TMS320C64x Image/Video Processing Library

Programmer's Reference

Literature Number: SPRU023B
October 2003

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

Welcome to the TMS320C64x™ Image/Video Library (IMGLIB). The IMGLIB is a collection of high-level optimized DSP functions for the TMS320C64x device. This source code library includes C-callable functions (ANSI–C language compatible) for general-purpose imaging functions that include compression, video processing, machine vision, and medical imaging type applications.

This document contains a reference for the IMGLIB functions and is organized as follows:

- ❏ Overview – an introduction to the TI 64x IMGLIB
- ❏ Installation – information on how to install and rebuild IMGLIB
- ❏ IMGLIB Functions – a description of the routines in the library and how they are organized
- ❏ IMGLIB Function Tables – a list of functions grouped by categories
- ❏ IMGLIB Reference – a detailed description of each IMGLIB function
- ❏ Information about performance and support

How to Use This Manual

These chapters describe the contents of the TMS320C64x IMGLIB:

- ❏ **Chapter 1 – Introduction** provides a brief introduction to the TI 64x IMGLIB, shows the organization of the routines contained in the library, and lists the features and benefits of the IMGLIB.
- ❏ **Chapter 2 – Installing and Using IMGLIB**, provides information on how to install, use, and rebuild the TI C64x IMGLIB.
- ❏ **Chapter 3 – IMGLIB Function Descriptions** provides a brief description of each IMGLIB function.
- ❏ **Chapter 4 – IMGLIB Function Tables** provides information about each IMGLIB function in table format for easy reference. The information shown

for each function includes the syntax, a brief description, and a page reference for obtaining more detailed information.

- **Chapter 5 – IMGLIB Reference** 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.
- **Appendix A – Performance and Support** describes performance considerations related to the C64xIMGLIB and provides information about software updates and customer support.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`.
- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- The TMS320C64x is also referred to in this reference guide as the C64x.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

Documentation:

TMS320/C64x Technical Overview (literature number SPRU395) gives an introduction to the C64x digital signal processor, and discusses the application areas that are enhanced by the C64x VelociTI.2 extensions to the C62x/C67x architecture.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced DMA (EDMA), expansion bus, clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

TMS320C6000 Chip Support Library (literature number SPRU401) describes the application programming interfaces (APIs) used to configure and control all on-chip peripherals.

TMS320C64x DSP Library (literature number SPRU565) describes high-level, C-callable, optimized DSP functions for general signal processing, math, and vector operations.

Image Processing Examples Using TMS320C64x Image Processing Library (literature number SPRA887) describes the usage and performance of key IMGLIB functions.

TMS320C6000 DSP Cache User's Guide (literature number SPRU656A) describes cache architectures in detail and presents how to optimize algorithms and function calls for better cache performance.

Trademarks

Windows is a registered trademark of Microsoft Corporation.

TMS320C62x and TMS320C64x is a trademark of Texas Instruments.

TMS320C6000 is a trademark of Texas Instruments.

Code Composer Studio is a trademark of Texas Instruments.

Contents

Tables

Chapter 1

Getting Started

This chapter introduces the TMS320C64x™ Image/Video Library (IMGLIB) and describes its features and benefits.

Topic **Page**

1.1 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. By using these routines, you can achieve 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 your image/video processing application development time.

1.2 Features and Benefits

The TI C64x IMGLIB contains commonly used image/video processing routines. Source code is provided 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
- ❏ Benchmarks (cycles and code size)
- ❏ Tested against reference C model

1.2.1 Software Routines

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

- ❏ Compression and decompression
- ❏ Image Analysis
- ❏ Picture filtering/format conversions

Chapter 2

Installing and Using IMGLIB

This chapter provides information on how to install, use, and rebuild IMGLIB.

Topic **Page**

2.1 Installing IMGLIB

Note:

You should read the README.txt file for specific details of the release.

The archive has the following structure:

```
 img64x.zip
|
+-- README.txt           Top-level README file
|
+-- lib
|   |
|   +-- img64x.lib       Library archive
|   |
|   +-- img64x.src       Full source archive
|   |                    (Hand-assembly and headers)
|   +-- img64x_sa.src    Full source archive
|   |                    (Linear asm and headers)
|   +-- img64x_c.src     Full source archive
|                        (C and headers)
|
+-- include              Unpacked header files
|
+-- examples             Example files
|
+-- doc
    |
    +-- img64xlib.pdf    This document
```

2.1.1 De-Archiving IMGLIB

The *lib* directory contains the library archive and the source archive. Please install the contents of the lib directory in a directory pointed by your C_DIR environment. If you choose to install the contents in a different directory, make sure you update the C_DIR environment variable, for example, by adding the following line in autoexec.bat file:

```
SET C_DIR=<install_dir>/lib;<install_dir>/include;%C_DIR%
```

or under Unix/csh:

```
setenv C_DIR "<install_dir>/lib;<install_dir>/
include; $C_DIR"
```

or under Unix/Bourne Shell:

```
C_DIR="<install_dir>/lib;<install_dir>/include;$C_DIR" ;
export C_DIR
```

2.2 Using IMGLIB

2.2.1 Calling an IMGLIB Function From C

In addition to correctly installing the IMGLIB software, you must follow these steps to include an IMGLIB function in your code:

- ❏ Include the function header file corresponding to the IMGLIB function
- ❏ Link your code with img64x.lib
- ❏ Use a correct linker command file for the platform you use. Remember most functions in img64x.lib are written assuming little endian mode of operation.

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

```
#include <IMG_fdct_8x8.h>
```

in your C file and compile and link using

```
cl6x main.c -z -o IMG_fdct_8x8_drv.out -lrts6400.lib
-limg64x.lib
```

Code Composer Studio Users

Assuming your C_DIR environment is correctly set up (as mentioned in Section 2.1*, Installing IMGLIB*), you must add IMGLIB in the Code Composer Studio environment by choosing img64x.lib from the menu *Project –> Add Files to Project*. Also, please make sure you link with the correct runtime support library.

2.2.2 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. Please refer to Section 8, *Runtime Environment,* of *TMS320C6000 Optimizing C Compiler User's Guide* (Literature Number SPRU187).

2.2.3 How IMGLIB is Tested – 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.

2.2.4 How IMGLIB Deals with Overflow and Scaling Issues

The IMGLIB functions implement the exact functionality of the reference C code. The user is expected to conform to the range requirements specified in the function API and also to be responsible for restricting the input range in such a way that the outputs do not overflow.

2.2.5 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. 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 interruptibility. 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 categories tolerate interrupts. That is, an interrupt can occur at any time without affecting the correctness of the function. The interruptiblity of the function only determines how long the kernel might delay the processing of the interrupt.

2.2.6 Code Composer Studio Users

If you set up a project Under Code Composer Studio, you could add IMGLIB by choosing img64x.lib from the menu *Project –> Add Files to Project*. Also

please make sure you link with the correct run-time support library and IMGLIB by having the following lines in your linker command file:

```
-lrts6400.lib
-limg64x.lib
```

The *include* directory contains the header files necessary to be included in the C code when you call an IMGLIB function from C code.

2.3 Rebuilding IMGLIB

If you would like to rebuild IMGLIB (for example, because you modified the source file contained in the archive), you will have to use the mk6x utility as follows:

```
mk6x img64x.src –l img64x.lib
```

Chapter 3

IMGLIB Function Descriptions

This chapter provides a brief description of each IMGLIB function listed in three catagories. It also gives representative examples of their areas of applicability.

Topic **Page**

3.1 IMGLIB Functions Overview

The C64x IMGLIB 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 DSP. Some of the functions provided and their areas of applicability are listed below. The areas of applicability are only provided as representative examples; users of this software will surely conceive many more creative uses.

3.2 Compression/Decompression

This sections describes the functions that are applicable to compression/decompression standards such as JPEG, MPEG video, and H.26x.

- IMG_fdct_8x8
- IMG_idct_8x8

  Forward and Inverse DCT (Discrete Cosine Transform) functions, `IMG_fdct_8x8` and `IMG_idct_8x8`, respectively, are provided. These functions have applicability in 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 such as:

  - 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.

- IMG_mad_8x8
- IMG_mad_16x16
- IMG_sad_8x8
- IMG_sad_16x16

  These functions are provided to enable high performance motion estimation algorithms used in applications such as MPEG Video Encode, or H.26x Encode. 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 high performance enabled by the functions provided can enable significant improvements in such systems.

- 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.

- 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` can be used in such systems to perform the quantization step.

- IMG_wave_horz
- IMG_wave_vert

  Wavelet processing is finding increasing use 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. Included in this release are utilities `IMG_wave_horz` and `IMG_wave_vert` for computing horizontal and vertical wavelet transforms. Together, they can be used to 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.

3.3 Image Analysis

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

- IMG_boundary

  Boundary and Perimeter computation functions, `IMG_boundary` and `IMG_perimeter`, are provided. These are commonly used structural operators in machine vision applications.

- 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, Closing, etc. that can be created from combinations of dilation and erosion. These functions are useful in machine vision and medical imaging applications.

- IMG_histogram

  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 will contain a count of the number of pixels in the image that have that particular intensity value. Histogram processing (such as histogram equalization or modification) are used in areas such as machine vision systems and image/video content generation systems.

- IMG_perimeter

  Boundary and Perimeter computation functions, `IMG_boundary` and `IMG_perimeter`, are provided. These are commonly used structural operators in machine vision applications.

- IMG_sobel

  Edge Detection is a commonly-used operation in machine vision systems. Many algorithms exist for edge detection, and one of the most commonly used ones is *Sobel Edge Detection*. The routine `IMG_sobel` provides an optimized implementation of this edge detection algorithm.

- IMG_thr_gt2max
- IMG_thr_gt2thr
- IMG_thr_le2min
- IMG_thr_le2thr

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 Picture Filtering/Format Conversions

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

- IMG_conv_3x3

  The convolution function is used to apply generic image filters with a 3x3 filter mask, such as image smoothing, sharpening, etc.

- IMG_corr_3x3

- IMG_corr_gen

  Correlation functions are provided to enable image matching. Image matching is useful in applications such as machine vision, medical imaging, and security/defense. Two versions of correlation functions are provided: `IMG_corr_3x3` implements highly optimized correlation for commonly used 3x3 pixel neighborhoods, and a more general version, `IMG_corr_gen`, can implement correlation for user specified pixel neighborhood dimensions within documented constraints.

- IMG_errdif_bin

  Error Diffusion with binary valued output is useful in printing applications. The most widely used error diffusion algorithm is the Floyd-Steinberg algorithm. An optimized implementation of this algorithm is provided in the function, IMG_errdif_bin.

- IMG_median_3x3

  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 routine `IMG_median_3x3`.

- 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.

- 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
- IMG_yc_demux_le16

  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.

Chapter 4

IMGLIB Function Tables

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

Topic **Page**

4.1 IMGLIB Function Tables

The routines included in the image library are organized into three functional categories and listed below in alphabetical order.

Table 4−1. Compression/Decompression

Function	Description	Page
void IMG_fdct_8x8(short *fdct_data, unsigned num_fdcts)	Forward Discrete Cosine Transform (FDCT)	5-2
void IMG_idct_8x8(short *idct_data, unsigned num_idcts)	Inverse Discrete Cosine Transform (IDCT)	5-5
void IMG_idct_8x8_12q4(short *idct_data, unsigned num_idcts)	Inverse Discrete Cosine Transform (IDCT)	5-7
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	5-9
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	5-12
void IMG_mpeg2_vld_intra(short *Wptr, short *outi, unsigned int *Mpeg2v, int dc_pred[3])	MPEG−2 Variable Length Decoding of Intra MBs	5-15
void IMG_mpeg2_vld_inter(short *Wptr, short *outi, unsigned int *Mpeg2v)	MPEG−2 Variable Length Decoding of Inter MBs	5-19
void IMG_quantize (short *data, int num_blks, int blk_sz, const short *recip_tbl, int q_pt)	Matrix Quantization with Rounding	5-21
unsigned IMG_sad_8x8(unsigned char *srclmg, unsigned char *reflmg, int pitch)	Sum of Absolute Differences on Single 8x8 block	5-23
unsigned IMG_sad_16x16(unsigned char *srclmg, unsigned char *reflmg, int pitch)	Sum of Absolute Differences on Single 16x16 block	5-25
void IMG_wave_horz (short *in_data, short *qmf, short *mqmf, short *out_data, int cols)	Horizontal Wavelet Transform	5-27
void IMG_wave_vert (short *in_data[ ], short *qmf,short *mqmf,short *out_ldata,short *out_hdata,int cols)	Vertical Wavelet Transform	5-32

Table 4–2. Image Analysis

Function	Description	Page
void IMG_boundary(unsigned char *in_data, int rows, int cols, int *out_coord, int *out_gray)	Boundary Structural Operator	5-36
void IMG_dilate_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Dilation	5-38
void IMG_erode_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Erosion	5-40
void IMG_histogram (unsigned char *in_data, int n, int accumulate, unsigned short *t_hist, unsigned short *hist)	Histogram Computation	5-42
void IMG_perimeter (unsigned char *in_data, int cols, unsigned char *out_data)	Perimeter Structural Operator	5-45
void IMG_sobel(const unsigned char *in_data, unsigned char *out_data, short cols, short rows)	Sobel Edge Detection	5-47
void IMG_thr_gt2max(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding – Clamp to 255	5-50
void IMG_thr_gt2thr(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding – Clip above threshold	5-52
void IMG_thr_le2min(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding – Clamp to zero	5-54
void IMG_thr_le2thr(unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Thresholding – Clip above threshold	5-56

Table 4–3. Picture Filtering/Format Conversions

Function	Description	Page
void IMG_conv_3x3(unsigned char *in_data, unsigned char *out_data, int cols, char *mask, int shift)	3x3 Convolution	5-58
void IMG_corr_3x3(const unsigned char *in_data, int *out_data, unsigned char *mask[3][3], iant x_dim, int n_out	3x3 Correlation	5-61
void IMG_corr_gen(short *in_data, short *h, short *out_data, int m, int cols)	Generalized Correlation	5-64
void IMG_errdif_bin(unsigned char errdif_data[ ], int cols, int rows, short err_buf[ ], unsigned char thresh)	Error Diffusion, Binary Output	5-68
void IMG_median_3x3(unsigned char *in_data, int cols, unsigned char *out_data)	3x3 Median Filter	5-72
void IMG_pix_expand(int n, unsigned char *in_data, short *out_data)	Pixel Expand	5-74
void IMG_pix_sat(int n, short *in_data, unsigned char *out_data)	Pixel Saturation	5-75
void IMG_yc_demux_be16(int n, unsigned char *yc, short *y, short *cr, short *cb)	YCbCr Demultiplexing (big endian source)	5-79
void IMG_yc_demux_le16(int n, unsigned char *yc, short *y, short *cr, short *cb)	YCbCr Demultiplexing (little endian source)	5-79
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	5-81

Chapter 5

IMGLIB Reference

This chapter 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.

Topic **Page**

5.1 Compression/Decompression

IMG_fdct_8x8 *Forward Discrete Cosine Transform (FDCT)*

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 Forward Discrete Cosine Transform (FDCT) is described by the following equation:

$$I(u,v) = \frac{\alpha(u)\alpha(v)}{4} \times \sum_{x=0}^{7}\sum_{y=0}^{7} i(x,y)\cos\left(\frac{(2x+1)u\pi}{16}\right)\cos\left(\frac{(2y+1)v\pi}{16}\right)$$

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.

Implementation Notes

- The code is setup to provide an early exit if it is called with num_fdcts = 0. In such case 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 some 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.
- ***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.

Benchmarks

Cycles	76 * num_fdcts + 50
	For num_fdtcs = 6, cycles = 496 For num_fdcts = 24, cycles = 1864
Code size	976 bytes

IMG_idct_8x8 *Inverse Discrete Cosine Transform (IDCT)*

void IMG_idct_8x8(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 12-bit DCT coefficients.

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

Note: This function simply converts the input data to 12Q4 format and calls the IMG_idct_8x8_12q4 function. It is provided here for backward compatibility with an earlier release of IMGLIB. The format conversion performed is a shift left by four of each input DCT coefficient. In a typical application this conversion can be integrated, for instance, in the de-quantization step without additional overhead, and then the routine IMG_idct_8x8_12q4 called directly.

Algorithm

The Inverse Discrete Cosine Transform (IDCT) is described by the following equation:

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

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.
- The code is set up to provide an early exit if it is called with num_idcts = 0. In such case it will run for 13 cycles.
- The IMG_idct_8x8 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.

Implementation Notes

- This function simply converts the input data to 12Q4 format and calls the routine IMG_idct_8x8_12q4.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is fully interruptible and fully re-entrant.

Benchmarks

Cycles	92 * num_idcts + 62 For num_idcts = 6, cycles = 614 For num_idcts = 24, cycles = 2270
Code size	968 bytes

IMG_idct_8x8_12q4 *Inverse Discrete Cosine Transform(IDCT)*

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 8x8 IDCTs on a list of 8x8 blocks.

Algorithm

The Inverse Discrete Cosine Transform (IDCT) is described by the following equation:

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

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 such case 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.

Implementation Notes

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

Benchmarks

Cycles	92 * num_idcts + 62 For num_idcts = 6, cycles = 614 For num_idcts = 24, cycles = 2270
Code size	968 bytes

IMG_mad_8x8 *8x8 Minimum Absolute Difference*

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 8x8 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 8x8 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 8x8 pixel block in a reference image which most closely matches the 8x8 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_8x8
(
    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.

Implementation Notes

- 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. This code thus executes 8 instructions per cycle, and computes 8 absolute differences per cycle.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt–tolerant, but not interruptible.

Benchmarks

Cycles	8 * sx * sy + 66 For sx = 4, sy = 4, cycles = 194 For sx = 64, sy = 32, cycles = 16,450
Code size	788 bytes

IMG_mad_16x16 *16x16 Minimum Absolute Difference*

void IMG_mad_16x16 (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 16x16 pixel block in a reference image which most closely matches the 16x16 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.

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

```
void IMG_mad_16x16
(
    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.

Implementation Notes

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

Benchmarks

Cycles	38 * sx * sy + 20 For sx = 4, sy = 4, cycles = 628 For sx = 64, sy = 32, cycles = 77,844
Code size	776 bytes

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

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 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. The structure is described below.
Mpeg2v	Pointer to a context structure containing 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 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 becasue 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 which 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 since they are accessed by this routine. If the layout is changed, the corresponding changes have to be made in code too.

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 theses cases 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 which 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 has to be performed only 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 which are in little Endian byte order.
- Wptr is allowed to overrun once in order 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 loactions whose read access does not cause any side effects (e.g. peripherals).
- Note that the AMR register is set to zero on exit.

Implementation Notes

- 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 look–up 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.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** This code is LITTLE ENDIAN.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	10 * (S – CB) + 55 * CB + 15 * NCB + 35 where S is the number of symbols in the MB, CB: the number of coded blocks and NCB the number of non-coded blocks (NCB = 6 – CB). For S = 120, CB = 6, NCB = 0, cycles = 1505 For S = 200, CB = 6, NCB = 0, cycles = 2305
Code size	1496 bytes
Data size	3584 bytes for lookup tables

IMG_mpeg2_vld_inter *MPEG-2 Variable Length Decoding of Inter MBs*

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 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 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 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 which illustrates 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 in order 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 loactions whose read access does not cause any side effects (e.g. peripherals).
- ❏ Note that the AMR register is set to zero on exit.

Implementation Notes

- ❏ 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 look–up 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.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** This code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	10 * S + 37 * CB + 15 * NCB + 34 where S is the number of symbols in the MB, CB: the number of coded blocks and NCB the number of non-coded blocks (NCB = 6 – CB). For S = 80, CB = 5, NCB = 1, cycles = 1032 For S = 192, CB = 6, NCB = 0, cycles = 2174
Code size	1212 bytes
Data size	1792 bytes for lookup tables

IMG_quantize *Matrix Quantization with Rounding*

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 ≤ 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 $0 \leq q\_pt \leq 31$.
- 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.

Implementation Notes

- The outer loop is unrolled 16 times to allow greater amounts of work to be performed in the inner loop. of work to be performed in the inner loop. The resulting loop-nest was 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 best use of the available memory bandwidth.
- SSHVR is used in the M-unit to avoid an S-unit bottleneck.
- Twin stack pointers have been used to speed up stack accesses.
- ***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.

Benchmarks

Cycles	(blk_size/16) * num_blks * 8 + 26 For blk_size = 64, num_blks = 8, cycles = 282 For blk_size = 256, num_blks = 24, cycles = 3098
Code size	580 bytes

IMG_sad_8x8 *Sum of Absolute Differences on Single 8x8 block*

unsigned sad_8x8(const unsigned char * restrict srcImg, const unsigned char * restrict refImg, int pitch)

Arguments

srcImg[64]	8x8 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 8x8 region pointed to in the reference image.

The code accepts a pointer to the 8x8 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_8x8
(
    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.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.

- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** This code blocks interrupts for 25 cycles.

Benchmarks

Cycles	31
Code size	164 bytes

IMG_sad_16x16 *Sum of Absolute Differences on Single 16x16 block*

unsigned sad_16x16(const unsigned char * restrict srcImg, const unsigned char * restrict refImg, int pitch)

Arguments

srcImg[256]	16x16 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 16x16 region pointed to in the reference image.

The code accepts a pointer to the 16x16 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_16x16
(
    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.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.

- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** This code blocks interrupts for 61 cycles.

Benchmarks

Cycles	67
Code size	168 bytes

IMG_wave_horz *Horizontal Wavelet Transform*

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 decimated by 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 and hence 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.

Implementation Notes

- The main ideas used for optimizing the code include 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 and hence this creates 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. Thus 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 in this fashion 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.
- ***Bank Conflicts****:* The code has no bank conflicts.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt–tolerant, but not interruptible.

Benchmarks

Cycles	2 * cols + 25 For cols = 256, cycles = 537 For cols = 512, cycles = 1049
Code size	360 bytes

IMG_wave_vert *Vertical Wavelet Transform*

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 which 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, for a given pair of output lines, the input context for the low-pass filter is offset by a number of lines from the input context for the high-pass filter. 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. In order 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 be used to 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:

```
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:

```
status = -1;
for (i = 0; i < M; i++)
{
    status = status * -1;
    hdata  = qmf[i] * status;
    filter[i] = hdata;
}
```

Implementation Notes

- 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, in order to save code size. This is done by loading the filter coefficients in a special order.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt–tolerant, but not interruptible.

Benchmarks

Cycles	4 * cols + 96 For cols = 256, cycles = 1120 For cols = 512, cycles = 2144
Code size	888 bytes

5.2 Image Analysis

IMG_boundary *Boundary Structural Operator*

void IMG_boundary(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
(
    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.

Implementation Notes

- Outer and inner loops are collapsed together.
- Inner loop is unrolled to process four pixels per iteration.
- ***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 will cause a bank conflict.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt–tolerant but not interruptible.

Benchmarks

Cycles	1.25 * (cols * rows) + 12 For cols = 128, rows = 3, cycles = 492 For cols = 720, rows = 8, cycles = 7212
Code size	132 bytes

IMG_dilate_bin *3x3 Binary Dilation*

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.

Implementation Notes

- The 3x3 dilation mask is applied to 32 output pixels simultaneously. This is done with 32-bit-wide bit-wise operators in the register file. In order 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 in order to make the rightmost two pixels well-defined.
- ***Bank Conflicts:*** No bank conflicts occur in this function.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	7 * cols/8 + 25 For cols = 128, cycles = 137 For cols = 720, cycles = 655
Code size	324 bytes

IMG_erode_bin *3x3 Binary Erosion*

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 3x3 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.

Implementation Notes

- The 3x3 erosion mask is applied to 32 output pixels simultaneously. This is done with 32-bit-wide bit-wise operators in the register file. In order 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 in order to make the rightmost two pixels well-defined.
- ***Bank Conflicts:*** No bank conflicts occur in this function.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	7 * cols/8 + 25 For cols = 128, cycles = 137 For cols = 720, cycles = 655
Code size	324 bytes

IMG_histogram *Histogram Computation*

void IMG_histogram (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.

Implementation 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	…

hst0,…,hst3 are the four histograms and bin0, bin1,… are the bins used. These are then summed together at the end in blocks of 4.

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

Benchmarks

Cycles	9/8 * n + 228
	For n = 512, cycles = 804 For n = 1024, cycles = 1380
Code size	552 bytes

IMG_perimeter *Perimeter Structural Operator*

void IMG_perimeter (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.

Implementation 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 ENIDAN.
- ❏ ***Interruptibility:*** The code is interrupt–tolerant but not interruptible.

Benchmarks

Cycles	10 * cols/16 + 55
	For cols = 128, cycles = 135
	For cols = 720, cycles = 505
Code size	600 bytes

IMG_sobel *Sobel Edge Detection*

void IMG_sobel(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.

Implementation Notes

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

Benchmarks

Cycles	11 * cols * (rows − 2)/8 + 23 For cols = 128, rows = 8, cycles = 1079 For cols = 720, rows = 8, cycles = 5963
Code size	688 bytes

IMG_thr_gt2max *Thresholding – Clamp to 255*

void IMG_thr_gt2max(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.

Please 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.

Implementation Notes

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

Benchmarks

Cycles	0.1875 * rows * cols + 22
	For cols = 32 and rows = 32, cycles = 214
Code size	144 bytes

IMG_thr_gt2thr *Thresholding – Clip above threshold*

void IMG_thr_gt2thr(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.

Please 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.

Implementation 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.

Benchmarks

Cycles	0.125 * rows * cols + 20
	For rows = 32 and cols = 32, cycles = 148
Code size	108 bytes

IMG_thr_le2min *Thresholding – Clamp to zero*

void IMG_thr_le2min(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.

Please 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.

Implementation 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.

Benchmarks

Cycles	0.1875 * rows * cols + 22
	For rows = 32 and cols = 32, cycles = 214
Code size	144 bytes

IMG_thr_le2thr *Thresholding – Clip below threshold*

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.

Please 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.

Implementation Notes

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

Benchmarks

Cycles	0.125 *cols * rows + 20
	For rows = 32, cols = 32, cycles = 148
Code size	108 bytes

5.3 Picture Filtering/Format Conversions

IMG_conv_3x3 *3x3 Convolution*

void IMG_conv_3x3(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]	3x3 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 ADD2 instruction to compute two pixels in a register.

Implementation Notes

- 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. This is achieved by the use of 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.
- ***Bank Conflicts****:* No bank conflicts occur in this function.
- ***Endian****:* The code is LITTLE ENDIAN.
- ***Interruptibility****:* The code is interrupt–tolerant, but not interruptible.

Benchmarks

Cycles	9 * cols/8 + 33 For cols = 256, cycles = 321 For cols = 720, cycles = 843
Code size	800 bytes

IMG_corr_3x3 *3x3 Correlation*

void IMG_corr_3x3(const unsigned char * restrict in_data, int * restrict out_data, const unsigned char mask[3][3], int x_dim, int n_out)

Arguments

in_data[ ]	Pointer to input array of 8-bit pixels
out_data[ ]	Pointer to output array of 32-bit values
mask[3][3]	Pointer to 8-bit mask
x_dim	Width of image
n_out	Number of outputs. Must be multiple of 8.

Description

This routine performs a point-by-point multiplication of the 3x3 mask with the input image. The result of the nine multiplications are then summed to produce a 32-bit sum. The sum is then stored in an output array. The image mask to be correlated is typically part of the input image or another image. 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.

In an application the correlation kernel is called once for every row as shown below:

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

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, mask, x_dim, 2 * x_dim);
```

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 will have to account for this when interpreting the results.

Algorithm Behavioral C code is provided below:

```
void IMG_corr_3x3
(
    const unsigned char *i_data,       /* input image       */
    int        *restrict o_data,       /* output image      */
    const unsigned char  mask[3][3],   /* convolution mask  */
    int                  x_dim,        /* width of image    */
    int                  n_out         /* number of outputs */
)
{
    int i, j, k;
    for (i = 0; i < n_out; i++)
    {
        int sum = 0;
        for (j = 0; j < 3; j++)
            for (k = 0; k < 3; k++)
                sum += i_data[j * x_dim + i + k] * mask[j][k];

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

Special Requirements

- ❏ The array pointed to by out_data must not alias with the array pointed to by in_data or mask.
- ❏ The number of outputs ‘n_out’ must be a multiple of 8. In cases where ‘n_out’ is not a multiple of 8, most applications can safely round ‘n_out’ up to the next multiple of 8 and ignore the extra outputs. This kernel does not round ‘n_out’ up for the user.

Implementation Notes

- ❏ The inner loops are unrolled completely. The outer loop is unrolled 8 times.
- ❏ We use 3 DOTPU4s to calculate the 3 rows of each output pixel. We then accumulate the 3 DOTPU4s to a 32-bit result and store them out. (Note that only 3 of every 4 8-bit MPYs in the DOTPU4 is actually used. The fourth MPY is unused.)

- We use non−aligned loads and stores to avoid alignment issues.
- Bank Conflicts: No bank conflicts occur.
- ***Endian**:* The code is LITTLE ENDIAN.
- ***Interruptibility**:* The code is fully interruptible.

Benchmarks

Cycles	1.5 * n_out + 22
	For n_out = 248, cycles = 394
Code size	296 bytes

IMG_corr_gen *Generalized Correlation*

void IMG_corr_gen(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).

Implementation Notes

- 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. In this case 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. This problem is solved with the following process:

  - 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. In order to annul this extra computation, 8 locations starting at out_data[cols–M] are zeroed out before returning to the calling function.

- ***Bank Conflicts:*** No bank conflicts occur.

- ***Endian:*** The code is ENDIAN NEUTRAL.

- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles (case 1 – even number of filter taps)	M * [floor[(cols – M + 8)/4] + 11] + 38 For M = 8, cols = 720, cycles = 1566
Cycles (case 2 – odd number of filter taps)	(M – 1) * [floor[(cols – E + 8)/4] + 11] + 3 * floor[(cols – E + 4)/4] + 48, where E = M + 1 For M = 9, cols = 720, cycles = 2102
Code size	604 bytes

IMG_errdif_bin *Error Diffusion, Binary Output*

void IMG_errdif_bin(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). In order 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.

Implementation Notes

- 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.
- ***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.

Benchmarks

Cycles	(4*cols + 11)*rows + 7 For cols = 128, rows = 128, cycles = 66,951 For cols = 720, rows = 480, cycles = 1,387,687
Code size	296 bytes

IMG_median_3x3 *3x3 Median Filter*

void IMG_median_3x3(const unsigned char * restrict in_data, int cols, unsigned char * restrict out_data)

Arguments

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

Description

This routine performs a 3x3 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 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:

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 MED values I10, I11, I12 are compared and their median value is retained, call it MED1.

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 alignement is required.

Implementation Notes

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

Benchmarks

Cycles	2 * cols + 32 For cols = 128, cycles = 288 For cols = 720, cycles = 1472
Code size	248 bytes

IMG_pix_expand *Pixel Expand*

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.

Implementation Notes

- ❏ 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.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** The code is interrupt–tolerant, but not interruptible.

Benchmarks

Cycles	3 * n/16 + 15 For n = 256, cycles = 63 For n = 1024, cycles = 207
Code size	108 bytes

IMG_pix_sat *Pixel Saturate*

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.

Special Requirements

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

Benchmarks

Cycles	3 * n/16 + 7 For n = 256, cycles = 55 For n = 1024, cycles = 199
Code size	116 bytes

IMG_yc_demux_be16 *YCbCR Demultiplexing (big endian source)*

void IMG_yc_demux_be16(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.

Please 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.

Implementation Notes

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

Benchmarks

Cycles	3 * n/8 + 18
	For n = 1024: 402 cycles
Code size	316 bytes

IMG_yc_demux_le16 *YCbCR Demultiplexing (little endian source)*

void IMG_yc_demux_le16(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.

Please 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.

Implementation Notes

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

Benchmarks

Cycles	3 * n/8 + 18
	For n = 1024: 402 cycles
Code size	352 bytes

IMG_ycbcr422p_rgb565

Planarized YCbCR 4:2:2/4:2:0 to RGB 5:6:5 color space conversion

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[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']
```

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.

1. Y'CbCr → RGB conversion with RGB levels that correspond to the 219−lev-

el range of Y'. Expected ranges are [16..235] for Y' and [16..240] for Cb and Cr.

```
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']
```

2. Y'CbCr → 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.

```
coeff[] = { 0x2543, 0x3313, -0x0C8A, -0x1A04, 0x408D };

[ 1.1644    0.0000    1.5960 ]    [ Y' -  16 ]      [ R']
[ 1.1644   -0.3918   -0.8130 ] *  [ Cb - 128 ]  =   [ G']
[ 1.1644    2.0172    0.0000 ]    [ Cr - 128 ]      [ B']
```

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]. General hue adjustment can not be performed, however, 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

Implementation Notes

- ❏ 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, 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. This however, cannot be fed directly to SPACKU4. This implies a shift left by 3 bits, which could be pretty expensive in terms of the number of shifts to be performed. Thus, to avoid being bottlenecked by so many shifts, the Y, Cr & 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.
- ***Bank Conflicts:*** No bank conflicts occur in this function.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt–tolerant but not interruptible.

Benchmarks

Cycles	12 * num_pixels/8 + 50 For num_pixels = 4096, cycles = 6,194 For num_pixels = 16,384, cycles = 24,626
Code size	952 bytes

Appendix A

Performance and Support

This appendix describes performance considerations related to the C64x IMGLIB and provides information about software updates and customer support issues.

Topic **Page**

A.1 Performance Considerations

The ceil( ) is used in some benchmark formulas to accurately describe the number of cycles. It returns a number rounded up, away from zero, to the nearest integer. For example, ceil(1.1) returns 2.

Although IMGLIB can be used as a first estimation of processor performance for a specific function, you should be aware that the generic nature of IMGLIB might add extra cycles not required for customer specific usage.

Benchmark cycles presented assume best case conditions, typically assuming all code and data are placed in L1 memory. Any extra cycles due to placement of code or data in L2/external memory or cache-associated effects (cache-hits or misses) are not considered when computing the cycle counts.

You should also be aware that execution speed in a system is dependent on where the different sections of program and data are located in memory. You should account for such differences when trying to explain why a routine is taking more time than the reported IMGLIB benchmarks.

For more information on additional stall cycles due to memory hierarchy, refer to the *Image Processing Examples Using TMS320C64x Image Processing Library (SPRA887)*. The *TMS320C6000 DSP Cache User's Guide (SPRU656)* presents how to optimize algorithms and function calls for better cache performance.

A.2 IMGLIB Software Updates

C64x IMGLIB software updates may be periodically released incorporating product enhancements and fixes as they become available. You should read the README.TXT available in the root directory of every release.

A.3 IMGLIB Customer Support

If you have questions or want to report problems or suggestions regarding the C64x IMGLIB, contact Texas Instruments at dsph@ti.com.

Appendix B

Glossary

A

address: The location of program code or data stored; an individually accessible memory location.

A-law companding: See *compress and expand (compand)*.

API: See *application programming interface.*

application programming interface (API): Used for proprietary application programs to interact with communications software or to conform to protocols from another vendor's product.

assembler: A software program that creates a machine language program from a source file that contains assembly language instructions, directives, and macros. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assert: To make a digital logic device pin active. If the pin is active low, then a low voltage on the pin asserts it. If the pin is active high, then a high voltage asserts it.

B

bit: A binary digit, either a 0 or 1.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*.

block: The three least significant bits of the program address. These correspond to the address within a fetch packet of the first instruction being addressed.

board support library (BSL): A set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control board level peripherals.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The C6x DSP supports booting from external ROM or the host port interface (HPI).

boundary: Boundary structural operator.

BSL: *See board support library.*

byte: A sequence of eight adjacent bits operated upon as a unit.

C

cache: A fast storage buffer in the central processing unit of a computer.

cache controller: System component that coordinates program accesses between CPU program fetch mechanism, cache, and external memory.

CCS: Code Composer Studio.

central processing unit (CPU): The portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data– and program–memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

chip support library (CSL): The CSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control all on–chip peripherals.

clock cycle: A periodic or sequence of events based on the input from the external clock.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

code: A set of instructions written to perform a task; a computer program or part of a program.

coder–decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

compiler: A computer program that translates programs in a high–level language into their assembly–language equivalents.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes: A–law (used in Europe) and ∝–law (used in the United States).

control register: A register that contains bit fields that define the way a device operates.

control register file: A set of control registers.

corr_3x3: 3x3 correlation with rounding.

corr_gen: Generalized correlation.

CSL: See *chip support library*.

D

device ID: Configuration register that identifies each peripheral component interconnect (PCI).

digital signal processor (DSP): A semiconductor that turns analog signals such as sound or light into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

dilate_bin: 3x3 binary dilation.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for and receives mastery of the memory bus so that data transfers can take place independent of the host.

DMA: See *direct memory access*.

DMA source: The module where the DMA data originates. DMA data is read from the DMA source.

DMA transfer: The process of transferring data from one part of memory to another. Each DMA transfer consists of a read bus cycle (source to DMA holding register) and a write bus cycle (DMA holding register to destination).

E

erode_bin: 3x3 binary erosion.

errdif_bin: Error diffusion, binary output.

evaluation module (EVM): Board and software tools that allow the user to evaluate a specific device.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

external memory interface (EMIF): Microprocessor hardware that is used to read to and write from off–chip memory.

F

fast fourier transform (FFT): An efficient method of computing the discrete Fourier transform algorithm, which transforms functions between the time domain and the frequency domain.

fdct_8x8: Forward discrete cosine transform (FDCT).

fetch packet: A contiguous 8–word series of instructions fetched by the CPU and aligned on an 8–word boundary.

FFT: *See fast fourier transform.*

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

frame: An 8–word space in the cache RAMs. Each fetch packet in the cache resides in only one frame. A cache update loads a frame with the requested fetch packet. The cache contains 512 frames.

G

global interrupt enable bit (GIE): A bit in the control status register (CSR) that is used to enable or disable maskable interrupts.

H

HAL: *Hardware abstraction layer* of the CSL. The HAL underlies the service layer and provides it a set of macros and constants for manipulating the peripheral registers at the lowest level. It is a low–level symbolic interface into the hardware providing symbols that describe peripheral registers/ bitfields and macros for manipulating them.

histogram: Histogram computation.

host: A device to which other devices (peripherals) are connected and that generally controls those devices.

host port interface (HPI): A parallel interface that the CPU uses to communicate with a host processor.

HPI: See *host port interface*; see also *HPI module*.

I

idct_8x8: Inverse discrete cosine transform (IDCT).

index: A relative offset in the program address that specifies which of the 512 frames in the cache into which the current access is mapped.

indirect addressing: An addressing mode in which an address points to another pointer rather than to the actual data; this mode is prohibited in RISC architecture.

instruction fetch packet: A group of up to eight instructions held in memory for execution by the CPU.

internal interrupt: A hardware interrupt caused by an on–chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, the user must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt service table (IST): A table containing a corresponding entry for each of the 16 physical interrupts. Each entry is a single-fetch packet and has a label associated with it.

Internal peripherals: Devices connected to and controlled by a host device. The C6x internal peripherals include the direct memory access (DMA) controller, multichannel buffered serial ports (McBSPs), host port interface (HPI), external memory-interface (EMIF), and runtime support timers.

IST: *See interrupt service table.*

L

least significant bit (LSB): The lowest-order bit in a word.

linker: A software tool that combines object files to form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher–numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

M

∝–law companding: See *compress and expand (compand)*.

mad_8x8: 8x8 minimum absolute difference.

mad_16x16: 16x16 minimum absolute difference.

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

median_3x3: 3x3 median filter.

memory map: A graphical representation of a computer system's memory, showing the locations of program space, data space, reserved space, and other memory–resident elements.

memory-mapped register: An on-chip register mapped to an address in memory. Some memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

most significant bit (MSB): The highest order bit in a word.

multichannel buffered serial port (McBSP): An on-chip full-duplex circuit that provides direct serial communication through several channels to external serial devices.

multiplexer: A device for selecting one of several available signals.

N

nonmaskable interrupt (NMI): An interrupt that can be neither masked nor disabled.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A state of being external to a device.

on chip: A state of being internal to a device.

P

perimeter: Perimeter structural operator.

peripheral: A device connected to and usually controlled by a host device.

pix_expand: Pixel expand.

pix_sat: Pixel saturate.

program cache: A fast memory cache for storing program instructions allowing for quick execution.

program memory: Memory accessed through the C6x program fetch interface.

PWR: *Power; see PWR module*.

PWR module: PWR is an API module that is used to configure the power–down control registers, if applicable, and to invoke various power–down modes.

Q

quantize: Matrix quantization with rounding.

R

random–access memory (RAM): A type of memory device in which the individual locations can be accessed in any order.

register: A small area of high speed memory located within a processor or electronic device that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction-set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RTOS: *Real–time operating system.*

S

scale_horz: Horizontal scaling.

scale_vert: Vertical scaling.

service layer: The top layer of the 2-layer chip support library architecture providing high-level APIs into the CSL and BSL. The service layer is where the actual APIs are defined and is the layer the user interfaces to.

sobel: Sobel edge detection.

synchronous–burst static random-access memory (SBSRAM): RAM whose contents does not have to be refreshed periodically. Transfer of data is at a fixed rate relative to the clock speed of the device, but the speed is increased.

synchronous dynamic random–access memory (SDRAM): RAM whose contents is refreshed periodically so the data is not lost. Transfer of data is at a fixed rate relative to the clock speed of the device.

syntax: The grammatical and structural rules of a language. All higher–level programming languages possess a formal syntax.

system software: The blanketing term used to denote collectively the chip support libraries and board support libraries.

T

tag: The 18 most significant bits of the program address. This value corresponds to the physical address of the fetch packet that is in that frame.

threshold: Image thresholding.

timer: A programmable peripheral used to generate pulses or to time events.

TIMER module: TIMER is an API module used for configuring the timer registers.

W

wave_horz: Horizontal wavelet transform.

wave_vert: Vertical wavelet transform.

word: A multiple of eight bits that is operated upon as a unit. For the C6x, a word is 32 bits in length.

Index

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W