the required amount of working data exceeds *TriMedia* cache size, many unwanted cash load / store actions happen. A solution to this problem is to divide image into vertical segments for processing (see fig.4).

In this case a whole image is processed by vertical segments, with widths adequate for cache size limitations. Note that this vertical decomposition of image segments is necessary only when a whole image is processed without division into tiles. In the case of tiled processing, appropriate tile size can be chosen to accommodate the size of cache. In contrast with the tiled processing, the described algorithm performs wavelet decomposition without any additional distortion (while tiles processing introduce symmetric extension instead of data near tiles boundaries).

For vertical decomposition in the case of 9/7 filter a special convolution-lifting scheme was proposed. The most important feature of this scheme is reduction in number of memory load operations. In the proposed algorithm all odd coefficients are calculated by the ordinary convolution while the even coefficients are calculated by a lifting-like scheme based on precalculated odd values. From a widely known lifting scheme

$$\begin{cases} Y(2n+1) = X(2n+1) + \alpha\left[X(2n) + X(2n+2)\right] \\ Y(2n) = X(2n) + \beta\left[Y(2n-1) + Y(2n+1)\right] \\ Y(2n+1) = Y(2n+1) + \gamma\left[Y(2n) + Y(2n+2)\right] \\ Y(2n) = Y(2n) + \delta\left[Y(2n-1) + Y(2n+1)\right] \\ Y(2n+1) = KY(2n+1) \\ Y(2n) = \dfrac{1}{K}Y(2n) \end{cases},$$

we may see the necessity to keep temporary values for each processing sample in a row. Hence, in case of a row-by-row vertical decomposition we need to reserve additional memory buffers for temporary coefficients and to perform memory load / store operations from/to these buffers. To avoid these unwanted effects we calculate every even coefficient as a sum of convolution of seven neighboring elements and a sum of two already computed odd coefficients:

$$Y(2n) = \frac{1}{K} Y'(2n) + \frac{\delta}{K^2}\left[Y(2n-1) + Y(2n+1)\right],$$

where

$$Y'(2n) = \alpha\beta\left[X(2n-2) + X(2n+2)\right] + \beta\left[X(2n-1) + X(2n+1)\right] + (1 + 2\alpha\beta)X(2n).$$

With the same computational cost this scheme requires only 7 memory loads for each two points processing (instead of 9 loads for the original scheme). Obviously this algorithm requires joint processing of two neighboring rows (corresponding to even end odd vertical samples).

Both wavelet filters (5/3 and 9/7) are implemented in the following files:

- *jpc\wavelet\Wavelet53.h*
- *jpc\wavelet\Wavelet53.c*
- *jpc\wavelet\Wavelet97.h*
- *jpc\wavelet\Wavelet97.c*
- *jpc\wavelet\WConsts.h*

File *WConsts.h* defines such values as wavelet decomposition coefficients, scaling constants, etc. Other files provide interfaces and implementations for respective filters.

The main functions of filters are *_2DWavelet53* and *_2DWavelet97*. These functions perform 2D wavelet decomposition of input byte-per-pixel images. Input arguments of these functions are input image (*unsigned char**), *DECOMPOSITION_LEVELS* – number of decomposition levels, *tile.cx* and *tile.cy* – tile width and height, *tile.xOffset* and *tile.yOffset* – offsets of the current tile from image top left corner.

_2Dwavelet53 and *_2Dwavelet97* functions functionality is affected by the tile position, odd- or even- alignment of tile y-offset.

The first decomposition iteration is different from the subsequent ones because the format of the input image for the first iteration is different from the data format for other iterations (input data for the first iteration are byte-per-pixel values, while for subsequent iterations they are 16bits-per-pixel). For algorithmic convenience some restrictions were imposed on possible tiles sizes / offsets. Each tile can have width equal to power of two and can start only at even row. The first restriction ensures that any tile and any decomposition subband always starts from even sample, the second restriction allows to use only one type of iteration procedure (iteration that starts at even row) at the first decomposition stage.

For all iterations except the first one the iteration alignment type is automatically defined basing on B.12 equations of the ISO/IEC 15444-1 standard (p.80). For subbands starting at even row the *IterationEven* procedure is called, for subbands started at odd row the *IterationOdd* procedure is called instead.

Each of *FirstIterationEven*, *IterationEven* and *IterationOdd* procedures consists of combined horizontal and vertical wavelet transformations. Horizontal decomposition is performed by the procedures *Lift53_8x16* and *Lift53_16x16* for 5/3 filter and *Conv97_8x16* and *Conv97_16x16* for 9/7 filter.

Vertical transformation for 5/3 filter is computed using the standard scheme, transformation for 9/7 filter is implemented by the convolution-lifting scheme as described above.

4.3 Passes-based rate control algorithm

The rate allocation algorithm is integrated into the coefficient bit modelling and arithmetic encoding process.

The algorithm processes one coefficient bit modelling (CBM) coding pass at a time. At each step, a codeblock to be processed is selected, based on contributions of code blocks to image reconstruction error. After a next CBM pass for the selected codeblock is encoded, the next "worse error case" codeblock is selected. This process continues until a pre-defined size of output data (arithmetic encoder output) is generated.

The pass location procedure consists of a search for appropriate code block throughout all code blocks of the image. To compute contribution of a codeblock to image reconstruction error, the procedure uses codeblock representation error and a pre-calculated weight. The weights are independent of the image data, and are precomputed for each decomposition subband, bitplane of a codeblock and coding pass type.

A simplified algorithm for passes weights estimation can be described as:

```
for each resolution level do
        for each subband do
                for NumBPS = 1 to MAX_BITPLANES do
                        NumPasses = (NumBPS > 0) ? (3 *NumBPS - 2) : 0
                        BitPos = 0
                        for Pass = NumPasses to 0 do
                                CumMSE = BandWeightL2 * ( 1 << (2*BitPos) )
                                 if ( (Pass + 2) % 3 = = 0 )
                                        BitPos += 1
                                PassWeight   = CumMSE + ( 3*NumBPS - Pass )
                                if ( Pass = = 0 )
                                        PassWeight += 100
                                if ( Pass%3 = = 1 )
                                        PassWeight += 40
                                PassWeight += 2*NumBPS
```

Here *NumBPS* is the number of bit planes we expect to find in a code block, *BitPos* is the current position of bitplanes, *CumMSE* is the cumulative mean squared error that would result if all passes starting from the current one are rejected, *BandWeightL2* is the squared band weight and *PassWeight* is the target pass weight.

Computed weights are further modified to ensure proper sequence of coding passes invocation.

All passes are ordered by their relative weights. This order is stored in ordering tables (see *Sources\Tier1\Order.c*). Each table element corresponds to the resolution level, band, number of bit planes, and the pass number.

In practical aspects some modifications to the algorithm may improve compressed image quality. For some types of images (for example for images with small contrast details) some quality improvement can be achieved when weights of passes belonging to the highest resolution levels are increased. For other types of images (smooth brightness variation), assigning higher weights to lower resolution levels could improve quality.

Interfaces and functionality of tier-1 and tier-2 operations are implemented in the following files:

- Tier-1 (*jpc\Tier-1*) – *MQCod.h, MQCod.c, MQEnc.h, MQEnc.c, T1Cod.h, T1Cod.c, T1Enc.h, T1Enc.c, T1Interface.h, T1Interface.c, Order.c.*
- Tier2 (*jpc\Tier-2*) – *T2Enc.h, T2Enc.c, Tagtree.h, Tagtree.c.*

The main function of Tier-1 block is *EncodeCblks* (*T1Interface*). This procedure processes all code blocks of image and calculates the number of non-0 bit planes present in each code block. After the most significant non-zero bit plane is determined, all other coefficients are normalized.. This procedure is performed by the *ConvertTopBitsBlockData* call. Due to the performance requirements, the current number of maximal allowed bit planes is 8. However for quality improvement reasons coefficients of some subbands are processed as 16 bits data. This is accomplished by the *ConvertBottomBitsBlockData* procedure for code blocks belonging to resolution levels less than or equal to *DECOMPOSITION_LEVELS - PROC16_LVLNO* (see *Consts.h*.for current *PROC16_LVLNO* value).

Function *CalcTagTreesSizes* estimates the possible size of all tag tree structures. This estimation is based on the assumption that all passes from all code blocks are encoded.

The main function for code blocks bit modeling and encoding is *RunEncoder* (*T1Interface.c*). In the main body of *RunEncoder* all code blocks are processed in the order defined by the earlier defined tables (as described above). Each pass is encoded by the arithmetic coder *EncodeCblk (T1Enc.c)* using the following functions: *MQEncCreate*, *EncSigPass*, *EncRefPass*, *EncClnPass*, *MQEncFlaush* (*MQEnc.c, T1Enc.c*).

During arithmetic coder execution, encoder output stream is saved for later packing into the final output stream. Each encoded pass is predictably terminated and after the each encoder execution the probability models are reset (see annex *C* of ISO/IEC 15444-1 standard for details). Such behavior is supported by *MQEncCreate* and *MQEncFlush* calls.

After each pass is encoded the output stream size is recalculated. If the output stream size plus all headers size does not exceed the pre-set output size (based on the preset rate value) the next pass is commenced, otherwise the procedure stops.

The final stage of the encoding process is the tier-2 module, i.e. tag array creation and output stream packetization. On this stage two tag array structures are created. The first one corresponds to the code blocks inclusions into the tile, the second one corresponds to the numbers of zero bit planes in each code block. Both of these tag arrays are created by *TagarrayCreate* (*Encoder.c*).

After tag arrays are created the output stream is packed by the *EncodePkts* procedure (*T2Enc.c*). This function sequentially takes the output stream of each code block and outputs it into the final output stream for each tile. During the packetization process each output stream of code block is accompanied by the proper header and is output into the tile output stream. Also the tag array information is continuously refreshed during the packetization procedure. Bit stream output is accomplished by the *BitStreamPutBit* procedure, tag arrays refreshment is done by *TagarrayGetLeaf* and *TagarrayEncode*.

4.4 FPGA-based CBM

The encoding process is optimized for faster execution on FastFrame1300 platform using the frontend FPGA matrix. The FPGA is utilized to carry out coefficient bit modeling, which constitutes one the main computational bottlenecks of the algorithm. The arithmetic encoder is implemented in software on TriMedia, since the available FPGA does not provide enough resources to implement the whole encoder there with enough parallelizm. The regular nature of coefficient bit modeling algorithm enables relatively compact implementation in FPGA, while a higher clock rate of TriMedia allow faster computations of arithmetic coder (sequential in its nature, considering a single coder). VideoIn and VideoOut ports are used for data exchange between TriMedia and FPGA.

FPGA processes data received from TriMedia portion by portion. One portion is one bit-plane of a codeblock. Currently only 64*64 codeblocks can be processed by FPGA, so the input image and each tile must be greater than 64*64 and must have horizontal and vertical sizes divisible by 64 if the FPGA is tobe used.

5. Demonstration application

The provided JPEG2000 encoder is supported by the demonstration application shell J2K. This is the C language application suitable to run on PC and TriMedia based platforms. This program can work in a single image-processing mode and in mode that emulates multi-frame source. After the program is started it checks for processing conditions consistency. This analysis includes validation of tiles width and height in accordance with the restrictions imposed by the DWT implementation. Also these checks include validation of size of globally allocated memory buffer which is restricted to the maximal allowable for TriMedia platform (16 M).

If no errors are found, the application tries to read file *JobScheduler.txt.* This file should contain the names of input files, each in new line, for example:

JobScheduler.txt

1.pbm

2.pbm

…

N.pbm

If the *JobScheduler.txt* file is correctly read all specified files are sequentially processed. The output file names are formed in accordance with the input file name and processing parameters, i.e., wavelet filter type, rate allocation type, and compression ratio. Output file name template is

[OutputName].[CompressionRatio]_[FilterType]_[AllocationVariant].jpc.

For example, for input file *1.pbm*, compressed with 1:20 ratio and filter 9/7 for the first allocation variant, the output file name generated is: *1.20_97_0.jpc*.

6. Performance estimation features

The provided JPEG2000 encoder has some features aimed to support performance estimation. Durations of all processing stages are measured and stored in the timing results array. Duration of any stage can be accessed as an element of *iTimes* array or output to the console by invocation of *JPEG2000_PrintTimingResults* function (*IO.c*). For example, performance measuring report may look like

```
--- 1.pbm ---
        Decoding time = 474696 ms
                Tile-1: 474696 ms
        Encoding time = 543164 ms
                Prepare tiles information = 117 ms
                Encode Main Header = 5 ms
                Encode Main Body = 543029 ms
                        Tile-1: 543029 ms
                                Stage0: Tile Preparation = 290 ms
                                Stage1: Perform Wavelet Analysis = 131022 ms
                                Stage2: Tier-1 = 379016 ms
                                Stage3: Encoding Tile Zero-Tree = 12951 ms
```

Here *1.pbm* is the name of the input file;

Decoding time is time taken for input file format decoding

Encoding time is time taken for image encoding;

Prepare tiles information is memory mapping for tile resolution levels, bands and code blocks;

Encode Main Header is creation of JPEG2000 file header;

Encode Main Body is the main procedure for image encoding.

Encode main body stage is further subdivided into several stages for each tile:

Tile Preparation is initializing of tile dependent structures (levels, bands etc.);

Perform Wavelet Analysis -- wavelet transformation stage,

Tier-1 – coefficient bit modeling and arithmetic encoder;

Encoding Tile Zero-Tree – file output stream creation.

7. Working parameters and constants

Most parameters and constants that control JPEG2000 encoder functionality are defined in files *Parameters.h* and *Consts.h*.

File *Consts.h* provide parameters that define general functionality of the encoder. The most important of them are:

IMAGE_PIXELS – number of pixels in processed image;

TILE_PIXELS – number of pixels in one tile;

TILE_AMOUNT – number of tiles;

Number of allowed decomposition levels:

MAX_LEVELS_ALLOWED - 6,

FIVE_LEVELS_ALLOWED - 5,

FOUR_LEVELS_ALLOWED - 4,

THREE_LEVELS_ALLOWED – 3,

TWO_LEVELS_ALLOWED - 2,

ONE_LEVELS_ALLOWED - 1,

ZERO_LEVELS_ALLOWED - 0;

DECOMPOSITION_LEVELS – automatically calculated number of decomposition levels, depending on the image size.

SUBBANDS_NUM - number of wavelet transformation subbands;

NUM_GUARD_BITS – number of guard bits;

CBLK_WIDTH_LOG2 – automatically calculated base-2 logarithm of code block width;

CBLK_HEIGHT_LOG2 – automatically calculated base-2 logarithm of code block height;

CBLK_AMOUNT – automatically calculated number of code blocks;

INITPLANES – maximal number of bit planes in a code block;

MAX_ALLOWED_PASSES maximal number of allowed CBM passes;

PASS_STREAM_MAX_LEN – automatically calculated length of one pass stream;

CBLK_OUT_STREAM_SIZE – automatically calculated length of code blocks output stream;

MAX_TREE_SIZE – automatically calculated length of zero tree;

MAX_TILE_OUT_SIZE – automatically calculated size of one tile output stream;

Do not edit anything in *Consts.h*!

All working parameters, like image and tile size should be set in file *Parameters.h*:

IMAGE_WIDTH – image width in pixels;

IMAGE_HEIGHT – image height in pixels;

TILE_WIDTH – tile width in pixels;

TILE_HEIGHT – tile height in pixels.

One can use file Parameters.h to tune the functionality of the JPEG2000 encoder.

To enable the FPGA usage for CBM, USE_FPGA constant should be defined in the makefile (this option can be used with TriMedia version only, not with x86).

8. User interfaces

JPEG2000 encoder provides a set of interfaces described in file JPEG2000.h. The following functions are provided:

- void JPEG2000_Initialize()

This function must be called once, immediately after the program is loaded.

- void JPEG2000_Encode(

```
char *pInputBuff,          // Input buffer
int iOutputFileSize,       // Target output file size
int iTableVariant,         // Rate allocation variant (0 or 1)
int iWavletTransform,      // Wavelet transformation 0 - 5/3, 1 – 9/7
uchar **pOutputBuff,       // Output buffer
int *pOutDataSize          // Output buffer length
```

)

If the *pInputBuff* is *NULL* then the input image is assumed to be already stored in the statically allocated memory buffer. Otherwise the copying is performed before the start of the encoding. *iOutputFileSize* defines the desired size of compressed image, *pOutDataSize* points to the variable where the exact resulting size of output file is to be stored. iT*ableVariant* and *iWaveletTransform* define processing mode, *pOutputBuff* points to the buffer for the compressed image. If tiles division is requested in the *Parameters.h* file, memory management operations for tiles support are provided inside the *JPEG2000_Encode* procedure.

It is recommended to use this function for small images compression without image tiling.

- void JPEG2000_EncodeTile(

```
int nTile,                 // Tile number
    uchar **pOutData,      // Pointer onto output buffer
    int *pOutDataSize      // Output buffer length
```

)

JPEG2000_EncodeTile assumes that each tile is stored in the statically allocated memory buffer (4.1). In such case all necessary actions related to JPEG2000 output stream creation are automatically performed in the *JPEG2000_EncodeTile* function. In case when the *nTile* is zero, JPEG2000 header is also created and stored at the very beginning of the output buffer. In case of the last tile processing, a necessary end of file marker will be attached to the end of the stream.

Note that *JPEG2000_InitDataStructurs* function must be called before each image compression to initialize compression parameters. Tiles must be compressed in their natural order, starting from number zero.

- void InitDataStructurs(

```
int iOutputFileSize,       // Target output file size
int iTableVariant,         // Rate allocation variant (0 or 1)
int iWavletTransform       // Wavelet transformation 0 - 5/3, 1 – 9/7
```

)

iOutputFileSize defines the desired size of compressed image. iT*ableVariant* and *iWaveletTransform* define the processing mode. If JPEG2000_EncodeTile is used, this function must be called once before image encoding.

- int JPEG2000_CheckConditions()

Use this function to verify the defined compression parameters. It returns the status code. If the returned value is greater than 300, there is an error in parameters. See also *JPEG2000_Error*.

- void JPEG2000_Error(

```
        unsigned int nErr          // Error status code
    )
```

This function can be useful to print error related information.

- unsigned char* JPEG2000_GetInputBuffer()

Returns pointer to the statically allocated memory buffer.

- int JPEG2000_GetInputBufferSize()

Returns size of the statically allocated memory buffer.

- unsigned char* JPEG2000_GetOutputBuffer(

```
        int nTile                  // Tile number
    )
```

Returns pointer to the compressed output stream for the given tile.

- int JPEG2000_GetOutputBufferSize(

```
        int nTile                  // Tile number
    )
```

Returns the size of compressed output stream for the given tile.

- unsigned char* JPEG2000_GetFullOutputBuffer()

Returns pointer to the compressed output stream for the current image.

- int JPEG2000_GetFullOutputBufferSize()

Returns the size of compressed output stream for the current image.

- void JPEG2000_GetTileRect(

```
        int nTile,             // Tile number
        int *cx,               // Horizontal tile size
        int *cy,               // Vertical tile size
        int *xOffset,          // Horizontal offset of a tile on the tile grid
        int *yOffset           // Vertical offset of a tile on the tile grid

    )
```

This function can be used by external module to properly organize image tiling. Returned values must be interpreted in terms defined in ISO_IEC_15444-1 2000(E).

- void JPEG2000_PrintTimingResults()

Print the profiling results for the current image. Must be called after image compression. An example of profiling report can be seen in section 6 of this document.

JPEG2000 Library User's Guide

9. Introduction

9.1 Purpose

This document provides a user description of JPEG2000 encoder library functionality and interfaces.

9.2 Definitions, Acronyms and Abbreviations

JPEG2000 – image compression algorithm described in ISO_IEC_15444-1 2000(E).

Tile – a rectangular array of points on the image. Each tile can be processed by encoder independently.

CBM – Coefficient Bit Modeling

DWT – Discrete Wavelet Transformation

Tier-1 – CBM and arithmetic encoder module.

Tier-2 – Zero-tree encoding and codestream creation module.

9.3 References

ISO_IEC_15444-1 2000(E) – Information technology – JPEG 2000 image coding system. Part1: Core coding system.

ISO/IEC 14492-1, Lossy/lossless coding of bi-level images, 2000.

JPEG-2000 Software Architecture Document, Alarity Corp., September, 2002.

10. Conditions

The JPEG2000 encoder library can be used by C or C++ programmer for image encoding on following hardware platforms:

- x86 platform with DOS, Windows, Unix (Linux, FreeBSD) and some others excluding MacOS. (It is recommended to use Microsoft Visual C compiler for Windows platform.)
- Philips TriMedia platform. (The Philips SDE is needed.)
- Alacron FastFrame 1300 platform (The Alacron's **alfast** library is needed).

The minimal recommended hardware requirements are:

- 16 Mb of RAM

11. Installation and removing

No installation is needed. Unpack archive lib.zip into a folder on your computer.

To uninstall JPEG2000 encoder library from your computer remove unpacked files and folders.

12. Overview

Image encoding process can be represented as shown in fig.1. It consists of five main stages: input image processing, wavelet transform, tier-1 and tier-2 processing and output file generation.

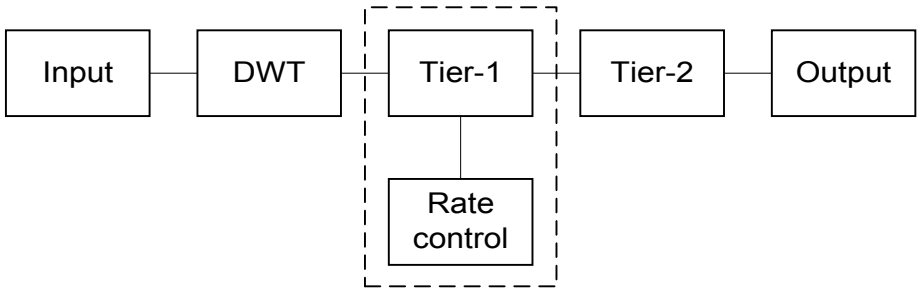


Fig.1 Image encoding process

The **Input unit** performs input image file reading and decoding. This module actually should be considered as an interface module and should not be treated as a part of JPEG2000 encoder. The only supported input format is grayscale 8 bit images. The example application provides input from PBM (Portable Bit Map) files, any other input formats may be supported, as long as it can be converted to 8 bits unsigned input data representation for DWT module input. This restriction comes from the finite precision arithmetic features used in DWT module optimized implementation.

The **Discrete wavelet transformation module** supports both standard wavelet filters recommended by ISO JPEG2000 – reversible 5/3 and irreversible 9/7.

The **Tier-1 module** provides encoding of resulting DWT coefficients with dynamic rate allocation control. Tier-1 uses original *Passes* allocation algorithm and implements single pass fidelity allocation method.

The **Tier-2 module** performs zero-tree encoding and JPEG2000 final output stream generation.

The **Output module** provides the output interface and, as the input module, does not belong to the JPEG2000 encoding algorithm.

The current software provides support of tiles processing in accordance with the requirements of ISO/IEC 15444-1 standard. However, several restrictions were introduced for the performance optimization reasons:

- Each tile component must start at even row.
- Width of each tile must be the power of two.
- Currently the permissible tile and image sizes are restricted to 1024*2048 and can be changed only with library recompilation.

The output image size (defining the compression ratio), the type of wavelet transformation filter and the type of rate allocation variant are specified for the encoder as external encoding parameters.

Two general modes are supported – internal tiles multiplexing (when division into tiles is performed inside the encoder) and external multiplexing (when tile data are passed into encoder from the outer source).

13. Summary of Capabilities

Current version of JPEG2000 encoder library provide the following list of features:

- Grayscale (256 levels of gray, 8 bit per pixel) image encoding
- Image tiling
- Two different wavelet filters, reversible 5/3 filter and irreversible 9/7 filter.

Encoding functions are optimized for fastest processing on TriMedia and FastFrame 1300 platforms. The following algorithm and features are implemented in the library.

- Original rate allocation algorithm working faster on big compression ratios.
- Original DWT providing high performance for TriMedia based platforms.
- Statically allocated memory buffer, there are no memory allocations in JPEG2000 encoder.
- There are no floating point operations in the JPEG2000 encoder.

The following standard features are not supported yet:

- More than one color component;
- Output stream layering;
- Regions of interest;
- Selective arithmetic coding bypass;
- Loseless compression;
- Only one progression order is supported (layer-resolution level-component-position).

14. User interfaces

JPEG2000 encoder provides a set of interfaces described in file JPEG2000.h. The following functions are provided:

- *void JPEG2000_Initialize*()

This function must be called once, immediately after the program is loaded.

- *void JPEG2000_Encode*(

```
        char *pInputBuff,        // Input buffer
        int iOutputFileSize,     // Target output file size
        int iTableVariant,       // Rate allocation variant (0 for MSE rate allocation
                                 principle,  // 1 for original rate allocation
                                 principle)
        int iWavletTransform,    // Wavelet transformation 0 – reversible 5/3 filter, 1 –
                                 // irreversible 9/7 filter
        uchar **pOutputBuff,     // Output buffer
        int *pOutDataSize        // Output buffer length
)
```

If the *pInputBuff* is *NULL* then the input image is assumed to be already stored in the statically allocated memory buffer. Otherwise the copying is performed before the start of the encoding. *iOutputFileSize* defines the desired size of compressed image, *pOutDataSize* points to the variable where the exact resulting size of output file is to be stored. iT*ableVariant* and *iWaveletTransform* define processing mode, *pOutputBuff* points to the buffer for the compressed image. If tiles division is requested in the *Parameters.h* file, memory management operations for tiles support are provided inside the *JPEG2000_Encode* procedure.

It is recommended to use this function for small images compression without image tiling.

- void *JPEG2000_EncodeTile*(

  int *nTile*, // Tile number

  uchar ***pOutData*, // Pointer onto output buffer

  int **pOutDataSize* // Output buffer length

)

JPEG2000_EncodeTile assumes that each tile is stored in the statically allocated memory buffer. In such case all necessary actions related to JPEG2000 output stream creation are automatically performed in the *JPEG2000_EncodeTile* function. In case when the *nTile* is zero, JPEG2000 header is also created and stored at the very beginning of the output buffer. In case of the last tile processing, a necessary end of file marker will be attached to the end of the stream.

Note that *JPEG2000_InitDataStructurs* function must be called before each image compression to initialize compression parameters. Tiles must be compressed in their natural order, starting from number zero.

- void *InitDataStructurs*(

  int *iOutputFileSize*, // Target output file size

  int *iTableVariant*, // Rate allocation variant (0 for MSE rate allocation principle, // 1 for original rate allocation principle)

  int *iWavletTransform* // Wavelet transformation 0 – reversible 5/3 filter, 1 –

  // irreversible 9/7 filter

)

iOutputFileSize defines the desired size of compressed image. iT*ableVariant* and *iWaveletTransform* define the processing mode. If JPEG2000_EncodeTile is used, this function must be called once before image encoding.

- int *JPEG2000_CheckConditions*()

Use this function to verify the defined compression parameters. It returns the status code. If the returned value is greater than 300, there is an error in parameters. See also *JPEG2000_Error*.

- void *JPEG2000_Error*(

  unsigned int *nErr* // Error status code

)

This function can be useful to print error related information. For brief errors list see Annex I.

- unsigned char* *JPEG2000_GetInputBuffer*()

Returns pointer to the statically allocated memory buffer.

- int *JPEG2000_GetInputBufferSize*()

Returns size of the statically allocated memory buffer.

- unsigned char* *JPEG2000_GetOutputBuffer*(

    int *nTile* // Tile number

)

Returns pointer to the compressed output stream for the given tile.

- int *JPEG2000_GetOutputBufferSize*(

    int *nTile* // Tile number

)

Returns the size of compressed output stream for the given tile.

- unsigned char* *JPEG2000_GetFullOutputBuffer*()

Returns pointer to the compressed output stream for the current image.

- int *JPEG2000_GetFullOutputBufferSize*()

Returns the size of compressed output stream for the current image.

- void *JPEG2000_GetTileRect*(

    int *nTile*, // Tile number

    int **cx*, // Horizontal tile size

    int **cy*, // Vertical tile size

    int **xOffset*, // Horizontal offset of a tile on the tile grid

    int **yOffset* // Vertical offset of a tile on the tile grid

)

This function can be used by external module to properly organize image tiling. Returned values must be interpreted in terms defined in ISO_IEC_15444-1 2000(E).

- void *JPEG2000_PrintTimingResults*()

Print the profiling results for the current image. Must be called after image compression. For example, performance measuring report for input file **1.pbm** may look like

```
--- 1.pbm ---
        Encoding time = 543164 ms
                Prepare tiles information = 117 ms
                Encode Main Header = 5 ms
                Encode Main Body = 543029 ms
                        Tile-1: 543029 ms
                                Stage0: Tile Preparation = 290 ms
                                Stage1: Perform Wavelet Analysis = 131022 ms
                                Stage2: Tier-1 = 379016 ms
                                Stage3: Encoding Tile Zero-Tree = 12951 ms
```

Encoding time is time taken for image encoding;

Prepare tiles information is memory mapping for tile resolution levels, bands and code blocks;

Encode Main Header is creation of JPEG2000 file header;

Encode Main Body is the main procedure for image encoding.

Encode main body stage is further subdivided into several stages for each tile:

Tile Preparation is initializing of tile dependent structures (levels, bands etc.);

Perform Wavelet Analysis -- wavelet transformation stage,

Tier-1 – coefficient bit modeling and arithmetic encoder;

Encoding Tile Zero-Tree – file output stream creation.

15. Demonstration application

The provided JPEG2000 encoder is supported by the demonstration application shell J2K. This is the C language application suitable to run on PC and TriMedia based platforms. This program can work in a single image-processing mode and in mode that emulates multi-frame source. After the program is started it checks for processing conditions consistency. This analysis includes validation of tiles width and height in accordance with the restrictions imposed by the DWT implementation. Also these checks include validation of size of globally allocated memory buffer which is restricted to the maximal allowable for TriMedia platform (16 M). The source codes of the example application are listed in Annex III.

If no errors are found, the application tries to read file *JobScheduler.txt.* This file should contain the names of input files, each in new line, for example:

JobScheduler.txt

1.pbm

2.pbm

…

N.pbm

If the *JobScheduler.txt* file is correctly read all specified files are sequentially processed. The output file names are formed in accordance with the input file name and processing parameters, i.e., wavelet filter type, rate allocation type, and compression ratio. Output file name template is

[OutputName].[CompressionRatio]_[FilterType]_[AllocationVariant].jpc.

For example, for input file *1.pbm*, compressed with 1:20 ratio and filter 9/7 for the first allocation variant, the output file name generated is: *1.20_97_0.jpc*.

To compile the demonstration application switch to folder **UsageExample_make** (see Annex II),choose PC, TriMedia or TriMedia+FPGA (FastFrame 1300) target platform by changing dir onto corresponding folder. For x86 platform, open the JPC.dsp with Microsoft Visual studio and Rebuild all ([Alt]-[B]-[R]). For TriMedia based platforms type **nmake** and press enter. The example binary will be created in the current folder.

16. Returned Error Codes List

Error code	Description
400	Tile height is too low
401	Tile width is too low
402	Tile width is too high
403	Tile height is too high
404	Vertical image slicing is not allowed
405	Tile height must be even
407	Output size is too low
408	Not enough memory
< 300	No errors

17. JPEG2000 Software Library Contents

The library distribution consists of the following files and folders:

UsageExample – this directory contains the JPEG2000 encoder functionality usage example.
- _make – directory contains files for making binaries
  - Debug – empty directory for compilation-time temporary files saving
  - PC – project for x86 binaries creation
    - J2K.dsp – Visual Studio project
  - TriMedia – files for TriMedia binaries creation
    - tmconfig – TriMedia configuration file
    - makefile – makefile for TriMedia binaries creation
  - TriMedia+FPGA – files for FastFrame 1300 binaries creation
    - tmconfig – TriMedia configuration file
    - makefile – makefile for TriMedia binaries creation
- pbm – directory contains PBM (P5) decoding implementation.
  - PBMdec.c – Portable Bitmap File (PBM) library.
  - PBMdec.h – PBM library interfaces definition.
- J2K.c – example application source.

CBM.hex – loadable CBM executable for FPGA (for FastFrame 1300 platform)
JPEG2000.a – JPEG2000 encoding library for TriMedia platform.
JPEG2000.h – JPEG2000 encoder interfaces definition.
JPEG2000.lib – JPEG2000 encoding library for x86 platform.
JPEG2000_FPGA.a – JPEG2000 encoding library for FastFrame1300 platform.

18. JPEG2000 Library Example Application

```
/*
 * Name:        J2K.c
 * Description: JPEG2000 library usage example
 * Project:     JPEG2000
 *
 * Copyright (c) 2002 Alarity Corp.
 *
```

```
 * Created:     16-July-2002 Vadim Vashkelis V1.00
 *
 * Revision history:
 *
 */

/*****************************************************************************\
* Includes.
\*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#include "..\JPEG2000.h"
#include "pbm/pbmdec.h"

/*****************************************************************************\
* Macros definitions.
\*****************************************************************************/

// Maximal allowed files
#define MAX_FILES_REQUESTED 369

#define MAX_FILES MAX_FILES_REQUESTED

/*****************************************************************************\
* Code.
\*****************************************************************************/

static int test(
          int iOutputFileSize,
          int iTableVariant,
          int iWavletTransform
        );

int main()
{
   int nError = JPEG2000_CheckConditions();
   int nCPU = JPEG2000_Initialize();

   if( nError >= 300 ){

      JPEG2000_Error( nError );

      return EXIT_FAILURE;

   }
   else
   {
      test( IMAGE_PIXELS / 20, 0 , 0 );
```

```
        test( IMAGE_PIXELS / 30, 0 , 0 );
        test( IMAGE_PIXELS / 50, 0 , 0 );
        test( IMAGE_PIXELS / 20, 1 , 0 );
        test( IMAGE_PIXELS / 30, 1 , 0 );
        test( IMAGE_PIXELS / 50, 1 , 0 );
        test( IMAGE_PIXELS / 20, 0 , 1 );
        test( IMAGE_PIXELS / 30, 0 , 1 );
        test( IMAGE_PIXELS / 50, 0 , 1 );
        test( IMAGE_PIXELS / 20, 1 , 1 );
        test( IMAGE_PIXELS / 30, 1 , 1 );
        test( IMAGE_PIXELS / 50, 1 , 1 );

        return EXIT_SUCCESS;
    }
}

int test(
          int iOutputFileSize,
          int iTableVariant,
          int iWavletTransform
        )
{
    FILE* in;
    FILE* out;

    char pInputNames[ MAX_FILES ][ 255 ];
    char pOutputNames[ MAX_FILES ][ 255 ];
    char pReverser[ 255 ];
    int nFiles;
    FILE* hSchedFile;
    int nFilesCount;
    int nDotPos;
    int nDecLen;
    int nDecimatingValue;

    if(iWavletTransform)
    {
        printf(
              "!!! Parameters: %d_97_%d !!!\n",
              iOutputFileSize,
              iTableVariant
             );
    }
    else
    {
        printf(
              "!!! Parameters: %d_53_%d !!!\n",
              iOutputFileSize,
              iTableVariant
             );
    }
```

```
/* Open the scheduler file. */
if ( !( hSchedFile = fopen( "JobScheduler.txt", "rb" ) ) )
{
   printf( "Error: Cannot open JobScheduler.txt\n" );
   exit( EXIT_FAILURE );
}

nFiles = 0;
while ( !feof(hSchedFile) && nFiles < MAX_FILES )
{
   fgets ( pInputNames[ nFiles ], 255, hSchedFile );
   if ( strchr( pInputNames[ nFiles ], '\r' ) != NULL )
      pInputNames[
                  nFiles
                 ]
                 [
                  strchr( pInputNames[nFiles], '\r' ) -
                  pInputNames[ nFiles ]
                 ] = '\0';

   strcpy ( pOutputNames[ nFiles ], pInputNames[ nFiles ] );
   nDotPos = strstr ( pOutputNames[ nFiles ], ".pbm" ) -
           pOutputNames[ nFiles ];

   // Compression ratio
   nDecimatingValue = IMAGE_PIXELS / iOutputFileSize;
   nDecLen = 0;
   while ( nDecimatingValue > 0 ){
      pReverser [ nDecLen ] = '0' + ( nDecimatingValue % 10 );
      nDecLen ++;
      nDecimatingValue = nDecimatingValue / 10;
   }
   while ( nDecLen > 0 ){
      nDecLen --;
      nDotPos ++;
      pOutputNames[ nFiles ][ nDotPos ] = pReverser[ nDecLen ];
   }
   // FILTER
   pOutputNames[ nFiles ][ nDotPos + 1 ] = '_';
   if( iWavletTransform ){
      pOutputNames[ nFiles ][ nDotPos + 2 ] = '9';
      pOutputNames[ nFiles ][ nDotPos + 3 ] = '7';
   } else {
      pOutputNames[ nFiles ][ nDotPos + 2] = '5';
      pOutputNames[ nFiles ][ nDotPos + 3] = '3';
   }
   // TABLE_VARIANT
   pOutputNames[ nFiles ][ nDotPos + 4 ] = '_';
   if( iTableVariant ){
      pOutputNames[ nFiles ][ nDotPos + 5 ] = '1';
   } else {
      pOutputNames[ nFiles ][ nDotPos + 5 ] = '0';
```

```
    }
    pOutputNames[ nFiles ][ nDotPos + 6 ] = '.';
    pOutputNames[ nFiles ][ nDotPos + 7 ] = 'j';
    pOutputNames[ nFiles ][ nDotPos + 8 ] = 'p';
    pOutputNames[ nFiles ][ nDotPos + 9 ] = 'c';
    pOutputNames[ nFiles ][ nDotPos + 10 ] = '\0';

    nFiles ++;
}

fclose ( hSchedFile );

for ( nFilesCount = 0; nFilesCount < nFiles; ++ nFilesCount )
{
    if( pInputNames[ nFilesCount ][ 0 ] != '\0' ){
        /* Open the input image file. */
        if ( !( in = fopen( pInputNames[nFilesCount], "rb" ) ) ) {
            printf(
                "error: cannot open input image file %s\n",
                pInputNames[nFilesCount]
                );
            exit( EXIT_FAILURE );
        }

        if (TILE_AMOUNT == 1)
        {
            int cx, cy, xOffset, yOffset;
            int pOutDataSize;
            unsigned char *pOutputBuff;

            JPEG2000_GetTileRect(0, &cx, &cy, &xOffset, &yOffset);

            pbm_GetTileSource (in, JPEG2000_GetInputBuffer(), 0,
                cx, cy, xOffset, yOffset);

            JPEG2000_Encode ( NULL,
                        iOutputFileSize,
                        iTableVariant,
                        iWavletTransform,
                        &pOutputBuff,
                        &pOutDataSize );

            out = fopen(pOutputNames[nFilesCount], "wb+");
            fwrite(pOutputBuff, 1, pOutDataSize, out);

            fclose(out);
        }
        else
        {
            int cx, cy, xOffset, yOffset, nTile;
            int pOutDataSize;
            unsigned char *pOutputBuff;
```

```
            JPEG2000_InitDataStructures(iOutputFileSize,
                                iTableVariant,
                                iWavletTransform);
            for (nTile = 0; nTile < TILE_AMOUNT; nTile++)
            {
               JPEG2000_GetTileRect(nTile, &cx, &cy, &xOffset, &yOffset);
               pbm_GetTileSource (in, JPEG2000_GetInputBuffer(), nTile,
                           cx, cy, xOffset, yOffset);
               JPEG2000_EncodeTile ( nTile,
                                &pOutputBuff,
                                &pOutDataSize );
            }
            out = fopen(pOutputNames[nFilesCount], "wb+");
            fwrite(
                 JPEG2000_GetFullOutputBuffer(), 1,
                 JPEG2000_GetFullOutputBufferSize(), out
                );
            fclose(out);
         }

         fclose ( in );
      }

      JPEG2000_PrintTimingResults( pInputNames[ nFilesCount ] );
   }

   /* Success at last! :) */
   return EXIT_SUCCESS;
}
```

19. Coefficient Bit Modeling (BitPlane Scanner) Interface

FE_FPGA _XCV300E-8FG456 Alacron FastFrame1300

1. General information

The function implements the Coefficient Bit Modelling (CBM) functionality of JPEG2000 encoder. That is, it accepts input data and input state of algorithm flags (“significance”, “refined” and “visited” bitplanes), and outputs the requested CBM pass output (stream of data and context labels to be input to the arithmetic encoder) and updated flags planes.

Since the function is to be used in a pass-by-pass encoding framework, a single function invocation produces a single coding pass. In order to allow independent codeblock processing without losing algorithm context (which is composed of three “flag” bitplanes – “Significance”, “Refined” and “Visited” flags), flags backup/load functionality is provided, allowing to load or backup the user-specified flag bitplane.

For interfacing with the function the following TriMedia interfaces are utilized:

- VideoIn port: receiving CBM output (data & context labels to be used as input to the arithmetic

encoder) and reading intermediate flags planes from the function). External (to TriMedia) clock is used.
- VideoOut port: loading of data planes, sign planes and intermediate flags planes into the FPGA. External (to TriMedia) clock is used.
- PCI address 0xFFF80000 (dumbus address space) - R/W - function control register
- PCI address 0xFFF80004 (dumbus address space) - R – byte counter of most recent CBM pass output data (not counting supplemental 0xFF bytes at end of data stream).

Source of all sync is VID_CLK2 signal.

VideoIn and VideoOut ports are used in the message-passing mode. Each data block (bitplane data – 512 bytes, CBM output data – variable size, with byte granularity) are transferred in a single message, therefore large enough buffers must be assigned to VideoIn port to accommodate the maximal possible amount of data. Several (5) supplemental 0xFF bytes are appended to the end of CBM pass output data stream to make sure enough data is transmitted for the VideoIn port to function properly.

For the Virtex300-6 FPGA and the current implementation, the maximal design frequency is 70 MHz.

The only supported codeblock size is 64x64 (hence a bitplane size is 64x64/8 = 512 bytes).

2. Notes on scanning algorithm implementation

Three different coding pass types are implemented, as defined in the JPEG2000 standard: Significance propagation, Magnitude refinement, Cleanup. During each pass, a single pre-loaded data bitplane (and, if necessary, a sign bitplane) is scanned, together with relevant flags and the output data stream, consisting of data bits and context labels, is produced. Relevant to the pass flags are updated.

To save on I/O operations, flag bitplanes consisting of all zeros do not have to be pre-loaded explicitly. For this case a special control bit exist in the control register, forcing the flags plane to be 0 irrespective of the flags state in FPGA RAM. When a flag plane is used in a coding pass, its state in RAM is correct after the pass. Note that if a flag bitplane is not used in a pass, it remains unmodified. For instance, after the significance propagation pass, the “Refined” flag bitplane remain unchanged, even if respective “Force to zero” flag was set.

In order to minimize the number of load/backup flags operations when switching between codeblocks, it is important to understand which flag planes is used by which pass type.

19.1 Significance propagation uses (reads and updates if necessary):

“Significance” and “Visited” flags.

19.2 Magnitude refinement uses (reads and updates if necessary):

“Significance”, “Refined”, “Visited” flags.

19.3 Clean-up uses (reads and updates if necessary):

“Significance”, “Visited” flags. “Visited” flags are reset to 0s after this pass, although they do not really have to be backed up/restored between different data bitplanes processing, since predictably no one point is “visited” for the new bitplane and all points are “visited” after all coding passes have been applied to a data bitplane.

Flags semantic and initial states:

"Significance" – "Significance" state of the point, as defined in the standard (that is, a point becomes "significant" (this flag set to 1) when the first non-0 data bit of the coefficient is encoded). Must be kept persistent throughout entire codeblock encoding (for all data bitplanes). Initial state at the start of codeblock encoding (at the start of the first "cleanup" pass) – all 0s.

"Refined" – Set to 1 when a point is "refined" (processed by the "magnitude refinement" pass) for the first time. Necessary to select a proper context, since it is different for the first refinement. Initial state at the start of the first "refinement" pass for the codeblock – all 0s.

"Visited" – Set to 1 when a point is processed either in "Significance propagation" or "Magnitude refinement" pass. Always reset to 0 in the clean-up pass. Necessary to identify points in a data bitplane that were not processed in either "significance propagation" or "magnitude refinement" pass, so that they are processed in the last, "cleanup" pass. Must be kept persistent throughout a data bitplane coding (3 passes). Do not need to be saved between different data bitplanes coding. Initial state at the start of each data bit plane encoding (at the start of the first "significance propagation" pass for each data plane) – all 0s.

Note that for the very first ("cleanup") pass applied to the most significant bitplane of a codeblock, actually no flag bitplane has to be loaded or saved. Necessary 0-states for "significance" flags can be requested by the 0-forcing bit in control register. No flag bitplane has to be backed up because (1) the significance flag bitplane is equal to the first (just processed) data bitplane, (2) the "refined" flag bitplane is not affected by the "cleanup" pass at all, (3) the "visited" flag bitplane, as the "significance" bitplane, is equal to the first data bitplane.

3. Main operations

3.1 Loading of data/sign bitplanes and flags bitplanes (from TriMedia VideoOut, Msg-passing Mode)

The FPGA function is set to "receive ready" state upon the "Load" command and remains in this state until another command is requested or the data block is transmitted through the VideoOut port. Immediately after the command the software must transmit the properly formatted bitplane data.
Size of expected data block is 512 bytes (64x64 single bitplane). Type of the loaded bitplane is determined by control register, AddrM[2:0] field. TM_VideoOut_Clock – External (VID_CLK2). ***TMVCO_En (bit 13)*** *must be set. Execution of the command is finished with transition to the "Idle" state.*

3.2 Scanning/coding pass (to TriMedia VideoIn, Msg-passing Mode)

The function starts coding upon the "Coding" command, outputting CBM output data bytes to VideoIn port as they become available. VideoIn port must be preset by software to the proper mode before the command issue. All necessary data and flags must be pre-loaded before, initial conditions are defined by control register fields Orient[1:0], Cn[1:0], VF_0Read, RF_0Read, SF_0Read. Modified flags are stored to their respective FPGA memory blocks. Execution of the command is finished with transition to the "Idle" state. Scanning proceed at VID_CLK2 clock.

3.3 Backing up flags (to TriMedia VideoIn, Msg-passing Mode)

The function starts transmitting flags plane data upon the "BackUp" command. VideoIn port must be preset by software to the proper mode before the command issue. Type of flags bitplane

(SignifFlag, RefFlag, VisFlag) is defined by control register, AddrM[2:0] field. Size of data block is 512 bytes (64x64, single bit plane). Execution of the command is finished with transition to the "Idle" state.

4. PCI registers (on dumbus)

4.1 Command register – PCI address 0xFFF80000 (DumBus)

Bit 15 Reset (not implemented)

Bit 14 ---------- not used

Bit 13 w/r TMVCO_En - TMVideoOut Clock External
When set to 1 - VID_CLK2 output is enabled to the TMVideoOut Clock line. TMVideoOut port should be set up to external clock source. The bit can remain always set to 1.

Bit 12,11,10 w/r AddrM[2:0]
Type (address) of the data bitplane to be loaded/stored. The field is defined for "Load" and "BackUp" commands. 000 - Sign, 001- Data, 010- SignifFlag, 011- RefFlag, 100- VisFlag.

Bit 9,8 w/r Mode[1:0]
00 – "Idle". Waiting for the next command. The state is entered immediately after configuration of XCV300 and upon completion of any other command.
01 – "Load" Load data or flags bitplane. Type (address) of the data loaded (Sign, Data, SignifFlag, RefFlag, VisFlag) is defined by AddrM[2:0].
10 – "BackUp" Save flags. Type (address) of the data backed up (SignifFlag, RefFlag, VisFlag) is defined by AddrM[2:0].
11 – "Code" Scanning (coding) of data. Coding pass type and initial conditions must be defined by Orient[1:0], Cn[1:0], VF_0Read, RF_0Read, SF_0Read fields.

Bit 6,5,4 w/r VF-0Read, RF_0Read, SF_0Read (in this respective order)
When a bit is set to 1, the respective flag bitplane is assumed to be all zeros. Current FPGA RAM contents is ignored and assumed to be 0. If a flag bitplane is used in the coding pass, it is always correctly set after the pass.

Bit 3,2 w/r Orient[1:0]
Orientation of the encoded data (as per JPEG2000 standard).

Bit 1,0 w/r Cn[1:0]
Coding pass type: 00 – Significance propagation pass. 01 – Refinement pass. 10 – Cleanup pass.

4.2 Code size register – PCI address 0xFFF80004 (DumBus)

Bit [15:0] r Code_Size(byte)
Counter of bytes, output to TMVideoIn during last coding pass. Cleared on receiving "Code" command.

5. Data formats

5.1 Input/Output bitplanes format

All input data values, data signs and flags are assumed to be formatted in a scan-order bitplane format. Each data block represents a single bit plane. Since the only allowed codeblock size is 64x64, the only allowed bit plane size is 64x64/8 = 512 bytes.

Each byte consists of two 4-bit nibbles. Each nibble corresponds to a 4-point vertical "column", of 64 of which a horizontal scanning "stripe" is constructed. The lower nibble corresponds to "more western" column, while the higher nibble correspond to "more eastern" column. That is, the first column in a stripe

goes to the least significant nibble, the next column goes to the most significant nibble, the next column goes to the least significant nibble of the next byte, and so on, in the algorithm scanning order.

Within each nibble, bit 0 (the least significant one) corresponds to the upper ("northern") point in a column, bit 3 (the most significant one) corresponds to the lower ("southern") point.

5.2 Scanning/coding output data format

The output of the CBM function is a variable-size stream of bytes. Each byte is encoded as follows:

Bits 7..6 – Data (only bit 7 is used in most cases, bits 7..6 are used only for "uniform" (runlength) context).

Bits 4..0 – Context label.

When the context label is 18 (0x12) ("Uniform" context), bits 7..6 of byte represent the run length. In this case, as per the standard, the most significant bit of this 2-bit runlength should be input first to the arithmetic encoder.

For all other contexts, the most significant bit (bit 7) of a byte is the data bit that should be input, together with the context label, to the arithmetic encoder.

At the end of data stream several (5) 0FFh bytes are added to satisfy the minimal data size condition of port.

6. FPGA Function state indication on FastFrame-1300 LEDs

Led01 - "Idle",
Led02 – "Load",
Led03 – "Backup",
Led04 – "Coding",
Led06 – always on.

20. TROUBLESHOOTING

There are several things you can try before you call Alacron Technical Support for help.

_____ Make sure the computer is plugged in. Make sure the power source is on.

_____ Go back over the hardware installation to make sure you didn't miss a page or a section.

_____ Go back over the software installation to make sure you have installed all necessary software.

_____ Run the Installation User Test to verify correct installation of both hardware and software.

_____ Run the user-diagnostics test for your main board to make sure it's working properly.

_____ Insert the Alacron CD-ROM and check the various Release Notes to see if there is any information relevant to the problem you are experiencing.

The release notes are available in the directory: \usr\alacron\alinfo

_____ Compile and run the example programs found in the directory: \usr\alacron\src\examples

_____ Find the appropriate section of the Programmer's Guide & Reference or the Library User's Manual for the particular library and problem you are experiencing. Go back over the steps in the guide.

_____ Check the programming examples supplied with the runtime software to see if you are using the software according to the examples.

_____ Review the return status from functions and any input arguments.

_____ Simplify the program as much as possible until you can isolate the problem. Turning off any operations not directly related may help isolate the problem.

_____ Finally, first save your original work. Then remove any extraneous code that doesn't directly contribute to the problem or failure.

21. ALACRON TECHNICAL SUPPORT

Alacron offers technical support to any licensed user during the normal business hours of 9 a.m. to 5 p.m. EST. We offer assistance on all aspects of processor board and PMC installation and operation.

21.1 Contacting Technical Support

To speak with a Technical Support Representative on the telephone, call the number below and ask for Technical Support:

Telephone: ***603-891-2750***

If you would rather FAX a written description of the problem, make sure you address the FAX to Technical Support and send it to:

Fax: ***603-891-2745***

You can email a description of the problem to **support@alacron.com**

Before you contact technical support have the following information ready:

_____ Serial numbers and hardware revision numbers of all of your boards. This information is written on the invoice that was shipped with your products.

_____ Also, each board has its serial number and revision number written on either in ink or in bar-code form.

_____ The version of the ALRT, ALFAST, or FASTLIB software that you are using.

_____ You can find this information in a file in the directory: **\usr\alfast\alinfo**

_____ The type and version of the host operating system, i.e., Windows 98.

_____ Note the types and numbers of all your software revisions, daughter card libraries, the application library and the compiler

_____ The piece of code that exhibits the problem, if applicable. If you email Alacron the piece of code, our Technical-Support team can try to reproduce the error. It is necessary, though, for all the information listed

above to be included, so Technical Support can duplicate your hardware and system environment.

21.2 Returning Products for Repair or Replacements

Our first concern is that you be pleased with your Alacron products.

If, after trying everything you can do yourself, and after contacting Alacron Technical Support, you feel your hardware or software is not functioning properly, you can return the product to Alacron for service or replacement. Service or replacement may be covered by your warranty, depending upon your warranty.The first step is to call Alacron and request a “Return Materials Authorization” (RMA) number.This is the number assigned both to your returning product and to all records of your communications with Technical Support. When an Alacron technician receives your returned hardware or software he will match its RMA number to the on-file information you have given us, so he can solve the problem you’ve cited.

When calling for an RMA number, please have the following information ready:

_____ Serial numbers and descriptions of product(s) being shipped back

_____ A listing including revision numbers for all software, libraries, applications, daughter cards, etc.

_____ A clear and detailed description of the problem and when it occurs

_____ Exact code that will cause the failure

_____ A description of any environmental condition that can cause the problem

All of this information will be logged into the RMA report so it’s there for the technician when your product arrives at Alacron.Put boards inside their anti-static protective bags. Then pack the product(s) securely in the original shipping materials, if possible, and ship to:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Clearly mark ***the outside of your package:***