TMS320C64x+ DSP Image/Video Processing Library (v2.0.1)

Programmer's Guide

![Texas Instruments logo]

Contents

List of Tables

Read This First

About This Manual

This document describes the TMS320C64x+ Image/Video Library 2 (IMGLIB2).

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.
- Registers in this document are shown in figures and described in tables.
  - Each register figure shows a rectangle divided into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties.
  - Reserved bits in a register figure designate a bit that is used for future device expansion.

Related Documentation From Texas Instruments

The following documents describe the TMS320C6000™ digital signal processor (DSP) devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C64x™ and TMS320C64x+ DSPs of the TMS320C6000 DSP family. The C64x/C64x+ DSP generation comprises fixed-point devices in the C6000 DSP platform. The C64x+ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set.

TMS320C64x to TMS320C64x+ CPU Migration Guide (literature number SPRAA84) describes migrating from the Texas Instruments TMS320C64x digital signal processor (DSP) to the TMS320C64x+ DSP. The objective of this document is to indicate differences between the two cores. Functionality in the devices that is identical is not included.

Trademarks

TMS320C6000, TMS320C64x are trademarks of Texas Instruments.

DSPImage/Video Processing Library

1   Introduction to the TI C64x+ IMGLIB

The Texas Instruments C64x+ IMGLIB is an optimized Image/Video Processing Functions Library for C programmers using TMS320C64x+ devices. It includes many C-callable, assembly-optimized, general-purpose image/video processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. Using these routines assures execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI IMGLIB can significantly shorten image/video processing application development time.

1.1   Features and Benefits

The TI C64x+ IMGLIB contains commonly used image/video processing routines, as well as source code that allows you to modify functions to match your specific needs.

IMGLIB features include:

- Optimized assembly code routines
- C and linear assembly source code
- C-callable routines fully compatible with the TI C6x compiler
- Host library to enable PC based development and testing
- CCS/VC++ projects to rebuild library
- Benchmarks (cycles)
- Tested against reference C model
- Test bench with reference input and output vectors

1.1.1   Software Routines

The rich set of software routines included in the IMGLIB is organized into three different functional categories as follows:

- Compression and decompression
- Image analysis
- Picture filtering/format conversions

In addition, a set of 22 low-level kernels have been included in Appendix A. These functions perform simple image operations such as addition, substraction, multiplication, etc and are intended to be used as a starting point for developing more complex kernels.

2   Installing and Using IMGLIB

This section provides the information needed to install the correct directory structure, and the proper steps to follow to use IMGLIB2.

2.1 Installing IMGLIB

IMGLIB is provided as a self-installing executable, imglibc64plus-2.x.x-Setup.exe. Upon installation, it produces the following directory structure:

```
 IMGLIB2
|
+--build              project files to builds host/target lib
|  |
|  +--host
|  +--target
|
+--docs               library documentation
|
+--include            Required include files
|
+--kernels            Kernel sources
|  |
|  +-- asm
|  +-- c
|  +-- intrinsics
|  +-- serial_asm
|
+--lib                host and target library
|  |
|  +--host
|  +--target
|
+--test_drivers       test bench with reference input/output vectors
|  |
|  +--drivers
|  |    |
|  |    +--set of test-cases
|
+--README.txt         Top-level README file
|
+--TI_license.pdf     License Agreement file
```

The default location for installing IMGLIB is C:\CCStudio_v3.3\c64plus. The user can modify the install directory to any location of choice.

2.2 Using IMGLIB2

2.2.1 Calling an IMGLIB2 Function From C

In addition to correctly installing the IMGLIB software, these steps must be followed to include an IMGLIB2 function in your code:

- Include the function header file corresponding to the IMGLIB function
- Link your code with imglib2.l64P
- Use the correct linker command file for your platform. Note that most functions in imglib2.l64P are written assuming little endian mode of operation.

For example, if you want to call the IMG_fdct_8x8 IMGLIB2 function, you would add:

```
#include <IMG_boundary_8.h>
```

in your C file, and compile and link using:

```
cl6x main.c -z -o IMG_boundary_drv.out -lrts64plus.lib
-limglib2.l64P
Note: The natural c version of the library is also provided. This can be used for debugging code.
#include <IMG_boundary_8_c.h>
cl6x main.c -z -o IMG_boundary_drv.out -lrts64plus.lib -limglib2_cn.l64P
```

2.2.1.1 Code Composer Studio Users

If you set up a project with Code Composer Studio, you can add IMGLIB by selecting Add Files to Project from the Project menu, and choosing imglib2.l64P from the list of libraries under the c64plus\imglib_v2xx folder. Also, ensure that you have linked to the correct runtime support library (rts64plus.lib). An alternate

to include the above two libraries in your project is to add the following lines in your linker command file:
-lrts64plus.lib
-limglib2.l64P
The include directory contains the header files necessary to be included in the C code when you call an IMGLIB2 function from C code, and should be added to the "include path" in CCS build options.

2.2.2 Calling an IMGLIB2 Function From VC++

The procedure remains the same as Section 2.2.1. The only diffrence is that imglib2_host.lib should be included in the corresponding VC project. The VC++ library uses the 'natural c' functions as its source.

2.2.3 Calling an IMGLIB Function From Assembly

The C64x+ IMGLIB functions were written to be used from C. Calling the functions from assembly language source code is possible as long as the calling function conforms to the Texas Instruments C6000 C-compiler calling conventions. See Runtime Environment, *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (SPRU187).

2.2.4 IMGLIB Testing - Allowable Error

IMGLIB is tested under the Code Composer Studio environment against a reference C implementation. Test routines that deal with fixed-point type results expect identical results between Reference C implementation and its assembly implementation. The test routines that deal with floating-point results typically allow an error margin of 0.000001 when comparing the results of reference C code and IMGLIB assembly code.

2.2.5 IMGLIB Overflow and Scaling Issues

The IMGLIB functions implement the exact functionality of the reference C code. You must conform to the range requirements specified in the function API, as well as restricting the input range so that the outputs do not overflow. Overflows or validity of input parameters is not checked in the functions.

2.2.6 Interrupt Behavior of IMGLIB Functions

All of the functions in this library are designed to be used in systems with interrupts. That is, it is not necessary to disable interrupts when calling any of these functions. The functions in the library will disable interrupts as needed to protect the execution of code in tight loops and so on. Functions in this library fall into three categories:

- Fully-interruptible: These functions do not disable interrupts. Interrupts are blocked by at most, 5 to 10 cycles at a time (not counting stalls) by branch delay slots.
- Partially-interruptible: These functions disable interrupts for long periods of time, with small windows of time when they can be interrupted. Examples include a function with a nested loop, where the inner loop is non-interruptible and the outer loop permits interrupts between executions of the inner loop.
- Non-interruptible: These functions disable interrupts for nearly their entire duration. Interrupts may happen for a short time during their setup and exit sequence.

Note that all three function categories tolerate interrupts. That is, an interrupt can occur at any time without affecting the functions' correctness. The function's ability to be interrupted only determines how long the kernel might delay the processing of the interrupt.

The interrupt handling behavior can be changed for the *intrinsic* and *natural c* functions in the library by appropriately modifying the build options. However, this change will not have any effect on the assembly functions in the library.

2.3 Rebuilding IMGLIB

If you would like to rebuild IMGLIB (for example, because you modified the source file contained in the archive, or to obtain a library with different compile options, or for debugging, etc.), you can use the corresponding CCS/VC++ projects in <install_dir>/build/target/ and <install_dir>/build/host/ .

2.4 IMGLIB2 Test Suite

A test suite for most of the functions is included. This test bench is comprised of a driver file, and reference input and output test vectors.

The test suite for each kernel is comprised of three files:

- <kernel_name>_d.c: This is the driver file which call the kernel and checks the output for correctness
- <kernel_name>_idat.c: The reference in put data file
- <kernel_name>_odat.c: The reference output data file

The driver file feeds the input dta to the kernel. The output from the kernel is tested against the refernce output data.

2.5 Building the Test Suite

A Cygwin-based build setup is provided with the release. The following steps must be followed to build the test suite:

- Set the path for C6x compile tools
- Go to <Install_dir>/test_drivers/common/
- Execute build_all.sh

These steps will build all available test applications. The test covers natural c as well as optimized code. The test applications can be cleaned by the command

```
$>build_all.sh clean
```

3   IMGLIB2 Function Descriptions

This section provides a brief description the functions within the IMGLIB2, organized in three categories: image analysis, picture filtering, and compression/decompression. It also provides examples of the function's application.

3.1 IMGLIB2 Functions Overview

The C64x+ IMGLIB2 provides a collection of C-callable high-performance routines that can serve as key enablers for a wide range of image/video processing applications. These functions are representative of the high performance capabilities of the C64x+ DSPs. The following sections describe some of the functions and their applications. These are only representative examples; there are many alternate uses as well.

All functions in the IMGLIB have been developed for the little-endian memory model. A few may work in the big-endian memory model. However, their functionality is not guaranteed.

3.2 Notational Conventions

Following are the conventions used for naming the functions. A suffix is placed after each function name based on the type of inputs it accepts. The various suffixes used are divided into four categories as described below.

- For all the Correlation and Convolution functions which do not involve the q-point math, the suffix will consist of the data length and sign of the input and masks/coefficients in the order '_ids_cds' where:
  - _ids denotes input (i), data length (d), and sign (s). For example, _i8s denotes that input consists of 8-bit signed data. For unsigned data, (s) will be omitted. For example, _i8 denotes that input consists of 8-bit unsigned data.
  - _cds denotes coefficients/masks (c), data length (d), and signed (s). For example, _c8s denotes that coefficients/masks are 8-bit signed. For unsigned coefficients/masks, (s) will be omitted. For example, _c8 denotes that coefficients/masks are 8-bit unsigned.
- For all the functions involving Q-point math, the suffix will be _iq. These functions operate on 32-bit input data and result in 32-bit output data which may be signed or unsigned according to the API

definition. For example, IMG_corr_gen_iq, works on 32-bit input data and result in 32-bit output data.

- For all the functions with two inputs of same data length and sign, the suffix will be _ds. For example, _8s denotes inputs which are 8-bit signed. For unsigned coefficients/masks, (s) will be omitted. For example, _8 denotes inputs which are 8-bit unsigned.
- For all the functions which have a single input of a particular data length and sign, the suffix will be _ds. For example, _8s denotes input consists of 8-bit signed data. For an unsigned data input 's' will be omitted. For example, _8 denotes input consists of 8-bit unsigned data

A few examples for the four categories of suffixes are represented in the table below

Table 1. Conventions Used for Naming Functions

Suffix Category	Suffix Notation	Suffix Example	Description	Example
1	_ids_cds	_i8_c16s	8-bit unsigned input and 16-bit signed masks/coefficients. No IQ format used for both the inputs	IMG_conv_5x5_i8_c16s
2	_iq	_iq	Input/Output in Q-point format	IMG_corr_gen_iq
3	_ds	_32s	32-bit signed or unsigned inputs. No IQ format used for both the inputs	IMG_vecsum_32s
4	_ds	_16s	16-bit signed input. No IQ format used for the input	IMG_boundary_16s

3.3 IMGLIB Image Analysis Functions Overview

This section provides a description of the functions that are applicable to image analysis.

3.3.1 Boundary and Perimeter Functions

- IMG_boundary_8
- IMG_boundary_16s
- IMG_perimeter_8
- IMG_perimeter_16

Boundary and perimeter computation functions IMG_boundary and IMG_perimeter, are provided. These are commonly-used structural operators in vision applications.

3.3.2 Dilation and Erosion Operation Functions

- IMG_dilate_bin
- IMG_erode_bin

The IMG_dilate_bin and IMG_erode_bin functions are morphological operators that are used to perform Dilation and Erosion operations on binary images. Dilation and erosion are the fundamental building blocks of various morphological operations such as Opening or Closing that can be created from combinations of dilation and erosion. These functions are useful in machine vision and medical imaging applications.

3.3.3 Edge Detection Function

- IMG_sobel_3x3_8
- IMG_sobel_3x3_16s
- IMG_sobel_3x3_16
- IMG_sobel_5x5_16s
- IMG_sobel_7x7_16s

Edge detection is a commonly-used operation in vision systems. Many algorithms exist for edge detection, and one of the most commonly used ones is Sobel edge detection. The above functions provide an optimized implementation of the Sobel operator with different mask sizes.

3.3.4 Histogram Function

- IMG_histogram_8
- IMG_histogram_16

The histogram routine provides the ability to generate an image histogram. An image histogram is basically a count of the intensity levels (or some other statistic) in an image. For example, for a grayscale image with 8-bit pixel intensity values, the histogram will consist of 256 bins corresponding to the 256 possible pixel intensities. Each bin contains a count of the number of pixels in the image that have that particular intensity value. Histogram processing (such as histogram equalization or modification) is used in areas such as vision systems and image/video content generation systems. The 16-bit version can operate on images with data resolution from 8 to 16 bits.

3.3.5 Image Threshold Function

- IMG_clipping_16s
- IMG_thr_gt2max_8
- IMG_thr_gt2thr_8
- IMG_thr_le2min_8
- IMG_thr_le2thr_8
- IMG_thr_gt2max_16
- IMG_thr_gt2thr_16
- IMG_thr_le2min_16
- IMG_thr_le2thr_16

Different forms of image thresholding operations are used for various reasons in image/video processing systems. For example, one form of thresholding may be used to convert grayscale image data to binary image data for input to binary morphological processing. Another form of thresholding may be used to clip image data levels into a desired range, and yet another form of thresholding may be used to zero out low-level perturbations in image data due to sensor noise. Thresholding is also used for simple segmentation in machine vision applications.

3.4 IMGLIB Picture Filtering Functions Overview

This section provides a description of the functions that are applicable to picture filtering and format conversions.

3.4.1 Color Space Conversion Functions

- IMG_ycbcr422p_rgb565

Color space conversion from YCbCr to RGB enables the display of digital video data generated, for instance, by an MPEG or JPEG decoder system on RGB displays.

- IMG_yc_demux_be16_8
- IMG_yc_demux_le16_8
- IMG_yc_demux_be16_16
- IMG_yc_demux_le16_16

These routines take a packed YCrYCb color buffer in big-endian or little-endian format and expands the constituent color elements into separate buffers in little-endian byte ordering.

3.4.2 Convolution Function

The convolution functions are used to apply generic filters to the input image. Filter sizes of 3x3, 5x5, 7x7, and 11x11 are supported. Typical applications include, but are not restricted to, image smoothing and sharpening.

The following functions operate on 16-bit image data:

- IMG_conv_3x3_i16s_c16s
- IMG_conv_3x3_i16_c16s
- IMG_conv_5x5_i16s_c16s
- IMG_conv_7x7_i16s_c16s
- IMG_conv_11x11_i16s_c16s

The following functions operate on 8-bit image data:

- IMG_conv_3x3_i8_c8s
- IMG_conv_5x5_i8_c8s
- IMG_conv_5x5_i8_c16s
- IMG_conv_7x7_i8_c8s
- IMG_conv_7x7_i8_c16s
- IMG_conv_11x11_i8_c8s

3.4.3 Correlation Functions

Correlation functions are provided to enable image matching. Image matching is useful in applications such as machine vision, medical imaging, and security/defense. The following functions implement highly optimized corrrelation for commonly-used filter sizes such as 3x3, 5x5, and 11x11.

- IMG_corr_3x3_i8_c8
- IMG_corr_3x3_i8_c16s
- IMG_corr_3x3_i16s_c16s
- IMG_corr_3x3_i16_c16s
- IMG_corr_5x5_i16s_c16s
- IMG_corr_11x11_i8_c16s
- IMG_corr_11x11_i16s_c16s

The functions below are more generic and can implement correlation for user-specified pixel neighborhood dimensions within documented constraints. The IMG_corr_gen_iq function handles 32-bit Q-point data.

- IMG_corr_gen_i16s_c16s
- IMG_corr_gen_iq

3.4.4 Error Diffusion Function

- IMG_errdif_bin_16

Error diffusion with binary valued output is useful in printing applications. The most widely-used error diffusion algorithm is the Floyd-Steinberg algorithm. This function provides an optimized implementation of this algorithm.

3.4.5 Median Filtering Function

- IMG_median_3x3_8
- IMG_median_3x3_16s
- IMG_median_3x3_16

Median filtering is used in image restoration, to minimize the effects of impulsive noise in imagery. Applications can cover almost any area where impulsive noise may be a problem, including security/defense, machine vision, and video compression systems. Optimized implementation of median filter for 3x3 pixel neighborhood is provided in the above routines.

3.4.6   Pixel Expand Functions

- IMG_pix_expand
- IMG_pix_sat

The routines IMG_pix_expand and IMG_pix_sat, respectively, expand 8-bit pixels to 16-bit quantities by zero extension, and saturate 16-bit signed numbers to 8-bit unsigned numbers. They can be used to prepare input and output data for other routines such as the horizontal and vertical scaling routines.

3.5   Compression/Decompression Functions Overview

This section describes the applicable functions for compression/decompression standards such as JPEG, MPEG video, and H.26x.

3.5.1   Forward and Inverse DCT Functions

The IMGLIB provides forward and inverse DCT (Discrete Cosine Transform) functions:

- IMG_fdct_8x8
- IMG_idct_8x8_12q4

These functions are applicable for a wide range of compression standards such as JPEG Encode/Decode, MPEG Video Encode/Decode, and H.26x Encode/Decode. These compression standards are used in diverse end-applications:

- JPEG is used in printing, photography, security systems, etc.
- MPEG video standards are used in digital TV, DVD players, set-top boxes, video-on-demand systems, video disc applications, multimedia/streaming media applications, etc.
- H.26x standards are used in video telephony and some streaming media applications.

Note that the inverse DCT function performs an IEEE 1180-1990 compliant inverse DCT, including rounding and saturation to signed 9-bit quantities. The forward DCT rounds the output values for improved accuracy. These factors can have significant effect on the final result in terms of picture quality, and are important to consider when implementing DCT-based systems or comparing the performance of different DCT-based implementations.

3.5.2   High Performance Motion Estimation Functions

The following functions are provided to enable high performance motion estimation algorithms that are used in applications such as MPEG Video Encode or H.26x Encode.

- IMG_mad_8x8
- IMG_mad_16x16
- IMG_sad_8x8
- IMG_sad_16x16

Video encoding is useful in video-on-demand systems, streaming media systems, video telephony, etc. Motion estimation is typically one of the most computation-intensive operations in video encoding systems; the provided functions enable high performance, which can significantly improve such systems.

3.5.3   MPEG-2 Variable Length Decoding Functions

- IMG_mpeg2_vld_intra
- IMG_mpeg2_vld_inter

The MPEG-2 variable length decoding functions provide a highly integrated and efficient solution for performing variable length decoding, run-length expansion, inverse scan, dequantization, saturation and mismatch control of MPEG-2 coded intra and non-intra macroblocks. The performance of any MPEG-2 video decoder system relies heavily on the efficient implementation of these decoding steps.

3.5.4 Quantization Function

- IMG_quantize

Quantization is an integral step in many image/video compression systems, including those based on widely used variations of DCT-based compression such as JPEG, MPEG, and H.26x. The routine IMG_quantize function can be used in such systems to perform the quantization step.

3.5.5 Wavelet Processing Functions

- IMG_wave_horz
- IMG_wave_vert

Wavelet processing is used in emerging standards such as JPEG2000 and MPEG-4, where it is typically used to provide highly efficient still picture compression. Various proprietary image compression systems are also wavelets-based. This release includes the utilities IMG_wave_horz and IMG_wave_vert for computing horizontal and vertical wavelet transforms. Together, they can compute 2-D wavelet transforms for image data. The routines are flexible enough, within documented constraints, to accommodate a wide range of specific wavelets and image dimensions.

4 IMGLIB Function Tables

This section provides tables containing all IMGLIB functions, a brief description of each, and a page reference for more detailed information.

Table 2. IMGLIB2 Image Analysis Functions

Function	Description	Page
void IMG_boundary_8 (unsigned char *in_data, int rows, int cols, int *out_coord, int *out_gray)	Boundary Structural Operator	Section 5.1
void IMG_boundary_16s(const short *restrict i_data, int rows, int cols, unsigned int *restrict o_coord, short *restrict o_grey)	Boundary Structural Operator for 16-bit input	Section 5.2
void IMG_clipping_16s(const short *restrict x, short rows, short cols, short *restrict r, short THRES_MAX, short THRES_MIN)	Image Clipping Operator for 16-bit input	Section 5.3
void IMG_yc_demux_be16_8(int n, unsigned char *yc, short *y, short *cr, short *cb)	YCbCr Demultiplexing (big endian source)	Section 5.29
void IMG_yc_demux_le16_8(int n, unsigned char *yc, short *y, short *cr, short *cb)	YCbCr Demultiplexing (little endian source)	Section 5.30
void IMG_dilate_bin (unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Dilation	Section 5.4
void IMG_erode_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Erosion	Section 5.5
void IMG_errdif_bin_8(unsigned char errdif_data[ ], int cols, int rows, short err_buf[ ], unsigned char thresh)	Error Diffusion, Binary Output	Section 5.6
void IMG_errdif_bin_16(unsigned short *restrict errdif_data, int cols, int rows, short *restrict err_buf, unsigned short thresh)	Error Diffusion, binary output	Section 5.7
void IMG_histogram_8(unsigned char *in_data, int n, int accumulate, unsigned short *t_hist, unsigned short *hist)	Histogram Computation	Section 5.8
void IMG_histogram_16(unsigned short *restrict in, short *restrict hist, short *restrict t_hist, int n, int accumulate, int img_bits)	Histogram Computation for 16-bit input	Section 5.9
void IMG_median_3x3_8(unsigned char *in_data, int cols, unsigned char *out_data)	3x3 Median Filter	Section 5.10
void IMG_perimeter_8(unsigned char *in_data, int cols, unsigned char *out_data)	Perimeter Structural Operator	Section 5.11
int IMG_perimeter_16 (const unsigned short *restrict in, int cols, unsigned short *restrict out)	Perimeter Structural Operator for 16-bit input	Section 5.12

Table 2. IMGLIB2 Image Analysis Functions (continued)

Function	Description	Page
void IMG_pix_expand(int n, unsigned char *in_data, short *out_data)	Pixel Expand	Section 5.13
void IMG_pix_sat(int n, short *in_data, unsigned char *out_data)	Pixel Saturation	Section 5.14
void IMG_sobel_3x3_8(const unsigned char *in_data, unsigned char *out_data, short cols, short rows)	Sobel Edge Detection	Section 5.15
void IMG_sobel_3x3_16s (constant short *restrict in, short *restrict out, short cols, short rows)	3x3 Sobel Edge Detection for 16-bit input	Section 5.16
void IMG_sobel_3x3_16 (constant unsigned short *restrict in, unsigned short *restrict out, short cols, short rows)	3x3 Sobel Edge Detection for 16-bit unsigned input	Section 5.17
void IMG_sobel_5x5_16s (const short *restrict in, short *restrict out, short cols, short rows)	5x5 Sobel Edge Detection for 16-bit input	Section 5.18
void IMG_sobel_7x7_16s (const short *restrict in, short *restrict out, short cols, short rows)	7x7 Sobel Edge Detection for 16-bit input	Section 5.19
void IMG_thr_gt2max_8(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding - Clamp to 255	Section 5.20
void IMG_thr_gt2max_16(const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)	Thresholding – Clamp to 255	Section 5.21
void IMG_thr_gt2thr_8(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding - Clip above threshold	Section 5.22
void IMG_thr_gt2thr_16(const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)	Thresholding – Clip above threshold	Section 5.23
void IMG_thr_le2min_8 (unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding - Clamp to zero	Section 5.24
void IMG_thr_le2min_16(const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)	Thresholding – Clamp to zero	Section 5.25
void IMG_thr_le2thr_8 (unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding - Clip above threshold	Section 5.26
void IMG_thr_le2thr_16(const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)	Thresholding – Clip above threshold	Section 5.27
void IMG_ycbcr422p_rgb565(short coeff[5], unsigned char *y_data, unsigned char *cb_data, unsigned char *cr_data, unsigned short *rgb_data, unsigned num_pixels)	Planarized YCbCr 4:2:2/4:2:0 to RGB 5:6:5 color space conversion	Section 5.31

Table 3. IMGLIB2 Picture Filtering Functions

Function	Description	Page
void IMG_conv_3x3_i8_c8s(unsigned char *in_data, unsigned char *out_data, int cols, char *mask, int shift)	3x3 Convolution	Section 6.1
void IMG_conv_3x3_i16s_c16s(const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	3x3 convolution for 16-bit inputs	Section 6.2
void IMG_conv_3x3_i16_c16s(const unsigned short *restrict imgin_ptr, unsigned short *restrict imgout_ptr, short width, const short *restrict mask_ptr, short shift)	3x3 convolution for unsigned 16-bit inputs	Section 6.3
void IMG_conv_5x5_i8_c8s(const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)	5x5 convolution for 8-bit inputs	Section 6.4
void IMG_conv_5x5_i16s_c16s(const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	5x5 convolution for 16-bit inputs	Section 6.5
void IMG_conv_5x5_i8_c16s(const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	5x5 convolution for 8-bit input and 16-bit masks	Section 6.6
void IMG_conv_7x7_i8_c8s(const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)	7x7 convolution for 8-bit inputs	Section 6.7
void IMG_conv_7x7_i16s_c16s(const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	7x7 convolution for 16-bit inputs	Section 6.8
void IMG_conv_7x7_i8_c16s(const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	7x7 convolution for 8-bit input and 16-bit masks	Section 6.9
void IMG_conv_11x11_i8_c8s(const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)	11x11 convolution for 8-bit inputs	Section 6.10
void IMG_conv_11x11_i16s_c16s(const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)	11x11 convolution for 16-bit inputs	Section 6.11
void IMG_corr_3x3_i8_c16s(const unsigned char *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr)	3x3 correlation for 8-bit input and 16-bit masks	Section 6.12
void IMG_corr_3x3_i16s_c16s(const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)	3x3 correlation for 16-bit inputs	Section 6.13
void IMG_corr_3x3_i8_c8(const unsigned char *restrict inptr, unsigned char *restrict outptr, int x_dim, const unsigned char *restrict mask_ptr, short shift, short round)	3x3 Correlation for unsigned 8-bit inputs	Section 6.14
void IMG_corr_3x3_i16_c16s(const unsigned short *restrict imgin_ptr, long *restrict imgout_ptr, const short *restrict mask_ptr, short pitch, short width	3x3 correlation for unsigned 16-bit inputs	Section 6.15
void IMG_corr_5x5_i16s_c16s(const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)	5x5 correlation for 16-bit inputs	Section 6.16
void IMG_corr_11x11_i16s_c16s(const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)	11x11 correlation for 16-bit inputs	Section 6.17
void IMG_corr_11x11_i8_c16s(const unsigned char *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr)	11x11 correlation for 8-bit input and 16-bit masks	Section 6.18
void IMG_corr_gen_i16s_c16s_(short *in_data, short *h, short *out_data, int m, int cols)	Generalized Correlation	Section 6.19
void IMG_corr_gen_iq(const int *restrict x, const short *restrict h, int *restrict y, int m, int x_dim, int x_qpt, int h_qpt, int y_qpt)	Generalized Correlation with q-point math	Section 6.20

Table 3. IMGLIB2 Picture Filtering Functions (continued)

Function	Description	Page
void IMG_median_3x3_16s (const short *restrict i_data, int n, short *restrict o_data)	3x3 Median Filtering for 16-bit input	Section 6.21
void IMG_median_3x3_16 (const short *restrict i_data, int n, short *restrict o_data)	3x3 Median Filtering for unsigned 16-bit input	Section 6.22
void IMG_yc_demux_be16_16(int n, const unsigned short *yc, short *restrict y, short *restrict cr, short *restrict cb)	YCbCr Demultiplexing (big endian source)	Section 6.23
void IMG_yc_demux_le16_16(int n, const unsigned short *yc, short *restrict y, short *restrict cr, short *restrict cb)	YCbCr Demultiplexing (little endian source)	Section 6.24

Table 4. Compression/Decompression Functions

Function	Description	
void IMG_fdct_8x8 (short *fdct_data, unsigned num_fdcts)	Forward Discrete Cosine Transform (FDCT)	Section 7.1
void IMG_idct_8x8_12q4 (short *idct_data, unsigned num_idcts)	Inverse Discrete Cosine Transform (IDCT)	Section 7.2
void IMG_mad_8x8 (unsigned char *ref_data, unsigned char *src_data, int pitch, int sx, int sy, unsigned int *match)	8x8 Minimum Absolute Difference	Section 7.3
void IMG_mad_16x16 (unsigned char *ref_data, unsigned char *src_data, int pitch, int sx, int sy, unsigned int *match)	16x16 Minimum Absolute Difference	Section 7.4
void IMG_mpeg2_vld_intra (short *Wptr, short *outi, unsigned int *Mpeg2v, int dc_pred[3])	MPEG-2 Variable Length Decoding of Intra MBs	Section 7.5
void IMG_mpeg2_vld_inter (short *Wptr, short *outi, unsigned int *Mpeg2v)	void IMG_mpeg2_vld_inter(short *Wptr, short *outi, unsigned int *Mpeg2v)	Section 7.6
void IMG_quantize (short *data, int num_blks, int blk_sz, const short *recip_tbl, int q_pt)	Matrix Quantization with Rounding	Section 7.7
unsigned IMG_sad_8x8 (unsigned char *srclmg, unsigned char *reflmg, int pitch)	Sum of Absolute Differences on Single 8x8 block	Section 7.8
unsigned IMG_sad_16x16 (unsigned char *srclmg, unsigned char *reflmg, int pitch)	Sum of Absolute Differences on Single 16x16 block	Section 7.9
void IMG_wave_horz (short *in_data, short *qmf, short *mqmf, short *out_data, int cols)	Horizontal Wavelet Transform	Section 7.10
void IMG_wave_vert (short *in_data[ ], short *qmf,short *mqmf,short *out_ldata,short *out_hdata,int cols)	Vertical Wavelet Transform	Section 7.11

5 IMGLIB Image Analysis Functions

5.1 IMG_boundary_8

IMG_boundary_8	***Boundary Structural Operator***

Syntax	void **IMG_boundary_8**(const unsigned char * restrict in_data, int rows, int cols, int * restrict out_coord, int * restrict out_gray)

Arguments

in_data[ ]	Input image of size rows * cols. Must be word aligned.
rows	Number of input rows.
cols	Number of input columns. Must be multiple of 4.
out_coord[ ]	Output array of packed coordinates. Must be word aligned.
out_gray[ ]	Output array of corresponding gray levels. Must be word aligned.

Description This routine scans an image for non-zero pixels. The locations of those pixels are stored to the array out_coord[ ] as packed Y/X pairs, with Y in the upper half, and X in the lower half. The gray levels of those pixels are stored in the out_gray[ ] array.

Algorithm Behavioral C code for the routine is provided below:

```
void IMG_boundary_8
(
    const unsigned char in_data,
    int rows, int cols,
    int out_coord,
    int out_gray
)
{
    int x, y, p;

    for (y = 0; y < rows; y++)
        for (x = 0; x < cols; x++)
            if ((p = in_data[x + y*cols] != 0)
            {
                *out_coord++ = ((y & 0xFFFF) << 16)
| (x & 0xFFFF);
                *out_gray++  = p;
            }
}
```

Special Requirements

- Array in_data[ ] must be word aligned.
- cols must be a multiple of 4.
- At least one row is being processed.
- Output buffers out_coord and out_gray should start in different banks and must be word aligned.
- No more than 32764 rows or 32764 columns are being processed.

Notes

- Bank Conflicts: No bank conflicts occur as long as out_coord and out_gray start in different banks. If they start in the same bank, every access to each array causes a bank conflict.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant but not interruptible.
- Outer and inner loops are collapsed together.
- Inner loop is unrolled to process four pixels per iteration.

5.2 *IMG_boundary_16s*

IMG_boundary_16s *Boundary structural operator for 16-bit input*

Syntax   void **IMG_boundary_16s**(const short *restrict i_data, int rows, int cols, unsigned int *restrict o_coord, short *restrict o_grey)

Arguments

i_data[ ]	Input image of size rows x cols
rows	Number of rows in input image
cols	Number of columns in input image
o_coor[ ]	Output array of packed coordinate
o_grey[ ]	Output array of corresponding grey levels

Description   This function scans an image for non-zero pixels. The locations of these pixels are stored to the array o_coord[ ] as packed Y-X co-ordinate pairs, with Y in the upper half, and X in the lower half. The grey levels of those pixels are stored in the o_grey[ ] array.

Algorithm   This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_boundary_16s
(
const  short  *restrict i_data,
          int                 rows,
          int                 cols,
     unsigned int *restrict o_coord,
          short *restrict  o_grey
)
{
int    x,         y,         p;
for (y = 0; y < rows; y++)
        for (x = 0; x < cols; x++)
            if ((p = i_data[x + y * cols]) != 0)
            {
                *o_coord++ = ((y) << 16) | (x);
*o_grey++ = p;
            }
}
```

Special Requirements

- rows should be minimum of 1 and maximum of 32767
- cols should be a multiple of 4,minimum of 4, and maximum of 32764
- Input array must be double-word aligned
- No alignment restrictions on output arrays
- Input and Output arrays should not overlap

Implementation Notes

- Outer and Inner loops are merged and four pixels are calculated per iteration
- Endian: The code is LITTLE ENDIAN.

Compatibility   Compatible for both C64x+ and C64x.

5.3   IMG_clipping_16s

IMG_clipping_16s	***Image Cipping Operator for 16-bit Input***

Syntax

void **IMG_clipping_16s**(const short *restrict x, short rows, short cols, short *restrict r, short THRES_MAX, short THRES_MIN)

Arguments

X	Input image of size rows x cols
rows	Number of rows in input image
cols	Number of columns in input image
r	Output image of size rows x cols
THRES_MAX	Maximum threshold level
THRES_MIN	Minimum threshold level

Description

The function truncates elements of a matrix to the maximum and minimum values defined by the user. Each element is 16-bit signed and the size of the matrix is user determined dimensions. The output matrix has the same size as that of the input matrix and each value will be truncated to minimum or maximum value defined by user based on whether it is less than the minimum value (THRES_MIN)or greater than the maximum value (THRES_MAX) respectively.

Algorithm

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_clipping_16s
(
    const short *restrict x,      /* Input Matrix Pointer    */
    short             rows,    /* Height of input matrix  */
    short             cols,    /* Width of input matrix   */
    short        *restrict r,    /* Output Matrix Pointer   */
    short          THRES_MAX,    /* Maximum Threshold Value */
    short          THRES_MIN     /* Minimum Threshold Value */
)
{
    int     i;
    for (i = 0; i < (rows * cols); i++)
    {
      r[i]      =   (x[i] > THRES_MAX) ? THRES_MAX : x[i];
      r[i]      =   (r[i] < THRES_MIN) ? THRES_MIN : r[i];
    }
}
```

Special Requirements

- (rows * cols) >= 8 and should be a multiples of 8.
- THRES_MAX >= THRES_MIN.
- Input and output arrays must be double word aligned.
- Input and Output arrays should not overlap.

Implementation Notes

- Outer and Inner loops are merged and eight pixels are calculated per iteration.
- Endian: The code is LITTLE ENDIAN.

Compatibility

- Compatible for both C64x+ and C64x.

5.4   *IMG_dilate_bin*

IMG_dilate_bin	***3x3 Binary Dilation***

Syntax

void **IMG_dilate_bin**(const unsigned char * restrict in_data, unsigned char * restrict out_data, const char * restrict mask, int cols)

Arguments

in_data[ ]	Binary input image (8 pixels per byte).
out_data[ ]	Filtered binary output image.
mask[3][3]	3x3 filter mask.
cols	Number of columns / 8. cols must be a multiple of 8.

Description

This routine dilate_bin() implements 3x3 binary dilation. The input image consists of binary valued pixels (0s or 1s). The dilation operator generates output pixels by ORing the pixels under the input mask together to generate the output pixel. The input mask specifies whether one or more pixels from the input are to be ignored.

Algorithm

The routine computes output for a target pixel as follows:

```
result = 0;
if (mask[0][0] != DONT_CARE) result |= input[y + 0][x + 0];
if (mask[0][1] != DONT_CARE) result |= input[y + 1][x + 1];
if (mask[0][2] != DONT_CARE) result |= input[y + 2][x + 2];
if (mask[1][0] != DONT_CARE) result |= input[y + 0][x + 0];
if (mask[1][1] != DONT_CARE) result |= input[y + 1][x + 1];
if (mask[1][2] != DONT_CARE) result |= input[y + 2][x + 2];
if (mask[2][0] != DONT_CARE) result |= input[y + 0][x + 0];
if (mask[2][1] != DONT_CARE) result |= input[y + 1][x + 1];
if (mask[2][2] != DONT_CARE) result |= input[y + 2][x + 2];
output[y][x] = result;
```

For this code, *DONT_CARE* is specified by a negative value in the input mask. Non-negative values in the mask cause the corresponding pixel to be included in the dilation operation.

Special Requirements

- Pixels are organized within each byte such that the pixel with the smallest index is in the LSB position, and the pixel with the largest index is in the MSB position. (That is, the code assumes a LITTLE ENDIAN bit ordering.)
- Negative values in the mask specify *DONT_CARE*, and non-negative values specify that pixels are included in the dilation operation.
- The input image needs to have a multiple of 64 pixels (bits) per row. Therefore, cols must be a multiple of 8.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- The 3×3 dilation mask is applied to 32 output pixels simultaneously. This is done with 32-bit-wide bit-wise operators in the register file. To do this, the code reads in a 34-bit-wide input window, and 40-bit operations are used to manipulate the pixels initially. Because the code reads a 34-bit context for each 32-bits of output, the input needs to be one byte longer than the output to make the rightmost two pixels well-defined.

5.5 *IMG_erode_bin*

IMG_erode_bin	***3x3 Binary Erosion***

Syntax

void **IMG_erode_bin**(const unsigned char * restrict in_data, unsigned char * restrict out_data, const char * restrict mask, int cols)

Arguments

in_data[ ]	Binary input image (8 pixels per byte).
out_data[ ]	Filtered binary output image.
mask[3][3]	3x3 filter mask.
cols	Number of columns / 8. cols must be a multiple of 8.

Description

This routine implements 3×3 binary erosion. The input image consists of binary valued pixels (0s or 1s). The erosion operator generates output pixels by ANDing the pixels under the input mask together to generate the output pixel. The input mask specifies whether one or more pixels from the input are to be ignored.

Algorithm

The routine computes output for a target pixel as follows:

```
result = 1;
if (mask[0][0] != DONT_CARE) result &= input[y + 0][x + 0];
if (mask[0][1] != DONT_CARE) result &= input[y + 1][x + 1];
if (mask[0][2] != DONT_CARE) result &= input[y + 2][x + 2];
if (mask[1][0] != DONT_CARE) result &= input[y + 0][x + 0];
if (mask[1][1] != DONT_CARE) result &= input[y + 1][x + 1];
if (mask[1][2] != DONT_CARE) result &= input[y + 2][x + 2];
if (mask[2][0] != DONT_CARE) result &= input[y + 0][x + 0];
if (mask[2][1] != DONT_CARE) result &= input[y + 1][x + 1];
if (mask[2][2] != DONT_CARE) result &= input[y + 2][x + 2];
output[y][x] = result;
```

For this code, *DONT_CARE* is specified by a negative value in the input mask. Non-negative values in the mask cause the corresponding pixel to be included in the erosion operation.

Special Requirements

- Pixels are organized within each byte such that the pixel with the smallest index is in the LSB position, and the pixel with the largest index is in the MSB position. (That is, the code assumes a LITTLE ENDIAN bit ordering.)
- Negative values in the mask specify *DONT_CARE*, and non-negative values specify that pixels are included in the erosion operation.
- The input image needs to have a multiple of 64 pixels (bits) per row. Therefore, cols must be a multiple of 8.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- The 3×3 erosion mask is applied to 32 output pixels simultaneously. This is done with 32-bit-wide bit-wise operators in the register file. To do this, the code reads in a 34-bit-wide input window, and 40-bit operations are used to manipulate the pixels initially. Because the code reads a 34-bit context for each 32-bits of output, the input needs to be one byte longer than the output to make the rightmost two pixels well-defined.

5.6 IMG_errdif_bin_8

IMG_errdif_bin_8	*Error Diffusion, Binary Output*

Syntax

void **IMG_errdif_bin_8**(unsigned char * restrict errdif_data, int cols, int rows, short * restrict err_buf, unsigned char thresh)

Arguments

errdif_data[ ]	Input/output image data.
cols	Number of columns in the image. Must be ≥ 2.
rows	Number of rows in the image.
err_buf[ ]	Buffer of size cols+1, where one row of error values is saved. Must be initialized to zeros prior to first call.
thresh	Threshold value in the range [0, 255].

Description

This routine implements the Floyd-Steinberg error diffusion filter with binary output.

Pixels are processed from left-to-right, top-to-bottom in an image. Each pixel is compared against a user-defined threshold. Pixels that are larger than the threshold are set to 255, and pixels that are smaller or equal to the threshold are set to 0. The error value for the pixel (e.g., the difference between the thresholded pixel and its original gray level) is propagated to the neighboring pixels using the Floyd-Steinberg filter (see below). This error propagation diffuses the error over a larger area, hence the term error diffusion.

The Floyd-Steinberg filter propagates fractions of the error value at pixel location X to four of its neighboring pixels. The fractional values used are:

	X	7/16
3/16	5/16	1/16

Algorithm

When a given pixel at location (x, y) is processed, it has already received error terms from four neighboring pixels. Three of these pixels are on the previous row at locations (x-1, y-1), (x, y-1), and (x+1, y-1), and one is immediately to the left of the current pixel at (x-1, y). To reduce the loop-carry path that results from propagating these errors, this implementation uses an error buffer to accumulate errors that are being propagated from the previous row. The result is an inverted filter, as shown below:

1/16	5/16	3/16
7/16	Y	

where Y is the current pixel location and the numerical values represent fractional contributions of the error values from the locations indicated that are diffused into the pixel at location Y location.

This modified operation requires the first row of pixels to be processed separately, since this row has no error inputs from the previous row. The previous row's error contributions in this case are essentially zero. One way to achieve this is with a special loop that avoids the extra calculation involved with injecting the previous row's errors. Another is to pre-zero the error buffer before processing the first row. This function supports the latter approach.

Behavioral C code for the routine is provided below:

```
void IMG_errdif_bin
(
    unsigned char *errdif_data,    /* Input/Output image ptr    */
    int           cols,            /* Number of columns (Width) */
    int           rows,            /* Number of rows    (Height) */
    short         err_buf,         /* row-to-row error buffer.   */
```

```
                    unsigned char thresh            /* Threshold from [0x00, 0xFF] */
)
{
    int   x, i, y;          /* Loop counters                          */
    int   F;                /* Current pixel value at [x,y]           */
    int   errA;             /* Error value at [x-1, y-1]              */
    int   errB;             /* Error value at [  x, y-1]              */
    int   errC;             /* Error value at [x+1, y-1]              */
    int   errE;             /* Error value at [x-1,   y]              */
    int   errF;             /* Error value at [  x,   y]              */

    /* ---------------------------------------------------------- */
    /*  Step through rows of pixels.                              */
    /* ---------------------------------------------------------- */
    for (y = 0, i = 0; y < rows; y++)
    {
        /* ---------------------------------------------------- */
        /*  Start off with our initial errors set to zero at    */
        /*  the start of the line since we do not have any      */
        /*  pixels to the left of the row.  These error terms   */
        /*  are maintained within the inner loop.               */
        /* ---------------------------------------------------- */
        errA = 0; errE = 0;
        errB = err_buf[0];

        /* ---------------------------------------------------- */
        /*  Step through pixels in each row.                    */
        /* ---------------------------------------------------- */
        for (x = 0; x < cols; x++, i++)
        {
            /* -------------------------------------------------- */
            /*  Load the error being propagated from pixel 'C'    */
            /*  from our error buffer.  This was calculated       */
            /*  during the previous line.                        */
            /* -------------------------------------------------- */
            errC = err_buf[x+1];

            /* -------------------------------------------------- */
            /*  Load our pixel value to quantize.                */
            /* -------------------------------------------------- */
            F = errdif_data[i];

            /* -------------------------------------------------- */
            /*  Calculate our resulting pixel.  If we assume     */
            /*  that this pixel will be set to zero, this also    */
            /*  doubles as our error term.                       */
            /* -------------------------------------------------- */
            errF = F + ((errE*7 + errA + errB*5 + errC*3) >> 4);

            /* -------------------------------------------------- */
            /*  Set pixels that are larger than the threshold to  */
            /*  255, and pixels that are smaller than the        */
            /*  threshold to 0.                                  */
            /* -------------------------------------------------- */
            if (errF > thresh)  errdif_data[i] = 0xFF;
            else                errdif_data[i] = 0;

            /* -------------------------------------------------- */
            /*  If the pixel was larger than the threshold, then  */
            /*  we need subtract 255 from our error.  In any     */
            /*  case, store the error to the error buffer.       */
            /* -------------------------------------------------- */
            if (errF > thresh)  err_buf[x] = errF = errF - 0xFF;
            else                err_buf[x] = errF;

            /* -------------------------------------------------- */
            /*  Propagate error terms for the next pixel.        */
            /* -------------------------------------------------- */
            errE = errF;
            errA = errB;
            errB = errC;
        }
    }
}
```

Special Requirements

- The number of columns must be at least 2.
- err_buf[ ] must be initialized to zeros for the first call and the returned err_buf [ ] should be provided for the next call.
- errdif_data[ ] is used for both input and output.
- The size of err_buf[ ] should be cols+1.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is ENDIAN NEUTRAL.
- Interruptibility: This function is interruptible. Maximum interrupt delay is 4*cols + 9 cycles.
- The outer loop has been interleaved with the prolog and epilog of the inner loop.
- Constants 7, 5, 3, 1 for filter-tap multiplications are shifted left 12 to avoid SHR 4 operation in the critical path.
- The inner loop is software-pipelined.

5.7 IMG_errdif_bin_16

IMG_errdif_bin_16 *Floyd-Steinberg Error Dffusion for 16-bit data*

Syntax void **IMG_errdif_bin_16**(unsigned short *restrict errdif_data, int cols, int rows, short *restrict err_buf, unsigned short thresh)

Arguments

errdif_data	Input/output image ptr
cols	Number of columns (width)
rows	Number of rows (height)
err_buf[cols+1]	Buffer where one row of errors is to be saved
thresh	Threshold in the range [0x00, 0xFF]

Description The code implements the Binary Floyd-Steinberg error diffusion filter. The following filter kernel is used:

```
            +---+
          P | 7 |
    +---+---+---+
    | 3 | 5 | 1 |
    +---+---+---+
```

Pixels are processed from left-to-right, top-to-bottom. Each pixel is compared against a user-defined threshold. Pixels that are larger than the threshold are set to 0xFFFF, and pixels that are smaller or equal to the threshold are set to 0. The error value for the pixel (e.g., the difference between the thresholded pixel and its original grey level) is propagated to the neighboring pixels according to the filter above. This error propagation diffuses the error over a larger area; hence, the term "error diffusion."

Algorithm The optimized code is based on the following C model:

```c
void IMG_errdif_bin_16_c
(
    unsigned short *restrict errdif_data, /* Input/Output image ptr    */
    int                      cols,        /* Number of columns (width) */
    int                      rows,        /* Number of rows   (height) */
    short          *restrict err_buf,     /* row-to-row error buffer.  */
    unsigned short           thresh       /* Threshold from [0x00, 0xFF] */
)
{
    int   x, i, y;              /* Loop counters                   */
    int   F;                    /* Current pixel value at [x,y]    */
    int   errA;                 /* Error value at [x-1, y-1]       */
    int   errB;                 /* Error value at [  x, y-1]       */
    int   errC;                 /* Error value at [x+1, y-1]       */
    int   errE;                 /* Error value at [x-1,   y]       */
    int   errF;                 /* Error value at [  x,   y]       */

    /* ------------------------------------------------------- */
    /*  Step through rows of pixels.                           */
    /* ------------------------------------------------------- */
    for (y = 0, i = 0; y < rows; y++)
    {
        /* --------------------------------------------------- */
        /*  Start off with the initial errors set to zero at the */
        /*  start of the line since there are no pixels to the */
        /*  left of the row.  These error terms are maintained */
        /*  within the inner loop.                             */
        /* --------------------------------------------------- */
        errA = 0; errE = 0;
        errB = err_buf[0];


        /* --------------------------------------------------- */
        /*  Step through pixels in each row.                   */
        /* --------------------------------------------------- */
```

```
for (x = 0; x < cols; x++, i++)
{
    /* ------------------------------------------------ */
    /*  Load the error being propagated from pixel 'C'  */
    /*  from the error buffer.  This was calculated     */
    /*  during the previous line.                       */
    /* ------------------------------------------------ */
    errC = err_buf[x+1];

    /* ------------------------------------------------ */
    /*  Load the pixel value to quantize.               */
    /* ------------------------------------------------ */
    F = errdif_data[i];

    /* ------------------------------------------------ */
    /*  Calculate the resulting pixel.  If we assume    */
    /*  that this pixel will be set to zero, this also  */
    /*  doubles as the error term.                      */
    /* ------------------------------------------------ */
    errF = F + ((errE*7 + errA + errB*5 + errC*3) >> 4);

    /* ------------------------------------------------ */
    /*  Set pixels that are larger than the threshold to */
    /*  255, and pixels that are smaller than the       */
    /*  threshold to 0.                                 */
    /* ------------------------------------------------ */
    if (errF > thresh)  errdif_data[i] = 0xFFFF;
    else                errdif_data[i] = 0;

    /* ------------------------------------------------ */
    /*  If the pixel was larger than the threshold, then */
    /*  subtract 255 from the error.  In any case, store */
    /*  the error to the error buffer.                  */
    /* ------------------------------------------------ */
    if (errF > thresh)  err_buf[x] = errF = errF - 0xFFFF;
    else                err_buf[x] = errF;

    /* ------------------------------------------------ */
    /*  Propagate error terms for the next pixel.       */
    /* ------------------------------------------------ */
    errE = errF;
    errA = errB;
    errB = errC;
}
    }
}
```

The processing of the filter itself is inverted so that the errors from previous pixels *propagate into* a given pixel at the time the pixel is processed, rather than *accumulate into* a pixel as its neighbors are processed. This allows us to maintain the image at 16-bit and reduces the number of accesses to the image array. The inverted filter kernel performs identically to the kernel's original form. In this form, the weights specify the weighting assigned to the errors coming from the neighboring pixels.

```
+---+---+---+
| 1 | 5 | 3 |
+---+---+---+
| 7 | P
+---+
```

Special Requirements

- Input and output buffers do not alias. 'cols should be even
- err_buf[ ] must be initialized to zeros for the first call and the returned err_buf should be provided for the subsequent calls

Memory Notes

- This kernel places no restrictions on the alignment of its input.
- No bank conflicts occur.
- The code is LITTLE ENDIAN.

Compatibility   This code is compatible for both C64x and C64x+.

5.8 IMG_histogram_8

IMG_histogram_8	*Histogram Computation*

Syntax

void **IMG_histogram_8** (const unsigned char * restrict in_data, int n, short accumulate, unsigned short * restrict t_hist, unsigned short * restrict hist)

Arguments

in_data[n]	Input image. Must be word aligned.
n	Number of pixels in input image. Must be multiple of 8.
accumulate	1: Add to existing histogram in hist[ ] -1: Subtract from existing histogram in hist[ ]
t_hist[1024]	Array of temporary histogram bins. Must be initialized to zero.
hist[256]	Array of updated histogram bins.

Description

This routine computes the histogram of the array in_data[ ] which contains n 8-bit elements. It returns a histogram in the array hist[ ] with 256 bins at 16-bit precision. It can either add or subtract to an existing histogram, using the accumulate control. It requires temporary storage for four temporary histograms, t_hist[ ], which are later summed together.

Algorithm

Behavioral C code for the function is provided below:

```
void IMG_histogram (unsigned char *in_data, int n, int accumulate, unsigned short
*t_hist,
unsigned short * hist)
{
        int pixel, j;
        for (j = 0; j < n; j++)
        {
                pixel = (int) in_data[j];
                hist[pixel] += accumulate;
        }
}
```

Special Requirements

- The temporary array of data, t_hist[ ], must be initialized to zero.
- The input array of data, in_data[ ], must be word-aligned.
- n must be a multiple of 8.
- The maximum number of pixels that can be profiled in each bin is 65535 in the main histogram.

Notes

- This code operates on four interleaved histogram bins. The loop is divided into two halves. The even half operates on even words full of pixels and the odd half operates on odd words. Each half processes 4 pixels at a time, and both halves operate on the same four sets of histogram bins. This introduces a memory dependency on the histogram bins which ordinarily would degrade performance. To break the memory dependencies, the two halves forward their results to each other via the register file, bypassing memory. Exact memory access ordering obviates the need to predicate stores.
- The algorithm is ordered as follows:
  1. Load from histogram for even half.
  2. Store odd_bin to histogram for odd half (previous iteration).
  3. If data_even = previous data_odd, increment even_bin by 2, else increment even_bin by 1, forward to odd.
  4. Load from histogram for odd half (current iteration).

5. Store even_bin to histogram for even half.
6. If data_odd = previous data_even increment odd_bin by 2, else increment odd_bin by 1, forward to even.
7. Go to 1.

- With this particular ordering, forwarding is necessary between even/odd halves when pixels in adjacent halves need to be placed in the same bin. The store is never predicated and occurs speculatively as it will be overwritten by the next value containing the extra forwarded value.
- The four histograms are interleaved with each bin spaced four half-words apart and each histogram starting in a different memory bank. This allows the four histogram accesses to proceed in any order without worrying about bank conflicts. The diagram below illustrates this (addresses are half-word offsets):

0	1	2	3	4	5	
hst0	hst1	hst2	hst3	hst0	hst1	
bin0	bin0	bin0	bin0	bin1	bin1	

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant, but not interruptible.

5.9 *IMG_histogram_16*

IMG_histogram_16 *Histogram Computation for 16-bit Input*

Syntax	void **IMG_histogram_16**(unsigned short *restrict in, short *restrict hist, short *restrict t_hist, int n, int accumulate, int img_bits)

Arguments

in	Input image of size n
hist	Array of updated histogram bins
t_hist	Array of temporary histogram bins
n	Nunber of pixels in input image
accumulate	1: add to existing histogram in hist[ ] -1: subtract from existing histogram in hist[ ]
img_bits	Number of valid data bits in a pixel

Description   This code takes a histogram of an array (of type short) with n number of pixels, with img_bits being the number of valid data bits in a pixel. It returns the histogram of corresponding number of bins at img_bits bits precision. It can either add or subtract to an existing histogram, using the accumulate control. It requires some temporary storage for four temporary histograms, which are later summed together. The length of the hist and the t_hist arrays depends on the value of img_bits. The length of the hist array is 2(img_bits) and that of t_hist is 4 * 2(img_bits) as there are no pixel values greater than or equal to 2(img_bits) in the given image.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements.

```
void IMG_histogram_16
(
    unsigned short *restrict        in,
            short *restrict      hist,
            short *restrict    t_hist,
             int                    n,
             int          accumulate,
             int             img_bits
)
{

    int    p0,    p1,    p2;
    int    p3,    i,    length;


    /* ------------------------------------------------- */
    /* this loop is unrolled four times, producing four  */
    /* interleaved histograms into a temporary buffer.   */
    /* ------------------------------------------------- */
    for (i = 0; i < n; i += 4)
    {
        p0 = in[i       ] * 4;
        p1 = in[i + 1] * 4 + 1;
        p2 = in[i + 2] * 4 + 2;
        p3 = in[i + 3] * 4 + 3;

        t_hist[p0]++;
        t_hist[p1]++;
        t_hist[p2]++;
        t_hist[p3]++;
    }

    /* ------------------------------------------------- */
    /*     Calculate the length of the histogram array   */
    /* ------------------------------------------------- */
    length = 1 << img_bits;
```

```
                    for (i = 0; i < length; i++)
                    {
                        hist[i] += (t_hist[i * 4 + 0] +
                                    t_hist[i * 4 + 1] +
                                    t_hist[i * 4 + 2] +
                                    t_hist[i * 4 + 3] ) * accumulate;
                    }
                }
```

Special Requirements

- n must be a multiple of 8 and greater than or equal to 8.
- The elements of arrays of data, t_hist are initialized to zero.
- in and t_hist arrays must be double-word aligned.
- hist array must be word-aligned
- Input and output arrays do not overlap
- img_bits must be at least 1

Implementation Notes

- This code operates on four interleaved histogram bins. The loop is divided into two halves. The even half operates on even double words full of pixels and the odd half operates on odd double words. Each half processes four pixels at a time, and both halves operate on the same four sets of histogram bins. This introduces a memory dependency on the histogram bins which ordinarily would degrade performance. To break the memory dependencies, the two halves forward their results to each other via the register file, bypassing memory. Exact memory access ordering obviates the need to predicate stores.
- The code is LITTLE ENDIAN.

Compatibility   Compatible for both C64x+ and C64x.

5.10 IMG_median_3x3_8

IMG_median_3x3_8 *3x3 Median Filter*

Syntax	void **IMG_median_3x3_8**(const unsigned char * restrict in_data, int cols, unsigned char * restrict out_data)

Arguments

in_data	Pointer to input image data. No alignment is required.
cols	Number of columns in input (or output). Must be multiple of 4.
out_data	Pointer to output image data. No alignment is required.

Description

This routine performs a 3×3 median filtering algorithm. The gray level at each pixel is replaced by the median of the nine neighborhood values. The function processes three lines of input data pointed to by in_data, where each line is cols' pixels wide, and writes one line of output data to out_data. For the first output pixel, two columns of input data outside the input image are assumed to be all 127.

The median of a set of nine numbers is the middle element, so that half of the elements in the list are larger and half are smaller. A median filter removes the effect of extreme values from data. It is a commonly used operation for reducing impulsive noise in images.

Algorithm

The algorithm processes a 3×3 region as three 3-element columns, incrementing through the columns in the image. Each column of data is first sorted into MAX, MED, and MIN values, resulting in the following arrangement:

I00	I01	I02	MAX
I10	I11	I12	MED
I20	I21	I22	MIN

Where I00 is the MAX of the first column, I10 is the MED of the first column, I20 is the MIN of the first column and so on.

The three MAX values I00, I01, I02 are then compared and their minimum value is retained, call it MIN0.

The three MAX values I00, I01, I02 are then compared and their minimum value is retained, call it MIN0.

The three MIN values I20, I21, I22 are compared and their maximum value is retained, call it MAX2.

The three values MIN0, MED1, MAX2 are then sorted and their median is the median value for the nine original elements.

After this output is produced, a new set of column data is read in, say I03, I13, I23. This data is sorted as a column and processed along with I01, I11, I21, and I02, I12, I22 as explained above. Since these two sets of data are already sorted, they can be re-used as is.

Special Requirements

- cols must be a multiple of 4.
- No alignment is required.

Implementation Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility:The code is fully interruptible.

5.11 IMG_perimeter_8

IMG_perimeter_8	*Perimeter Structural Operator*

Syntax

void **IMG_perimeter_8** (const unsigned char * restrict in_data, int cols, unsigned char * restrict out_data)

Arguments

in_data[ ]	Input image data. Must be double-word aligned.
cols	Number of input columns. Must be multiple of 16.
out_data[ ]	Output boundary image data.

Description

This routine produces the boundary of an object in a binary image. It echoes the boundary pixels with a value of 0xFF and sets the other pixels to 0x00. Detection of the boundary of an object in a binary image is a segmentation problem and is done by examining spatial locality of the neighboring pixels. This is done by using the four connectivity algorithm:

```
        pix_top
 pix_lft     pix_cent     pix_rgt
        pix_bot
```

The output pixel at location pix_cent is echoed as a boundary pixel if pix_cent is non-zero and any one of its four neighbors is zero. The four neighbors are as shown above.

Algorithm

Behavioral C code for the routine is provided below:

```
void IMG_perimeter (unsigned char *in_data, int cols, unsigned char *out_data)
{
        Int                 icols, count = 0;
        unsigned char       pix_lft, pix_rgt, pix_top;
        unsigned char       pix_bot, pix_cent;
    for(icols = 1; icols < (cols-1); icols++ )
        {
                pix_lft = in_data[icols - 1];
                pix_cent = in_data[icols + 0];
                pix_rgt = in_data[icols + 1];
                pix_top = in_data[icols - cols];
                pix_bot = in_data[icols + cols];
                if (((pix_lft==0)||(pix_rgt==0)||(pix_top==0)||(pix_bot==0))
                && (pix_cent > 0))
                {
                        out_data[icols] = pix_cent;
                        count++;
                }
                else
                {
                        out_data[icols] = 0;
                }
        }
        return(count);
}
```

Special Requirements

- Array in_data[ ] must be double-word aligned.
- cols must be a multiple of 16.
- This code expects three input lines each of width cols pixels and produces one output line of width (cols – 1) pixels.

Notes

- Double word wide loads are used to bring in pixels from three consecutive lines.
- The instructions CMPEQ4/CMPGTU4 are used to compare if pixels are greater than or equal to zero. Comparison results are re-used between adjacent comparisons.
- Multiplies replace some of the conditional operations to reduce the bottleneck on the

predication registers as well as on the .L, .S, and .D units.

- XPND4 and BITC4 are used to perform expansion and bit count.
- The loop is unrolled once and computes 16 output pixels per iteration.
- Bank Conflicts: No bank conflicts occur.
- Endian: This code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant but not interruptible.

5.12 IMG_perimeter_16

IMG_perimeter_16	***Perimeter structural operator for 16-bit input***

Syntax   void **IMG_perimeter_16** (const unsigned short*restrict in, int cols, unsigned short *restrict out)

Arguments

in	Pointer to input image 16-bit unsigned
cols	Number of columns in the input image
out	Pointer to output image 16-bit unsigned

Description   This function computes and returns the boundary of an object in a binary image. It echoes the boundary pixels with a value of 0xFFFF and sets the other pixels to 0x0000. Detection of the boundary of an object in a binary image is a segmentation problem and is done by examining spatial locality of the neighboring pixels, using the four connectivity algorithm:
pix_up
pix_lft pix_cent pix_rgt
pix_dn

The output pixel at location pix_cent is echoed as a boundary pixel, if pix_cent is non-zero and any one of its four neighbors (shown above) is zero. Perimeter pixels retain their original grey level in the output. Non-perimeter pixels are set to zero in the output. Pixels on the far left and right edges of a row are defined as *not* being on the perimeter, and are always forced to zero.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
int IMG_perimeter_16
(
    const unsigned short *restrict in,
                      int            cols,
    unsigned short       *restrict out
)
{
    int             i,            count;
    unsigned short  pix_lft,    pix_rgt,    pix_top;
    unsigned short  pix_bot,    pix_mid;
    count = 0;
     for(i = 0; i < cols; i++)
     {
        pix_lft   =   in[i - 1];
        pix_mid   =   in[i     ];
        pix_rgt   =   in[i + 1];
        pix_top   =   in[i - cols];
        pix_bot   =   in[i + cols];

        if (((pix_lft == 0) || (pix_rgt == 0) ||
            (pix_top == 0) || (pix_bot == 0)) && (pix_mid > 0))
        {
            out[i] = pix_mid;
            count++;
        }
        else
        {
            out[i] = 0;
        }
    }
    if (out[cols - 1]) count--;
    if (out[0])        count--;
    out[0] = out[cols - 1] = 0;
    return count;
}
```

Special Requirements

- cols must be a multiple of 8
- Input and output arrays must be double-word aligned
- Input and output arrays should not overlap

Implementation Notes

- Eight output pixels are calculated per iteration
- Each function call calculates one new row of output for three rows of input
- The code is LITTLE ENDIAN.

Compatibility   Compatible for both C64x+ and C64x.

5.13 IMG_pix_expand

IMG_pix_expand	*Pixel Expand*

Syntax

void **IMG_pix_expand**(int n, const unsigned char * restrict in_data, short * restrict out_data)

Arguments

	n	Number of samples to process. Must be multiple of 16.
	in_data	Pointer to input array (unsigned chars). Must be double-word aligned.
	out_data	Pointer to output array (shorts). Must be double-word aligned.

Description

This routine takes an array of unsigned chars (8-bit pixels), and zero-extends them to signed 16-bit values (shorts).

Algorithm

Behavioral C code for the routine is provided below:

```
void IMG_pix_expand (int n, unsigned char *in_data, short *out_data)

{
    int j;
    for (j = 0; j < n; j++)
        out_data[j] = (short) in_data[j];
}
```

Special Requirements

- in_data and out_data must be double-word aligned.
- n must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant, but not interruptible
- The loop is unrolled 16 times, loading bytes with LDDW. It uses UNPKHU4 and UNPKLU4 to unpack the data and store the results with STDW.

5.14 IMG_pix_sat

IMG_pix_sat	*Pixel Saturate*

Syntax

void **IMG_pix_sat**(int n, const short * restrict in_data, unsigned char * restrict out_data)

Arguments

n	Number of samples to process. Must be multiple of 32.
in_data	Pointer to input data (shorts).
out_data	Pointer to output data (unsigned chars).

Description

This routine performs the saturation of 16-bit signed numbers to 8-bit unsigned numbers. If the data is over 255, it is clamped to 255. If it is less than 0, it is clamped to 0.

Algorithm

Behavioral C code for the routine is provided below:

```
void IMG_pix_sat_cn
(
    int          n,
    const short  in_data,
    unsigned char out_data
)
{
    int i, pixel;
    for (i = 0; i < n; i++)
    {
        pixel = in_data[i];
        if (pixel > 0xFF)
        {
            out_data[i] = 0xFF;
        } else if (pixel < 0x00)
        {
            out_data[i] = 0x00;
        } else
        {
            out_data[i] = pixel;
        }
    }
}
```

Special Requirements

- The input size n must be a multiple of 32. The code behaves correctly if n is zero.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- The inner loop has been unrolled to fill a 6 cycle loop. This allows the code to be interruptible.
- The prolog and epilog have been collapsed into the kernel.

5.15 IMG_sobel_3x3_8

IMG_sobel_3x3_8	Sobel Edge Detection

Syntax

void **IMG_sobel_3x3_8**(const unsigned char *in_data, unsigned char *out_data, short cols, short rows)

Arguments

in_data[ ]	Input image of size cols * rows.
out_data[ ]	Output image of size cols * (rows-2).
cols	Number of columns in the input image. Must be multiple of 2.
rows	Number of rows in the input image. cols * (rows-2) must be multiple of 8.

Description

This routine applies horizontal and vertical Sobel edge detection masks to the input image and produces an output image which is two rows shorter than the input image. Within each row of the output, the first and the last pixel will not contain meaningful results.

Algorithm

The Sobel edge-detection masks shown below are applied to the input image separately. The absolute values of the mask results are then added together. If the resulting value is larger than 255, it is clamped to 255. The result is then written to the output image.

Horizontal Mask			Vertical Mask		
-1	-2	-1	-1	0	1
0	0	0	-2	0	2
1	2	1	-1	0	1

This is a C model of the Sobel implementation. This C code is functionally equivalent to the assembly code without restrictions. The assembly code may impose additional restrictions.

```
void IMG_sobel
(
    const unsigned char    *in,   /* Input image data   */
    unsigned char          *out,  /* Output image data  */
    short cols, short rows        /* Image dimensions   */
)
{
    int H, O, V, i;
    int i00, i01, i02;
    int i10,      i12;
    int i20, i21, i22;
    int w = cols;

    /* ------------------------------------------------------------------ */
    /*  Iterate over entire image as a single, continuous raster line.    */
    /* ------------------------------------------------------------------ */
    for (i = 0; i < cols*(rows-2) - 2; i++)
    {
        /* -------------------------------------------------------------- */
        /*  Read in the required 3x3 region from the input.               */
        /* -------------------------------------------------------------- */
        i00=in[i   ]; i01=in[i   +1]; i02=in[i   +2];
        i10=in[i+ w];                 i12=in[i+ w+2];
        i20=in[i+2*w]; i21=in[i+2*w+1]; i22=in[i+2*w+2];

        /* -------------------------------------------------------------- */
        /*  Apply horizontal and vertical filter masks.  The final filter */
        /*  output is the sum of the absolute values of these filters.    */
        /* -------------------------------------------------------------- */

        H = -   i00 - 2*i01 -   i02 +
```

```
                                    +   i20 + 2*i21 +   i22;

                        V = -   i00          +   i02
                            - 2*i10          + 2*i12
                            -   i20          +   i22;

                        O = abs(H) + abs(V);

                        /* ------------------------------------------------------------- */
                        /*  Clamp to 8-bit range.  The output is always positive due to  */
                        /*  the absolute value, so we only need to check for overflow.    */
                        /* ------------------------------------------------------------- */
                        if (O > 255) O = 255;

                        /* ------------------------------------------------------------- */
                        /*  Store it.                                                     */
                        /* ------------------------------------------------------------- */
                        out[i + 1] = O;
                    }
                }
```

Special Requirements

- cols must be a multiple of 2.
- At least eight output pixels must be processed; i.e., cols * (rows-2) must be a multiple of 8.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant, but not interruptible.
- The values of the left-most and right-most pixels on each line of the output are not computed.
- Eight output pixels are computed per iteration using loop unrolling and packed operations.
- The last stage of the epilog is kept to accommodate for the exception of storing only 6 outputs in the last iteration.

5.16 IMG_sobel_3x3_16s

IMG_sobel_3x3_16s *3x3 Sobel Edge Detection for 16-bit Input*

Syntax

void **IMG_sobel_3x3_16s** (const short *restrict in, const short *restrict out, short cols, short rows))

Arguments

in[ ]	Image input of size rows x cols
out[ ]	Image output of size (rows - 2) x cols
cols	Number of columns in the input image
rows	Number of rows in the input image

Description

This function applies horizontal and vertical Sobel edge detection masks to the input image and produces an output image which is two rows shorter than the input image. Within each row of the output, the first and the last pixel will not contain meaningful results.

Algorithm

The Sobel edge-detection masks shown below are applied to the input image separately. The absolute values of the mask results are then added together. If the resulting value is larger than 32767, it is clamped to 32767. The result is then written to the output image.

Horizontal Mask		
-1	-2	-1
0	0	0
1	2	1

Vertical Mask		
-1	0	1
-2	0	2
-1	0	1

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_sobel_3x3_16s
(
    const short *restrict   in,  /* Input image data   */
          short *restrict  out,  /* Output image data  */
          short            cols, /* image columns      */
          short            rows  /* Image rows         */
)
{
    int    H,      O,      V;
    int    i;
    int    i00,    i01,    i02;
    int    i10,            i12;
    int    i20,    i21,    i22;

    /* ---------------------------------------------------------- */
    /* Iterate over entire image as a single, continuous raster line*/
    /* ---------------------------------------------------------- */

    for (i = 0; i < (cols * (rows - 2) - 2); i++)
    {
        /* ---------------------------------------------------- */
        /*  Read in the required 3x3 region from the input.     */
        /* ---------------------------------------------------- */

        i00 =  in[i            ];
        i01 =  in[i        + 1];
```

```
            i02 =    in[i           + 2];
            i10 =    in[i +    cols    ];
            i12 =    in[i +    cols + 2];
            i20 =    in[i + 2 * cols    ];
            i21 =    in[i + 2 * cols + 1];
            i22 =    in[i + 2 * cols + 2];

            /* ----------------------------------------------------- */
            /*  Apply horizontal and vertical filter masks. The final */
    /*  filter output is the sum of the absolute values of      */
            /*  these filters.                                          */
            /* ----------------------------------------------------- */

            H   =    - i00 - 2 * i01 -  i02
                     + i20 + 2 * i21 +  i22;

            V   =    - i00      +     i02
                     - 2 * i10  + 2 * i12
                     - i20      +     i22;

            O   =    abs(H) + abs(V);

            /* ----------------------------------------------------- */
            /*  Clamp to 16-bit range.  The output is always positive */
            /*  due to the absolute value, so we only need to check    */
            /*  for overflow                                            */
            /* ----------------------------------------------------- */

            O   =    (O > 32767) ? 32767 : O;

            /* ----------------------------------------------------- */
            /* Store the output result                                  */
            /* ----------------------------------------------------- */

            out[i + 1] = O;
        }
    }
```

Special Requirements

- cols must be a multiple of 2 and greater than 3
- rows must be greater than or equal to 3
- Input and output arrays have no alignment requirements
- Input and output arrays should not overlap

Implementation Notes

- The values of the left-most and right-most pixels on each line of the output are not well defined
- Loop is unrolled by two manually, and further unroll by two is performed by the compiler to calculate four output samples per iteration
- The code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

5.17 *IMG_sobel_3x3_16*

IMG_sobel_3x3_16	***3x3 Sobel Edge Detection for Unsigned 16-bit input***

Syntax

void **IMG_sobel_3x3_16** (const unsigned short *restrict in, unsigned short *restrict out, short cols, short rows))

Arguments

in[ ]	Image input of size rows x cols
out[ ]	Image output of size (rows - 2) x cols
cols	Number of columns in the input image
rows	Number of rows in the input image

Description

The IMG_sobel filter is applied to the input image. The input image dimensions are given by the arguments 'cols' and 'rows'. The output image is 'cols' pixels wide and 'rows - 2' pixels tall.

To see how the implementation is going to work on the input buffer, imagine the following input buffer.

```
yyyyyyyyyyyyyyyy
yxxxxxxxxxxxxxxy
yxxxxxxxxxxxxxxy
yxxxxxxxxxxxxxxy
yxxxxxxxxxxxxxxy
yyyyyyyyyyyyyyyy
```

The output buffer would be:

```
tXXXXXXXXXXXXXXz
zXXXXXXXXXXXXXXz
zXXXXXXXXXXXXXXz
zXXXXXXXXXXXXXXt
```

Where:

X = IMG_sobel(x) The algorithm is applied to that pixel. The correct output is obtained; the data around the pixels that are worked on is used.

t = Whatever was in the output buffer in that position is kept there.

z = IMG_sobel(y) The algorithm is applied to that pixel. The output is not meaningful, because the necessary data to process the pixel is not available. This is because for each output pixel, input pixels from the right and from the left of the output pixel are needed; however, this data doesn't exist.

This means that only (rows-2) lines will be processed, and then all the pixels inside each line will be processed Even though the results for the first and last pixels in each line will not be relevant, it makes the control much simpler and ends up saving cycles. Also, the first pixel in the first processed line and the last pixel in the last processed line will not be computed. It is not necessary, since the results would not be valid. The following horizontal and vertical masks that are applied to the input buffer to obtain one output pixel.

Horizontal				**Vertical**		
-1	-2	-1		-1	0	1
0	0	0		-2	0	2
1	2	1		-1	0	1

Algorithm

This is a C model of the Sobel implementation..

```
void IMG_sobel_3x3_16
(
    const unsigned short *in,      /* Input image data  */
    unsigned short       *out,     /* Output image data */
    short cols, short rows         /* Image dimensions  */
```

```
)
{
    /* ----------------------------------------------- */
    /*  Intermediate values.                           */
    /* ----------------------------------------------- */
    int H;    /* Horizontal mask result           */
    int V;    /* Vertical mask result             */
    int O;    /* Sum of horizontal and vertical masks  */
    int i;    /* Input pixel offset               */
    int o;    /* Output pixel offset.             */
    int xy;   /* Loop counter.                    */

    /* ----------------------------------------------- */
    /*  Input values.                                  */
    /* ----------------------------------------------- */
    int i00, i01, i02;
    int i10,      i12;
    int i20, i21, i22;

    /* ----------------------------------------------- */
    /*  Step through the entire image.  We step        */
    /*  through 'rows - 2' rows in the output image,   */
    /*  since those are the only rows that are fully   */
    /*  defined for our filter.                        */
    /* ----------------------------------------------- */
    for (xy = 0, i = cols + 1, o = 1;
         xy < cols*(rows-2) - 2;
         xy++, i++, o++)
    {

        /* ------------------------------------------- */
        /*  Read necessary data to process an input    */
        /*  pixel.  The following instructions are     */
        /*  written to reflect the position of the     */
        /*  input pixels in reference to the pixel     */
        /*  being processed, which would correspond    */
        /*  to the blank space left in the middle.     */
        /* ------------------------------------------- */
        i00=in[i-cols-1]; i01=in[i-cols]; i02=in[i-cols+1];
        i10=in[i     -1];                 i12=in[i     +1];
        i20=in[i+cols-1]; i21=in[i+cols]; i22=in[i+cols+1];

        /* ------------------------------------------- */
        /*  Apply the horizontal mask.                 */
        /* ------------------------------------------- */
        H = -i00 - 2*i01 -   i02 +   i20 + 2*i21 + i22;

        /* ------------------------------------------- */
        /*  Apply the vertical mask.                   */
        /* ------------------------------------------- */
        V = -i00 +   i02 - 2*i10 + 2*i12 -   i20 + i22;

        O = abs(H) + abs(V);

        /* ------------------------------------------- */
        /*  If the result is over 65535 (largest valid */
        /*  pixel value), saturate (clamp) to 65535.   */
        /* ------------------------------------------- */
        if (O > 0xFFFF) O = 0xFFFF;

        /* ------------------------------------------- */
        /*  Store the result.                          */
        /* ------------------------------------------- */
        out[o] = O;
    }
}
```

Four output pixels are computed per iteration using loop unrolling and packed operations.

Special Requirements   At least four output pixels must be processed. The input image width must be even (eg. 'cols' must be even). rows >= 3

Implementation Notes

- No bank conflicts occur
- No bank conflicts occur
- The values of the left-most and right-most pixels on each line of the output are not well-defined
- The code is LITTLE ENDIAN

Compatibility   Compatible for C64x+ and C64x.

5.18 IMG_sobel_5x5_16s

IMG_sobel_5x5_16s *5x5 Sobel Edge Detection for 16-bit Input*

Syntax	void **IMG_sobel_5x5_16s** (const short *restrict in, short *restrict out, short cols, short rows)
Arguments	

in[ ]	Image input of size rows x cols
out[ ]	Image output of size (rows - 4) x cols
cols	Number of columns in the input image
rows	Number of rows in the input image

Description This function applies horizontal and vertical Sobel edge detection masks to the input image and produces an output image which is four rows shorter than the input image. Within each row of the output, the first two and the last two pixels will not contain meaningful results.

Algorithm The Sobel edge-detection masks shown below are applied to the input image separately. The absolute values of the mask results are then added together. If the resulting value is larger than 32767, it is clamped to 32767. The result is then written to the output image.

Horizontal Mask						Vertical Mask				
-1	-4	-6	-4	-1		1	2	0	-2	-1
-2	-8	-12	-8	-2		4	8	0	-8	-4
0	0	0	0	0		6	12	0	-12	-6
2	8	12	8	2		4	8	0	-8	-4
1	4	6	4	1		1	2	0	-2	-1

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_sobel_5x5_16s
(
    const short *restrict  in,   /* Input image data   */
          short *restrict out,   /* Output image data  */
          short          cols,   /* image columns      */
          short          rows    /* Image rows         */
)
{
    int    H,       O,       V;
    int    i;

    int    i00,     i01,     i02;
    int    i03,     i04,     i10;
    int    i11,     i12,     i13;
    int    i14,     i20,     i21;
    int    i23,     i24,     i30;
    int    i31,     i32,     i33;
    int    i34,     i40,     i41;
    int    i42,     i43,     i44;

    for (i = 0; i < cols * (rows - 4) - 4; i++)
    {

        i00 = in[i                ];
        i01 = in[i +             1];
        i02 = in[i +             2];
        i03 = in[i +             3];
        i04 = in[i +             4];

        i10 = in[i +     cols     ];
        i11 = in[i +     cols + 1];
```

```
                        i12 = in[i +    cols + 2];
                        i13 = in[i +    cols + 3];
                        i14 = in[i +    cols + 4];

                        i20 = in[i + 2 * cols    ];
                        i21 = in[i + 2 * cols + 1];
                        i23 = in[i + 2 * cols + 3];
                        i24 = in[i + 2 * cols + 4];

                        i30 = in[i + 3 * cols    ];
                        i31 = in[i + 3 * cols + 1];
                        i32 = in[i + 3 * cols + 2];
                        i33 = in[i + 3 * cols + 3];
                        i34 = in[i + 3 * cols + 4];

                        i40 = in[i + 4 * cols    ];
                        i41 = in[i + 4 * cols + 1];
                        i42 = in[i + 4 * cols + 2];
                        i43 = in[i + 4 * cols + 3];
                        i44 = in[i + 4 * cols + 4];

                        H = -     i00 - 4 * i01 - 6 * i02 - 4 * i03 -     i04
                            - 2 * i10 - 8 * i11 -12 * i12 - 8 * i13 - 2 * i14
                            + 2 * i30 + 8 * i31 +12 * i32 + 8 * i33 + 2 * i34
                            +     i40 + 4 * i41 + 6 * i42 + 4 * i43 +     i44;

                        V = +     i00 + 2 * i01            - 2 * i03 -     i04
                            + 4 * i10 + 8 * i11            - 8 * i13 - 4 * i14
                            + 6 * i20 +12 * i21            -12 * i23 - 6 * i24
                            + 4 * i30 + 8 * i31            - 8 * i33 - 4 * i34
                            +     i40 + 2 * i41            - 2 * i43 -     i44;

                        O = abs(H) + abs(V);
                        O = (O > 32767) ? 32767 : O;

                        out[i + 2]= O;
                    }
                }
```

Special Requirements

- cols must be a multiple of 2 and greater than 5
- rows must be greater than or equal to 5
- cols x (rows-4)-4>=2
- Input and output arrays should be half-word aligned
- Input and output arrays do not overlap

Implementation Notes

- The values of the left-most and right-most pixels on each line of the output are not well defined
- The loop computes two output pixels per iteration
- The code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

5.19 IMG_sobel_7x7_16s

IMG_sobel_7x7_16s *7x7 Sobel Edge Detection for 16-bit Input*

Syntax	void **IMG_sobel_7x7_16s** (const short *restrict in, short *restrict out, short cols, short rows)

Arguments

in[ ]	Image input of size rows x cols
out[ ]	Image output of size (rows - 6) x cols
cols	Number of columns in the input image
rows	Number of rows in the input image

Description This function applies horizontal and vertical sobel edge detection masks to the input image and produces an output image which is six rows shorter than the input image. Within each row of the output, the first three and the last three pixels will not contain meaningful results.

Algorithm The Sobel edge-detection masks shown below are applied to the input image separately. The absolute values of the mask results are then added together. If the resulting value is larger than 32767, it is clamped to 32767. The result is then written to the output image.

Horizontal Mask								Vertical Mask						
-1	-1	-1	-2	-1	-1	-1		-1	-1	-1	0	1	1	1
-1	-1	-1	-2	-1	-1	-1		-1	-1	-1	0	1	1	1
-1	-1	-1	-2	-1	-1	-1		-1	-1	-1	0	1	1	1
0	0	0	0	0	0	0		-2	-2	-2	0	2	2	2
1	1	1	2	1	1	1		-1	-1	-1	0	1	1	1
1	1	1	2	1	1	1		-1	-1	-1	0	1	1	1
1	1	1	2	1	1	1		-1	-1	-1	0	1	1	1

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_sobel_7x7_16s
(
    const short *restrict  in,   /* Input image data   */
            short *restrict out,  /* Output image data  */
            short           cols, /* image columns      */
            short           rows  /* Image rows         */
)
{
    int   H,       O,      V;
    int   i;
    int   i00,   i01,    i02;
    int   i03,   i04,    i05;
    int   i06,   i10,    i11;
    int   i12,   i13,    i14;
    int   i15,   i16,    i20;
    int   i21,   i22,    i23;
    int   i24,   i25,    i26;
    int   i30,   i31,    i32;
    int   i34,   i35,    i36;
    int   i40,   i41,    i42;
    int   i43,   i44,    i45;
    int   i46,   i50,    i51;
    int   i52,   i53,    i54;
    int   i55,   i56,    i60;
    int   i61,   i62,    i63;
    int   i64,   i65,    i66;

    for (i = 0; i < (cols * (rows - 6) - 6); i++)
```

```
{
    /* ------------------------------------------------------ */
    /*  Read in the required 7x7 region from the input.     */
    /* ------------------------------------------------------ */
    i00 = in[i    ];    i01 = in[i + 1];   i02 = in[i + 2];
    i03 = in[i + 3];    i04 = in[i + 4];   i05 = in[i + 5];
    i06 = in[i + 6];

    i10 = in[i + cols    ];     i11 = in[i + cols + 1];
    i12 = in[i + cols + 2];     i13 = in[i + cols + 3];
    i14 = in[i + cols + 4];     i15 = in[i + cols + 5];
    i16 = in[i + cols + 6];

    i20 = in[i + 2 * cols    ]; i21 = in[i + 2 * cols + 1];
    i22 = in[i + 2 * cols + 2]; i23 = in[i + 2 * cols + 3];
    i24 = in[i + 2 * cols + 4]; i25 = in[i + 2 * cols + 5];
    i26 = in[i + 2 * cols + 6];

    i30 = in[i + 3 * cols    ]; i31 = in[i + 3 * cols + 1];
    i32 = in[i + 3 * cols + 2]; i34 = in[i + 3 * cols + 4];
    i35 = in[i + 3 * cols + 5]; i36 = in[i + 3 * cols + 6];

    i40 = in[i + 4 * cols    ]; i41 = in[i + 4 * cols + 1];
    i42 = in[i + 4 * cols + 2]; i43 = in[i + 4 * cols + 3];
    i44 = in[i + 4 * cols + 4]; i45 = in[i + 4 * cols + 5];
    i46 = in[i + 4 * cols + 6];

    i50 = in[i + 5 * cols    ]; i51 = in[i + 5 * cols + 1];
    i52 = in[i + 5 * cols + 2]; i53 = in[i + 5 * cols + 3];
    i54 = in[i + 5 * cols + 4]; i55 = in[i + 5 * cols + 5];
    i56 = in[i + 5 * cols + 6];

    i60 = in[i + 6 * cols    ];
    i61 = in[i + 6 * cols + 1]; i62 = in[i + 6 * cols + 2];
    i63 = in[i + 6 * cols + 3]; i64 = in[i + 6 * cols + 4];
    i65 = in[i + 6 * cols + 5]; i66 = in[i + 6 * cols + 6];

    /* ------------------------------------------------------ */
    /* Apply horizontal and vertical filter masks. The final */
    /* filter output is the sum of the absolute values of    */
    /* these filters.                                        */

    /* ------------------------------------------------------ */

    H = -    i00 - i01 - i02 - 2 * i03 - i04 - i05 - i06
        -    i10 - i11 - i12 - 2 * i13 - i14 - i15 - i16
        -    i20 - i21 - i22 - 2 * i23 - i24 - i25 - i26
        +    i40 + i41 + i42 + 2 * i43 + i44 + i45 + i46
        +    i50 + i51 + i52 + 2 * i53 + i54 + i55 + i56
        +    i60 + i61 + i62 + 2 * i63 + i64 + i65 + i66;


    V = -    i00 - i01 - i02          + i04 + i05 + i06
        -    i10 - i11 - i12          + i14 + i15 + i16
        -    i20 - i21 - i22          + i24 + i25 + i26
        -    2 * i30 - 2 * i31 - 2 * i32 + 2 * i34 + 2 * i35
        +    2 * i36
        -    i40 -    i41 -    i42 +    i44 +    i45
        +    i46

        -    i50 -    i51 -    i52 +    i54 +    i55
        +    i56
        -    i60 -    i61 -    i62 +    i64 +    i65
        +    i66;

    O = abs(H) + abs(V);

    /* ------------------------------------------------------ */
    /* Clamp to 16-bit range.  The output is always positive */
    /* due to the absolute value, so we only need to check   */
    /* overflow.                                             */
    /* ------------------------------------------------------ */
    O = (O > 32767) ? 32767 : O;
```

```
                          /* -------------------------------------------------- */
                          /* Store it.                                          */
                          /* -------------------------------------------------- */
                          out[i + 3] = O;
                    }
              }
```

Special Requirements

- cols must be a multiple of 2 and greater than 7
- rows must be greater than or equal to 7
- cols x (rows-6)-6>=2
- Input and output arrays should be half-word aligned
- Input and output arrays do not overlap

Implementation Notes

- The values of the three left-most and three right-most pixels on each line of the output are not well defined
- The loop computes two output pixels per iteration
- The code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

5.20 IMG_thr_gt2max_8

IMG_thr_gt2max_8 *Thresholding - Clamp to 255*

Syntax   void **IMG_thr_gt2max_8**(const unsigned char * restrict in_data, unsigned char * restrict out_data, short cols, short rows, unsigned char threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double–word aligned.
out_data[ ]	Pointer to output image data. Must be double–word aligned.
cols	Number of image columns.
rows	Number of image rows. (col*rows) must be multiple of 16.
threshold	Threshold value.

Description   This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output have exactly the same dimensions.

Pixels that are below or equal to the threshold value are written to the output unmodified. Pixels that are greater than the threshold are set to 255 in the output image.

See the functions IMG_thr_le2min, IMG_thr_le2thr and IMG_thr_gt2thr for other thresholding functions.

Algorithm   Behavioral C code for this routine is provided below:

```
void IMG_thr_gt2max(const unsigned char *in_data, unsigned char *out_data, short
cols, short rows, unsigned char threshold)
{
    int i;

    for (i = 0; i < rows * cols; i++)
        out_data[i] = in_data[i] > threshold ? 255 : in_data[i];
}
```

Special Requirements

- Input and output buffers do not alias.
- Input and output buffers must be double-word aligned.
- rows* cols must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: This code is LITTLE ENDIAN.
- Interruptibility: The code is interrup-tolerant but not interruptible.

5.21 IMG_thr_gt2max_16

IMG_thr_gt2max_16 *Thresholding - Clamp to 65535*

Syntax

void **IMG_thr_gt2max_16**(const unsigned short *restrict in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Number of IDCTs to perform.
cols	Number of columns in input image.
rows	Number of rows in input image.
threshold	Threshold value.

Description

This routine performs thresholding operation on an input image pointed to by in_data[ ]. The dimensions of the input image are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output images are exactly of the same dimensions.

Pixels that are below the threshold value are written to the output unmodified. Pixels that are greater than the threshold are set to 65535 in the output image.

The exact thresholding function performed is described by the following transfer function diagram:

```
       65535_|         _____
             |        |  |
             |        |  |
         O   |        |  |
         U   |        |  |
         T   th _|. . . . .|
         P   |          /.
         U   |        /  .
         T   |      /    .
             |    /      .
          0_|/_____._____
             |        |  |
             0       th  65535

                   INPUT
```

See the IMGLIB functions IMG_thr_le2thr_16, IMG_thr_gt2thr_16 and IMG_thr_le2min_16 for other thresholding operations.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_thr_gt2max_16_c ( const unsigned short *restrict in_data, unsigned
short *restrict out_data, short cols, short rows, unsigned
short threshold )
{
    int i, pixels = rows * cols;

    /* ------------------------------------------------------------------ */
    /*  Step through input image copying pixels to the output.  If the    */
    /*  pixels are above our threshold, set them to the threshold value.  */
    /* ------------------------------------------------------------------ */
     #pragma UNROLL(16)
    for (i = 0; i < pixels; i++)
        out_data[i] = in_data[i] > threshold ? 0xffff : in_data[i];
}
```

Special Requirements

- The input and output buffers do not alias.
- The input and output buffers must be double-word aligned.

- The total number of pixels rows*cols must be at least 16 and a multiple of 16

Implementation Notes

- The loop is unrolled 16x. Packed-data processing techniques allow us to process all 16 pixels in parallel.
- CMPGT2 is used for comparison, but the unsigned value must be changed to signed value first, using XOR instructions.

Memory Note

- This code is ENDIAN NEUTRAL.
- The input and output arrays must be double-word aligned
- No bank conflicts occur, regardless of the relative alignment of in_data[ ] and out_data[ ].

Compatibility   This code is compatible for C64x+ and C64x.

5.22 IMG_thr_gt2thr_8

IMG_thr_gt2thr_8   *Thresholding - Clip Above Threshold*

Syntax

void **IMG_thr_gt2thr_8**(const unsigned char * restrict in_data, unsigned char * restrict out_data, short cols, short rows, unsigned char threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Pointer to output image data. Must be double-word aligned.
cols	Number of image columns.
rows	Number of image rows. (cols*rows) must be multiple of 16.
threshold	Threshold value.

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output have exactly the same dimensions.

Pixels that are below or equal to the threshold value are written to the output unmodified. Pixels that are greater than the threshold are set to the threshold value in the output image.

See the functions IMG_thr_le2min, IMG_thr_le2thr and IMG_thr_gt2max for other thresholding functions.

Algorithm

Behavioral C code for this routine is provided below:

```
void IMG_thr_gt2thr(const unsigned char *in_data, unsigned char *out_data, short
cols, short rows, unsigned char threshold)
{
    int i;

    for (i = 0; i < rows * cols; i++)
        out_data[i] = in_data[i] > threshold ? thr : in_data[i];
}
```

Special Requirements

- Input and output buffers do not alias.
- Input and output buffers must be double-word aligned.
- rows* cols must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: This code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant but not interruptible.

5.23 *IMG_thr_gt2thr_16*

IMG_thr_gt2thr_16 *Unsigned 16-bit Thresholding - Clip Above Threshold*

Syntax

void **IMG_thr_gt2thr_16**(const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)

Arguments

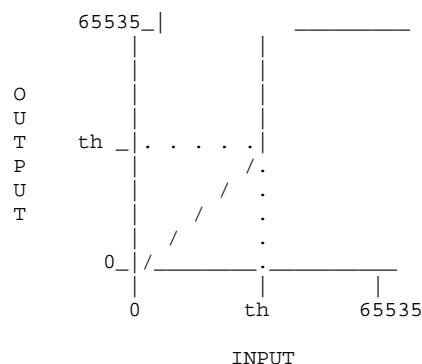|	
in_data[]	Pointer to input image data. Must be double-word aligned.
outdata[]	Pointer to output image data. Must be double-word aligned.
cols	Number of columns in input image.
rows	Number of rows in input image.
threshold	Threshold value

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given in the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output are exactly the same dimensions.

Pixels that are below the threshold value are written to the output unmodified. Pixels that are greater than the threshold are set to the threshold value in the output image.

The exact thresholding function performed is described by the following transfer function diagram:

```
        65535_|
              |
              |
         O    |
         U    |
         T   th _|. . . . . _____
         P     |         /.
         U     |        /  .
         T     |       /   .
               |     /     .
            0_|/_____._____
              |       |        |
              0      th      65535

                     INPUT
```

See the IMGLIB functions IMG_thr_le2thr_16, IMG_thr_gt2max_16 and IMG_thr_le2min_16 for other thresholding operations.

Algorithm

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements

```
void IMG_thr_gt2thr_16_c ( const unsigned short *in_data, unsigned
short *restrict out_data, short cols, short rows, unsigned
short threshold )
{
    int i, pixels = rows * cols;

    /* ------------------------------------------------------------------ */
    /*  Step through input image copying pixels to the output.  If the    */
    /*  pixels are above our threshold, set them to the threshold value.  */
    /* ------------------------------------------------------------------ */
    #pragma UNROLL(16)
    for (i = 0; i < pixels; i++)
        out_data[i] = in_data[i] > threshold ? threshold : in_data[i];
}
```

Special Requirements

- The input and output buffers do not alias.

- The input and output buffers must be double-word aligned.
- The total number of pixels rows*cols must be at least 16 and a multiple of 16.

Implementation Notes

- The loop is unrolled 16x. Packed-data processing techniques allow us to process all 16 pixels in parallel.
- Compare using MIN2, but first change the unsigned values to signed values, using XOR 0x8000.

Memory Note   This code is ENDIAN NEUTRAL.

Compatibility   This code is compatible for both C64x and C64x+.

5.24 *IMG_thr_le2min_8*

IMG_thr_le2min_8	*Thresholding - Clamp to Zero*

Syntax

void **IMG_thr_le2min_8**(const unsigned char * restrict in_data, unsigned char * restrict out_data, short cols, short rows, unsigned char threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Pointer to output image data. Must be double-word aligned.
cols	Number of image columns.
rows	Number of image rows. (cols*rows) must be multiple of 16.
threshold	Threshold value.

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output have exactly the same dimensions.

Pixels that are above the threshold value are written to the output unmodified. Pixels that are less than or equal to the threshold are set to zero in the output image.

See the functions IMG_thr_gt2thr, IMG_thr_le2thr and IMG_thr_gt2max for other thresholding functions.

Algorithm

Behavioral C code for this routine is provided below:

```
void IMG_thr_le2min(const unsigned char *in_data, unsigned char *out_data, short
cols, short rows, unsigned char threshold)
{
    int i;

    for (i = 0; i < rows * cols; i++)
        out_data[i] = in_data[i] <= threshold ? 0 : in_data[i];
}
```

Special Requirements

- Input and output buffers do not alias.
- Input and output buffers must be double-word aligned.
- rows* cols must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: This code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant but not interruptible.

5.25 IMG_thr_le2min_16

IMG_thr_le2min_16 *Unsigned 16-bit Thresholding - Clamp to Zero*

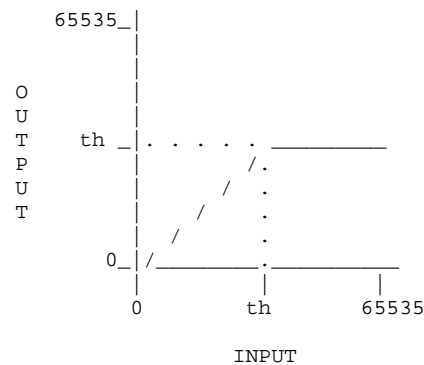| Syntax	void **IMG_thr_le2min_16** (const unsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Pointer to output image data. Must be double-word aligned.
cols	Number of columns in input image.
rows	Number of rows in input image.
threshold	Threshold value.

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given in the arguments' cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output are exactly the same dimensions.

Pixels that are above the threshold value are written to the output unmodified. Pixels that are less than or equal to the threshold are set to 0 in the output image.

The exact thresholding function performed is described by the following transfer function diagram:

```
        65535_|
              |                            /
              |                         /
        O     |                      /
        U     |                   /
        T   th _|. . . . . /
        P     |          |
        U     |          |
        T     |          |
              |          |
           0_|_____|_____
              |          |       |
              0         th     65535

                    INPUT
```

Algorithm

This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_thr_le2min_16_c( const unsigned short *in_data, unsigned
short *restrict out_data, short cols, short rows, unsigned
short threshold )
{
    int i, pixels = rows * cols;

    /* ------------------------------------------------------------------ */
    /*  Step through input image copying pixels to the output.  If the    */
    /*  pixels are below or equal to our threshold, set them to 0.        */
    /* ------------------------------------------------------------------ */
    for (i = 0; i < pixels; i++)
        out_data[i] = in_data[i] <= threshold ? 0 : in_data[i];
}

}
```

Special Requirements

- The input and output buffers do not alias.
- The input and output buffers must be double-word aligned.
- The total number of pixels rows*cols must be at least 16 and a multiple of 16.

Implementation Notes

- The loop is unrolled 16x. Packed-data processing techniques allow us to process all 16 pixels in parallel.

Memory Note

- This code is ENDIAN NEUTRAL
- No bank conflicts occur, regardless of the relative alignment of in_data[ ] and out_data[ ].

Compatibility   This code is compatible for both C64x+ and C64x.

5.26 IMG_thr_le2thr_8

IMG_thr_le2thr_8	*Thresholding - Clip Below Threshold*

Syntax

void **IMG_thr_le2thr_8**(const unsigned char * restrict in_data, unsigned char * restrict out_data, short cols, short rows, unsigned char threshold)

Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Pointer to output image data. Must be double-word aligned.
cols	Number of image columns.
rows	Number of image rows. (cols*rows) must be multiple of 16.
threshold	Threshold value.

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output have exactly the same dimensions.

Pixels that are above the threshold value are written to the output unmodified. Pixels that are less than or equal to the threshold are set to the threshold value in the output image.

See the functions IMG_thr_gt2thr, IMG_thr_le2min and IMG_thr_gt2max for other thresholding functions.

Algorithm

Behavioral C code for this routine is provided below:

```
void IMG_thr_le2thr(const unsigned char *in_data, unsigned char *out_data,
short cols, short rows, unsigned char threshold)
{
    int i;

    for (i = 0; i < rows * cols; i++)
        out_data[i] = in_data[i] <= threshold ? threshold : in_data[i];
}
```

Special Requirements

- Input and output buffers do not alias.
- Input and output buffers must be double-word aligned.
- rows* cols must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: This code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant but not interruptible.
- The loop is unrolled 16x. Packed-data processing techniques allow the parallel processing of all 16 pixels.

5.27 IMG_thr_le2thr_16

IMG_thr_le2thr_16 *Unsigned 16-bit Thresholding - Clip Below Threshold*

Syntax void **IMG_thr_le2thr_16**(constunsigned short *in_data, unsigned short *restrict out_data, short cols, short rows, unsigned short threshold)
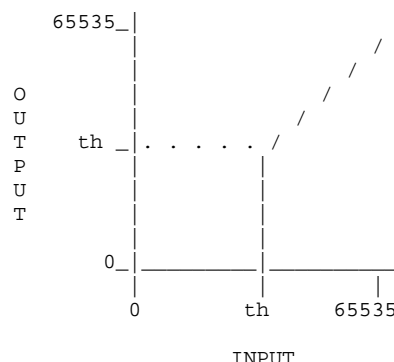
Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data	Pointer tooutput image data. Must be double-word aligned.
cols	Number of columns in input image.
rows	Number of rows in input image.
threshold	Threshold value.

Description This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given in the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output are exactly the same dimensions.

Pixels that are above the threshold value are written to the output unmodified. Pixels that are less than or equal to the threshold are set to the threshold value in the output image.

The exact thresholding function performed is described by the following transfer function diagram:

```
     65535_|
           |                              /
           |                           /
       O   |                        /
       U   |                     /
       T   th _|_____  /
       P       |          .
       U       |          .
       T       |          .
           |          .
       0_|_____._____
           |          |        |
           0          th      65535

                    INPUT
```

See the IMGLIB functions IMG_thr_gt2thr_16, IMG_thr_le2min_16 and IMG_thr_gt2max_16 for other thresholding operations.

Algorithm This is the C code implementation without any restrictions. However intrinsic code has restrictions as listed in the special requirements.

```
void IMG_thr_le2thr_16_c ( const unsigned short *in_data, unsigned
short *restrict out_data, short cols, short rows, unsigned short threshold )
{
    int i, pixels = rows * cols;

    /* ------------------------------------------------------------------ */
    /*  Step through input image copying pixels to the output.  If the    */
    /*  pixels are below our threshold, set them to the threshold value.  */
    /* ------------------------------------------------------------------ */
    for (i = 0; i < pixels; i++)
        out_data[i] = in_data[i] <= threshold ? threshold : in_data[i];
}
```

Special Requirement

- The input and output buffers do not alias.
- The input and output buffers must be double-word aligned.
- The total number of pixels rows*cols must be at least 16 and a multiple of 16.

Implementation Notes

- The loop is unrolled 16x.
- For comparison, MAX2 command is used, but before that the unsigned values must be changed into signed values using XOR 0x8000.

Compatibility　　　This code is compatible for both C64x and C64x+.

5.28 IMG_thr_le2thr

IMG_thr_le2thr	***Thresholding - Clip Below Threshold***

Syntax

void **IMG_thr_le2thr**(const unsigned char * restrict in_data, unsigned char * restrict out_data, short cols, short rows, unsigned char threshold)
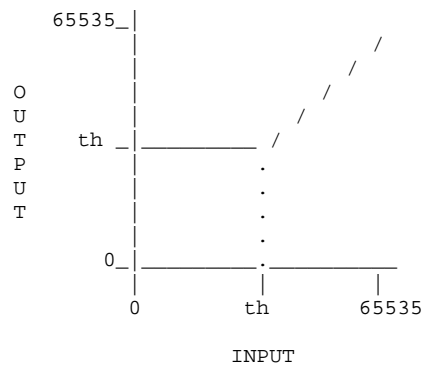
Arguments

in_data[ ]	Pointer to input image data. Must be double-word aligned.
out_data[ ]	Pointer to output image data. Must be double-word aligned.
cols	Number of image columns.
rows	Number of image rows. (cols*rows) must be multiple of 16.
threshold	Threshold value.

Description

This routine performs a thresholding operation on an input image in in_data[ ] whose dimensions are given by the arguments cols and rows. The thresholded pixels are written to the output image pointed to by out_data[ ]. The input and output have exactly the same dimensions.

Pixels that are above the threshold value are written to the output unmodified. Pixels that are less than or equal to the threshold are set to the threshold value in the output image.

See the functions IMG_thr_gt2thr, IMG_thr_le2min and IMG_thr_gt2max for other thresholding functions.

Algorithm

Behavioral C code for this routine is provided below:

```
void IMG_thr_le2thr(const unsigned char *in_data, unsigned char *out_data,
short cols, short rows, unsigned char threshold)
{
    int i;

    for (i = 0; i < rows * cols; i++)
        out_data[i] = in_data[i] <= threshold ? threshold : in_data[i];
}
```

Special Requirements

- Input and output buffers do not alias.
- Input and output buffers must be double-word aligned.
- rows* cols must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: This code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant but not interruptible.
- The loop is unrolled 16x. Packed-data processing techniques allow the parallel processing of all 16 pixels.

5.29 *IMG_yc_demux_be16_8*

IMG_yc_demux_be16_8 *YCbCR Demultiplexing (big endian source)*

Syntax	void **IMG_yc_demux_be16_8**(int n, const unsigned char * restrict yc, short * restrict y, short * restrict cr, short * restrict cb)

Arguments

n	Number of luma points. Must be multiple of 16.
yc	Packed luma/chroma inputs. Must be double-word aligned.
y	Unpacked luma data. Must be double-word aligned.
cr	Unpacked chroma r data. Must be double-word aligned.
cb	Unpacked chroma b data. Must be double-word aligned.

Description

This routine de-interleaves a 4:2:2 BIG ENDIAN video stream into three separate LITTLE ENDIAN 16-bit planes. The input array yc is expected to be an interleaved 4:2:2 video stream. The input is expected in BIG ENDIAN byte order within each 4-byte word. This is consistent with reading the video stream from a word-oriented BIG ENDIAN device, while the C6000 device is in a LITTLE ENDIAN configuration. In other words, the expected pixel order is:

	Word 0				Word 1				Word 2			
Byte#	0	1	2	3	4	5	6	7	8	9	10	11
	cb0	y1	cr0	y0	cb2	y3	cr2	y2	cb4	y5	cr4	y4

The output arrays y, cr, and cb are expected to not overlap. The de-interleaved pixels are written as half-words in LITTLE ENDIAN order.

This function reads the byte-oriented pixel data, zero-extends it, and then writes it to the appropriate result array. Both the luma and chroma values are expected to be unsigned. The data is expected to be in an order consistent with reading byte oriented data from a word-oriented peripheral that is operating in BIG ENDIAN mode, while the CPU is in LITTLE ENDIAN mode. This function unpacks the byte-oriented data so that further processing may proceed in LITTLE ENDIAN mode.

See the function IMB_yc_demux_le16 for code which handles input coming from a LITTLE ENDIAN device.

Algorithm

Behavioral C code for the routine is provided below:

```
void yc_demux_be16(int n, unsigned char *yc, short *y,
                   short *cr, short *cb )
{
    int i;
    for (i = 0; i < (n >> 1); i++)
    {
        y[2*i+0] = yc[4*i + 3];
        y[2*i+1] = yc[4*i + 1];
        cr[i]    = yc[4*i + 2];
        cb[i]    = yc[4*i + 0];
    }
}
```

Special Requirements

- The input and output data must be aligned to double-word boundaries.
- n must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur.

- Endian: The code is LITTLE ENDIAN.
- Interruptibility: This code is fully interruptible.
- The loop has been unrolled a total of 16 times to allow for processing 8 pixels in each datapath.
- Double-word-wide loads and stores maximize memory bandwidth utilization.
- This code uses _gmpy4() to ease the L/S/D unit bottleneck on ANDs. The _gmpy4(value, 0x00010001) is equivalent to value & 0x00FF00FF, as long as the size field of GFPGFR is equal to 7. (The polynomial does not matter.)

5.30 IMG_yc_demux_le16_8

IMG_yc_demux_le16_8 *YCbCR Demultiplexing (little endian source)*

Syntax		void **IMG_yc_demux_le16_8**(int n, const unsigned char * restrict yc, short * restrict y, short * restrict cr, short * restrict cb)
Arguments		
	n	Number of luma points. Must be multiple of 16.
	yc	Packed luma/chroma inputs. Must be double-word aligned.
	y	Unpacked luma data. Must be double-word aligned.
	cr	Unpacked chroma r data. Must be double-word aligned.
	cb	Unpacked chroma b data. Must be double-word aligned.

Description

This routine de-interleaves a 4:2:2 LITTLE ENDIAN video stream into three separate LITTLE ENDIAN 16-bit planes. The input array yc is expected to be an interleaved 4:2:2 video stream. The input is expected in LITTLE ENDIAN byte order within each 4-byte word. This is consistent with reading the video stream from a word-oriented LITTLE ENDIAN device, while the C6000 device is in a LITTLE ENDIAN configuration. In other words, the expected pixel order is:

	Word 0				Word 1				Word 2			
Byte#	0	1	2	3	4	5	6	7	8	9	10	11
	y0	cr0	y1	cb0	y2	cr2	y3	cb2	y4	cr4	y5	cb4

The output arrays y, cr, and cb are expected to not overlap. The de-interleaved pixels are written as half-words in LITTLE ENDIAN order.

This function reads the byte-oriented pixel data, zero-extends it, and then writes it to the appropriate result array. Both the luma and chroma values are expected to be unsigned. The data is expected to be in an order consistent with reading byte oriented data from a word-oriented peripheral that is operating in LITTLE ENDIAN mode, while the CPU is in LITTLE ENDIAN mode. This function unpacks the byte-oriented data so that further processing may proceed in LITTLE ENDIAN mode.

See the function IMB_yc_demux_be16 for code which handles input coming from a BIG ENDIAN device.

Algorithm

Behavioral C code for the routine is provided below:

```
void IMG_yc_demux_le16(int n, unsigned char *yc, short *y,
                    short *cr, short *cb )
{
    int i;
    for (i = 0; i < (n >> 1); i++)
    {
        y[2*i+0] = yc[4*i + 0];
        y[2*i+1] = yc[4*i + 2];
        cr[i]    = yc[4*i + 1];
        cb[i]    = yc[4*i + 3];
    }
}
```

Special Requirements

- The input and output data must be aligned to double-word boundaries.
- n must be a multiple of 16.

Notes

- Bank Conflicts: No bank conflicts occur.

- Endian: The code is LITTLE ENDIAN.
- Interruptibility: This code is fully interruptible.
- The loop has been unrolled a total of 16 times to allow for processing 8 pixels in each data path.
- Double-word-wide loads and stores maximize memory bandwidth utilization.
- This code uses _gmpy4() to ease the L/S/D unit bottleneck on ANDs. The _gmpy4(value, 0x00010001) is equivalent to value & 0x00FF00FF, as long as the size field of GFPGFR is equal to 7. (The polynomial does not matter.)

5.31 *IMG_ycbcr422p_rgb565*

IMG_ycbcr422p_rgb565 *Planarized YCbCR 4:2:2/4:2:0 to RGB 5:6:5 Color Space Conversion*

Syntax

void **IMG_ycbcr422p_rgb565**(const short * restrict coeff, const unsigned char * restrict y_data, const unsigned char * restrict cb_data, const unsigned char * restrict cr_data, unsigned short * restrict rgb_data, unsigned num_pixels)

Arguments

coeff[5]	Matrix coefficients
y_data	Luminence data (Y'). Must be double-word aligned.
cb_data	Blue color-diff (B'-Y'). Must be word aligned.
cr_data	Red color-diff (R'-Y'). Must be word aligned.
rgb_data	RGB 5:6:5 packed pixel out. Must be double-word aligned.
num_pixels	Number of luma pixels to process. Must be multiple of 8.

Description

This kernel performs Y'CbCr to RGB conversion. The coeff[] array contains the color-space-conversion matrix coefficients. The y_data, cb_data and cr_data pointers point to the separate input image planes. The rgb_data pointer points to the output image buffer, and must be word aligned. The kernel is designed to process arbitrary amounts of 4:2:2 image data, although 4:2:0 image data may be processed as well. For 4:2:2 input data, the y_data, cb_data and cr_data arrays may hold an arbitrary amount of image data. For 4:2:0 input data, only a single scan-line (or portion thereof) may be processed at a time.

The coefficients in the coeff array must be in signed Q13 form.

This code can perform various flavors of Y'CbCr to RGB conversion, as long as the offsets on Y, Cb, and Cr are -16, -128, and -128, respectively, and the coefficients match the pattern shown. The kernel implements the following matrix form, which involves 5 unique coefficients:

coeff[] = { 0x2000, 0x2BDD, -0x0AC5, -0x1658, 0x3770 };

[1.0000	0.0000	1.3707]		[Y' - 16]		[R']
[1.0000	-0.3365	-0.6982]	*	[Cb - 128]	=	[G']
[1.0000	1.7324	0.0000]		[Cr - 128]		[B']

Below are some common coefficient sets, along with the matrix equation that they correspond to. Coefficients are in signed Q13 notation, which gives a suitable balance between precision and range.

Y'CbCr $\rightarrow$ RGB conversion with RGB levels that correspond to the 219-level range of Y'. Expected ranges are [16..235] for Y' and [16..240] for Cb and Cr.

[coeff[0]	0.0000	coeff[1]]		[Y' - 16]		[R']
[coeff[0]	coeff[2]	coeff[3]]	*	[Cb - 128]	=	[G']
[coeff[0]	coeff[4]	0.0000]		[Cr - 128]		[B']

Y'CbCr $\rightarrow$ RGB conversion with the 219-level range of Y' expanded to fill the full RGB dynamic range. (The matrix has been scaled by 255/219). Expected ranges are [16..235] for Y' and [16..240] for Cb and Cr.

$$
\begin{bmatrix} 1.1644 & 0.0000 & 1.5960 \\ 1.1644 & -0.3918 & -0.8130 \\ 1.1644 & 2.0172 & 0.0000 \end{bmatrix} * \begin{bmatrix} Y' - 16 \\ Cb - 128 \\ Cr - 128 \end{bmatrix} = \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}
$$

Other scalings of the color differences (B'-Y') and (R'-Y') (sometimes incorrectly referred to as U and V) are supported, as long as the color differences are unsigned values centered around 128 rather than signed values centered around 0, as noted above.

In addition to performing plain color-space conversion, color saturation can be adjusted by scaling coeff[1] through coeff[4]. Similarly, brightness can be adjusted by scaling coeff[0]. However, general hue adjustment cannot be performed, due to the two zeros hard-coded in the matrix.

Algorithm   Behavioral C code for the routine is provided below:

```
void IMG_ycbcr422pl_to_rgb565
(
    const short       coeff[5],  /* Matrix coefficients.          */
    const unsigned char *y_data,  /* Luminence data        (Y')    */
    const unsigned char *cb_data,  /* Blue color-difference (B'-Y') */
    const unsigned char *cr_data,  /* Red color-difference  (R'-Y') */
    unsigned short    *rgb_data, /* RGB 5:6:5 packed pixel output. */
    unsigned          num_pixels /* # of luma pixels to process.  */
)
{
    int    i;                    /* Loop counter                  */
    int    y0, y1;               /* Individual Y components       */
    int    cb, cr;               /* Color difference components   */
    int    y0t,y1t;              /* Temporary Y values            */
    int    rt, gt, bt;           /* Temporary RGB values          */
    int    r0, g0, b0;           /* Individual RGB components      */
    int    r1, g1, b1;           /* Individual RGB components      */
    int    r0t,g0t,b0t;          /* Truncated RGB components       */
    int    r1t,g1t,b1t;          /* Truncated RGB components       */
    int    r0s,g0s,b0s;          /* Saturated RGB components       */
    int    r1s,g1s,b1s;          /* Saturated RGB components       */
    short  luma = coeff[0];      /* Luma scaling coefficient.     */
    short  r_cr = coeff[1];      /* Cr's contribution to Red.     */
    short  g_cb = coeff[2];      /* Cb's contribution to Green.   */
    short  g_cr = coeff[3];      /* Cr's contribution to Green.   */
    short  b_cb = coeff[4];      /* Cb's contribution to Blue.    */
    unsigned short  rgb0, rgb1;  /* Packed RGB pixel data         */
    /* ---------------------------------------------------------------- */
    /*  Iterate for num_pixels/2 iters, since we process pixels in pairs.  */
    /* ---------------------------------------------------------------- */
    i = num_pixels >> 1;
    while (i-->0)
    {
        /* ---------------------------------------------------------------- */
        /*  Read in YCbCr data from the separate data planes.          */
        /*                                                             */
        /*  The Cb and Cr channels come in biased upwards by 128, so   */
        /*  subtract the bias here before performing the multiplies for */
        /*  the color space conversion itself.  Also handle Y's upward */
        /*  bias of 16 here.                                           */
        /* ---------------------------------------------------------------- */
        y0 = *y_data++  - 16;
        y1 = *y_data++  - 16;
        cb = *cb_data++ - 128;
        cr = *cr_data++ - 128;
        /* ================================================================ */
        /*  Convert YCrCb data to RGB format using the following matrix: */
        /*                                                             */
        /*      [ Y' - 16 ]   [ coeff[0] 0.0000   coeff[1] ]    [ R']  */
        /*      [ Cb - 128 ] * [ coeff[0] coeff[2] coeff[3] ] = [ G']  */
        /*      [ Cr - 128 ]   [ coeff[0] coeff[4] 0.0000   ]    [ B']  */
        /*                                                             */
        /*  We use signed Q13 coefficients for the coefficients to make */
        /*  good use of our 16-bit multiplier.  Although a larger Q-point */
        /*  may be used with unsigned coefficients, signed coefficients */
```

```
                      /*  add a bit of flexibility to the kernel without significant   */
                      /*  loss of precision.                                           */
                      /* ============================================================= */
                      /* ------------------------------------------------------------- */
                      /*  Calculate chroma channel's contribution to RGB.              */
                      /* ------------------------------------------------------------- */
                      rt  = r_cr * (short)cr;
                      gt  = g_cb * (short)cb + g_cr * (short)cr;
                      bt  = b_cb * (short)cb;
                      /* ------------------------------------------------------------- */
                      /*  Calculate intermediate luma values.  Include bias of 16 here. */
                      /* ------------------------------------------------------------- */
                      y0t = luma * (short)y0;
                      y1t = luma * (short)y1;


                      /* ------------------------------------------------------------- */
                      /*  Mix luma, chroma channels.                                   */
                      /* ------------------------------------------------------------- */
                      r0  = y0t + rt; r1 = y1t + rt;
                      g0  = y0t + gt; g1 = y1t + gt;
                      b0  = y0t + bt; b1 = y1t + bt;
                      /* ============================================================= */
                      /*  At this point in the calculation, the RGB components are     */
                      /*  nominally in the format below.  If the color is outside the  */
                      /*  our RGB gamut, some of the sign bits may be non-zero,         */
                      /*  triggering saturation.                                       */
                      /*                                                               */
                      /*                     3     2 2       1 1                       */
                      /*                     1     1 0       3 2         0             */
                      /*                    [ SIGN | COLOR   | FRACTION ]              */
                      /*                                                               */
                      /*  This gives us an 8-bit range for each of the R, G, and B     */
                      /*  components.  (The transform matrix is designed to transform  */
                      /*  8-bit Y/C values into 8-bit R,G,B values.)  To get our final */
                      /*  5:6:5 result, we "divide" our R, G and B components by 4, 8, */
                      /*  and 4, respectively, by reinterpreting the numbers in the    */
                      /*  format below:                                                */
                      /*                                                               */
                      /*         Red,    3     2 2     1 1                             */
                      /*         Blue    1     1 0     6 5             0                */
                      /*                [ SIGN | COLOR | FRACTION    ]                 */
                      /*                                                               */
                      /*                 3     2 2       1 1                           */
                      /*         Green   1     1 0       5 4             0             */
                      /*                [ SIGN | COLOR | FRACTION    ]                 */
                      /*                                                               */
                      /*  "Divide" is in quotation marks because this step requires no */
                      /*  actual work.  The code merely treats the numbers as having a */
                      /*  different Q-point.                                           */
                      /* ============================================================= */
                      /* ------------------------------------------------------------- */
                      /*  Shift away the fractional portion, and then saturate to the  */
                      /*  RGB 5:6:5 gamut.                                             */
                      /* ------------------------------------------------------------- */
                      r0t = r0 >> 16;
                      g0t = g0 >> 15;
                      b0t = b0 >> 16;
                      r1t = r1 >> 16;
                      g1t = g1 >> 15;
                      b1t = b1 >> 16;
                      r0s = r0t < 0 ? 0 : r0t > 31 ? 31 : r0t;
                      g0s = g0t < 0 ? 0 : g0t > 63 ? 63 : g0t;
                      b0s = b0t < 0 ? 0 : b0t > 31 ? 31 : b0t;
                      r1s = r1t < 0 ? 0 : r1t > 31 ? 31 : r1t;
                      g1s = g1t < 0 ? 0 : g1t > 63 ? 63 : g1t;
                      b1s = b1t < 0 ? 0 : b1t > 31 ? 31 : b1t;
                      /* ------------------------------------------------------------- */
                      /*  Merge values into output pixels.                             */
                      /* ------------------------------------------------------------- */
                      rgb0 = (r0s << 11) + (g0s <<  5) + (b0s <<  0);
                      rgb1 = (r1s << 11) + (g1s <<  5) + (b1s <<  0);
                      /* ------------------------------------------------------------- */
                      /*  Store resulting pixels to memory.                            */
                      /* ------------------------------------------------------------- */
```

```
                    *rgb_data++ = rgb0;
                    *rgb_data++ = rgb1;
                }
                return;
        }
```

Special Requirements

- The number of luma samples to be processed must be a multiple of 8.
- The input Y array and the output image must be double-word aligned.
- The input Cr and Cb arrays must be word aligned.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- Pixel replication is performed implicitly on chroma data to reduce the total number of multiplies required. The chroma portion of the matrix is calculated once for each Cb and Cr pair, and the result is added to both Y' samples.
- Matrix Multiplication is performed as a combination of MPY2s and DOTP2s. Saturation to 8-bit values is performed using SPACKU4, which takes in 4 signed 16-bit values and saturates them to unsigned 8-bit values. The output of Matrix Multiplication would ideally be in a Q13 format. However, this cannot be fed directly to SPACKU4. This implies a shift left by 3 bits, which could increase the number of shifts to be performed. Thus, to avoid being bottlenecked by so many shifts, the Y, Cr, and Cb data are shifted left by 3 before multiplication. This is possible because they are 8-bit unsigned data. Due to this, the output of Matrix Multiplication is in a Q16 format, which can be directly fed to SPACKU4.
- Because the loop accesses four different arrays at three different strides, no memory accesses are allowed to parallelize in the loop. No bank conflicts occur as a result. The epilog has been completely removed, while the prolog is left as is. However, some cycles of the prolog are performed using the kernel cycles to help reduce code-size. The setup code is merged along with the prolog for speed.

6 IMGLIB2 Picture Filtering Functions

This section provides detailed specifications and descriptions for IMGLIB2 picture filtering functions.

6.1 IMG_conv_3x3_i8_c8s

IMG_conv_3x3_i8_c8s *3x3 Convolution*

Syntax	void **IMG_conv_3×3_i8_c8s**(const unsigned char * restrict in_data, unsigned char * restrict out_data, int cols, const char * restrict mask, int shift)
Arguments	

in_data[ ]	Input image
out_data[ ]	Output image
cols	Number of columns in the input image. Must be multiple of 8
mask[3][3]	3×3 mask
shift	Shift value

Description　　The convolution kernel accepts three rows of cols input pixels and produces one output row of cols pixels using the input mask of 3 by 3. The user-defined shift value is used to shift the convolution value down to the byte range. The convolution sum is also range limited to 0.255. The shift amount is non-zero for low pass filters, and zero for high pass and sharpening filters.

Algorithm　　This is the C equivalent of the assembly code without restrictions. The assembly code is hand optimized and restrictions apply as noted.

```
void IMG_conv_3x3(unsigned char *in_data, unsigned char *out_data, int cols, char
*mask, int shift)
{
    unsigned char   *IN1,*IN2,*IN3;
    unsigned char   *OUT;

    short    pix10,  pix20,  pix30;
    short    mask10, mask20, mask30;

    int      sum,       sum00,  sum11;
    int      i;
    int      sum22,    j;

    IN1     =   in_data;
    IN2     =   IN1 + x_dim;
    IN3     =   IN2 + x_dim;
    OUT     =   out_data;

    for (j = 0; j < cols; j++)
    {
        sum = 0;

        for (i = 0; i < 3; i++)
        {
            pix10  =   IN1[i];
            pix20  =   IN2[i];
            pix30  =   IN3[i];

            mask10 =   mask[i];
            mask20 =   mask[i + 3];
            mask30 =   mask[i + 6];

            sum00  =   pix10 * mask10;
            sum11  =   pix20 * mask20;
            sum22  =   pix30 * mask30;

            sum    +=   sum00 + sum11+ sum22;
        }
```

```
                        IN1++;
                        IN2++;
                        IN3++;

                        sum = (sum >> shift);
                        if ( sum <  0  )        sum = 0;
                        if ( sum > 255 )        sum = 255;
                        *OUT++   =        sum;
                }
        }
```

Special Requirements

- cols output pixels are produced when three lines, each with a width of cols pixels, are given as input.
- cols must be a multiple of 8.
- The array pointed to by out_data should not alias with the array pointed to by in_data.
- The mask to the kernel should be such that the sum for each pixel is less than or equal to 65536. This restriction arises because of the use of the ADD2 instruction to compute two pixels in a register.

Notes

- Bank Conflicts: No bank conflicts occur in this function.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- This code is designed to take advantage of the 8-bit multiplier capability provided by MPYSU4/MPYUS4. The kernel uses loop unrolling and computes eight output pixels for every iteration.
- The eight bit elements in each mask are replicated four times to fill a word by using the PACKL4 and PACK2 instructions.
- The image data is brought in using LDNDW. The results of the multiplications are summed using ADD2. The output values are packed using SPACK2 and stored using STNDW, which writes eight 8-bit values at a time.

6.2 *IMG_conv_3x3_i16s_c16s*

IMG_conv_3x3_i16s_c16s *3x3 Convolution for 16-bit Input*

Syntax	void **IMG_conv_3x3_i16s_c16s** (const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image 16-bit signed
imgout_ptr	Pointer to output image 16-bit signed
width	Number of outputs to be calculated
pitch	Number of columns in the input image
mask_ptr	Pointer to 3x3 mask used-16 bit signed
shift	User specified shift value

Description The convolution kernel accepts three rows of'pitch input pixels and produces one row of width output pixels using the input mask of 3 by 3. This convolution performs a point by point multiplication of 3 by 3 masks with the input image. The result of 9 multiplications are then summed to produce a 32-bit convolution intermediate sum. Overflow while accumulation is not handled. However assumptions are made on filter gain to avoid overflow. The user-defined shift value is used to shift this convolution sum down to the short range and store in an output array. The result being stored is also saturated to the -32768 to 32767 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input, output image pixels and the masks are provided as 16-bit signed values.

Algorithm This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_3x3_i16s_c16s
(
    const short  *restrict    imgin_ptr,
          short  *restrict    imgout_ptr,
          short                   width,
          short                   pitch,
    const short  *restrict    mask_ptr,
          short                   shift
)
{
    int    i,      j,      k;
    int    sum;

    for (i = 0; i < width; i++)
    {
        sum = 0;
        for (j = 0; j < 3; j++)
        {
            for (k = 0; k < 3; k++)
                sum += imgin_ptr[j * pitch + i + k] *
                    mask_ptr[j * 3 + k];
        }
        sum  >>= shift ;
        sum    = (sum > 32767)? 32767 : (sum < -32768 ? -32768 : sum);

        imgout_ptr[i] = sum;

    }
}
```

Special Requirements

- Width must be >= 2 and multiples of 2
- Pitch should be >= width
- Internal accuracy of the computations is 32 bits. To ensure correctness on a 16 bit

input data, the maximum permissible filter gain in terms of bits is 16-bits i.e. the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{16} - 1$

- Output array must be word aligned
- Input and Mask array must be half-word aligned
- The input and output arrays should not overlap

Implementation Notes

- The inner loop is unrolled completely to form a single loop
- Two output samples are calculated per iteration
- The code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.3 IMG_conv_3x3_i16_c16s

IMG_conv_3x3_i16_c16s *3x3 Convolution for Unsigned 16-bit Input*

Syntax	void **IMG_conv_3x3_i16_c16s** (const unsigned short *restrict inptr, unsigned short *restrict outptr, int x_dim, const short *restrict mask, int shift)

Arguments

inptr	Pointer to an input array of unsigned 16-bit pixels
outptr	Pointer to an output array of 16-bit pixels
x_dim	Number of output pixels
mask	Pointer to 16-bit filter mask
shift	User specified shift value

Description   The convolution kernel accepts three rows of 'x_dim' input points and produces one output row of 'x_dim' points using the input mask of 3 by 3. The user-defined shift value is used to shift the convolution value, down to the 16-bit range. The convolution sum is also range-limited to 40 bits. The shift amount is non-zero for low pass filters, and zero for high pass and sharpening filters..

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_3x3_i16_c16s_c(const unsigned short *restrict inptr,
                  unsigned short      *restrict outptr,
                  int                 x_dim,
                  const short          *restrict mask,
                  int                 shift)
{
    const unsigned short    *IN1,*IN2,*IN3;
    unsigned short          *OUT;

    unsigned short   pix10,  pix20,  pix30;    /* rev. from short to ushort */
    short    mask10, mask20, mask30;

    long    sum;                                /* rev. from int to long  */
    long    sum00,  sum11, sum22;
    int     i, j;

    /*------------------------------------------------------------------*/
    /* Set imgcols to the width of the image and set three pointers for  */
    /* reading data from the three input rows. Also set the output poin- */
    /* ter.                                                              */
    /*------------------------------------------------------------------*/

    IN1     =   inptr;
    IN2     =   IN1 + x_dim;
    IN3     =   IN2 + x_dim;
    OUT     =   outptr;

    /*------------------------------------------------------------------*/
    /* The j: loop iterates to produce one output pixel per iteration.   */
    /* The mask values and the input values are read using the i loop.   */
    /* The convolution sum is then computed. The convolution sum is      */
    /* then shifted and range limited to 0..255                         */
    /*------------------------------------------------------------------*/

    for (j = 0; j < x_dim ; j++)
    {
        /*------------------------------------------------------------------*/
        /* Initialize convolution sum to zero, for every iteration of    */
        /* outer loop. The inner loop computes convolution sum.          */
        /*------------------------------------------------------------------*/

        sum = 0;
```

```
                for (i = 0; i < 3; i++)
                {
                    pix10  =   IN1[i];
                    pix20  =   IN2[i];
                    pix30  =   IN3[i];

                    mask10 =   mask[i];
                    mask20 =   mask[i + 3];
                    mask30 =   mask[i + 6];

                    sum00  =  (long)pix10 * mask10;
                    sum11  =  (long)pix20 * mask20;
                    sum22  =  (long)pix30 * mask30;

                    sum   +=   sum00 + sum11+ sum22;
                }

                /*-------------------------------------------------------------*/
                /*  Increment input pointers and shift sum and range limit to  */
                /*  0...65535.                                                 */
                /*-------------------------------------------------------------*/

                IN1++;
                IN2++;
                IN3++;

                sum = (sum >> shift);

                if ( sum <  0  )       sum = 0;
                  if ( sum > 65535 )     sum = 65535;

                /*-------------------------------------------------------------*/
                /* Store output sum into the output pointer OUT                */
                /*-------------------------------------------------------------*/

                *OUT++   =        sum;
            }
        }
```

Special Requirements

- x_dim must be a multiple of 4.
- I/O buffers do not overlap.
- I/O and mask arrays should be half-word aligned.
- Appropriate shift is used to avoid saturation.
- Internal accuracy of the computations is 40 bits. Accuracy will not be lost in internal computations. The final results are saturated to 16 bit.

Compatibility   Compatible for both C64x and C64x+.

6.4 IMG_conv_5x5_i8_c8s

IMG_conv_5x5_i8_c8s *5x5 Convolution for 8-bit Input*

Syntax	void **IMG_conv_5x5_i8_c8s** (const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output image (8-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 5x5 mask used (8-bit signed).
shift	User-specified shift value.

Description   The convolution kernel accepts five rows of pitch input pixels and produces one row of width output pixels using the input mask of 5 by 5. This convolution performs a point by point multiplication of 5 by 5 masks with the input image. The result of 25 multiplications are then summed to produce a 32-bit convolution intermediate sum. The user defined shift value is used to shift this convolution sum down to the byte range and store in an output array. The result being stored is also saturated to the range 0 to 255 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire 'width' is covered. The input pixels are provided as 8-bit unsigned values and the masks are provided as 8-bit signed values. Output will be in unsigned 8-bit.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_5x5_i8_c8s
(
    const unsigned char    *restrict    imgin_ptr,
          unsigned char    *restrict   imgout_ptr,
          short                              width,
          short                              pitch,
    const char                 *restrict    mask_ptr,
          short                              shift
)
{
    int    i,      j,       k;
    int    sum;

    for (i = 0; i < width ; i++)
sum = 0;
       for (j = 0; j < 5; j++)

            for (k = 0; k < 5; k++)

                sum += (unsigned char)imgin_ptr[j * pitch + i + k] *
                    (char)mask_ptr[j * 5 + k];

        sum  = (sum >> shift);
        sum  = (sum > 255) ?  255 : (sum < 0 ? 0 : sum);
        imgout_ptr[i] = sum ;
    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.
- Output array must be word-aligned.
- No alignment restrictions for mask and input array.
- Input and output arrays should not overlap.

Implementation Notes

- The inner loop is manually unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for both C64x and C64x+.

6.5 *IMG_conv_5x5_i16s_c16s*

IMG_conv_5x5_i16s_c16s *5x5 Convolution for 16-bit Input*

Syntax	void **IMG_conv_5x5_i16_c16s** (const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (16-bit signed).
imgout_ptr	Pointer to output image (16-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 5x5 mask used (16-bit signed).
shift	User-specified shift value.

Description   The convolution kernel accepts five rows of pitch input pixels and produces one row of width output pixels using the input mask of 5 by 5. This convolution performs a point by point multiplication of 5 by 5 masks with the input image. The result of 25 multiplications are then summed to produce a 32-bit convolution intermediate sum. Overflow during accumulation is not handled; however, assumptions are made on filter gain to avoid overflow. The user defined shift value is used to shift this convolution sum down to the short range and store in an output array. The result being stored is also saturated to the -32768 to 32767 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input, output image pixels, and the masks are provided as 16-bit signed values.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_5x5_i16s_c16s
(
    const short   *restrict    imgin_ptr,
          short   *restrict    imgout_ptr,
          short                    width,
          short                    pitch,
    const short   *restrict     mask_ptr,
          short                    shift
)
{
    int     i,       j,       k;
    int     sum;

    for (i = 0; i < width; i++)
    {
        sum = 0;
        for (j = 0; j < 5; j++)
        {
            for (k = 0; k < 5; k++)
                sum += imgin_ptr[j * pitch + i + k] *
                    mask_ptr[j * 5 + k];
        }
        sum  >>= shift ;
        sum    = (sum > 32767)? 32767 : (sum < -32768 ? -32768 : sum);

        imgout_ptr[i] = sum;

    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.

- Internal accuracy of the computations is 32 bits. To ensure correctness on a 16-bit input data, the maximum permissible filter gain in terms of bits is 16-bits (i.,e., the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{16} - 1$).
- Output array must be word-aligned.
- Input and output arrays should not overlap.

Implementation Notes

- The inner loop is unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for C64x+.

6.6 *IMG_conv_5x5_i8_c16s*

IMG_conv_5x5_i8_c16s *5x5 Convolution for 16-bit Input and 16-bit masks*

Syntax	void **IMG_conv_5x5_i8_c16s** (const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output image (8-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 5x5 mask used (16-bit signed).
shift	User-specified shift value.

Description   The convolution kernel accepts five rows of pitch input pixels and produces one row of width output pixels using the input mask of 5 by 5. This convolution performs a point-by-point multiplication of 5 by 5 masks with the input image. The result of 25 multiplications are then summed to produce a 32-bit convolution intermediate sum. The user-defined shift value is used to shift this convolution sum down to the byte range and store in an output array. The result being stored is also saturated to the range 0 to 255 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input pixels are provided as 8-bit signed values. The masks are provided as 16-bit signed values. Output will be in unsigned 8-bit values.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_5x5_i8_c16s
(
    const unsigned char    *restrict    imgin_ptr,
          unsigned char    *restrict    imgout_ptr,
          short                          width,
          short                          pitch,
    const short            *restrict    mask_ptr,
          short                          shift
)
{
    int    i,      j,      k;
    int    sum;

    for (i = 0; i < width ; i++)
sum = 0;
      for (j = 0; j < 5; j++)

            for (k = 0; k < 5; k++)

                 sum += (unsigned char)imgin_ptr[j * pitch + i + k] *
                        mask_ptr[j * 5 + k];

        sum  = (sum >> shift);
        sum = (sum >  255) ?  255 : (sum < 0 ? 0 : sum);
        imgout_ptr[i] = sum ;
    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.
- Output array must be word-aligned.
- Mask array should be half-word aligned.

- No alignment restrictions on input array.
- Input and output arrays should not overlap.

Implementation Notes

- The inner loop is manually unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for C64x+.

6.7 IMG_conv_7x7_i8_c8s

IMG_conv_7x7_i8_c8s *7x7 Convolution for 8-bit Input*

Syntax	void **IMG_conv_7x7_i8_c8s** (const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output image (8-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 7x7 mask used (8-bit signed).
shift	User-specified shift value.

Description   The convolution kernel accepts seven rows of pitch input pixels and produces one row of width output pixels using the input mask of 7 by 7. This convolution performs a point-by-point multiplication of 7 by 7 masks with the input image. The result of 49 multiplications are then summed to produce a 32-bit convolution intermediate sum. The user-defined shift value is used to shift this convolution sum down to the byte range and store in an output array. The result being stored is also saturated to the range 0 to 255 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input pixels are provided as 8-bit unsigned values. The masks are provided as 8-bit signed values. Output will be in unsigned 8-bit values.

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_7x7_i8_c8s
(
    const unsigned char    *restrict    imgin_ptr,
          unsigned char    *restrict  imgout_ptr,
          short                              width,
          short                              pitch,
    const char      *restrict               mask_ptr,
          short                              shift
)
{
    int    i,      j,      k;
    int    sum;

    for (i = 0; i < width ; i++)
sum = 0;
      for (j = 0; j < 7; j++)

          for (k = 0; k < 7; k++)

              sum += (unsigned char)imgin_ptr[j * pitch + i + k] *
                    (char)mask_ptr[j * 7 + k];

        sum  = (sum >> shift);
        sum = (sum >  255) ?  255 : (sum < 0 ? 0 : sum);
        imgout_ptr[i] = sum ;
    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.
- Output array must be word-aligned.

- No alignment restrictions on input or mask arrays.
- Input and output arrays should not overlap.

Implementation Notes

- The outer loop is unrolled to calculate two output samples per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for both C64x and C64x+.

6.8 *IMG_conv_7x7_i16s_c16s*

IMG_conv_7x7_i16s_c16s *7x7 Convolution for 16-bit Input*

Syntax	void **IMG_conv_7x7_i16s_c16s** (const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (16-bit signed).
imgout_ptr	Pointer to output image (16-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 7x7 mask used (16-bit signed).
shift	User-specified shift value.

Description   The convolution kernel accepts seven rows of pitch input pixels and produces one row of width output pixels using the input mask of 7 by 7. This convolution performs a point-by-point multiplication of 7 by 7 masks with the input image. The result of 49 multiplications are then summed to produce a 32-bit convolution intermediate sum. Overflow during accumulation is not handled; however, assumptions are made on filter gain to avoid overflow. The user-defined shift value is used to shift this convolution sum down to the short range and store in an output array. The result being stored is also saturated to the -32768 to 32767 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input, output image pixels, and the masks are provided as 16-bit signed values

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_7x7_i16s_c16s
(
    const short   *restrict    imgin_ptr,
         short    *restrict    imgout_ptr,
         short                     width,
         short                     pitch,
    const short   *restrict     mask_ptr,
         short                     shift
)

{
    int      i,       j,        k;
    int      sum;

    for (i = 0; i < width; i++)
    {
        sum = 0;
        for (j = 0; j < 7; j++)
        {
            for (k = 0; k < 7; k++)
                sum += imgin_ptr[j * pitch + i + k] *
                        mask_ptr[j * 7 + k];
        }
        sum  >>= shift ;
        sum    = (sum > 32767)? 32767 : (sum < -32768 ? -32768 : sum);

        imgout_ptr[i] = sum;

    }
}
```

Special Requirements

- Width must be >=8 and multiples of 8.

- Pitch should be >= width.
- Internal accuracy of the computations is 32 bits. To ensure correctness on a 16 bit input data, the maximum permissible filter gain in terms of bits is 16-bits (i.e., the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{16} - 1$).
- Output and output arrays must be double-word aligned.
- Mask array should be half-word aligned.
- Input and output arrays should not overlap.

Implementation Notes

- The inner loop simultaneously operates on 8 output pixels.
- Code is LITTLE ENDIAN

Compatibility   Compatible for C64x+.

6.9 *IMG_conv_7x7_i8_c16s*

IMG_conv_7x7_i8_c16s *7x7 Convolution for 8-bit Input and 16-bit Masks*

Syntax

void **IMG_conv_7x7_i8_c16s** (const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output image (8-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 7x7 mask used (16-bit signed).
shift	User-specified shift value.

Description

The convolution kernel accepts seven rows of'pitch input pixels and produces one row of width output pixels using the input mask of 7 by 7. This convolution performs a point-by-point multiplication of 7 by 7 masks with the input image. The result of 49 multiplications are then summed to produce a 32-bit convolution intermediate sum. The user-defined shift value is used to shift this convolution sum down to the byte range and store in an output array. The result being stored is also saturated to the range 0 to 255 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input pixels are provided as 8-bit unsigned values. The masks are provided as 16-bit signed vales. Output will be in unsigned 8-bit values.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_7x7_i8_c16s
(
    const unsigned char    *restrict    imgin_ptr,
          unsigned char    *restrict    imgout_ptr,
          short                              width,
          short                              pitch,
    const short                 *restrict    mask_ptr,
          short                              shift
)
{
    int     i,      j,      k;
    int     sum;

    for (i = 0; i < width ; i++)
{
sum = 0;
        for (j = 0; j < 7; j++)
          {
              for (k = 0; k < 7; k++)
              {
                  sum += (unsigned char)imgin_ptr[j * pitch + i + k] *
                      mask_ptr[j * 7 + k];
                          }
                      }
        sum  = (sum >> shift);
        sum = (sum >  255) ?  255 : (sum < 0 ? 0 : sum);
        imgout_ptr[i] = sum ;
    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.
- Output array must be word-aligned.
- Mask pointer should be half-word aligned.
- No restrictions on the alignment of input array.

Implementation Notes

- The outer loop is manually unrolled by two to calculate two output samples per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for C64x+.

6.10 IMG_conv_11x11_i8_c8s

IMG_conv_11x11_i8_c8s *7x7 Convolution for 8-bit Input and 8-bit Masks*

Syntax	void **IMG_conv_11x11_i8_c8s** (const unsigned char *restrict imgin_ptr, unsigned char *restrict imgout_ptr, short width, short pitch, const char *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output image (8-bit unsigned).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 11x11 mask used (8-bit signed).
shift	User-specified shift value.

Description The convolution kernel accepts eleven rows of pitch input pixels and produces one row of 'width' output pixels using the input mask of 11 by 11. This convolution performs a point-by-point multiplication of 11 by 11 masks with the input image. The result of 121 multiplications are then summed to produce a 32-bit convolution intermediate sum. The user-defined shift value is used to shift this convolution sum down to the byte range and store in an output array. The result being stored is also saturated to the range 0 to 255 inclusive. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input pixels are provided as 8-bit unsigned values and the masks are provided as 8-bit signed values. Output will be in unsigned 8-bit values.

Algorithm This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_11x11_i8_c8s
(
    const unsigned char    *restrict   imgin_ptr,
          unsigned char    *restrict  imgout_ptr,
          short                               width,
          short                               pitch,
    const char                   *restrict   mask_ptr,
          short                               shift
)
{
    int     i,      j,      k;
    int     sum;

    for (i = 0; i < width ; i++) {
sum = 0;
        for (j = 0; j < 11; j++) {
          for (k = 0; k < 11; k++) {
              sum += (unsigned char)imgin_ptr[j * pitch + i + k] *
                      (char)mask_ptr[j * 11 + k];
          }
        }
        sum  = (sum >> shift);
        sum = (sum >  255) ?  255 : (sum < 0 ? 0 : sum);
        imgout_ptr[i] = sum ;
          }
    }
}
```

Special Requirements

- Width must be >=2 and multiples of 2.
- Pitch should be >= width.
- Output array must be word-aligned.

- No alignment restrictions on input and mask arrays.
- Input and output arrays should not overlap.

Implementation Notes

- The outer loop is manually unrolled by two to calculate two output samples per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for both C64x and C64x+.

6.11 IMG_conv_11x11_i16s_c16s

IMG_conv_11x11_i16s_c16s *11x11 Convolution for 16-bit Inputs*

Syntax

void **IMG_conv_11x11_i16s_c16s** (const short *restrict imgin_ptr, short *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift)

Arguments

imgin_ptr	Pointer to input image (16-bit signed)
imgout_ptr	Pointer to output image (16-bit unsigned
width	Number of outputs to be calculated
pitch	Number of columns in the input image
mask_ptr	Pointer to 11x11 mask used (16-bit signed)
shift	User-specified shift value

Description

The convolution kernel accepts eleven rows of pitch input pixels and produces one row of width output pixels using the input mask of 11 by 11. This convolution performs a point-by-point multiplication of 11 by 11 masks with the input image. The result of 121 multiplications are then summed to produce a 40-bit convolution intermediate sum. Overflow while accumulation is not handled; however, assumptions are made on filter gain to avoid overflow. The user defined shift value is used to shift this convolution sum down to the short range and store in an output array. The result being stored is also range limited between -32768 to 32767 and will be saturated accordingly. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The input, output image pixels and the masks are provided as 16-bit signed values.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_conv_11x11_i16s_c16s
(
    const short   *restrict    imgin_ptr,
          short   *restrict    imgout_ptr,
          short                 width,
          short                 pitch,
    const short   *restrict    mask_ptr,
          short                 shift
)

{
    int     i,      j,         k;
    long    sum;

    for (i = 0; i < width; i++)
    {
        sum = 0;
        for (j = 0; j < 11; j++)
        {
            for (k = 0; k < 11; k++)
                sum += imgin_ptr[j * pitch + i + k] *
                        mask_ptr[j * 11 + k];
        }
        sum  >>= shift ;
        sum    = (sum > 32767)? 32767 : (sum < -32768 ? -32768 : sum);

        imgout_ptr[i] = sum;

    }
}
```

Special Requirements

- Width must be >=4 and multiples of 4
- Pitch should be >= width
- Internal accuracy of the computations is 40 bits. To ensure correctness on a 16 bit input data, the maximum permissible filter gain in terms of bits is 24-bits i.e. the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{24} - 1$
- Output array must be double-word aligned
- Input and mask arrays should be half-word aligned
- Input and output arrays should not overlap.

Implementation Notes

- The outer loop is unrolled by four to calculate four output samples per iteration.
- Code is LITTLE ENDIAN

Compatibility   Compatible for C64x+.

6.12 IMG_corr_3x3_i8_c16s

IMG_corr_3x3_i8_c16s *3x3 Correlation for 8-bit Input and 16-bit Masks*

Syntax	void **IMG_corr_3x3_i8_c16s** (const unsigned char *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)

Arguments

imgin_ptr	Pointer to input image (8-bit signed)
imgout_ptr	Pointer to output image (32-bit signed)
width	Number of outputs to be calculated
pitch	Number of columns in the input image
mask_ptr	Pointer to 3x3 mask used (16-bit signed)

Description

The correlation kernel accepts three rows of pitch input pixels and produces one row of width output pixels using the input mask of 3x3. This correlation performs a point-by-point multiplication of 3x3 masks with the input image. The result of the nine multiplications are then summed to produce a 32-bit sum and then stored in an output array. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The masks are provided as 16-bit signed values, the input image pixels are provided as 8-bit unsigned values, and the output pixels will be 32-bit signed. The image mask to be correlated is typically part of the input image or another image.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_3x3_i8_c16s
(
    const   unsigned char    *restrict  imgin_ptr,
                     int      *restrict imgout_ptr,
                     short                     width,
                     short                     pitch,
    const            short    *restrict    mask_ptr
)
{
    int     i,          j,          k;
    int     sum;
    for (i = 0; i < width ; i++)
    {
        sum = 0;
        for (j = 0; j < 3; j++)
            for (k = 0; k < 3; k++)
sum += imgin_ptr[j * pitch + i + k] *
        mask_ptr[j * 3 + k];
        imgout_ptr[i] =  sum;
    }
 }
```

Special Requirements

- Width must be >= 2 and multiples of 2.
- Pitch should be >= width.
- Output array must be double-word aligned.
- No alignment restrictions on input array.
- mask_ptr should be half-word aligned.
- Input and output arrays should not overlap.

Implementation Notes

- The inner loop is manually unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.13 *IMG_corr_3x3_i16s_c16s*

IMG_corr_3x3_i16s_c16s *3x3 Correlation for 16-bit Inputs*

Syntax

void **IMG_corr_3x3_i16s_c16s** (const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift, int round)

Arguments

imgin_ptr	Pointer to input image (16-bit signed)
imgout_ptr	Pointer to output image (32-bit signed)
width	Number of outputs to be calculated
pitch	Number of outputs to be calculated
mask_ptr	Pointer to 3x3 mask used (16-bit signed)
shift	User-specified shift amount
round	User-specified round value

Description

The correlation kernel accepts three rows of pitch input pixels and produces one row of width output pixels using the input mask of 3x3. This correlation performs a point by point multiplication of 3x3 masks with the input image. The result of the 9 multiplications are then summed to produce a 40-bit sum which is added to user-specified round value, right-shifted by the specified value, and then stored in an output array. Overflow and saturation of the accumulated sum is not handled. However assumptions are made on filter gain to avoid them. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The masks are provided as 16-bit signed values and the input image pixels are provided as 16-bit signed values and the output pixels will be 32-bit signed. The image mask to be correlated is typically part of the input image or another image.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_3x3_i16s_c16s
(
    const  short    *restrict  imgin_ptr,
           int      *restrict imgout_ptr,
           short                    width,
           short                    pitch,
    const  short    *restrict  mask_ptr,
           short                    shift,
           int                      round
)
{
    int    i,        j,        k;
    long     sum;
    for (i = 0; i < width ; i++)
    {
        sum = round;
        for (j = 0; j < 3; j++)
            for (k = 0; k < 3; k++)
sum += imgin_ptr[j * pitch + i + k] *          mask_ptr[j * 3 + k];
        sum  = (sum >> shift);
        imgout_ptr[i] =  (int)sum;
    }
 }
```

Special Requirements

- Width must be >= 2 and multiples of 2
- Pitch should be >= width.
- Internal accuracy of the computations is 40 bits. To ensure correctness on a 16 bit input data, the maximum permissible filter gain in terms of bits is 24-bits i.e. the

cumulative sum of the absolute values of the filter coefficients should not exceed 2^{24} - 1.

- Output array must be double word aligned.
- Input and mask array should be half-word aligned.
- Input and output arrays should not overlap
- Shift is appropriate to produce a 32-bit result.
- Range of filter co-efficients is -32767 to 32767.

Implementation Notes

- The inner loop is manually unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.14 IMG_corr_3x3_i8_c8

IMG_corr_3x3_i8_c8 *3x3 Correlation for unsigned 8-bit inputs*

Syntax	void **IMG_corr_3x3_i8_c8**(const unsigned char *inptr, int *restrict outptr, int n_out, int x_dim, const unsigned char *mask, const short shift, int round)

Arguments

inptr	Pointer to input image (8-bit signed)
outptr	Pointer to output image (32-bit signed)
n_out	Number of outputs to be calculated
x_dim	Number of columns in the input image
mask	Pointer to 3x3 mask used 16-bit signed
shift	User-specified shift amount
round	User-specified round value

Description

The correlation performs a point by point multiplication of the 3 by 3 mask with the input image. The result of the nine multiplications are then summed up together to produce a convolution sum. A rounding constant is added to the sum and shifted by user specified amount.

The image mask to be correlated is typically part of the input image and indicates the area of the best match between the input image and mask. The mask is moved one column at a time, advancing the mask over the portion of the row specified by 'n_out'. When 'n_out' is larger than 'x_dim', multiple rows will be processed.

An application may call this kernel once per row to calculate the correlation for an entire image:

```
for (i = 0; i < rows; i++)
{
    IMG_corr_3x3(&i_data[i * x_dim],
    &o_data[i * n_out],n_out, x_dim, mask, shift, round);
}
```

Alternately, the kernel may be invoked for multiple rows at a time, although the two outputs at the end of each row will have meaningless values. For example:

```
IMG_corr_3x3(i_data, o_data,2 * x_dim, x_dim, mask, shift, round);
```

This will produce two rows of outputs into o_data. The outputs at locations o_data[x_dim - 2], o_data[x_dim - 1], o_data[2*x_dim - 2] and o_data[2*x_dim - 1] will have meaningless values. This is harmless, although the application must account for this when interpreting the results.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements.

```
void IMG_corr_3x3_i8_c8_cn
(
    const unsigned char *restrict i_data,      /* input image           */
    int                       *restrict o_data,       /* output image
  */
    int                               n_out,        /* number of
outputs */
    int                               x_dim,        /* width of image
     */
    const unsigned char *restrict  mask,        /* convolution mask */
    const short                           shift,          /* result shift
amount */
    int                               round         /* rounding
constant */
)
```

```
{
  in[t i, j, k;
  for (i=0; i<n_out; i++)
 {
    int sum=round;

    for (j=0; j<3;++)
      sum+=i_data[j*x_dim+i+k]*mask[j*3+k];

 }
}
```

Special Requirements

- The array pointed to by outptr does not alias with the array pointed to by inptr
- x_dim>=4 and is a multiple of 2
- n_out should be a multiple of 4
- This kernel is developed for LITTLE ENDIAN target

Implementation Notes

- Data for the input image pixels is reused by pre-loading them outside the loop and issuing moves to bring them to the appropriate registers once inside the loop. this is done to minimize the loads from nine to six within the loop, for each pair of pixels in the present computation of the correlation. The loop is unrolled once so that eighteen multiples for the two output pixels can schedule in 9 cycles leading to 4.5 cycles per output pixel. In addition, the trivial loop that did the loads three at a time, per row, is collapsed to increase parallel operations.
- The code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.15 *IMG_corr_3x3_i16_c16s*

IMG_corr_3x3_i16_c16s *3x3 Correlation for unsigned 16-bit Inputs*

Syntax   void **IMG_corr_3x3_i16_c16s** (const unsigned short *i_data, long *restrict o_data, const unsigned short mask[3][3], int x_dim, int n_out)

Arguments

i_data	Pointer to input image (16-bit signed)
o_data	Pointer to output image (40(64)-bit signed))
mask_ptr	Pointer to 3x3 mask used (16-biit signed)
x_dim	Number of columns in the input image
n_out	Number of outputs to be calculated

Description   The correlation kernel accepts three rows of x_dim input pixels and produces one row of width output pixels using the input 3x3 mask. This correlation performs a point by point multiplication of 3x3 masks with the input image. The result of the 9 multiplications are then summed together to produce a 40-bit sum and stored in an output array. The mask is moved one column at a time, advancing the mask over the entire image until the n_out points are generated. The masks are provided as 16-bit signed values and the input image pixels are provided as 16-bit unsigned values and the output pixels will be 40-bit signed.

The image mask to be correlated is typically part of an input image and indicates the area of the best match between the input image and mask.

An application may call this kernel once per row to calculate the correlation for an entire image:

```
for (i = 0; i < rows; i++)
    {
        IMG_corr_3x3(&i_data[i * x_dim], &o_data[i * n_out],
                mask, x_dim, n_out);
    }
```

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_3x3_i16_c16s
(
    const unsigned short *i_data,        /* input image        */
    long         *restrict o_data,       /* output correlation data */
    unsigned short  mask[3][3],           /* correlation mask  */
    int               x_dim,             /* width of image     */
    int               n_out              /* number of outputs */
)
{
    int i, j, k;

    for (i = 0; i < n_out; i++)
    {
        long sum = 0;                    /* temporary var. long data type */

        for (j = 0; j < 3; j++)
            for (k = 0; k < 3; k++)
                sum += (long) i_data[j * x_dim + i + k] * mask[j][k];

        o_data[i] = sum;
    }
}
```

Special Requirements

- Input and output buffers do not alias.

- n_out should be a multiple of 4.

Memory Notes

- All arrays should be half-word aligned.
- No bank conflicts occur.
- Code is LITTLE ENDIAN.

Implementation Notes

- The inner loops are unrolled completely, and the outer loop is unrolled 4 times.
- Half-word unsigned multiplication is used here.
- Non-aligned loads and stores are used to avoid alignment issues.

Compatibility   Compatible for both C64x and C64x+.

6.16 IMG_corr_5x5_i16s_c16s

IMG_corr_5x5_i16s_c16s *5x5 Correlation for 16-bit Inputs*

Syntax	void **IMG_corr_5x5_i16s_c16s** (const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, short shift, int round)

Arguments

imgin_ptr	Pointer to input image (16-bit signed).
imgout_ptr	Pointer to output correlation result (32-bit signed).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 5x5 mask used (16-bit signed).
shift	User-specified shift amount.
round	User-specified round value.

Description

The correlation kernel accepts five rows of pitch input pixels and produces one row of width output pixels using the input mask of 5x5. This correlation performs a point-by-point multiplication of 5x5 masks with the input image. The result of the 25 multiplications are then summed to produce a 40-bit sum. A rounding const is added and the result is shifted and stored in the output array. Overflow and saturation of the accumulated sum is not handled; however, assumptions are made on filter gain to avoid them. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The masks are provided as 16-bit signed values and the input image pixels are provided as 16-bit signed values and the output pixels will be 32-bit signed. The image mask to be correlated is typically part of the input image or another image.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_5x5_i16s_c16s
(
    const   short    *restrict  imgin_ptr,
            int      *restrict imgout_ptr,
            short                   width,
            short                   pitch,
    const   short    *restrict   mask_ptr,
            short               shift,
            int                 round
)
{
    int    i,        j,        k;
    long    sum;
    for (i = 0; i < width ; i++)
    {
        sum = round;
        for (j = 0; j < 5; j++)
            for (k = 0; k < 5; k++)
sum += imgin_ptr[j * pitch + i + k] *        mask_ptr[j * 5 + k];
            sum  = (sum >> shift);
            imgout_ptr[i] =  (int)sum;
    }
 }
```

Special Requirements

- Width must be >= 2 and multiples of 2.
- Pitch should be >= width.
- Internal accuracy of the computations is 40 bits. To ensure correctness on a 16-bit input data, the maximum permissible filter gain in terms of bits is 24-bits i.e. the cumulative sum of the absolute values of the filter coefficients should not exceed 2^{24}

– 1.
- Output array must be double word aligned.
- Input and mask arrays should be half-word aligned.
- The input and output arrays should not overlap.
- Shift is appropriate to produce a 32-bit result.
- Range of filter co-efficients is -32767 to 32767.

Implementation Notes

- The inner loop is manually unrolled completely to form a single loop and two output samples are calculated per iteration.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.17 IMG_corr_11x11_i16s_c16s

IMG_corr_11x11_i16s_c16s *11x11 Correlation for 16-bit Inputs*

Syntax	void **IMG_corr_11x11_i16s_c16s** (const short *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)

Arguments

imgin_ptr	Pointer to input image (16-bit signed).
imgout_ptr	Pointer to output correlation result (32-bit signed).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 11x11 mask used (16-bit signed).
round	User-specified round value.

Description   The correlation kernel accepts 11 rows of pitch input pixels and produces one row of width output pixels using the input mask of 11x11. This correlation performs a point-by-point multiplication of 11x11 masks with the input image. The result of the 121 multiplications are then summed to produce a 40-bit sum which is added to user specified round value and then stored in an output array. Overflow and saturation of the accumulated sum is not handled; hooowever, assumptions are made on filter gain to avoid them. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The masks are provided as 16-bit signed values, the input image pixels are provided as 16-bit signed values, and the output pixels will be 32-bit signed. The image mask to be correlated is typically part of the input image or another image

Algorithm   This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
# define SHIFT 7
void IMG_corr_11x11_i16s_c16s
(
    const  short    *restrict  imgin_ptr,
           int      *restrict imgout_ptr,
           short                    width,
           short                    pitch,
    const  short    *restrict   mask_ptr,
           int                      round
)
{
    int     i,          j,          k;
    long sum;
    for (i = 0; i < width ; i++)
    {
        sum = round;
        for (j = 0; j < 11; j++)
            for (k = 0; k < 11; k++)
sum += imgin_ptr[j * pitch + i + k] *          mask_ptr[j * 11 + k];
            sum  = (sum >> SHIFT);
            imgout_ptr[i] =  sum;
    }
}
```

Special Requirements

- Width must be >= 2 and multiples of 2.
- Pitch should be >= width.
- Internal accuracy of the computations is 40 bits. To ensure correctness on a 16-bit input data, the maximum permissible filter gain in terms of bits is 24-bits (i.e., the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{24} - 1$).

- Output array must be double-word aligned.
- Input and mask arrays should be half-word aligned.
- The input and output arrays should not overlap.

Implementation Notes

- The inner loop is manually unrolled by two to calculate two output samples per iteration.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.18   IMG_corr_11x11_i8_c16s

IMG_corr_11x11_i8_c16s *11x11 Correlation for 8-bit input and 16-bit masks*

Syntax	void **IMG_corr_11x11_i8_c16s** (const unsigned char *restrict imgin_ptr, int *restrict imgout_ptr, short width, short pitch, const short *restrict mask_ptr, int round)

Arguments

imgin_ptr	Pointer to input image (8-bit signed).
imgout_ptr	Pointer to output correlation result (32-bit signed).
width	Number of outputs to be calculated.
pitch	Number of columns in the input image.
mask_ptr	Pointer to 11x11 mask used (16-bit signed).
round	User-specified round value.

Description The correlation kernel accepts 11 rows of pitch input pixels and produces one row of width output pixels using the input mask of 11x11. This correlation performs a point-by-point multiplication of 11x11 masks with the input image. The result of the 121 multiplications are then summed together to produce a 40-bit sum which is added to user specified round value and then stored in an output array. Overflow and saturation of the accumulated sum is not handled; however, assumptions are made on filter gain to avoid them. The mask is moved one column at a time, advancing the mask over the entire image until the entire width is covered. The masks are provided as 16-bit signed values, the input image pixels are provided as 16-bit signed values, and the output pixels will be 32-bit signed. The image mask to be correlated is typically part of the input image or another image

Algorithm This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_11x11_i8_c16s
(
    const  unsigned char      *restrict  imgin_ptr,
           int                *restrict imgout_ptr,
           short                           width,
           short                           pitch,
    const  short              *restrict mask_ptr,
           int                             round
)
{
    int    i,         j,          k;
    int    sum;
    for (i = 0; i < width ; i++)
    {
        sum = round;
        for (j = 0; j < 11; j++)
            for (k = 0; k < 11; k++)
sum += imgin_ptr[j * pitch + i + k] *           mask_ptr[j * 11 + k];
        imgout_ptr[i] =  sum;
    }
}
```

Special Requirements

- Width must be >= 2 and multiples of 2.
- Pitch should be >= width.
- Internal accuracy of the computations is 32 bits. To ensure correctness on 8-bit input data, the maximum permissible filter gain in terms of bits is 24-bits (i.e., the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{24} - 1$).
- Output array must be double-word aligned.
- No alignment restrictions on Input array.

- Mask should be half-word aligned.
- The input and output arrays should not overlap.

Implementation Notes

- The inner loop is manually unrolled by two to calculate two output samples per iteration.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for C64x+.

6.19 IMG_corr_gen_i16s_c16s

IMG_corr_gen_i16s_c16s *Generalized Correlation*

Syntax	void **IMG_corr_gen_i16s_c16s**_(const short *in_data, short *h, short *out_data, int M, int cols)

Arguments

in_data[ ]	Input image data (one line of width cols). Must be word aligned.
h[M]	1xM tap filter
out_data[ ]	Output array of size cols – M + 8. Must be double–word aligned.
M	Number of filter taps.
cols	Width of line of image data.

Description
This routine performs a generalized correlation with a 1xM tap filter. It can be called repetitively to form an arbitrary MxN 2-D generalized correlation function. The correlation sums are stored as half words. The input pixel, and mask data are assumed to be shorts. No restrictions are placed on the number of columns in the image (cols) or the number of filter taps (M).

Algorithm
Behavioral C code for the routine is provided below:

```
void IMG_corr_gen_cn
(
    const short *in_data,
    const short *h,
    short       *out_data,
    int          M,
    int          cols
)
{
    int i, j;
    for (j = 0; j < cols - M; j++)
        for (i = 0; i < M; i++)
            out_data[j] += in_data[i + j] * h[i];
}
```

Special Requirements

- Array in_data[ ] must be word-aligned, array out_data[ ] must be double-word aligned, and array h[ ] must be half-word aligned.
- The size of the output array must be at least (cols - m + 8).
- Internal accuracy of computations is 16 bits. The convolution sum should not exceed 16 bits (signed) at any stage
- cols > M

Implementation Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant, but not interruptible.
- Since this function performs generalized correlation, the number of filter taps can be as small as one. Hence, it is not beneficial to pipeline this loop in its original form. In addition, collapsing of the loops causes data dependencies and degrades the performance.
- However, loop order interchange can be used effectively. For example, the outer loop of the natural C code is exchanged to be the inner loop that is to be software pipelined in the optimized assembly code. It is beneficial to pipeline this loop because typical image dimensions are larger than the number of filter taps. Note however, that the number of data loads and stores increase within this loop compared to the natural C code.

- Unrolling of the outer loop assumes that there are an even number of filter taps (M). Two special cases arise:
  - m = 1. In this case, a separate version that processes just 1 tap is used and the code directly starts from this loop without executing the version of the code for even number of taps.
  - m is odd. In this case, the even version of the loop is used for as many even taps as possible and then the last tap is computed using the odd tap special version created for m = 1.
- The inner loop is unrolled 8 times, assuming that the loop iteration (cols – M) is a multiple of 8. In most typical images, cols is a multiple of 8 but since M is completely general, (cols – M) may not be a multiple of 8. If (cols – M) is not a multiple of 8, then the inner loop iterates fewer times than required and certain output pixels may not be computed. Use the following process to solve this problem:
  - Eight is added to (cols – M) so that the next higher multiple of 8 is computed. This implies that in certain cases, up to 8 extra pixels may be computed. To annul this extra computation, 8 locations starting at out_data[cols – M] are zeroed out before returning to the calling function.

6.20 *IMG_corr_gen_iq*

IMG_corr_gen_iq	*Correlation fwith Q-point Math*

Syntax

void **IMG_corr_gen_iq** (const int *restrict x, const short *restrict h, int *restrict y, int m, int x_dim, int x_qpt, int h_qpt, int y_qpt)

Arguments

x	Input image data (one line of width 'x_dim').
h[m]	1 x m tap filter
y	Correlation output array of size 'x_dim – m'
m	Number of filter taps
x_dim	Width of input image data
x_qpt	Q-format used by the input pixel array
h_qpt	Q-format used by the filter mask array
y_qpt	Q-format to be used for the output array

Description

The function performs a generalized correlation with a 1 by 'm' tap filter. It can be called repetitively to form an arbitrary 'm x n' 2D generalized correlation kernel. The input data, mask data and output data are in Q-formats. The data type of input image array and output is int where as it is short for mask array. The intermediate correlation sum is accumulated to a 64-bit value in an intermediate Q-format. This sum is shifted by a suitable value to get the final output in the specified output Q-format. If the width of the input image is x_dim and the mask is m then the output array must have at-least a dimension of (x_dim - m). Overflow may occur while accumulating the intermediate sum in 64-bits or while converting the intermediate sum to the final sum in 32-bits. In either of the cases, no saturation will be performed by this function. However assumptions on filter gain are made to avoid overflow.

Algorithm

This is the C code implementation without any restrictions. However, intrinsic code has restrictions as listed in the special requirements:

```
void IMG_corr_gen_iq
(
const        int      *restrict x,
const        short      *restrict h,
             int      *restrict y,
int               m,
int            x_dim,
int            x_qpt,
int            h_qpt,
int            y_qpt
)
{
    int          i,             j;
    int          q_pt;
    long long    temp_y;
    q_pt = x_qpt + h_qpt - y_qpt;
    for (j = 0; j < x_dim - m; j++)
    {
        temp_y = 0;
          for (i = 0; i < m; i++)
            {
            temp_y  += x[i + j] * h[i];
            }
temp_y >>= q_pt;
        y[j]     = (int)temp_y;
      }
}
```

Special Requirements

- Length of filter (m) should be a minimum of 2 and also multiple of 2
- Minimum value of 'm' is 2.
- Minimum value of 'x_dim' is 'm' + 2.
- The following assumption is made on the Q-formats: $y\_qpt \leq x\_qpt + h\_qpt$. .
- Both the input arrays and the output array should be double-word aligned
- The input and output matrices should not overlap
- Internal accuracy of the computations is 64 bits. To ensure correctness on 32-bit input data, the maximum permissible filter-gain in bits is 32-bits (i.e., the cumulative sum of the absolute values of the filter coefficients should not exceed $2^{32} - 1$) .

Implementation Notes

- The inner loop is unrolled twice and two output values are calculated per iteration of the outer loop.
- Saturation is performed appropriately at all stages of computation.
- Code is LITTLE ENDIAN.

Compatibility   Compatible for both C64x and C64x+.

6.21 IMG_median_3x3_16s

IMG_median_3x3_16s *3x3 Median Filtering for 16-bit input*

Syntax	void **IMG_median_3x3_16s** (const short *restrict i_data, int n, short *restrict o_data)

Arguments

i_data	Pointer to input array of size 3 x n
n	Width of the input image
o_data	Pointer to output array of size 1 x n

Description This function performs a 3x3 median filter operation on 16-bit signed values. The median filter comes under the class of non-linear signal processing algorithms. The grey level at each pixel is replaced by the median of the nine neighboring values. The median of a set of nine numbers is the middle element so that half of the elements in the list are larger and half are smaller. The i_data points to an array which consists of three rows of pixel values. The median value is calculated corresponding to the middle row of i_data, and written into memory location pointed by o_data. The first two values in the output array will not be containing any meaningful data. The 3rd value in the output array will be the median of 2nd value in the middle row of input array and so on. The nth value in the output array will be the median of the (n-1)th value in the mid row of input array. Hence, the output array will not contain the median values corresponding to first and last elements in the middle row of input image. .

Algorithm The algorithm processes a 3x3 region as three 3-element columns, incrementing through the columns in the image. Each column of data is first sorted into MAX, MED, and MIN values, resulting in the following arrangement:

Column 0	Column 1	Column 2	Columnwise sorted
Prev_col0_0	Prev_col1_0	Cur_col_0	MAX
Prev_col0_1	Prev_col1_1	Cur_col_1	MED
Prev_col0_2	Prev_col1_2	Cur_col_2	MIN

After sorting all the three columns in the descending order specified above, The MIN of MAX (i.e., row 0) , the MEDium of MEDium values (row 1) and MAX of MIN values (row 2) is taken and their MEDium value is calculated to get the median of the above considered 3x3 region. After this, the pointers moves by a column, that is, prev_col1 becomes prev_col0, cur_col becomes prev_col1, and a new column is loaded into cur_col.

Special Requirements

- The minimum value for width of input image 'n' is 4.
- Width of input image 'n' should be a multiple of 4.
- Input and output arrays must be double word aligned.
- Input and output arrays should not overlap .

Implementation Notes

- The loop is manually unrolled by two and further unrolled twice using pragma directives to the compiler, resulting in a total unroll of four.
- Four output pixels are calculated per iteration, after unrolling is taken into consideration.
- Valid output starts from third element to nth element in the output array which corresponds to median values starting from second element to 'n - 1'th element in the middle row of input.
- Code is LITTLE ENDIAN.

Compatibility Compatible for both C64x and C64x+.

6.22   IMG_median_3x3_16

IMG_median_3x3_16 *3x3 Median Filtering for 16-bit input*

Syntax	void **IMG_median_3x3_16** (const short *restrict i_data, int n, short *restrict o_data)

Arguments

	i_data	Pointer to input array of size 3 x n
	n	Width of the input image
	o_data	Pointer to output array of size 1 x n

Description This kernel performs a 3x3 median filter operation on 16-bit unsigned values. The median filter comes under the class of non-linear signal processing algorithms. Rather than replace the grey level at a pixel by a weighted average of the nine pixels including and surrounding it, the grey level at each pixel is replaced by the median of the nine values. The median of a set of nine numbers is the middle element so that half of the elements in the list are larger and half are smaller. Median filters remove the effects of extreme values from data, such as salt and pepper noise, although using a wide filter may result in unacceptable blurring of sharp edges in the original image.

Algorithm The algorithm is same as IMG_median_3x3_16s, with the difference of unsigned input.

Special Requirements

- The length 'n' must be a multiple of four

Memory Notes

- No bank conflicts occur.
- No alignment restrictions on input/output buffers.

Compatibility Compatible for both C64x and C64x+.

6.23 *IMG_yc_demux_be16_16*

IMG_yc_demux_be16_16 *YCbCR Demultiplexing (16-bit big endian source)*

Syntax	void **IMG_yc_demux_be16_16** (int n, const unsigned short * yc, short *restrict y, short *restrict cr, short *restrict cb);

Arguments

n	Number of luma points. Must be multiple of 16.
yc	Packed luma/chroma inputs. Must be double-word aligned.
y	Unpacked luma data. Must be double-word aligned.
cr	Unpacked chroma r data. Must be double-word aligned.
cb	Unpacked chroma b data. Must be double-word aligned.

Description   The input array 'yc' is expected to be an interleaved 4:2:2 video stream. The input is expected in BIG ENDIAN byte order within each 4-byte word. This is consistent with reading the video stream from a word-oriented BIG ENDIAN device while the C6000 device is in a LITTLE ENDIAN configuration. In other words, the expected pixel order is:

```
             Word 0          Word 1          Word 2
       +--------------+--------------+--------------+--
Byte#  | 0   1   2   3| 4   5   6   7| 8   9  10  11|...
       |  cb0      y1 |  cr0      y0 |  cb1      y2 |...
       +--------------+--------------+--------------+--
```

The output arrays 'y', 'cr', and 'cb' are expected to not overlap. The de-interleaved pixels are written as half-words in LITTLE ENDIAN order.

This function reads the halfword-oriented pixel data, zero-extends it, and then writes it to the appropriate result array. Both the luma and chroma values are expected to be unsigned. The data is expected to be in an order consistent with reading byte oriented data from a word-oriented peripheral that is operating in BIG ENDIAN mode, while the CPU is in LITTLE ENDIAN mode. This results in a pixel ordering which is not immediately obvious. This function correctly reorders the pixel values so that further processing may proceed in LITTLE ENDIAN mode.

Algorithm

```
void IMG_yc_demux_be16_16_c
(
    int n,                  /* Number of luma pixels    */
    const unsigned short *yc, /* Interleaved luma/chroma  */
    short *restrict y,      /* Luma plane (16-bit)      */
    short *restrict cr,     /* Cr chroma plane (16-bit) */
    short *restrict cb      /* Cb chroma plane (16-bit) */
)
{
    int i;

    for (i = 0; i < (n >> 1); i++)
    {
        /*  0   1   2   3  */
        /* cb0 y1  cr0  y0 */

        y[2*i+0] = yc[4*i + 3];
        y[2*i+1] = yc[4*i + 1];
        cr[i]    = yc[4*i + 2];
        cb[i]    = yc[4*i + 0];
    }
}
```

Special Requirements

- Input and output arrays are double-word aligned.
- The input must be a multiple of 16 luma pixels long.

Compatibility   Compatible for both C64x and C64x+.

6.24 IMG_yc_demux_le16_16

IMG_yc_demux_le16_16 *YCbCR Demultiplexing (16-bit little endian source*

Syntax	void **IMG_yc_demux_le16_16** (int n, const unsigned short * yc, short *restrict y, short *restrict cr, short *restrict cb)

Arguments

n	Number of luma points. Must be multiple of 16.
yc	Packed luma/chroma inputs. Must be double-word aligned.
y	Unpacked luma data. Must be double-word aligned.
cr	Unpacked chroma r data. Must be double-word aligned.
cb	Unpacked chroma b data. Must be double-word aligned.

Description   The input array 'yc' is expected to be an interleaved 4:2:2 video stream. The input is
expected in LITTLE ENDIAN byte order within each 4-byte word. This is consistent with
reading the video stream from a word-oriented LITTLE ENDIAN device while the C6000
device is in a LITTLE ENDIAN configuration. In other words, the expected pixel order is:

```
            Word 0            Word 1            Word 2
      +----------------+----------------+----------------+--
Byte# | 0    1 | 2    3 | 4    5 | 6    7 | 8    9 | 10   11 |...
      |   y0   |   cr0  |   y1   |   cb0  |   y2   |   cr2  |...
      +----------------+----------------+----------------+--
```

The output arrays 'y', 'cr', and 'cb' are expected to not overlap. The de-interleaved pixels
are written as half-words in LITTLE ENDIAN order. **Note:** Please see the IMGLIB
function IMB_yc_demux_be16_16 for code which handles input coming from a BIG
ENDIAN device.

This function reads the halfword-oriented pixel data, zero-extends it, and then writes it to
the appropriate result array. Both the luma and chroma values are expected to be
unsigned. The data is expected to be in an order consistent with reading byte-oriented
data from a word-oriented peripheral that is operating in LITTLE ENDIAN mode, while
the CPU is in LITTLE ENDIAN mode. This function unpacks the byte-oriented data so
that further processing may proceed in LITTLE ENDIAN mode.

Special Requirements

- Input and output arrays are double-word aligned.
- The input must be a multiple of 16 luma pixels long.

Compatibility   Compatible for both C64x and C64x+.

7   Compression/Decompression IMGLIB2 Reference

This section provides a list of the routines within the IMGLIB organized into functional categories. The functions within each category are listed in alphabetical order and include arguments, descriptions, algorithms, benchmarks, and special requirements.

7.1   *IMG_fdct_8x8*

IMG_fdct_8x8	*Forward Discrete Cosine Transform (FDCT)*

Syntax   void **IMG_fdct_8x8**(short *fdct_data, unsigned num_fdcts)

Arguments

	fdct_data	Pointer to `num_fdct' 8x8 blocks of image data. Must be double-word aligned.
	num_fdcts	Number of FDCTs to perform. Note that IMG_fdct_8x8 requires exactly `num_fdcts' blocks of storage starting at the location pointed to by `fdct_data', since the transform is executed completely in place.

Description   This routine implements the forward discrete cosine transform (FDCT). Output values are rounded, providing improved accuracy. Input terms are expected to be signed 11Q0 values, producing signed 15Q0 results. A smaller dynamic range may be used on the input, producing a correspondingly smaller output range. Typical applications include processing signed 9Q0 and unsigned 8Q0 pixel data, producing signed 13Q0 or 12Q0 outputs, respectively. No saturation is performed.

Algorithm   The FDCT is described by the following equation:

$$Iuv = \frac{\alpha u \alpha v}{4}$$

$$\sum_{x=0}^{7} \sum_{y=0}^{7} ixy \cos\left(\frac{2x+1u\pi}{16}\right) \cos\left(\frac{2y+1v\pi}{16}\right)$$

where

$$z = 0 \Rightarrow \alpha z = \frac{1}{\sqrt{2}}$$

$$z \neq 0 \Rightarrow \alpha z = 1$$

i(x,y) : pixel values (spatial domain)

I(u,v) : transform values (frequency domain)

This particular implementation uses the Chen algorithm for expressing the FDCT. Rounding is performed to provide improved accuracy.

Special Requirements

- The fdct_data[ ] array must be aligned on a double-word boundary.
- Stack must be aligned on a double-word boundary.
- Input terms are expected to be signed 11Q0 values; i.e., in the range [-512,511], producing signed 15Q0 results. Larger inputs may result in overflow.
- The IMG_fdct_8x8 routine accepts a list of 8x8 pixel blocks and performs FDCTs on each. Pixel blocks are stored contiguously in memory. Within each pixel block, pixels are expected in left-to-right, top-to-bottom order.
- Results are returned contiguously in memory. Within each block, frequency domain terms are stored in increasing horizontal frequency order from left to right, and increasing vertical frequency order from top to bottom.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible. Interrupts are blocked out only in branch delay slots.
- The code is set up to provide an early exit if it is called with num_fdcts = 0. In that situation, it will run for 13 cycles.
- Both vertical and horizontal loops have been software pipelined.
- For performance, portions of the optimized assembly code outside the loops have been interscheduled with the prolog and epilog code of the loops. Also, twin stack pointers are used to accelerate stack accesses. Finally, pointer values and cosine term registers are reused between the horizontal and vertical loops to reduce the impact of pointer and constant re-initialization.
- To save code size, prolog and epilog collapsing have been performed in the optimized assembly code to the extent that it does not impact performance.
- To reduce register pressure and save code, the horizontal loop uses the same pair of pointer registers for both reading and writing. The pointer increments are on the loads to permit prolog and epilog collapsing, since loads can be speculated.

7.2 IMG_idct_8x8_12q4

IMG_idct_8x8_12q4 *Inverse Discrete Cosine Transform(IDCT)*

Syntax void **IMG_idct_8x8_12q4**(short *idct_data, unsigned num_idcts)

Arguments

idct_data	Pointer to `num_idcts' 8x8 blocks of DCT coefficients. Must be double-word aligned.
num_idcts	Number of IDCTs to perform.

Description This routine performs an IEEE 1180-1990 compliant IDCT, including rounding and saturation to signed 9-bit quantities. The input coefficients are assumed to be signed 16-bit DCT coefficients in 12Q4 format.

This function performs a series of 8×8 IDCTs on a list of 8x8 blocks.

Algorithm The IDCT is described by the following equation:

$$ixy = \frac{1}{4} = \sum_{u=0}^{7} \sum_{v=0}^{7} luv \cos\left(\frac{2x+1u\pi}{16}\right) \cos\left(\frac{2y+1v\pi}{16}\right)$$

where

$$z = 0 \Rightarrow \alpha z = \frac{1}{\sqrt{2}}$$
$$z \neq 0 \Rightarrow \alpha z = 1$$

i(x,y) : pixel values (spatial domain)

i(x,y) : pixel values (spatial domain)

I(u,v) : transform values (frequency domain)

This particular implementation uses the Even-Odd decomposition algorithm for expressing the IDCT. Rounding is performed so that the result meets the IEEE 1180-1990 precision and accuracy specification.

Special Requirements

- The idct_data[ ] array must be aligned on a double-word boundary.
- Input DCT coefficients are expected to be in the range +2047 to -2048 inclusive. Output terms are saturated to the range +255 to -256 inclusive; i.e., inputs are in a 12Q4 format and outputs are saturated to a 9Q0 format.
- The code is set up to provide an early exit if it is called with num_idcts = 0. In this situation, it will run for 13 cycles.
- The routine accepts a list of 8x8 DCT coefficient blocks and performs IDCTs on each. Coefficient blocks are stored contiguously in memory. Within each block, frequency domain terms are stored in increasing horizontal frequency order from left to right, and increasing vertical frequency order from top to bottom.
- Results are returned contiguously in memory. Within each pixel block, pixels are returned in left-to-right, top-to-bottom order.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible and fully re-entrant.
- All levels of looping are collapsed into single loops which are pipelined. The outer loop focuses on 8-pt IDCTs, whereas the inner loop controls the column-pointer to handle jumps between IDCT blocks. (The column-pointer adjustment is handled by a four-phase rotating *fix-up* constant which takes the place of the original inner-loop.)
- For performance, portions of the outer-loop code have been inter-scheduled with the

prologs and epilogs of both loops. Finally, cosine term registers are reused between the horizontal and vertical loops to save the need for re-initialization.

- To save code size, prolog and epilog collapsing have been performed to the extent that performance is not affected. The remaining prolog and epilog code has been inter-scheduled with code outside the loops to improve performance.
- The code may perform speculative reads of up to 128 bytes beyond the end of the IDCT array. The speculatively accessed data is ignored.

7.3 *IMG_mad_8x8*

IMG_mad_8x8	***8x8 Minimum Absolute Difference***

Syntax

void **IMG_mad_8x8**(const unsigned char * restrict ref_data, const unsigned char * restrict src_data, int pitch, int sx, int sy, unsigned int * restrict match)

Arguments

	*ref_data	Pointer to a pixel in a reference image which constitutes the top-left corner of the area to be searched. The dimensions of the search area are given by (sx + 8) x (sy + 8).
	src_data[8*8]	Pointer to 8×8 source image pixels. Must be word aligned.
	pitch	Width of reference image.
	sx	Horizontal dimension of the search space.
	sy	Vertical dimension of the search space.
	match[2]	Result. Must be word aligned. match[0]: Packed best match location. The upper half-word contains the horizontal pixel position and the lower half-word the vertical pixel position of the best matching 8×8 block in the search area. The range of the coordinates is [0,sx-1] in the horizontal dimension and [0,sy-1] in the vertical dimension, where the location (0,0) represents the top-left corner of the search area. match[1]: Minimum absolute difference value at the best match location.

Description

This routine locates the position of the top-left corner of an 8×8 pixel block in a reference image which most closely matches the 8×8 pixel block in src_data[ ], using the sum of absolute differences metric. The source image block src_data[ ] is moved over a range that is sx pixels wide and sy pixels tall within a reference image that is pitch pixels wide. The pointer *ref_data points to the top-left corner of the search area within the reference image. The match location as well as the minimum absolute difference value for the match are returned in the match[2] array. The search is performed in top-to-bottom, left-to-right order, with the earliest match taking precedence in the case of ties.

Algorithm

Behavioral C code for the routine is provided below: The assembly implementation has restrictions as noted under Special Requirements.

```
void IMG_mad_8×8
(
    const unsigned char *restrict refImg,
    const unsigned char *restrict srcImg,
    int pitch, int sx, int sy,
    unsigned int *restrict match
)
{
    int i, j, x, y, matx, maty;
    unsigned matpos, matval;

    matval = ~0U;
    matx   = maty = 0;

    for (x = 0; x < sx; x++)
        for (y = 0; y < sy; y++)
        {
            unsigned acc = 0;

            for (i = 0; i < 8; i++)
                for (j = 0; j < 8; j++)
                    acc += abs(srcImg[i*8 + j] -
                        refImg[(i+y)*pitch + x + j]);
```

```
                        if (acc < matval)
                        {
                            matval = acc;
                            matx   = x;
                            maty   = y;
                        }
                    }

            matpos     = (0xffff0000 & (matx << 16)) |
                         (0x0000ffff & maty);
            match[0] = matpos;
            match[1] = matval;

        }
```

Special Requirements

- It is assumed that src_data[ ] and ref_data[ ] do not alias in memory.
- The arrays src_data[ ] and match[ ] must be word aligned.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- The inner loops that perform the 8x8 MADs are completely unrolled and the outer two loops are collapsed together. In addition, all source image data is preloaded into registers.
- The data required for any one row is brought in using nonaligned loads. SUBABS4 and DOTPU4 are used together to do the MAD computation.
- To save instructions and fit within an 8 cycle loop, the precise location of a given match is not stored. Rather, the loop iteration that it was encountered on is stored. A short divide loop after the search loop converts this value into X and Y coordinates of the location.
- The inner loop comprises 64 instructions that are executed in 8 cycles, with 64 absolute differences accumulated in a single iteration. The source pixels are pre-read into registers. Thus, this code executes 8 instructions per cycle, and computes 8 absolute differences per cycle.

7.4 *IMG_mad_16x16*

IMG_mad_16x16	*16×16 Minimum Absolute Difference*

Syntax

void **IMG_mad_16×16** (const unsigned char * restrict ref_data, const unsigned char * restrict src_data, int pitch, int sx, int sy, unsigned int * restrict match)

Arguments

	*ref_data	Pointer to a pixel in a reference image which constitutes the top-left corner of the area to be searched. The dimensions of the search area are given by (sx + 16) x (sy + 16).
	src_data[16*16]	Pointer to 16x16 source image pixels.
	pitch	Width of reference image.
	sx	Horizontal dimension of the search space.
	sy	Vertical dimension of the search space.
	match[2]	Result. match[0]: Packed best match location. The upper half-word contains the horizontal pixel position and the lower half-word the vertical pixel position of the best matching 16x16 block in the search area. The range of the coordinates is [0,sx-1] in the horizontal dimension and [0,sy-1] in the vertical dimension, where the location (0,0) represents the top-left corner of the search area. match[1]: Minimum absolute difference value at the best match location.

Description

This routine locates the position of the top-left corner of an 16×16 pixel block in a reference image which most closely matches the 16×16 pixel block in src_data[ ], using the sum of absolute differences metric. The source image block src_data[ ] is moved over a range that is sx pixels wide and sy pixels tall within a reference image that is pitch pixels wide. The pointer *ref_data points to the top-left corner of the search area within the reference image. The match location and the minimum absolute difference value for the match are returned in the match[2] array.

Algorithm

Behavioral C code for the routine is provided below: The assembly implementation has restrictions as noted under Special Requirements.

```
void IMG_mad_16×16
(
    const unsigned char *restrict refImg,
    const unsigned char *restrict srcImg,
    int pitch, int sx, int sy,
    unsigned int *restrict match
)
{
    int i, j, x, y, matx, maty;
    unsigned matpos, matval;

    matval = ~0U;
    matx   = maty = 0;

    for (x = 0; x < sx; x++)
        for (y = 0; y < sy; y++)
        {
            unsigned acc = 0;

            for (i = 0; i < 16; i++)
                for (j = 0; j < 16; j++)
                    acc += abs(srcImg[i*16 + j] -
                            refImg[(i+y)*pitch + x + j]);

            if (acc < matval)
            {
```

```
                        matval = acc;
                        matx   = x;
                        maty   = y;
                }
        }

        matpos   = (0xffff0000 & (matx << 16)) |
                   (0x0000ffff & maty);
        match[0] = matpos;
        match[1] = matval;
}
```

Special Requirements

- It is assumed that src_data[ ] and ref_data[ ] do not alias in memory.
- sy must be a multiple of 2.
- There are no alignment restrictions.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is fully interruptible.
- The two outer loops are merged, as are the two inner loops. The inner loop process 2 lines of 2 search locations in parallel.
- The search is performed in top-to-bottom, left-to-right order, with the earliest match taking precedence in the case of ties.
- Further use is made of SUBABS4 and DOTPU4. The SUBABS4 takes the absolute difference on four 8 bit quantities packed into a 32 bit word. The DOTPU4 performs four 8 bit wide multiplies and adds the results together.

7.5 *IMG_mpeg2_vld_intra*

IMG_mpeg2_vld_intra *MPEG-2 Variable Length Decoding of Intra MBs*

Syntax	void **IMG_mpeg2_vld_intra**(const short *restrict Wptr, short *restrict outi, IMG_mpeg2_vld *restrict Mpeg2v, int dc_pred[3], int mode_12Q4, int num_blocks, int bsbuf_words)

Arguments

Wptr[]	Pointer to array that contains quantization matrix. The elements of the quantization matrix in Wptr[] must be ordered according to the scan pattern used (zigzag or alternate scan). Video format 4:2:0 requires one quantization matrix of 64 array elements. For formats 4:2:2 and 4:4:4, two quantization matrices, one for luma and one for chroma, must be specified in the array now containing 128 array elements.
outi[6*64]	Pointer to the context object containing the coding parameters of the MB to be decoded and the current state of the bitstream buffer. Tthe structure is described below.
Mpeg2v	Pointer to the context structure containing the coding parameters of the MB to be decoded and the current state of the bitstream buffer.
dc_pred[3]	Intra DC prediction array. The first element of dc_pred is the DC prediction for Y, the second for Cr, and the third for Cb.
mode_12Q4	0: Coefficients are returned in normal 16-bit integer format. Otherwise: Coefficients are returned in 12Q4 format (normal 16-bit integer format left shifted by 4). This mode is useful for directly passing the coefficients into the IMG_idct_8x8_12q4 routine.
num_blocks	Number of blocks that the MB contains. Valid values are 6 for 4:2:0, 8 for 4:2:2, and 12 for 4:4:4 format.
bsbuf_words	Size of bitstream buffer in words. Must be a power of 2. Bitstream buffer must be aligned at an address boundary equal to its size in bytes because the bitstream buffer is addressed circularly by this routine.

Description   This routine takes a bitstream of an MPEG-2 intra coded macroblock (MB) and returns the decoded IDCT coefficients. The routine checks the coded block pattern (cbp) and performs DC and AC coefficient decoding including variable length decode, run-length expansion, inverse zigzag ordering, de-quantization, saturation, and mismatch control. An example program is provided that illustrates the usage of this routine. The structure IMG_mpeg2_vld is defined as follows:

```
typedef struct {
    unsigned int  *bsbuf;        // pointer to bitstream buffer
    unsigned int  next_wptr;     // next word to read from buffer
    unsigned int  bptr;          // bit position within word
    unsigned int  word1;         // word aligned buffer
    unsigned int  word2;         // word aligned buffer
    unsigned int  top0;          // top 32 bits of bitstream
    unsigned int  top1;          // next 32 bits of bitstream
    unsigned char *scan;         // inverse zigzag scan matrix
    unsigned int  intravlc;      // intra_vlc_format
    unsigned int  quant_scale;   // quantiser_scale
    unsigned int  dc_prec;       // intra_dc_precision
    unsigned int  cbp;           // coded_block_pattern
    unsigned int  fault;         // fault condition (returned)
} IMG_mpeg2_vld;
```

The Mpeg2v variables should have a fixed layout because they are accessed by this routine. If the layout is changed, the corresponding changes also have to be made in

code.

The routine sets the fault flag Mpeg2v.fault to 1 if an invalid VLC code was encountered or the total run went beyond 63. In these situations, the decoder has to resynchronize.

Before calling the routine, the bitstream variables in Mpeg2v have to be initialized. If bsbuf is a circular buffer and bsptr contains the number of bits in the buffer that have already been consumed, then next_wptr, bptr, word1, word2, top0 and top1 are initialized as follows:

1. next_wptr: bsptr may not be a multiple of 32, therefore it is set to the next lower multiple of 32.

    ```
    next_wptr = (bsptr >> 5);
    ```

2. bptr: bptr is the bit pointer that points to the current bit within the word pointed to by next_wptr.

    ```
    bptr = bsptr & 31;
    bptr_cmpl = 32 - bptr;
    ```

3. word1 and word2: Read the next 3 words from the bitstream buffer bsbuf. bsbuf_words is the size of the bitstream buffer in words (word0 is a temporary variable not passed in Mpeg2v).

    ```
    word0 = bsbuf[next_wptr];
    next_wptr = (next_wptr+1) & (bsbuf_words -1);
    word1 = bsbuf[next_wptr];
    next_wptr = (next_wptr+1) & (bsbuf_words -1);
    word2 = bsbuf[next_wptr];
    next_wptr = (next_wptr+1) & (bsbuf_words -1);
    ```

4. top0 and top1: Shift words word0, word1, word2 by bptr to the left so that the current bit becomes the left-most bit in top0 and top0 and top1 contain the next 64 bits to be decoded.

    ```
    s1 = word0 << bptr;
    s2 = word1 >> bptr_cmpl; /*unsigned shift*/
    top0 = s1 + s2;
    s3 = word1<< bptr;
    s4 = word2 >> bptr_cmpl; /*unsigned shift*/
    top1 = s3 + s4;
    ```

    Note that the routine returns the updated state of the bitstream buffer variables, top0, top1, word1, word2, bptr and next_wptr. If all other functions which access the bitstream in a decoder system maintain the buffer variables in the same way, then the above initialization procedure only has to be performed once at the beginning.

Algorithm   This routine is implemented as specified in the MPEG-2 standard text (ISO/IEC 13818-2).

Special Requirements

- The bitstream must be stored in memory in 32-bit words in little Endian byte order.
- Wptr is allowed to overrun once to detect if a decoded run causes the total run to exceed 63. The maximum overrun that can occur is the error mark 66 because it is the highest value that can be decoded for a run value. Therefore, 67 half-words behind the weighting matrix array should be memory locations whose read access does not cause any side effects, such as peripherals.
- Note that the AMR register is set to zero on exit.

Notes

- Bank Conflicts: No bank conflicts occur.
- This code is LITTLE ENDIAN.
- Interruptibility: This code is interrupt-tolerant but not interruptible.
- The instruction NORM is used to detect the number of leading zeros or ones in a code word. This value, together with additional bits extracted from the code word, is then used as an index into lookup tables to determine the length, run, level, and sign. Escape code sequences are directly extracted from the code word.
- DC coefficients are decoded without lookup tables by exploiting the relatively simple relationship between the number of leading zeros and dc_size and the length of the code word.

7.6 IMG_mpeg2_vld_inter

IMG_mpeg2_vld_inter *MPEG-2 Variable Length Decoding of Inter MBs*

Syntax	void **IMG_mpeg2_vld_inter**(const short *Wptr, short *outi, IMG_mpeg2_vld *Mpeg2v, int mode_12Q4, int num_blocks, int bsbuf_words)

Arguments

Wptr[]	Pointer to array that contains quantization matrix. The elements of the quantization matrix in Wptr[] must be ordered according to the scan pattern used (zigzag or alternate scan). Video format 4:2:0 requires one quantization matrix of 64 array elements. For formats 4:2:2 and 4:4:4, two quantization matrices, one for luma and one for chroma, must be specified in the array now containing 128 array elements.
outi[6*64]	Pointer to the IDCT coefficients output array (6*64 elements), elements must be set to zero prior to function call.
Mpeg2v	Pointer to the context object containing the coding parameters of the MB to be decoded and the current state of the bitstream buffer. The structure is described below.
mode_12Q4	0: Coefficients are returned in normal 16-bit integer format. Otherwise: Coefficients are returned in 12Q4 format (normal 16-bit integer format left shifted by 4). This mode is useful for directly passing the coefficients into the IMG_idct_8x8_12q4 routine.
num_blocks	Number of blocks that the MB contains. Valid values are 6 for 4:2:0, 8 for 4:2:2, and 12 for 4:4:4 format.
bsbuf_words	Size of bitstream buffer in words. Must be a power of 2. Bitstream buffer must be aligned at an address boundary equal to its size in bytes because the bitstream buffer is addressed circularly by this routine.

Description	This routine takes a bitstream of an MPEG-2 non-intra coded macroblock (MB) and returns the decoded IDCT coefficients. The routine checks the coded block pattern (cbp) and performs coefficient decoding including variable length decode, run-length expansion, inverse zigzag ordering, de-quantization, saturation, and mismatch control. An example program is provided illustrating the usage of this routine.
	See the description of the IMG_mpeg2_vld_intra routine for further information about the usage of this routine.
Algorithm	This routine is implemented as specified in the MPEG-2 standard text (ISO/IEC 13818-2).

Special Requirements

- The bitstream must be stored in memory in 32-bit words which are in little Endian byte order.
- Wptr is allowed to overrun once to detect if a decoded run causes the total run to exceed 63. The maximum overrun that can occur is the error mark 66 because it is the highest value that can be decoded for a run value. Therefore, 67 half-words behind the weighting matrix array should be memory locations whose read access does not cause any side effects, such as peripherals.
- Note that the AMR register is set to zero on exit.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: This code is LITTLE ENDIAN.
- Interruptibility: This code is interrupt-tolerant but not interruptible.
- The instruction NORM is used to detect the number of leading zeros or ones in a code word. This value, together with additional bits extracted from the codeword, is then used as an index into lookup tables to determine the length, run, level, and sign. Escape code sequences are directly extracted from the code word.
- The special case of the first coefficient of a block is handled by modifying the prolog of the decoding loop.

7.7 *IMG_quantize*

IMG_quantize	*Matrix Quantization With Rounding*

Syntax

void **IMG_quantize** (short *data, int num_blks, int blk_size, const short *recip_tbl, int q_pt)

Arguments

	data[ ]	Pointer to data to be quantized. Must be double-word aligned and contain num_blks * blk_size elements.
	num_blks	Number of blocks to be processed. May be zero.
	blk_size	Block size. Must be multiple of 16 and ≥ 32
	recip_tbl[ ]	Pointer to quantization values (reciprocals) . Must be double-word aligned and contain blk_size elements.
	q_pt	Q-point of quantization values. 0 3 q_pt ≤ 31

Description

This routine quantizes a list of blocks by multiplying their contents with a second block of values that contains reciprocals of the quantization terms. This step corresponds to the quantization that is performed in 2-D DCT-based compression techniques, although the routine may be used on any signed 16-bit data using signed 16-bit quantization terms.

The routine merely multiplies the contents of the quantization array recip_tbl[ ] with the data array data[ ]. Therefore, it may be used for inverse quantization as well, by setting the Q-point appropriately.

Algorithm

Behavioral C code for the routine is provided below:

```
void IMG_quantize (short *data, int num_blks, int  blk_size, const short
*recip_tbl, int q_pt)
{
    short recip;
    int i, j, k, quot, round;

    round = q_pt ? 1 << (q_pt - 1) : 0;

    for (i = 0; i < blk_size; i++)
    {
        recip = recip_tbl[i];
        k = i;

        for (j = 0; j < num_blks; j++)
        {
            quot = data[k] * recip + round;
            data[k] = quot >> q_pt;
            k += blk_size;
        }
    }
}
```

Special Requirements

- The number of blocks, num_blks, may be zero.
- The block size, blk_size, must be at least 32 and a multiple of 16.
- The Q-point, q_pt, controls rounding and final truncation; it must be in the range $0 \leq$ q_pt ≤ 31.
- Both input arrays, data[ ] and recip_tbl[ ], must be double-word aligned.
- The data[ ] array must contain num_blks * blk_size elements, and the recip_tbl[ ] array must contain blk_size elements.

Notes

- Bank Conflicts: No bank conflicts occur, regardless of the relative orientation of recip_tbl[ ] and data[ ].
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: This code is fully interruptible, with a maximum interrupt latency of 16 cycles due to branch delay slots.
- The outer loop is unrolled 16 times to allow greater amounts of work to be performed in the inner loop. The resulting loop-nest is then collapsed and pipelined as a single loop, since the code is not bottlenecked on bandwidth.
- Reciprocals and data terms are loaded in groups of four with double-word loads, making the best use of the available memory bandwidth.
- SSHVR is used in the M-unit to avoid an S-unit bottleneck.
- Twin stack pointers are used to speed up stack accesses.

7.8 IMG_sad_8x8

IMG_sad_8x8   *Sum of Absolute Differences on Single 8×8 Block*

Syntax

unsigned **IMG_sad_8×8**(const unsigned char * restrict srcImg, const unsigned char * restrict refImg, int pitch)

Arguments

srcImg[64]	8×8 source block. Must be double-word aligned.
refImg[]	Reference image.
pitch	Width of reference image.

Description

This function returns the sum of the absolute differences between the source block and the 8×8 region pointed to in the reference image.

The code accepts a pointer to the 8×8 source block (srcImg), and a pointer to the upper-left corner of a target position in a reference image (refImg). The width of the reference image is given by the pitch argument.

Algorithm

Behavioral C code for the routine is provided below:

```
unsigned sad_8×8
(
    const unsigned char *restrict srcImg,
    const unsigned char *restrict refImg,
    int pitch
)
{
    int i, j;
    unsigned sad = 0;

    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            sad += abs(srcImg[j+i*8] – refImg[j+i*pitch]);

    return sad;
}
```

Special Requirements

- The array srcImg[64] must be aligned at a double-word boundary.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is ENDIAN NEUTRAL.
- Interruptibility: The code is fully interruptible.

7.9   IMG_sad_16x16

IMG_sad_16x16	***Sum of Absolute Differences on Single 16×16 Block***

Syntax

unsigned **IMG_sad_16×16**(const unsigned char * restrict srcImg, const unsigned char * restrict refImg, int pitch)

Arguments

srcImg[256]	16×16 source block. Must be double-word aligned.
refImg[]	Reference image.
pitch	Width of reference image.

Description

This function returns the sum of the absolute differences between the source block and the 16×16 region pointed to in the reference image.

The code accepts a pointer to the 16×16 source block (srcImg), and a pointer to the upper-left corner of a target position in a reference image (refImg). The width of the reference image is given by the pitch argument.

Algorithm

Behavioral C code for the routine is provided below:

```
unsigned sad_16×16
(
    const unsigned char *restrict srcImg,
    const unsigned char *restrict refImg,
    int pitch
)
{
    int i, j;
    unsigned sad = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sad += abs(srcImg[j+i*16] - refImg[j+i*pitch]);

    return sad;
}
```

Special Requirements

- The array srcImg[256] must be aligned at a double-word boundary.

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is ENDIAN NEUTRAL.
- Interruptibility: The code is fully interruptible.

7.10 IMG_wave_horz

IMG_wave_horz	***Horizontal Wavelet Transform***

Syntax

void **IMG_wave_horz** (const short * restrict in_data, const short * restrict qmf, const short * restrict mqmf, short * restrict out_data, int cols)

Arguments

in_data[cols]	Pointer to one row of input pixels. Must be word aligned.
qmf[8]	Pointer to Q.15 qmf filter-bank for low-pass filtering. Must be double-word aligned.
mqmf[8]	Pointer to Q.15 mirror qmf filter bank for high-pass filtering. Must be double-word aligned.
out_data[cols]	Pointer to row of reference/detailed decimated outputs.
cols	Number of columns in the input image. Must be multiple of 2 and ≥ 8.

Description

This routine performs a 1-D Periodic Orthogonal Wavelet decomposition. It also performs the row decomposition component of a 2-D wavelet transform. An input signal x[n] is low pass and high pass filtered and the resulting signals are decimated by a factor of two. This results in a reference signal r1[n] which is the decimated output obtained by dropping the odd samples of the low pass filter output, and a detail signal d[n] obtained by dropping the odd samples of the highpass filter output. A circular convolution algorithm is implemented, so the wavelet transform is periodic. The reference signal and the detail signal are each half the size of the original signal.

Algorithm

Behavioral C code for the routine wave_horz is provided below:

```
void IMG_wave_horz
(
    const short *restrict in_data,  /* Row of input pixels  */
    const short *restrict qmf,      /* Low-pass QMF filter  */
    const short *restrict mqmf,     /* High-pass QMF filter */
    short       *restrict out_data, /* Row of output data   */
    int                   cols      /* Length of input.     */
);

{
    int    i, res, iters;
    int    j, sum, prod;
    short *xptr  = in_data;
    short *yptr  = out_data;
    short *x_end = &in_data[cols - 1];
    short  xdata, hdata;
    short *xstart;
    short *filt_ptr;
    int    M = 8;

    /* ------------------------------------------------- */
    /*  Set our loop trip count and starting x posn.     */
    /*  'xstart' is used in the high-pass filter loop.   */
    /* ------------------------------------------------- */
    iters  = cols;
    xstart = in_data + (cols - M)  + 2;

    /* ------------------------------------------------- */
    /*  Low pass filter.  Iterate for cols/2 iterations  */
    /*  generating cols/2 low pass sample points with    */
    /*  the low-pass quadrature mirror filter.           */
    /* ------------------------------------------------- */
    for (i = 0; i < iters; i += 2)
    {
        /* --------------------------------------------- */
        /*  Initialize our sum to the rounding value     */
        /*  and reset our pointer.                       */
        /* --------------------------------------------- */
```

```
                    sum  = Qr;
                    xptr = in_data + i;

                    /* ------------------------------------------- */
                    /*  Iterate over the taps in our QMF.          */
                    /* ------------------------------------------- */
                    for (j = 0; j < M; j++)
                    {
                        xdata = *xptr++;
                        hdata =  qmf[j];
                        prod  =  xdata * hdata;
                        sum  += prod;
                        if (xptr > x_end) xptr = in_data;
                    }

                    /* ------------------------------------------- */
                    /*  Adjust the Qpt of our sum and store result. */
                    /* ------------------------------------------- */
                    res    = (sum >> Qpt);
                    *out_data++ = res;
                }

                /* ------------------------------------------------- */
                /*  High pass filter.  Iterate for cols/2 iters      */
                /*  generating cols/2 high pass sample points with   */
                /*  the high-pass quadrature mirror filter.          */
                /* ------------------------------------------------- */
                for (i = 0; i < iters ; i+=2)
                {
                    /* ------------------------------------------- */
                    /*  Initialize our sum and filter pointer.     */
                    /* ------------------------------------------- */
                    sum  = Qr;
                    filt_ptr  = mqmf + (M - 1);

                    /* ------------------------------------------- */
                    /*  Set up our data pointer.  This is slightly */
                    /*  more complicated due to how the data wraps */
                    /*  around the edge of the buffer.             */
                    /* ------------------------------------------- */
                    xptr = xstart;
                    xstart += 2;
                    if (xstart > x_end) xstart = in_data;

                    /* ------------------------------------------- */
                    /*  Iterate over the taps in our QMF.          */
                    /* ------------------------------------------- */
                    for ( j = 0; j < M; j++)
                    {
                        xdata = *xptr++;
                        hdata = *filt_ptr--;
                        prod  = xdata * hdata;
                        if (xptr > x_end) xptr = in_data;
                        sum  += prod;
                    }

                    /* ------------------------------------------- */
                    /*  Adjust the Qpt of our sum and store result. */
                    /* ------------------------------------------- */
                    res = (sum >> Qpt);
                    *out_data++ =  res;
                }
            }
```

Special Requirements

- This function assumes that the number of taps for the qmf and mqmf filters is 8, and that the filter coefficient arrays qmf[ ] and mqmf[ ] are double-word aligned.
- The array in_data[ ] is assumed to be word aligned.
- This function assumes that filter coefficients are maintained as 16-bit Q.15 numbers.
- It is also assumed that input data is an array of shorts, to allow for re-use of this function to perform Multi Resolution Analysis where the output of this code is

feedback as input to an identical next stage.

- The transform is a dyadic wavelet, requiring the number of image columns cols to be a multiple of 2. Cols must also be at least 8.

Notes

- Bank Conflicts: The code has no bank conflicts.
- Endian: The code is ENDIAN NEUTRAL.
- Interruptibility: The code is interrupt-tolerant, but not interruptible.
- Optimizing the code includes issuing one set of reads to the data array and performing low-pass and high pass filtering together to maximize the number of multiplies. The last six elements of the low-pass filter and the first six elements of the high-pass filter use the same input. This is used to appropriately change the output pointer to the low-pass filter after six iterations. However, for the first six iterations, pointer wraparound can occur, creating a dependency. Prereading those six values outside the array prevents the checks that introduce this dependency. In addition, the input data is read as word wide quantities and the low-pass and high-pass filter coefficients are stored in registers, allowing for the input loop to be completely unrolled. Therefore, the assembly code has only one loop. A predication register is used to reset the low-pass output pointer after three iterations. The merging of the loops allows for the maximum number of multiplies with the minimum number of reads.
- This code can implement the Daubechies D4 filter bank for analysis with four vanishing moments. The length of the analyzing low-pass and high-pass filters is 8, in this case.

7.11 IMG_wave_vert

IMG_wave_vert	*Vertical Wavelet Transform*

Syntax

void **IMG_wave_vert** (const short * restrict * restrict in_data, const short * restrict qmf, const short * restrict mqmf, short * restrict out_ldata, short * restrict out_hdata, int cols)

Arguments

*in_data[8]	Pointer to an array of 8 pointers that point to input data line buffers. Each of the 8 lines has cols number of elements and must be double-word aligned.
qmf[8]	Pointer to Q.15 QMF filter bank for low-pass filtering. Must be word aligned.
mqmf[8]	Pointer to Q.15 mirror QMF filter bank for high-pass filtering. Must be word aligned.
out_ldata[ ]	Pointer to one line of low-pass filtered outputs consisting of cols number of elements. Must be double-word aligned.
out_hdata[ ]	Pointer to one line of high-pass filtered outputs consisting of cols number of elements. Must be double-word aligned.
cols	Width of each line in the input buffer. Must be a multiple of 2.

Description

This routine performs the vertical pass of a 2-D wavelet transform. A vertical filter is applied on 8 lines that are pointed to by the pointers contained in the array in_data[ ]. Instead of transposing the input image and re-using the horizontal wavelet function, the vertical filter is applied directly to the image data as-is, producing a single line of high-pass and a single line of low-pass filtered outputs. The vertical filter is traversed over the entire width of the line.

In a traditional wavelet implementation, the input context for the low-pass filter is offset by a number of lines from the input context for the high-pass filter for a given pair of output lines. The amount of offset is determined by the number of filter taps and is generally num_taps - 2 rows (this implementation is fixed at 8 taps, so the offset would be 6 rows).

This implementation breaks from the traditional model so that it can re-use the same input context for both low-pass and high-pass filters simultaneously. The result is that the low-pass and high-pass outputs must instead be offset by the calling function. To write the low-pass filtered output to the top half and the high pass-filtered output to the bottom half of the output image, the respective start pointers have to be set to:

```
out_lstart = o_im + ((rows >> 1) - 3) * cols
out_hstart = o_im + (rows >> 1) * cols
```

Where o_im is the start of the output image, rows is the number of rows of the input image, and cols is the number of cols of the output image. The following table illustrates how the pointers out_ldata and out_hdata need to be updated at the start of each call to this function:

Call Number	out_ldata	out_hdata
1	out_lstart	out_hstart
2	out_lstart + cols	out_hstart + cols
3	out_lstart + 2 * cols	out_hstart + 2 * cols

At this point out_ldata wraps around to become o_im, while out_hdata proceeds as usual:

| 4 | o_im | out_hstart + 3 * cols |

Corresponding to the output pointer update scheme described above, the input buffer lines have to be filled starting with the 6th row from the bottom of the input image. That is, for the first call of the wave_vert function, the eight input line buffers consist of the last six plus the first two lines of the image. For the second call, the input line buffers contain the last four plus the first 4 lines of the image, and so on.

The routine can obtain maximum performance by using a working buffer of ten input lines to effectively mix processing and data transfer through DMAs. At the start of the routine, eight input lines are loaded into the first 8 line buffers and processing begins. In the background, the next two lines are fetched. The pointers are moved up by 2, namely ptr[i] = ptr[i+2] and the last two lines now point to lines 9 and 10 and processing starts again. In the background, the next two lines are loaded into the first two lines of the line buffer. Pointers move up again by two but now the last two point to line 0 and 1. This pattern then repeats.

Algorithm   Behavioral C code for the routine wave_vert is provided below:

```c
void IMG_wave_vert
(
    short **in_data,    /* Array of row pointers */
    short *lp_filt,     /* Low pass QMF filter   */
    short *hp_filt,     /* High pass QMF filter  */
    short *out_ldata,   /* Low pass output data  */
    short *out_hdata,   /* High pass output data */
    int   cols      /* Length of rows to process */
)
{
    int   i, j;
    /* ------------------------------------------------------------------- */
    /*  First, perform the low-pass filter on the eight input rows.        */
    /* ------------------------------------------------------------------- */
    for (i = 0; i < cols; i++)
    {
        int sum = 1 << 14;

        for (j = 0; j < 8; j++)
            sum += in_data[j][i] * lp_filt[j];
        out_ldata[i] = sum >> 15;
    }
    /* ------------------------------------------------------------------- */
    /*  Next, perform the high-pass filter on the same eight input rows.   */
    /* ------------------------------------------------------------------- */
    for (i = 0; i < cols; i++)
    {
        int sum = 1 << 14;

        for (j = 0; j < 8; j++)
            sum += in_data[j][i] * hp_filt[7 - j];
        out_hdata[i] = sum >> 15;
    }
}
```

Special Requirements

- Since the wavelet transform is dyadic, cols must be a multiple of 2.
- The filters qmf[ ] and mqmf[ ] are assumed to be word aligned and have 8 taps.
- The input data on any line, and the output arrays out_ldata[ ] and out_hdata[ ] must be double-word aligned.
- The mqmf filter is constructed from the qmf as follows:

```c
status = -1;
for (i = 0; i < M; i++)
{
    status = status * -1;
    hdata  = qmf[i] * status;
    filter[i] = hdata;
}
```

Notes

- Bank Conflicts: No bank conflicts occur.
- Endian: The code is LITTLE ENDIAN.
- Interruptibility: The code is interrupt-tolerant, but not interruptible.
- The low-pass and high-pass filtering are performed together. This implies that the low-pass and high-pass filters be overlapped in execution so that the input data array may be read once and both filters can be executed in parallel.
- The inner loop that advances along each filter tap is totally optimized by unrolling. Double-word loads are performed, and paired multiplies are used to perform four iterations of low-pass filter in parallel.
- For the high-pass kernel, the same loop is reused, to save code size. This is done by loading the filter coefficients in a special order.

Appendix A Low Level Kernels

Often during Image Processing algorithm development, it is required to perform low -evel operations on the input image data. These are basic operations like Image Addition, Image Multiplication, etc. This appendix provides example code for such operations. The intent is for user to either use the code provided in this Appendix as is, or use the code examples to develop more complex kernels. The following kernel implementations are provided:

Table A-1. Table 4. Low-level kernels and Their Performance

Kernel Name	Description	Clocks/Pixel
IMG_mulS_16s	Multiply pixels with a constant 16-bit data	0.375
IMG_mulS_8	Multiply pixels with a constant 8-bit data	0.1875
IMG_addS_16s	Add pixels with a constant 16-bit data	0.25
IMG_addS_8	Add pixels with a constant 8-bit data	0.125
IMG_subS_16s	Subtract pixels with a constant 16bit data	0.25
IMG_subS_8	Subtract pixels with a constant 8-bit data	0.125
IMG_not_16	Bitwise NOT operation on each pixel 16-bit data	0.25
IMG_not_8	Bitwise NOT operation on each pixel 8-bit data	0.125
IMG_andS_16	Bitwise AND operation of each pixel with a constant data 16-bit data	0.25
IMG_andS_8	Bitwise AND operation of each pixel with a constant data 8-bit data	0.125
IMG_orS_16	Bitwise OR operation of each pixel with a constant data 16-bit data	0.25
IMG_orS_8	Bitwise OR operation of each pixel with a constant data 8-bit data	0.125
IMG_and_16	Combines corresponding pixels of two images by a bitwise AND 16-bit data	0.375
IMG_and_8	Combines corresponding pixels of two images by a bitwise AND 8-bit data	0.1875
IMG_or_16	Combines corresponding pixels of two images by a bitwise OR 16-bit data	0.375
IMG_or_8	Combines corresponding pixels of two images by a bitwise OR 8-bit data	0.1875
IMG_mul_16s	Multiply corresponding pixels from two images 16-bit data	0.5
IMG_mul_8	Multiply corresponding pixels from two images 8-bit data	0.25
IMG_add_16s	Add corresponding pixels from two images 16-bit data	0.375
IMG_add_8	Add corresponding pixels from two images 8-bit data	0.1875
IMG_sub_16s	Subtract corresponding pixels from two images 16-bit data	0.375
IMG_sub_8	Subtract corresponding pixels from two images 8-bit data	0.1875

A.1 *IMG_mulS_16s*

```
/*-----------------------------------------------------------------**
** This function performs multiplication of each pixel in a image **
** with a constant value. The image consist of 16bits per pixel.  **
** The constant is 16bits in size                                 **
**-----------------------------------------------------------------*/


void IMG_mulS_16s
(
    short * restrict imgR,  /* Read pointer for the input image */
    int * restrict imgW,    /* Write pointer for the output image */
    short constData,        /* Constant data */
    int count               /* Number of samples in the image */
)
{
    int i;
    long long pix3_pix2_pix1_pix0;
    int pix3_pix2, pix1_pix0;
    double respix1_respix0, respix3_respix2;

    long long pix7_pix6_pix5_pix4;
    int pix7_pix6, pix5_pix4;
    double respix5_respix4, respix7_respix6;


    int cData_cData;

    cData_cData = (constData << 16) | constData;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = _mpy2 (pix1_pix0, cData_cData);
        respix3_respix2 = _mpy2 (pix3_pix2, cData_cData);
        *((double *)imgW) = respix1_respix0;
        imgW += 2;
        *((double *)imgW) = respix3_respix2;
        imgW += 2;

        pix7_pix6_pix5_pix4 = _amem8(imgR);
        pix7_pix6 = _hill (pix7_pix6_pix5_pix4);
        pix5_pix4 = _loll (pix7_pix6_pix5_pix4);
        imgR += 4;

        respix5_respix4 = _mpy2 (pix5_pix4, cData_cData);
        respix7_respix6 = _mpy2 (pix7_pix6, cData_cData);
        *((double *)imgW) = respix5_respix4;
        imgW += 2;
        *((double *)imgW) = respix7_respix6;
        imgW += 2;
    }
}
```

A.2 IMG_mulS_8

```
/*-----------------------------------------------------------------**
** This function performs multiplication of each pixel in a image **
** with a constant value. The image consist of 8 bits per pixel.  **
** The constant is 8 bits in size                                 **
**-----------------------------------------------------------------*/

void IMG_mulS_8
(
    unsigned char * restrict imgR,  /* Read pointer for the input image */
    short * restrict imgW,          /* Write pointer for the output image */
    char constData,                 /* Constant data */
    int count                       /* Number of samples in the image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;
    int p7_p6_p5_p4, p3_p2_p1_p0;
    double rp3_rp2_rp1_rp0, rp7_rp6_rp5_rp4;

    int cD_cD_cD_cD;

    cD_cD_cD_cD = (constData << 24) | (constData << 16) |
                  (constData << 8)  | (constData);

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        rp3_rp2_rp1_rp0 = _mpysu4 (cD_cD_cD_cD, p7_p6_p5_p4);
        rp7_rp6_rp5_rp4 = _mpysu4 (cD_cD_cD_cD, p3_p2_p1_p0);
        *((double *)imgW) = rp3_rp2_rp1_rp0;
        imgW += 4;
        *((double *)imgW) = rp7_rp6_rp5_rp4;
        imgW += 4;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        rp3_rp2_rp1_rp0 = _mpysu4 (cD_cD_cD_cD, p7_p6_p5_p4);
        rp7_rp6_rp5_rp4 = _mpysu4 (cD_cD_cD_cD, p3_p2_p1_p0);
        *((double *)imgW) = rp3_rp2_rp1_rp0;
        imgW += 4;
        *((double *)imgW) = rp7_rp6_rp5_rp4;
        imgW += 4;
    }
}
```

A.3 IMG_addS_16s

```
/*-----------------------------------------------------------------**
** This function performs addition of each pixel in a image with  **
** a constant value. The image consist of 16bits per pixel. The   **
** constant is 16bits in size                                     **
**-----------------------------------------------------------------*/

void IMG_addS_16s
(
    short * restrict imgR, /* Read pointer for the input image */
    short * restrict imgW, /* Write pointer for the output image */
    short constData,       /* Constant data */
    int count              /* Number of samples in the image */
)
{
    int i;
    long long pix3_pix2_pix1_pix0;
    int pix3_pix2, pix1_pix0;
    int respix1_respix0, respix3_respix2;

    int cData_cData;

    cData_cData = (constData << 16) | constData;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = _add2 (pix1_pix0, cData_cData);
        respix3_respix2 = _add2 (pix3_pix2, cData_cData);

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;

        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = _add2 (pix1_pix0, cData_cData);
        respix3_respix2 = _add2 (pix3_pix2, cData_cData);

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;
    }
}
```

A.4   IMG_addS_8

```
/*-----------------------------------------------------------------**
** This function performs addition of each pixel in a image with  **
** a constant value. The image consist of 8 bits per pixel. The   **
** constant is 8 bits in size                                     **
**-----------------------------------------------------------------*/

void IMG_addS_8
(
    char * restrict imgR,  /* Read pointer for the input image */
    char * restrict imgW,  /* Write pointer for the output image */
    char constData,        /* Constant data */
    int count              /* Number of samples in the image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;
    int p3_p2_p1_p0, p7_p6_p5_p4;
    int r7_r6_r5_r4, r3_r2_r1_r0;

    int cD_cD_cD_cD;

    cD_cD_cD_cD = (constData << 24) | (constData << 16) |
                  (constData << 8)  | constData;

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = _add4 (p7_p6_p5_p4, cD_cD_cD_cD);
        r3_r2_r1_r0 = _add4 (p3_p2_p1_p0, cD_cD_cD_cD);

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = _add4 (p7_p6_p5_p4, cD_cD_cD_cD);
        r3_r2_r1_r0 = _add4 (p3_p2_p1_p0, cD_cD_cD_cD);

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;
    }
}
```

A.5 IMG_subS_16s

```
/*--------------------------------------------------------------------**
** This function performs subtraction of each pixel in a image with  **
** a constant value. The image consist of 16bits per pixel. The      **
** constant is 16bits in size                                        **
**--------------------------------------------------------------------*/

void IMG_subS_16s
(
    short * restrict imgR,  /* Read pointer for the input image */
    short * restrict imgW,  /* Write pointer for the output image */
    short constData,        /* Constant data */
    int count               /* Number of samples in the image */
)
{
    int i;
    long long pix3_pix2_pix1_pix0;
    int pix3_pix2, pix1_pix0;
    int respix1_respix0, respix3_respix2;

    int cData_cData;

    cData_cData = (constData << 16) | constData;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = _sub2 (pix1_pix0, cData_cData);
        respix3_respix2 = _sub2 (pix3_pix2, cData_cData);

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;

        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = _sub2 (pix1_pix0, cData_cData);
        respix3_respix2 = _sub2 (pix3_pix2, cData_cData);

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;
    }
}
```

A.6 IMG_subS_8

```
/*----------------------------------------------------------------**
** This function performs subtraction of each pixel in a image with  **
** a constant value. The image consist of 8 bits per pixel. The      **
** constant is 8 bits in size                                        **
**----------------------------------------------------------------*/

void IMG_subS_8
(
    char * restrict imgR,  /* Read pointer for the input image */
    char * restrict imgW,  /* Write pointer for the output image */
    char constData,        /* Constant data */
    int count              /* Number of samples in the image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;
    int p3_p2_p1_p0, p7_p6_p5_p4;
    int r7_r6_r5_r4, r3_r2_r1_r0;

    int    cD_cD_cD_cD;

    cD_cD_cD_cD = (constData << 24) | (constData << 16) |
                  (constData << 8)  | constData;

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = _sub4 (p7_p6_p5_p4, cD_cD_cD_cD);
        r3_r2_r1_r0 = _sub4 (p3_p2_p1_p0, cD_cD_cD_cD);

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = _sub4 (p7_p6_p5_p4, cD_cD_cD_cD);
        r3_r2_r1_r0 = _sub4 (p3_p2_p1_p0, cD_cD_cD_cD);

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;
    }
}
```

A.7 *IMG_not_16*

```
/*----------------------------------------------------------**
** This function performs bitwise NOT operation on a image  **
** Each image consist of 16 bits per sample                 **
**----------------------------------------------------------*/

void IMG_not_16
(
    unsigned short * restrict imgR,  /* Image read pointer */
    unsigned short * restrict imgW,  /* Image write pointer */
    int count                        /* Number of samples in image */
)
{
    int i;
    long long pix3_pix2_pix1_pix0;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        imgR += 4;
        _amem8(imgW) = ~pix3_pix2_pix1_pix0;
        imgW += 4;

        pix3_pix2_pix1_pix0 = _amem8(imgR);
        imgR += 4;
        _amem8(imgW) = ~pix3_pix2_pix1_pix0;
        imgW += 4;
    }
}
```

A.8 IMG_not_8

```
/*----------------------------------------------------------**
** This function performs bitwise NOT operation on a image  **
** Each image consist of 8 bits per sample                  **
**----------------------------------------------------------*/

void IMG_not_8
(
    unsigned char * restrict imgR,  /* Image read pointer */
    unsigned char * restrict imgW,  /* Image write pointer */
    int count                       /* Number of samples in image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        imgR += 8;
        _amem8(imgW) = ~p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        imgR += 8;
        _amem8(imgW) = ~p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;
    }
}
```

A.9 IMG_andS_16

```
/*------------------------------------------------------------------**
** This function performs bit wise AND of each pixel in a image    **
** with a constant value. The image consist of 16bits per pixel.   **
** The constant is 16bits in size                                  **
**------------------------------------------------------------------*/


void IMG_andS_16
(
    unsigned short * restrict imgR,    /* Read pointer for the input image */
    unsigned short * restrict imgW,    /* Write pointer for the output image */
    short constData,                   /* Constant data */
    int count                          /* Number of samples in the image */
)
{
    int i;
    long long pix3_pix2_pix1_pix0;
    int pix3_pix2, pix1_pix0;
    int respix1_respix0, respix3_respix2;

    int cData_cData;

    cData_cData = (constData << 16) | constData;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = pix1_pix0 & cData_cData;
        respix3_respix2 = pix3_pix2 & cData_cData;

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;

        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = pix1_pix0 & cData_cData;
        respix3_respix2 = pix3_pix2 & cData_cData;

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;
    }
}
```

A.10 IMG_andS_8

```
/*-----------------------------------------------------------------**
** This function performs AND of each pixel in a image with      **
** a constant value. The image consist of 8 bits per pixel. The  **
** constant is 8 bits in size                                    **
**-----------------------------------------------------------------*/

void IMG_andS_8
(
    unsigned char * restrict imgR,  /* Read pointer for the input image */
    unsigned char * restrict imgW,  /* Write pointer for the output image */
    char constData,                 /* Constant data */
    int count                       /* Number of samples in the image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;
    int p3_p2_p1_p0, p7_p6_p5_p4;
    int r7_r6_r5_r4, r3_r2_r1_r0;

    int cD_cD_cD_cD;

    cD_cD_cD_cD = (constData << 24) | (constData << 16) |
                  (constData << 8)  | constData;

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 =  p7_p6_p5_p4 & cD_cD_cD_cD;
        r3_r2_r1_r0 =  p3_p2_p1_p0 & cD_cD_cD_cD;

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = p7_p6_p5_p4 & cD_cD_cD_cD;
        r3_r2_r1_r0 = p3_p2_p1_p0 & cD_cD_cD_cD;

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;
    }
}
```

A.11 IMG_orS_16

```
/*------------------------------------------------------------------**
** This function performs bit wise OR of each pixel in a image      **
** with a constant value. The image consist of 16bits per pixel.    **
** The constant is 16bits in size                                   **
**------------------------------------------------------------------*/


void IMG_orS_16
(
    unsigned short * restrict imgR, /* Read pointer for the input image */
    unsigned short * restrict imgW, /* Write pointer for the output image */
    short constData,                /* Constant data */
    int count                       /* Number of samples in the image */
)
{

    int i;
    long long pix3_pix2_pix1_pix0;
    int pix3_pix2, pix1_pix0;
    int respix1_respix0, respix3_respix2;

    int cData_cData;

    cData_cData = (constData << 16) | constData;

    for (i = 0; i < count >> 3; i += 8) {
        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = pix1_pix0 | cData_cData;
        respix3_respix2 = pix3_pix2 | cData_cData;

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;

        pix3_pix2_pix1_pix0 = _amem8(imgR);
        pix3_pix2 = _hill (pix3_pix2_pix1_pix0);
        pix1_pix0 = _loll (pix3_pix2_pix1_pix0);
        imgR += 4;

        respix1_respix0 = pix1_pix0 | cData_cData;
        respix3_respix2 = pix3_pix2 | cData_cData;

        _amem8(imgW) = _itoll (respix3_respix2, respix1_respix0);
        imgW += 4;
    }
}
```

A.12 IMG_orS_8

```
/*------------------------------------------------------------------**
** This function performs bit wise OR of each pixel in a image      **
** with a constant value. The image consist of 8 bits per pixel.    **
** The constant is 16bits in size                                   **
**------------------------------------------------------------------*/


void IMG_orS_8
(
    unsigned char * restrict imgR,  /* Read pointer for the input image */
      unsigned char * restrict imgW,  /* Write pointer for the output image */
    char constData,                      /* Constant data */
    int count                       /* Number of samples in the image */
)
{
    int i;
    long long p7_p6_p5_p4_p3_p2_p1_p0;
    int p3_p2_p1_p0, p7_p6_p5_p4;
    int r7_r6_r5_r4, r3_r2_r1_r0;

    int    cD_cD_cD_cD;

    cD_cD_cD_cD = (constData << 24) | (constData << 16) |
                  (constData << 8)  | constData;

    for (i = 0; i < count >> 4; i += 16) {
        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 =  p7_p6_p5_p4 | cD_cD_cD_cD;
        r3_r2_r1_r0 =  p3_p2_p1_p0 | cD_cD_cD_cD;

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;

        p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR);
        p7_p6_p5_p4 = _hill (p7_p6_p5_p4_p3_p2_p1_p0);
        p3_p2_p1_p0 = _loll (p7_p6_p5_p4_p3_p2_p1_p0);
        imgR += 8;

        r7_r6_r5_r4 = p7_p6_p5_p4 | cD_cD_cD_cD;
        r3_r2_r1_r0 = p3_p2_p1_p0 | cD_cD_cD_cD;

        _amem8(imgW) = _itoll (r7_r6_r5_r4, r3_r2_r1_r0);
        imgW += 8;
    }
}
```

A.13 IMG_and_16

```
/*-----------------------------------------------------------**
** This function performs bitwise AND operation on 2 images **
** Each image consist of 16bits per sample                  **
**-----------------------------------------------------------*/

void IMG_and_16
(
    unsigned short * restrict imgR1,  /* Image 1 read pointer */
    unsigned short * restrict imgR2,   /* Image 2 read pointer */
    short * restrict imgW,            /* Output image pointer */
    int count                          /* Number of samples in image */
)
{
    int i;
    long long im1_p3_p2_p1_p0, im2_p3_p2_p1_p0;
    long long res_p3_p2_p1_p0;

    for (i = 0; i < count >> 3; i += 8) {
        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        res_p3_p2_p1_p0 = im1_p3_p2_p1_p0 & im2_p3_p2_p1_p0;

        _amem8(imgW) = res_p3_p2_p1_p0;
        imgW += 4;

        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        res_p3_p2_p1_p0 = im1_p3_p2_p1_p0 & im2_p3_p2_p1_p0;

        _amem8(imgW) = res_p3_p2_p1_p0;
        imgW += 4;
    }
}
```

A.14 IMG_and_8

```
/*-----------------------------------------------------------**
** This function performs bitwise AND operation on 2 images **
** Each image consist of 8 bits per sample                  **
**-----------------------------------------------------------*/

void IMG_and_8
(
    unsigned char * restrict imgR1,   /* Image 1 read pointer */
    unsigned char * restrict imgR2,   /* Image 2 read pointer */
    char * restrict imgW,             /* Output image pointer */
    int count                         /* Number of samples in image */
)
{
    int i;
    long long im1_p7_p6_p5_p4_p3_p2_p1_p0, im2_p7_p6_p5_p4_p3_p2_p1_p0;
    long long res_p7_p6_p5_p4_p3_p2_p1_p0;

    for (i = 0; i < count >> 4; i += 16) {
        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        res_p7_p6_p5_p4_p3_p2_p1_p0 = im1_p7_p6_p5_p4_p3_p2_p1_p0 & im2_p7_p6_p5_p4_p3_p2_p1_p0;

        _amem8(imgW) = res_p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;

        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        res_p7_p6_p5_p4_p3_p2_p1_p0 = im1_p7_p6_p5_p4_p3_p2_p1_p0 & im2_p7_p6_p5_p4_p3_p2_p1_p0;

        _amem8(imgW) = res_p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;
    }
}
```

A.15 *IMG_or_16*

```
/*----------------------------------------------------------**
** This function performs bitwise OR operation on 2 images  **
** Each image consist of 16bits per sample                  **
**----------------------------------------------------------*/

void IMG_or_16
(
    unsigned short * restrict imgR1,  /* Image 1 read pointer */
    unsigned short * restrict imgR2,  /* Image 2 read pointer */
    short * restrict imgW,            /* Output image pointer */
    int count                         /* Number of samples in image */
)
{
    int i;
    long long im1_p3_p2_p1_p0, im2_p3_p2_p1_p0;
    long long res_p3_p2_p1_p0;

    for (i = 0; i < count >> 3; i += 8) {
        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        res_p3_p2_p1_p0 = im1_p3_p2_p1_p0 | im2_p3_p2_p1_p0;

        _amem8(imgW) = res_p3_p2_p1_p0;
        imgW += 4;

        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        res_p3_p2_p1_p0 = im1_p3_p2_p1_p0 | im2_p3_p2_p1_p0;

        _amem8(imgW) = res_p3_p2_p1_p0;
        imgW += 4;
    }
}
```

A.16 IMG_or_8

```
/*-----------------------------------------------------------**
** This function performs bitwise OR operation on 2 images  **
** Each image consist of 8 bits per sample                  **
**-----------------------------------------------------------*/

void IMG_or_8
(
    unsigned char * restrict imgR1, /* Image 1 read pointer */
    unsigned char * restrict imgR2, /* Image 2 read pointer */
    char * restrict imgW,           /* Output image pointer */
    int count                       /* Number of samples in image */
)
{
    int i;
    long long im1_p7_p6_p5_p4_p3_p2_p1_p0, im2_p7_p6_p5_p4_p3_p2_p1_p0;
    long long res_p7_p6_p5_p4_p3_p2_p1_p0;

    for (i = 0; i < count >> 4; i += 16) {
        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        res_p7_p6_p5_p4_p3_p2_p1_p0 = im1_p7_p6_p5_p4_p3_p2_p1_p0 | im2_p7_p6_p5_p4_p3_p2_p1_p0;

        _amem8(imgW) = res_p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;

        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        res_p7_p6_p5_p4_p3_p2_p1_p0 = im1_p7_p6_p5_p4_p3_p2_p1_p0 | im2_p7_p6_p5_p4_p3_p2_p1_p0;

        _amem8(imgW) = res_p7_p6_p5_p4_p3_p2_p1_p0;
        imgW += 8;
    }
}
```

A.17   *IMG_mul_16s*

```
/*--------------------------------------------------------------------**
** This function performs multiplication of corresponding samples of  **
** two images Each image consist of 16bits per samples.               **
**--------------------------------------------------------------------*/

void IMG_mul_16s
(
    short * restrict imgR1,    /* Image 1 read pointer */
    short * restrict imgR2,    /* Image 2 read pointer */
    int * restrict imgW,       /* Output image pointer */
    int count                  /* Number of samples in image */
)
{
    int i;
    long long img1_p3_p2_p1_p0, img2_p3_p2_p1_p0;
    int img1_p3_p2, img1_p1_p0, img2_p3_p2, img2_p1_p0;
    double r1_r0, r3_r2;

    for (i = 0; i < count >> 2; i += 4) {
        img1_p3_p2_p1_p0 = _amem8(imgR1);
        img1_p3_p2 = _hill (img1_p3_p2_p1_p0);
        img1_p1_p0 = _loll (img1_p3_p2_p1_p0);
        imgR1 += 4;

        img2_p3_p2_p1_p0 = _amem8(imgR2);
        img2_p3_p2 = _hill (img2_p3_p2_p1_p0);
        img2_p1_p0 = _loll (img2_p3_p2_p1_p0);
        imgR2 += 4;

        r1_r0 = _mpy2 (img1_p1_p0, img2_p1_p0);
        r3_r2 = _mpy2 (img1_p3_p2, img2_p3_p2);
        *((double *)imgW) = r1_r0;
        imgW += 2;
        *((double *)imgW) = r3_r2;
        imgW += 2;
    }
}
```

A.18 IMG_mul_8

```
/*----------------------------------------------------------------------**
** This function performs multiplication of corresponding samples of  **
** two images Each image consist of 8 bits per samples.               **
**----------------------------------------------------------------------*/

void IMG_mul_8
(
    char * restrict imgR1,    /* Image 1 read pointer */
    char * restrict imgR2,    /* Image 2 read pointer */
    short * restrict imgW,    /* Output image pointer */
    int count                 /* Number of samples in image */
)
{
    int i;
    long long img1_p7_p6_p5_p3_p2_p1_p0, img2_p7_p6_p5_p3_p2_p1_p0;
    int img1_p7_p6_p5_p4, img1_p3_p2_p1_p0, img2_p7_p6_p5_p4, img2_p3_p2_p1_p0;
    double r3_r2_r1_r0, r7_r6_r5_r4;

    for (i = 0; i < count >> 3; i += 8) {
        img1_p7_p6_p5_p3_p2_p1_p0 = _amem8(imgR1);
        img1_p7_p6_p5_p4 = _hill (img1_p7_p6_p5_p3_p2_p1_p0);
        img1_p3_p2_p1_p0 = _loll (img1_p7_p6_p5_p3_p2_p1_p0);
        imgR1 += 8;

        img2_p7_p6_p5_p3_p2_p1_p0 = _amem8(imgR2);
        img2_p7_p6_p5_p4 = _hill (img2_p7_p6_p5_p3_p2_p1_p0);
        img2_p3_p2_p1_p0 = _loll (img2_p7_p6_p5_p3_p2_p1_p0);
        imgR2 += 8;

        r3_r2_r1_r0 = _mpyu4 (img1_p3_p2_p1_p0, img2_p3_p2_p1_p0);
        r7_r6_r5_r4 = _mpyu4 (img1_p7_p6_p5_p4, img2_p7_p6_p5_p4);
        *((double *)imgW) = r3_r2_r1_r0;
        imgW += 4;
        *((double *)imgW) = r7_r6_r5_r4;
        imgW += 4;
    }
}
```

A.19 IMG_add_16s

```
/*--------------------------------------------**
** This function performs addition of 2 images **
** Each image consist of 16bits per samples.   **
**--------------------------------------------*/

void IMG_add_16s
(
    short * restrict imgR1, /* Image 1 read pointer */
    short * restrict imgR2, /* Image 2 read pointer */
    short * restrict imgW,  /* Output image pointer */
    int count               /* Number of samples in image */
)
{
    int i;
    long long im1_p3_p2_p1_p0, im2_p3_p2_p1_p0;
    int im1_p3_p2, im1_p1_p0, im2_p3_p2, im2_p1_p0;
    int res_p3_p2, res_p1_p0;

    for (i = 0; i < count >> 3; i += 8) {
        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        im1_p3_p2 = _hill (im1_p3_p2_p1_p0);
        im1_p1_p0 = _loll (im1_p3_p2_p1_p0);

        im2_p3_p2 = _hill (im2_p3_p2_p1_p0);
        im2_p1_p0 = _loll (im2_p3_p2_p1_p0);

        res_p3_p2 = _add2 (im1_p3_p2, im2_p3_p2);
        res_p1_p0 = _add2 (im1_p1_p0, im2_p1_p0);

        _amem8(imgW) = _itoll (res_p3_p2, res_p1_p0);
        imgW += 4;

        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        im1_p3_p2 = _hill (im1_p3_p2_p1_p0);
        im1_p1_p0 = _loll (im1_p3_p2_p1_p0);

        im2_p3_p2 = _hill (im2_p3_p2_p1_p0);
        im2_p1_p0 = _loll (im2_p3_p2_p1_p0);

        res_p3_p2 = _add2 (im1_p3_p2, im2_p3_p2);
        res_p1_p0 = _add2 (im1_p1_p0, im2_p1_p0);

        _amem8(imgW) = _itoll (res_p3_p2, res_p1_p0);
        imgW += 4;
    }
}
```

A.20 IMG_add_8

```
/*--------------------------------------------**
** This function performs addition of 2 images **
** Each image consist of 8bits per samples.    **
**--------------------------------------------*/

void IMG_add_8
(
    char * restrict imgR1, /* Image 1 read pointer */
    char * restrict imgR2, /* Image 2 read pointer */
    char * restrict imgW,  /* Output image pointer */
    int count              /* Number of samples in image */
)
{
    int i;
    long long im1_p7_p6_p5_p4_p3_p2_p1_p0, im2_p7_p6_p5_p4_p3_p2_p1_p0;
    int im1_p7_p6_p5_p4, im1_p3_p2_p1_p0, im2_p7_p6_p5_p4, im2_p3_p2_p1_p0;
    int res_p7_p6_p5_p4, res_p3_p2_p1_p0;

    for (i = 0; i < count >> 4; i += 16) {
        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        im1_p3_p2_p1_p0 = _loll (im1_p7_p6_p5_p4_p3_p2_p1_p0);
        im1_p7_p6_p5_p4 = _hill (im1_p7_p6_p5_p4_p3_p2_p1_p0);

        im2_p3_p2_p1_p0 = _loll (im2_p7_p6_p5_p4_p3_p2_p1_p0);
        im2_p7_p6_p5_p4 = _hill (im2_p7_p6_p5_p4_p3_p2_p1_p0);

        res_p3_p2_p1_p0 = _add4 (im1_p3_p2_p1_p0, im2_p3_p2_p1_p0);
        res_p7_p6_p5_p4 = _add4 (im1_p7_p6_p5_p4, im2_p7_p6_p5_p4);

        _amem8(imgW) = _itoll (res_p7_p6_p5_p4, res_p3_p2_p1_p0);
        imgW += 8;

        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        im1_p3_p2_p1_p0 = _loll (im1_p7_p6_p5_p4_p3_p2_p1_p0);
        im1_p7_p6_p5_p4 = _hill (im1_p7_p6_p5_p4_p3_p2_p1_p0);

        im2_p3_p2_p1_p0 = _loll (im2_p7_p6_p5_p4_p3_p2_p1_p0);
        im2_p7_p6_p5_p4 = _hill (im2_p7_p6_p5_p4_p3_p2_p1_p0);

        res_p3_p2_p1_p0 = _add4 (im1_p3_p2_p1_p0, im2_p3_p2_p1_p0);
        res_p7_p6_p5_p4 = _add4 (im1_p7_p6_p5_p4, im2_p7_p6_p5_p4);

        _amem8(imgW) = _itoll (res_p7_p6_p5_p4, res_p3_p2_p1_p0);
        imgW += 8;

    }
}
```

A.21 IMG_sub_16s

```
/*---------------------------------------------------**
** This function performs subtraction of 2          **
** images. Each image consist of 16bits per samples. **
**---------------------------------------------------*/

void IMG_sub_16s
(
    short * restrict imgR1,    /* Image 1 read pointer */
    short * restrict imgR2,     /* Image 2 read pointer */
    short * restrict imgW,      /* Output image pointer */
    int count                 /* Number of samples in image */
)
{
    int i;
    long long im1_p3_p2_p1_p0, im2_p3_p2_p1_p0;
    int im1_p3_p2, im1_p1_p0, im2_p3_p2, im2_p1_p0;
    int res_p3_p2, res_p1_p0;

    for (i = 0; i < count >> 3; i += 8) {
        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        im1_p3_p2 = _hill (im1_p3_p2_p1_p0);
        im1_p1_p0 = _loll (im1_p3_p2_p1_p0);

        im2_p3_p2 = _hill (im2_p3_p2_p1_p0);
        im2_p1_p0 = _loll (im2_p3_p2_p1_p0);

        res_p3_p2 = _sub2 (im1_p3_p2, im2_p3_p2);
        res_p1_p0 = _sub2 (im1_p1_p0, im2_p1_p0);

        _amem8(imgW) = _itoll (res_p3_p2, res_p1_p0);
        imgW += 4;

        im1_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 4;
        imgR2 += 4;

        im1_p3_p2 = _hill (im1_p3_p2_p1_p0);
        im1_p1_p0 = _loll (im1_p3_p2_p1_p0);

        im2_p3_p2 = _hill (im2_p3_p2_p1_p0);
        im2_p1_p0 = _loll (im2_p3_p2_p1_p0);

        res_p3_p2 = _sub2 (im1_p3_p2, im2_p3_p2);
        res_p1_p0 = _sub2 (im1_p1_p0, im2_p1_p0);

        _amem8(imgW) = _itoll (res_p3_p2, res_p1_p0);
        imgW += 4;
    }
}
```

A.22 IMG_sub_8

```
/*----------------------------------------------------**
** This function performs subtraction of 2          **
** images. Each image consist of 8 bits per samples. **
**----------------------------------------------------*/

void IMG_sub_8
(
    char * restrict imgR1,    /* Image 1 read pointer */
    char * restrict imgR2,       /* Image 2 read pointer */
    char * restrict imgW,        /* Output image pointer */
    int count                 /* Number of samples in image */
)
{
    int i;
    long long im1_p7_p6_p5_p4_p3_p2_p1_p0, im2_p7_p6_p5_p4_p3_p2_p1_p0;
    int im1_p7_p6_p5_p4, im1_p3_p2_p1_p0, im2_p7_p6_p5_p4, im2_p3_p2_p1_p0;
    int res_p7_p6_p5_p4, res_p3_p2_p1_p0;

    for (i = 0; i < count >> 4; i += 16) {
        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        im1_p3_p2_p1_p0 = _loll (im1_p7_p6_p5_p4_p3_p2_p1_p0);
        im1_p7_p6_p5_p4 = _hill (im1_p7_p6_p5_p4_p3_p2_p1_p0);

        im2_p3_p2_p1_p0 = _loll (im2_p7_p6_p5_p4_p3_p2_p1_p0);
        im2_p7_p6_p5_p4 = _hill (im2_p7_p6_p5_p4_p3_p2_p1_p0);

        res_p3_p2_p1_p0 = _sub4 (im1_p3_p2_p1_p0, im2_p3_p2_p1_p0);
        res_p7_p6_p5_p4 = _sub4 (im1_p7_p6_p5_p4, im2_p7_p6_p5_p4);

        _amem8(imgW) = _itoll (res_p7_p6_p5_p4, res_p3_p2_p1_p0);
        imgW += 8;

        im1_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR1);
        im2_p7_p6_p5_p4_p3_p2_p1_p0 = _amem8(imgR2);
        imgR1 += 8;
        imgR2 += 8;

        im1_p3_p2_p1_p0 = _loll (im1_p7_p6_p5_p4_p3_p2_p1_p0);
        im1_p7_p6_p5_p4 = _hill (im1_p7_p6_p5_p4_p3_p2_p1_p0);

        im2_p3_p2_p1_p0 = _loll (im2_p7_p6_p5_p4_p3_p2_p1_p0);
        im2_p7_p6_p5_p4 = _hill (im2_p7_p6_p5_p4_p3_p2_p1_p0);

        res_p3_p2_p1_p0 = _sub4 (im1_p3_p2_p1_p0, im2_p3_p2_p1_p0);
        res_p7_p6_p5_p4 = _sub4 (im1_p7_p6_p5_p4, im2_p7_p6_p5_p4);

        _amem8(imgW) = _itoll (res_p7_p6_p5_p4, res_p3_p2_p1_p0);
        imgW += 8;

    }
}
```

Appendix B Benchmarks

This appendix lists the benchmarks of various functions within each category. These benchmarks are approximate details and dependent on the compiler version and subject to change for the newer versions. The test environment for the listed benchmarks are as listed below:

- Compiler version 6.0.9
- Single cycle access of (L1) flat memory. No other memory overheads are considered.

B.1 *Benchmarks for Image Analysis Functions*

These benchmarks are subject to change with the version of the compiler and/or considering other memory overheads. The performance formulae are indicative and actual figures might vary slightly based on the input dimensions and compiler version.

Table B-1. Benchmarks for Image Analysis Functions

Function	Int C Code		Input Image Size (rows x cols)	Performance Formulae[1]	
	C64x+	C64x		C64x+	C64x
IMG_boundary_8	473	479	3 x 120	$5 \times \dfrac{rows \times cols}{4} + 20$	$5 \times \dfrac{rows \times cols}{4} + 20$
IMG_boundary_16s	476	479	3 x 120	$5 \times \dfrac{rows \times cols}{4} + 20$	$5 \times \dfrac{rows \times cols}{4} + 20$
IMG_clipping_16s	4126	8217	128 x 128	$2 \times \dfrac{rows \times cols}{8} + 10$	$4 \times \dfrac{rows \times cols}{8} + 10$
IMG_dilate_bin	144	143	3x80	$5 \times \dfrac{cols}{4} + 40$	$5 \times \dfrac{cols}{4} + 40$
IMG_erode_bin	144	143	3x80	$5 \times \dfrac{cols}{4} + 40$	$5 \times \dfrac{cols}{4} + 40$
IMG_errdif_bin_8	5735	5711	4x128	$rows*(11*cols + 12) + 50$	$rows*(11*cols + 10) + 40$
IMG_errdif_bin_16	422	422	8x8	rows * (cols * 8 + 16) + 40	rows * (cols * 8 + 16) + 40
IMG_histogram_8	1073	1141	n=512	$10 \times \dfrac{n}{8} + 430$	$13 \times \dfrac{cols}{16} + 50$
IMG_histogram_16	98987	99008	n = 512 img_bits/ pixel = 16	$10 \times \dfrac{n}{8} + 3 \times \dfrac{(2^{img\_bits})}{2} + 40$	$10 \times \dfrac{n}{8} + 3 \times \dfrac{(2^{img\_bits})}{2} + 50$
IMG_median_3x3_8	563	626	n=256	$8 \times \dfrac{n}{4} + 50$	$9 \times \dfrac{n}{4} + 50$
IMG_perimeter_8	646	669	3x720	$13 \times \dfrac{cols}{16} + 50$	$14 \times \dfrac{cols}{16} + 40$

[1]   Intrinsic C code implementation benchmarks
NC → Not Compatible

Table B-1. Benchmarks for Image Analysis Functions (continued)

Function	Int C Code		Input Image Size (rows x cols)	Performance Formulae[1]	
	C64x+	C64x		C64x+	C64x
IMG_perimeter_16	1216	1311	3 x 720	$13 \times \dfrac{cols}{8} + 40$	$14 \times \dfrac{cols}{8} + 50$
IMG_pix_expand	223	426	1072	$3 \times \dfrac{cols}{16} + 20$	$6 \times \dfrac{cols}{16} + 25$
IMG_pix_sat	149	149	640	$3 \times \dfrac{cols}{16} + 30$	$3 \times \dfrac{cols}{16} + 30$
IMG_sobel_3x3_8	624	670	8 x 64	$12 \times \dfrac{((rows\ -\ 2) \times cols)}{8} + 48$	$13 \times \dfrac{((rows\ -\ 2) \times cols)}{8} + 46$
IMG_sobel_3x3_16s	947	NC	8 x 64	$9 \times \dfrac{((rows\ -\ 2) \times cols)\ -\ 2}{4} + 80$	NC
IMG_sobel_3x3_16	2912	2912	3 * 256	$15 \times \dfrac{cols * (rows - 2)}{4} + 32$	$15 \times \dfrac{cols * (rows - 2)}{4} + 32$
IMG_sobel_5x5_16s	1950	NC	8 x 64	$15 \times \dfrac{((rows\ -\ 4) \times cols)\ -\ 4}{2} + 30$	NC
IMG_sobel_7x7_16s	1233	NC	8 x 64	$19 \times \dfrac{((rows\ -\ 6) \times cols)\ -\ 6}{2} + 40$	NC
IMG_thr_gt2max_8	192	417	32 x 32	$3 * (rows * cols)/16 + 30$	$6 * (rows * cols)/16 + 30$
IMG_thr_gt2max_16	362	362	3 x 256	$7 * (rows * cols)/16 + 26$	$7 * (rows * cols)/16 + 26$
IMG_thr_gt2thr_8	155	411	32 x 32	$2 * (rows * cols)/16 + 27$	$6 * (rows * cols)/16 + 30$
IMG_thr_gt2thr_16	314	314	3 x 256	$6 * (rows * cols)/16 + 26$	$6 * (rows * cols)/16 + 26$
IMG_thr_le2min_8	223	418	32 x 32	$3 * (rows * cols)/16 + 30$	$6 * (rows * cols)/16 + 34$
IMG_thr_le2min_16	362	362	3 x 256	$7 * (rows * cols)/16 + 26$	$7 * (rows * cols)/16 + 26$
IMG_thr_le2thr_8	154	410	32 x 32	$2 * (rows * cols)/16 + 26$	$6 * (rows * cols)/16 + 26$
IMG_thr_le2thr_16	314	314	3 x 256	$6 * (rows * cols)/16 + 26$	$6 * (rows * cols)/16 + 26$
IMG_yc_demux_ be16_8	483	488	luma = 1024	$7 \times \dfrac{luma}{16} + 35$	$7 \times \dfrac{luma}{16} + 40$
IMG_yc_demux_ le16_8	418	488	luma = 1024	$6 \times \dfrac{luma}{16} + 35$	$7 \times \dfrac{luma}{16} + 40$
IMG_ycbcr422p_rgb565	1268	1266	luna=640	$15 \times \dfrac{luma}{8} + 65$	$15 \times \dfrac{luma}{8} + 65$

B.2 Benchmarks for Picture Filtering / Format Conversion Functions

Table B-2. Benchmarks for Picture Filtering Functions

Function	Int C Code		No. of Outputs (width)	Performance Formulae[1]	
	C64x+	C64x		C64x+	C64x
IMG_conv_3x3_i8_c8s	760	775	480	$6 \times \dfrac{cols}{4} + 40$	$6 \times \dfrac{cols}{4} + 55$
IMG_conv_3x3_i16s_c16s	557	NC	256	$4 \times \dfrac{width}{2} + 30$	NC
IMG_conv_3x3_i16_c16s	930	930	256	$14 \times \dfrac{width}{4} + 34$	$14 \times \dfrac{width}{4} + 34$
IMG_conv_5x5_i8_c8s	1151	1148	256	$17 \times \dfrac{width}{4} + 30$	$17 \times \dfrac{width}{4} + 30$
IMG_conv_5x5_i16s_c16s	1592	NC	256	$12 \times \dfrac{width}{2} + 30$	NC
IMG_conv_5x5_i8_c16s	1711	NC	256	$13 \times \dfrac{width}{2} + 40$	NC
IMG_conv_7x7_i8_c8s	1845	1847	256	$15 \times \dfrac{width}{2}$	$15 \times \dfrac{width}{2}$
IMG_conv_7x7_i16s_c16s	3178	NC	256	$90 \times \dfrac{width}{8}$	NC
IMG_conv_7x7_i8_c16s	6035	NC	256	$48 \times \dfrac{width}{2}$	NC
IMG_conv_11x11_i8_c8s	5597	5417	256	$120 \times \dfrac{width}{2}$	$120 \times \dfrac{width}{2}$
IMG_conv_11x11_i16s_c16s	9920	NC	256	$160 \times \dfrac{width}{4}$	NC
IMG_corr_3x3_i8_c16s	674	NC	256	$5 \times \dfrac{width}{2} + 30$	NC
IMG_corr_3x3_i16_c16s	1202	1202	256	$18 \times \dfrac{n\_out}{4} + 50$	$18 \times \dfrac{n\_out}{4} + 50$
IMG_corr_3x3_i8_c8	248	394	296	$6 \times \dfrac{n\_out}{4} + 40$	$6 \times \dfrac{n\_out}{4} + 55$
IMG_corr_3x3_i16_c16s	558	NC	256	$4 \times \dfrac{width}{2} + 30$	NC
IMG_corr_5x5_i16s_c16s	1595	NC	256	$12 \times \dfrac{width}{2} + 30$	NC

[1] Intrinsic C code implementation benchmarks
NC → Not Compatible

Table B-2. Benchmarks for Picture Filtering Functions (continued)

Function	Int C Code		No. of Outputs (width)	Performance Formulae[1]	
	C64x+	C64x		C64x+	C64x
IMG_corr_11x11_ i16s_c16s	10671	NC	256	$90 \times \dfrac{width}{2}$	NC
IMG_corr_11x11_ i8_c16s	11044	NC	256	$90 \times \dfrac{width}{2}$	NC
IMG_corr_gen_ i16s_c16s	2094	2094	For odd no. of taps (m) X = 720, m = 9	$(m-1)*\left[floor\left(\dfrac{(X-m+7)}{4}\right)+11\right]+3*floor\left(\dfrac{(x-m+3)}{4}\right)+40$	
	1557	1557	For even no. of taps (m) X = 720, m = 8	$m*\left[floor\left(\dfrac{X-m+8}{4}\right)+11\right]+30$	
IMG_corr_gen_iq	14232	14219	Input width x_dim = 720 Filter Length m =10 Outputs: x_dim − m = 710	$\left(\dfrac{m}{2}\times 4+20\right)\times\dfrac{(x\_dim - m)}{2}+10$	$\left(\dfrac{m}{2}\times 4+20\right)\times\dfrac{(x\_dim - m)}{2}+10$
IMG_median_3x3_ 16s	734	1187	3 x 256	$11\times\dfrac{n}{4}+30$	$18\times\dfrac{n}{4}+30$
IMG_median_3x3_16	975	975	n = 256	$14\times\dfrac{n}{4}+79$	$14\times\dfrac{n}{4}+79$
IMG_yc_demux_ be16_16	522	522	num_luma = 1024	$4\times\dfrac{num\_luma}{8}+10$	$4\times\dfrac{num\_luma}{8}+10$
IMG_yc_demux_ le16_16	522	522	num_luma = 1024	$4\times\dfrac{num\_luma}{8}+10$	$4\times\dfrac{num\_luma}{8}+10$

B.3 *Benchmarks for Compression/Decompression Functions*

Table B-3. Benchmarks for Compression/Decompression Functions

| Function | ASM Code | | No. of Outputs (width) | Performance Formulae | |
	C64x+	C64x		C64x+	C64x
IMG_fdct_8x8	368	NC	num_fdcts=6	52 × num_fdcts + 56	NC
IMG_idct_8x8_12q4	614	NC	num_idcts=6	72 × num_idcts + 63	NC
IMG_mad_8x8	194	NC	sx=4, sy=4	8 × sx × sy + 66	NC
IMG_mad_16x16	628	NC	sx=4, sy=4	38 × sx × sy + 20	NC
IMG_mpeg2_vld_intra	1505	1505	S=120, CB=6, NCB=0	10 × (S-CB) + 55 × CB + 15 × NCB + 35	10 × (S-CB) + 55 × CB + 15 × NCB + 35
IMG_mpeg2_vld_inter	1032	1032	S=80, CB=5, NCB=1	10 ×S + 37 × CB + 15 × NCB + 34	10 × S + 37 × CB + 15 × NCB + 34
IMG_quantize	282	NC	blk_size=64, num_blks=8	(blk_size/16) × num_blks 8 + 26	NC
IMG_sad_8x8	31	NC			NC
IMG_sad_16x16	67	NC			NC
IMG_wave_horz	545	798	256	$4 \times \dfrac{cols}{2} + 30$	$6 \times \dfrac{cols}{2} + 30$
IMG_wave_vert	3302	4875	800	$8 \times \dfrac{cols}{2} + 100$	$12 \times \dfrac{cols}{2} + 75$

Appendix C Revision History

This document has been revised because of the following technical change(s).

Table C-1. Additions, Deletes

Location	Description of Change
Global	Changed instances of **I**64P to **I**64P
Global	Removed all references to these functions: IMG_idct_8x8 IMG_pix_expand_nM32 IMG_median_5x5_16s
Section 1.1.1	Added the last paragraph to this section
Section 2.1	Changed: The DSPLIB is provided in the file img64plus.zip. The file must be unzipped to provide the following directory structure. To IMGLIB is provided as a self installing executable imglibc64plus-2.x.x-Setup.exe. Upon installation, it produces the following directory srtucture.
Section 2.1	Changed the last paragrah of this section.
Section 2.2.2	Changed: img64plus2_0_host.lib To : imglib2_host.lib
Section 3.3.1	Added IMG_boundary_8 and IMG_PERIMETER_8 and edited text
Section 3.3.3	Added IMG_sobel_3x3-8 and edited text
Section 3.3.4	Added IMG_histogram_8 and edited text
Section 3.3.5	Added four functions and edited text
Section 3.4.1	Added new functions and edited text
Section 3.4.2	Revised the entire section
Section 3.4.3	Revised the entire section
Section 3.4.5	Added IMG_median_3x3_8 and edited text
Section 3.5.1	Added IMG_idct_8x8_12q4
Section 6.14	In the Requirements section, changed from n_out should be a multiple of **2** to n_out should be a multiple of **4**
Section 7.6 and Section 7.5	For argument mod_12Q4, replaced IMG_idct_8x8 with IMG_idct_8.8_**12q4**
Table B-2	Added benchmark formula for IMG_corr_gen_i16s_c16s()