TMS320C62x Image/Video Processing Library Programmer's Reference

Literature Number SPRU400
March 2000

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

Welcome to the TMS320C62x image/video Library, or IMGLIB for short. The IMGLIB is a collection of 22 high-level optimized DSP functions for the TMS320C62x 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 ’62x 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 catagories
- IMGLIB Reference – a detailed description of each IMGLIB function
- Information about performance, warranty, and support

How to Use This Manual

The information in this document describes the contents of the TMS320C62x IMGLIB in several different ways.

- Chapter 1 provides a brief introduction to the TI ’62x IMGLIB, shows the organization of the routines contained in the library, and lists the features and benefits of the IMGLIB.

- Chapter 2 provides information on how to install, use, and rebuild the TI ’C62x IMGLIB.

- Chapter 3 provides a brief description of each IMGLIB function.

- Chapter 4 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 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 describes performance considerations related to the ’62x IMGLIB and provides information about warranty issues, 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 TMS320C62x is also referred to in this reference guide as the ’C62x.

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.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x digital signal processors, development tools, and third-party support.

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.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/6701 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 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

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

TMS320C62x DSP Library (literature number SPRU402) describes the 32 high-level, C-callable, optimized DSP functions for general signal processing, math, and vector operations.

Contents

Tables

Chapter 1

Introduction

This chapter introduces the TMS320C62x Image/Video Library (IMGLIB) and describes its features and benefits.

Topic **Page**

1.1 Introduction to the TI '62x IMGLIB

The TI 'C62x IMGLIB is an optimized Image/Video Processing Functions Library for C programmers using TMS320C62x 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 'C62x 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-callable routines fully compatible with the TI 'C6x compiler
- ❏ Benchmarks (cycles and code size)
- ❏ Tested against reference C model

NOTE: Although the code provided in this software release has been optimized for 'C62x DSP devices, it will also be operational on other members of the TI 'C6000 DSP family as new devices are made available.

1.3 Software Routines

The rich set of software routines included in the IMGLIB are organized into three different functional catagories 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 and rebuild IMGLIB.

Topic **Page**

2.1 How to Install IMGLIB

Note:

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

The archive has the following structure:

```
img62x.zip
      |
     +-- README.txt      Top-level README file
      |
     +-- lib
      |    |
      |   +-- img62x.lib    Library archive
      |   +-- img62x.src    Full source archive (asm and
headers)
      |
     +-- include
      |    |
      |   +-- header files     Unpacked header files
     +-- doc
          |
         +-- img62xlib.pdf  pdf document of API
```

First Step: De-Archive 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 img62x.lib
- ❑ Use a correct linker command file for the platform you use. Remember most functions in img62x.lib are written assuming little-endian mode of operation.

For example, if you want to call the fdct_8x8 IMGLIB function you would add

```
#include <fdct_8x8.h>
```

in your C file and compile and link using

```
cl6x main.c –z –o fdct_8x8_drv.out –lrts6201.lib
–limg62x.lib
```

Code Composer Studio Users

Assuming your C_DIR environment is correctly set-up (as mentioned in Section 2.1*, How to Install IMGLIB*), you would have to add IMGLIB in the Code Composer Studio environment by choosing img62x.lib from the menu *Project –> Add Files to Project*. Also please make sure you link with the correct runtime support library and IMGLIB by having the following lines in your linker command file:

```
–lrts6201.lib
–limg62x.lib
```

2.2.2 Calling an IMGLIB Function from Assembly

The ’C62x 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 ’C62x 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 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 deals floating point 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 additionally be responsible to restrict the input range in such a way that the outputs do not overflow.

2.2.5 Code Composer Studio Users

If you set up a project Under Code Composer Studio, you could add IMGLIB by choosing img62x.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:

```
–lrts6201.lib
–limg62x.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 How to Rebuild 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 img62x.src –l img62x.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 ’C62x 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 ’C62x 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

- ❑ fdct_8x8
- ❑ idct_8x8

Forward and Inverse DCT (Discrete Cosine Transform) functions, `fdct_8x8` and `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, 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 provides rounding of 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 performance of different DCT based implementations.

- mad_8x8
- mad_16x16

  Functions *8x8 Minimum Absolute Difference,* `mad_8x8`, and *16x16 Minimum Absolute Difference,* `mad_16x16,` 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 and the high performance enabled by the functions provided can enable significant improvements in such systems.

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

- wave_horz
- 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 `wave_horz` and `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 be able to accommodate a wide range of specific wavelets and image dimensions.

3.3 Image Analysis

- boundary

  Boundary and *Perimeter* computation functions, `boundary` and `perimeter`, are provided. These are commonly used structural operators in Machine Vision applications.

- dilate_bin

- erode_bin

  Morphological operators for performing *Dilation* and *Erosion* operations on binary images are provided, “dilate_bin” and “erode_bin” respectively. 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.

- histogram

  The routine *histogram* 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 gray scale 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.

- perimeter

  Boundary and *Perimeter* computation functions, `boundary` and `perimeter`, are provided. These are commonly used structural operators in Machine Vision applications.

- 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 `sobel` provides an optimized implementation of this edge detection algorithm.

- threshold

  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 gray-scale image data to binary image data for input to binary morphological processing. Another form of thresholding may be used to clamp 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. This latter form of thresholding is addressed in the routine `threshold`.

3.4 Picture Filtering/Format Conversions

- corr_3x3
- corr_gen

  Correlation functions are provided to enable image matching. Image matching is useful in applications such as Machine Vision, Medical Imaging, Security/Defense. Two versions of correlation functions are provided: `corr_3x3` implements highly optimized correlation for commonly used 3x3 pixel neighborhoods, and a more general version, `corr_gen`, can implement correlation for user specified pixel neighborhood dimensions within documented constraints.

- 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 “errdif_bin”.

- 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 `median_3x3`.

- pix_expand
- pix_sat

  The routines `pix_expand` and `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.

- scale_horz
- scale_vert

Horizontal and Vertical Scaling functions, `scale_horz` and `scale_vert`, are provided. These functions implement Polyphase FIR Filtering for horizontal and vertical re-sizing of images. Within documented constraints, the functions are flexible enough to be able to accommodate a wide range of image dimensions, scale factors, and numbers of filter taps. These functions may be used in concert to implement 2-D image resizing, or individually for 1-D image resizing, depending on the application. Also provided are support functions for Pixel Expansion and Saturation (see explanations below) that may be used with the scaling functions.

Scaling functions are universally used in image/video processing applications; that is, wherever there is a need to convert one image size to another. Applications include systems for displays, printing, photography, security, digital TV, video telephony, defense, streaming media, etc.

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 fdct_8x8(short fdct_data[ ], unsigned num_fdcts)	Forward Discrete Cosine Transform (FDCT)	5-2
void idct_8x8(short idct_data[ ], unsigned num_idcts)	Inverse Discrete Cosine Transform (IDCT)	5-4
void mad_8x8(void *ref_data, void * src_data, int pitch, void *motvec)	8x8 Minimum Absolute Difference	5-7
void mad_16x16(void *ref_data, void * src_data, int pitch, void *motvec)	16x16 Minimum Absolute Difference	5-9
void quantize (short *data, int num_blks, int blk_sz, const short *recip_tbl, int q_pt)	Matrix Quantization with Rounding	5-12
void wave_horz (short *in_data, short *qmf, short *mqmf, short *out_data, int cols)	Horizontal Wavelet Transform	5-14
void wave_vert (short *in_data[ ], short *qmf,short *mqmf,short *out_ldata,short *out_hdata,int cols,int M)	Vertical Wavelet Transform	5-18

Table 4–2. Image Analysis

Function	Description	Page
void boundary(unsigned char *in_data, int rows, int cols, int *XY, int *out_data)	Boundary Structural Operator	5-22
void dilate_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Dilation	5-24
void erode_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)	3x3 Binary Erosion	5-25
void histogram (unsigned char *in_data, int n, int accumulate, unsigned short *t_hist, unsigned short *hist)	Histogram Computation	5-27
void perimeter (unsigned char *in_data, int cols, unsigned char *out_data)	Perimeter Structural Operator	5-30
void sobel(const unsigned char *in_data, unsigned char *out_data, short cols, short rows)	Sobel Edge Detection	5-32
void threshold(const unsigned char *in_data, unsigned char *out_data, short cols, short rows, unsigned char threshold)	Image Thresholding	5-33

Table 4–3. Picture Filtering/Format Conversions

Function	Description	Page
void corr_3x3(const unsigned char *in_data, unsigned char *out_data, int rows, int cols, unsigned char *mask, short roundval)	3x3 Correlation with Rounding	5-36
void corr_gen(short *in_data, short *h, short *out_data, int m, int cols)	Generalized Correlation	5-39
void errdif_bin(unsigned char errdif_data[ ], int cols, int rows, short err_buf[ ], unsigned char thresh)	Error Diffusion, Binary Output	5-41
void median_3x3(unsigned char *in_data, int cols, unsigned char *out_data)	3x3 Median Filter	5-45
void pix_expand(int n, unsigned char *in_data, short *out_data)	Pixel Expand	5-46
void pix_sat(int n, short *in_data, unsigned char *out_data)	Pixel Saturation	5-47
void scale_horz(unsigned short *in_data, unsigned int n_x, short *out_data, unsigned int n_y, short *hh, unsigned int l_hh, unsigned int n_hh, short * patch)	Horizontal Scaling	5-49
void scale_vert(short *in_data, short *out_data, int cols, short *ptr_hh, short *mod_hh, int l_hh, int start_line)	Vertical Scaling	5-51

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

5.1.1 fdct_8x8 *Forward Discrete Cosine Transform (FDCT)*

void fdct_8x8(short fdct_data[ ], unsigned num_fdcts)

Arguments

fdct_data	pointer to 'num_fdct' 8x8 blocks of image data.
num_fdcts	number of FDCTs to perform. Note that 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

The routine fdct_8x8() 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

- Input terms are expected to be signed 11Q0 values, producing signed 15Q0 results. Larger inputs may result in overflow.
- The 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.
- Input values are stored in shorts, and may be in the range [–512,511]. Larger input values may result in overflow.
- Stack is aligned to a word boundary.

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 inter-scheduled 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. Also, code outside the loops has been scheduled to pack as tightly into fetch packets as possible to avoid alignment padding NOPs.
- 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 ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	160 * num_fdcts + 48 For num_fdtcs = 6, cycles = 1008 For num_fdcts = 24, cycles = 3888
Code size	1216 bytes

5.1.2 idct_8x8 *Inverse Discrete Cosine Transform (IDCT)*

void idct_8x8(short idct_data[ ], unsigned num_idcts)

Arguments

idct_data	pointer to 'num_idcts' 8x8 blocks of DCT coefficients, plus an additional 8x8 block for scratch
num_idcts	number of IDCTs to perform. Note that idct_8x8 requires num_idcts + 1 blocks of storage starting at the location pointed to by 'idct_data'

Description

The routine idct_8x8() 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 cosine terms.

The function idct_8x8() 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

- 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 signed 12-bit range and outputs are saturated to a signed 9-bit range.
- The code is setup to provide an early exit if it is called with num_idcts = 0. In such case it will run for 35 cycles.
- The 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.
- The idct_data[ ] array should be aligned to a 32-bit (word) boundary.
- The routine requires one 8x8-block's worth of extra storage at the end of the list of DCT blocks. The caller must provide room for 'num_idcts + 1' blocks of data in the idct_data[ ] array. The original contents of the extra block are ignored and overwritten with intermediate results by idct_8x8().
- The optimized assembly code requires '(168 * num_idcts) + 62' cycles to process 'num_idcts' blocks. When 'num_idcts' is zero, the function takes an early exit and runs for only 35 cycles (again, including overhead).

Implementation Notes

- The idct_8x8() function returns its results in-place, although it generates intermediate results out-of-place. As a result, when processing N blocks, it requires N+1 blocks of storage, with the

extra block occurring immediately after the valid input data. The initial value of this extra block is ignored, as its value is overwritten with the intermediate results of the IDCT.

- For performance, portions of the optimized code outside the loops have been inter-scheduled 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 save the need for messy pointer and constant re-initialization.

- To save code size, prolog and epilog collapsing have been performed to the extent that it does not impact performance. Also, code outside the loops has been scheduled to pack as tightly into fetch packets as possible to avoid alignment padding NOPs.

- The IDCTs cannot be performed completely in-place due to the transpose that each pass performs. In order to save data memory, the horizontal pass works from the end of the array towards the beginning, writing its result one IDCT block later in memory, thus performing the IDCT nearly-in-place. The vertical pass performs its IDCTs in the opposite direction, working from the start of the array towards the end, writing the results in-place. A nice side effect of this is that the pointer values at the end of the horizontal loop are a fixed offset relative to their required values for the vertical loop, regardless of the number of IDCTs performed. This makes the pointer re-initialization exceptionally cheap.

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

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

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

Benchmarks

Cycles	(168 * num_idcts) + 62 For num_idcts = 6, cycles = 1070 For num_idcts = 24, cycles = 4094
Code size	1344 bytes

5.1.3 mad_8x8 — *8x8 Minimum Absolute Difference*

void mad_8x8(void *ref_data, void * src_data, int pitch, void *motvec)

Arguments

ref_data	pointer to array of image pixels that constitute a reference image, with H columns and V rows
src_data	pointer to 8x8 source image pixels
pitch	linearizes 2-D array to 1-D array based on the relation: 1d[(i*pitch)+j] = 2d[i][j]
motvec	motion vector (output) first element motvec[0]: packed coordinates h,v second element: madval

Description The routine mad_8x8() returns the location of the minimum absolute difference (MAD) between an 8x8 search block and some point in a search window of size HxV. The location is returned as ‘h’ and ‘v’ value packed as two 16-bit quantities in a 32-bit int. The corresponding MAD value is also returned.

Algorithm Behavioral C code for the routine mad_8x8() is provided below:

```
void mad_8x8(unsigned char *ref_data, unsigned char *src_data, unsigned int pitch,
unsigned int *motvec)
{
      unsigned int              i, j, k, i2;
      unsigned int              bptch, vptch;
      unsigned int              matpos, pitch_8;
      unsigned int              matval;
      unsigned int              accm0;
      short                     s0;
      short                     img_i0, srcIm;
      int                       mh, mv;
      int                       matx, maty;
      bptch = (8–1) * pitch;
      vptch = (V*pitch) – 1;
      matval = 0xffffffff;
      pitch_8 = pitch – 8;
      k = 0;
```

```
        for(mh = 0; mh < H; mh++)
        {
                for (mv = 0; mv < V; mv++)
                {
                    accm0 = 0;
                    j = 0;
                    for (i = 0; i < 8; i++)
                        {
                                for (i2 = 0; i2 < 8; i2++)
                                {
                                        srcIm = src_data[j];
                                        img_i0 = ref_data[k];
                                        s0 = (img_i0 – srcIm);
                                        accm0 += abs(s0);
                                        j++;
                                        k++;
                                }

                                k+= pitch_8;
                        }

                        k –=  bptch;
                        if (accm0 < matval)
                        {
                                matval = accm0;
                                matx = mh;
                                maty = mv;
                        }
                }

                k –= vptch;
        }
/*************************************************************************/
/* Return packed x,y coordinates corresponding to minimum MAD.           */
/* Also return the minimum MAD as matval.                                */
/*************************************************************************/
        matpos = (0xffff0000 & (matx << 16)) | (0x0000ffff & maty);
        motvec[0] = matpos;
        motvec[1] = matval;
}
```

Special Requirements

- ❏ It is assumed that src_data and ref_data do not alias in memory.
- ❏ No special alignment of byte-level src_data or ref_data is expected.

Implementation Notes

- The inner loop is unrolled four times, the outer loop is unrolled once. Delay slot stuffing and outer loop branch overhead are minimized.
- Motion vectors are contained in the array 'motvec' in packed coordinates h:v respectively representing horizontal and vertical vector components. 'motvec' array also contains the MAD value.

  motvec[0]: hl:vl

  motvec[1]: madval
- ***Bank Conflicts:*** At most one bank conflict can occur over the entire kernel.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	62 * H * V + 21 For H = 4, V = 4, cycles = 1013 For H = 64, V = 32, cycles = 126,997
Code size	768 bytes

5.1.4 mad_16x16 — *16x16 Minimum Absolute Difference*

void mad_16x16(void *ref_data, void * src_data, int pitch,
void *motvec)

Arguments

ref_data	pointer to array of image pixels that constitutes a reference image, with H columns and V rows
src_data	pointer to 16x16 source image pixels
pitch	linearizes 2-D array to 1-D array based on the relation: 1d[(i*pitch)+j] = 2d[i][j]
motvec	motion vector (output) first element motvec[0]: packed coordinates h,v second element: madval

Description The routine mad_16x16 returns the location of the minimum absolute difference (MAD) between an 16x16 search block and some point in a search window of size HxV. The location is returned as 'h' and 'v' value packed as two 16-bit quantities in a 32-bit int. The corresponding MAD value is also returned.

Algorithm Behavioral C code for the routine mad_16x16() is provided below:

```
void mad_16x16(unsigned char *ref_data, unsigned char *src_data,
unsigned int pitch,    unsigned int *motvec)
{
       unsigned  int       i, j, k, i2;
       unsigned int        bptch, vptch;
       unsigned int        matpos, pitch_16;
       unsigned int        matval;
       unsigned int        accm0;
       short               s0;
       short               img_i0, srcIm;
       int                 mh, mv;
       int                 matx, maty;
       bptch = (16–1) * pitch;
       vptch = (V*pitch) – 1;
       matval = 0xffffffff;
       pitch_16 = pitch – 16;
       k = 0;
       for(mh = 0; mh < H; mh++)
       {
              for (mv = 0; mv < V; mv++)
              {
                     accm0 = 0;
                     j     = 0;
                     for (i = 0; i < 16; i++)
                     {
                            for (i2 = 0; i2 < 16; i2++)
                            {
                                   srcIm = src_data[j];
                                   img_i0 = ref_data[k];

                                   s0 = (img_i0 – srcIm);
                                   accm0 += abs(s0);
                                   j++;
                                   k++;
                            }

                            k += pitch_16;
                     }
                     k –=  bptch;
                     if (accm0 < matval)
```

```
                {
                    matval = accm0;
                    matx = mh;
                    maty = mv;
                }
            }
            k –= vptch;
        }
        /*************************************************************/
        /* Return packed x,y coordinates corresponding to            */
        /* minimum MAD. Also return the minimum MAD as               */
        /* matval.                                                   */
        /*************************************************************/
        matpos = (0xffff0000 & (matx << 16)) | (0x0000ffff & maty);
        motvec[0] = matpos;
        motvec[1] = matval;
}
```

Special Requirements No special alignment of src_data or ref_data is expected.

Implementation Notes

- The inner loop is unrolled four times, and the outer loop is unrolled once in the optimized assembly code. Delay slot stuffing and outer loop branch overhead are minimized.
- Motion vectors are contained in the array ‘motvec’ in packed coordinates h:v respectively representing horizontal and vertical vector components. ‘motvec’ array also contains the MAD value.

  motvec[0]: hl:vl

  motvec[1]: madval
- ***Bank Conflicts:*** At most one bank conflict can occur over the entire function.
- ***Endian****:* The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	231*H*V + 21 For H = 4, V = 4, cycles = 3717 For H = 64, V = 32, cycle = 473,109
Code size	768 bytes

5.1.5 quantize — *Matrix Quantization with Rounding*

void quantize (short *data, int num_blks, int blk_size,
const short *recip_tbl, int q_pt)

Arguments

data	pointer to data to be quantized
num_blks	number of 64-element blocks processed
blk_size	block size, multiple of 16
recip_tbl	pointer to Quantization Values (reciprocals)
q_pt	Q-point of Quantization values

Description

The routine quantize() 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 quantize() may be used on any signed 16-bit data using signed 16-bit quantization terms.

quantize() merely multiplies the contents of the quantization array with the data being quantized. Therefore, it may be used for inverse quantization as well, by setting the Q-point appropriately.

Algorithm

Behavioral C code for the routine quantize() is provided below:

```
void quantize (short *data, int num_blks, int blk_size, const short *recip_tbl,
int q_pt)
{
      short                  recip;
      int                    i, j, k, quot, round;

/*****************************************************************/
/* Set rounding term as 0.5, effectively                         */
/*****************************************************************/
      round = q_pt ? 1 << (q_pt – 1) : 0;
/*****************************************************************/
/* Outer loop: Step between quant matrix elements.               */
/*****************************************************************/


      for (i = 0; i < blk_size; i++)
      {
```

```
/*********************************************************************/
/*Load a reciprocal and point to appropriate element of block        */
/*********************************************************************/
            recip = recip_tbl[i];
            k = i;

/*********************************************************************/
/*Inner loop: Step between blocks of elements, quantizing            */
/*********************************************************************/

            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, must be at least 1. The block size (number of elements in each block) must be at least 16, and a multiple of 16. The Q-point, q_pt, controls rounding and final truncation; it must be in the range from 0 <= q_pt <= 31.
- ❑ Both input arrays, data[ ] and recip_tbl[ ], must be word aligned.
- ❑ The data[ ] array must be 'num_blks * blk_size' elements, and the recip_tbl[ ] array must be 'blk_size' elements.
- ❑ The block size, blk_size, must be a multiple of 16.
- ❑ The number of blocks, num_blks, must be at least 1.

Implementation Notes

- ❑ The outer loop is unrolled 16 times to allow greater amounts of work to be performed in the inner loop.
- ❑ Reciprocals and data terms are loaded in pairs with word-wide loads, making better use of the available memory bandwidth.
- ❑ The outer loop has been interleaved with the prolog and epilog of the inner loop.
- ❑ Epilog code from the inner loop has been moved into the exit-code delay slots through creative use of branch delay slots.

- ❏ Twin stack pointers have been used to speed up stack accesses.
- ❏ The inner loop steps through individual blocks, while the outer loop steps through reciprocals for quantization. This eliminates redundant loads for the quantization terms.
- ❏ The direction of travel for the inner loop oscillates with each iteration of the outer loop to simplify pointer updating in the outer loop and reduce register pressure. (e.g. in the first iteration of the outer loop, the inner loop steps forward through memory; in the second iteration of the outer loop, the inner loop steps backwards through memory, etc.)
- ❏ A total of 14 words of stack frame are used for saving the Save-On-Entry registers.
- ❏ ***Bank Conflicts:*** No bank conflicts occur, regardless of the relative orientation of recip_tbl[ ] and data[ ].
- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(blk_size/16) * (4 + num_blks * 12) + 26 For blk_size = 64, num_blks = 8, cycles = 426 For blk_size = 256, num_blks = 24, cycles = 4696
Code size	1024 bytes

5.1.6 wave_horz — *Horizontal Wavelet Transform*

void wave_horz (short *in_data, short *qmf, short *mqmf, short *out_data, int cols)

Arguments

in_data	pointer to one row of input pixels
qmf	pointer to qmf filter-bank for low-pass filtering
mqmf	pointer to mirror qmf filter bank for high-pass filtering
out_data	pointer to row of detailed/reference decimated outputs
cols	number of columns in the input image

Description

The routine wave_horz() performs a 1-D Periodic Orthogonal Wavelet decomposition. It also performs the row decomposition component of a 2D 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 high-pass 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:

```
#define Qpt 15
#define Qr 16384
void wave_horz( short *in_data, short *qmf, short *mqmf, short *out_data,
int cols)
{
      int               i;
      short             *xptr = in_data;
      short             *x_end = &in_data[cols – 1];
      int               j, sum, prod;
      short             xdata, hdata;
      short             *filt_ptr;
      int               M = 8;
      /*********************************************************************/
      /* iters: number of iterations = half the width of the input line   */
      /* xstart: starting point for the high pass filter input data       */
      /*********************************************************************/
      int  iters = cols;
      short *xstart = in_data + (cols – M) + 2;

      /*********************************************************************/
      /* Since the output of the low pass filter is decimated by          */
      /* eliminating odd output samples, the loop counter i               */
      /* increments by 2 for every iteration of the loop. Let the         */
      /* input data be {d0, . . . , d–7} and the low pass filter be {h0, . . . , h7} */
      /* Outputs y0, y1, . . . , . . . are generated as:                  */
      /* y0 = h0d0 + h1d1 + h2d2 + h3d3 + h4d4 + h5d5 + h6d6 + h7d7       */
      /* y1 = h0d2 + h1d3 + h2d4 + h3d5 + h4d6 + h5d7 + h6d8 + h7d9       */
      /* If the input array access d goes past the end of the array       */
      /* the pointer is wrapped around. Since the filter is in floating   */
      /* point it is implemented in Q15 math. Qr is the associated        */
      /* round value.                                                     */
      /*********************************************************************/
```

```
      for ( i = 0; i < iters; i+= 2)
      {
            sum = Qr;
            xptr = in_data + i;
            for (j = 0; j < M; j++)
            {
               xdata = *xptr++;
               hdata = qmf[j];
               prod = xdata * hdata;
               sum += prod;
               if (xptr > x_end) xptr = in_data;
            }
            *out_data++ = (sum >> Qpt);
      }

/*************************************************************************/
/* Since the output of the high pass filter is decimated by              */
/* eliminating odd output samples, the loop counter I                    */
/* increments by 2 for every iteration of the loop. Let the              */
/* input data be {d0, d1, . . . , dN-1} where N = cols and M = 8         */
/* Let the high pass filter be {g0, . . . , g7}.                         */
/* Outputs y0, y1, . . . are generated as:                               */
/*  y0 = g7dN-M+2 + g6dN-M+1 + . . . + g0d1                              */
/*  y1 = g7dN-M+2 + g6dN-M+1 + . . . + g0d1                              */
/*                                                                       */
/* If the input array access d goes past the end of the array            */
/* the pointer is wrapped around. Since the filter is in floating        */
/* point it is implemented in Q15 math. Filt_ptr points to the           */
/* end of the high-pass filter array and moves in reverse                */
/* direction.                                                            */
/*************************************************************************/
      for (i = 0; i < iters ; i+=2)
      {
            sum = Qr;
            filt_ptr = mqmf + (M - 1);
            xptr = xstart;
            xstart += 2;
            if (xstart > x_end) xstart = in_data;
            for (j = 0; j < M; j++)
            {
               xdata = *xptr++;
               hdata = *filt_ptr--;
               prod = xdata * hdata;
```

```
            if (xptr > x_end) xptr = in_data;
            sum += prod;
         }

         *out_data++ = (sum >> Qpt);
      }
}
```

Special Requirements

- ❑ This function assumes that the number of taps for the qmf and mqmf filters is 8, and that the filter coefficients are word aligned.
- ❑ Input data is assumed to be word aligned so that word-wide loads may be performed.
- ❑ This function assumes that filter coefficients are maintained as shorts (16-bits).
- ❑ 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 fedback as input to an identical next stage.
- ❑ The transform is a dyadic wavelet, requiring the number of image columns to be a power of 2.

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 wrap-around can occur and hence this creates a dependency. Pre-reading 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 filterbank 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 because data reuse reduces the number of loads and stores in the loop to 3.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	(4 * cols) + 5 For cols = 256, cycles = 1029 For cols = 512, cycles = 2058
Code size	640 bytes

5.1.7 wave_vert *Vertical Wavelet Transform*

void wave_vert (short *in_data[ ], short *qmf, short *mqmf, short *out_ldata, short *out_hdata, int cols, int M)

Arguments

in_data	pointer to an array of pointer for input data line buffers
qmf	pointer to qmf filter-bank for low-pass filtering
mqmf	pointer to mirror qmf filter bank for high-pass filtering
out_ldata	pointer to lowpass filtered outputs
out_hdata	pointer to highpass filtered outputs
cols	width of each line in the input buffer
M	number of filter taps (always 8 in this implementation)

Description

The routine wave_vert() performs the vertical pass of 2-D wavelet transform. It performs a vertical filter on 8 rows which are pointed to by the pointers contained in an array of pointers. It produces two lines worth of output, one being the low-pass and the other being the high pass filtered result. The vertical filter is traversed over the entire width

of the line and 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. This requires the caller to implement a C code close to what is presented here.

Algorithm Behavioral C code for the routine wave_vert() is provided below:

```
#define Qpt 15
#define Qr 16384
#define Qs 32767
void wave_vert(short *in_data[ ], short *qmf, short *mqmf, short *out_ldata,
short *out_hdata, int cols, int M)
{
   int               i,  j;
   int               sum_h, sum_l;
   int               prod_h, prod_l;
   short             xdata, hdata, ldata;
   short             *filt_ptr;
   /*********************************************************************/
   /* Low-Pass filter: in_data contains 8 pointers which point to       */
   /* input lines. The filters are placed vertically and input data     */
   /* is read from 8 separate lines. Qr is round value for Q15          */
   /* math. M is assumed to be 8 and is the number of filter            */
   /* taps for D4. Low-Pass filter output is in out_ldata.              */
   /*********************************************************************/
   for ( i = 0; i < cols; i++)
   {
            sum_l = Qr;
            filt_ptr = qmf;
            for ( j = 0; j < M; j++)
            {
               xdata = in_data[j][i];
               ldata = *filt_ptr++;
               prod_l = xdata * ldata;
               sum_l += prod_l;
            }
            *out_ldata++ = (sum_l >> Qpt);
   }
   /*********************************************************************/
   /* High-Pass filter: in_data contains 8 pointers which point to      */
   /* input lines. The filters are placed vertically and input data     */
   /* is read from 8 separate lines. Qr is round value for Q15          */
   /* math. M is number of filter taps and is assumed to be 8.          */
   /* High-Pass filter output is in out_hdata.                          */
   /*********************************************************************/
```

```
   for ( i = 0; i < cols; i++)
   {
            sum_h    = Qr;
            filt_ptr = mqmf + M – 1;
            for ( j = 0; j < M; j++)
            {
               xdata = in_data[j][i];
               hdata = *filt_ptr––;
               prod_h = xdata * hdata;
               sum_h += prod_h;
            }
            *out_hdata++ = (sum_h >> Qpt);
   }
}
```

Special Requirements

- ❏ Since the wavelet transform is dyadic cols should be a multiple of 2. No checking is done in the code to ensure this.
- ❏ The input filters qmf and mqmf are assumed to be word-aligned and have 8 taps.
- ❏ The input data on any line is assumed to be word-aligned.
- ❏ The code can be used to obtain maximum performance by using a working buffer of ten input lines to effectively mix processing and data transfer through DMA's. On every even iteration, the first eight lines are used as input for processing while the last two lines are filled up using DMA. On every odd iteration, {line2, ..line10} are used for processing while the next two input lines are brought in on line0 and line 1. The input data on the next iteration starts reading lines from {line4, 0.line10, line0, line1}. This pattern then repeats.
- ❏ The output pointers need to be set as follows on the following calls to the code:

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*
4	*out_data*	*out_hstart + 3 * cols*

where

out_lstart = *out_data* + ((*rows* > > 1) + 3) * *cols*,
out_hstart = *out_data* + (*rows* > > 1) * *cols*,

and *out_data* is the start of the output line.

Notice that on the fourth call to the function o*ut_ldata* wraps to point to the beginning of the output line.

Implementation Notes

- The inner loop that advances along each filter tap is unrolled. Word-wide data loads are performed and split multiplies are used to perform two iterations of low-pass filtering in parallel. By loading the filter coefficients in a special fashion, the low-pass filter kernel is re-used for performing the high-pass filter, thereby saving code size.
- In order to eliminate bank conflicts, successive lines in the line buffer are separated by exactly one word so that loads to any successive lines may be parallelized together.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	8 * cols + 48 For cols = 256, cycles = 2096 For cols = 512, cycles = 4144
Code size	736 bytes

5.2 Image Analysis

5.2.1 boundary — *Boundary Structural Operator*

void boundary(unsigned char *in_data, int rows, int cols, int *XY, int *out_data)

Arguments

in_data	pointer to input image data
rows	number of input rows
cols	number of input columns
XY	pointer to output array of packed XY coordinates
out_data	pointer to output gray-value data

Description

The routine boundary() assumes that an object in the image occupies non-zero gray level values and the background has zero value. It returns the coordinates of the non-zero pixels in the image and the associated gray level values. The XY coordinates are returned as packed values. The gray level values are returned in an int array to avoid bank conflicts.

Algorithm

Behavioral C code for the routine boundary() is provided below:

```
void boundary (unsigned char *in_data, int rows, int
cols, int *XY, int *out_data)
{
/*************************************************************************/
/* rows_n and cols_n maintain current row and column index. The          */
/* loop iterates rows*cols times and checks if the pixel at any given    */
/* location is non-zero. If it is non-zero the associated gray level is  */
/* stored and the coordinates are stored as a packed quantity with       */
/* the row in the upper 16-bits and the column in the lower 16 bits.     */
/*************************************************************************/
         int i, xword, pixel;
         int rows_n = 0;
         int cols_n = 0;
         int ncols = cols;
```

```
         for(i = 0; i < (rows*cols); i++)
         {
/*************************************************************************/
/* Prepare the packed coordinate row|col to store if the pixel           */
/* that is loaded is non-zero. If it is non-zero store out the           */
/* packed coordinates and the gray level.                                */
/*************************************************************************/

            xword = (rows_n << 16) + cols_n;
            pixel = *in_data++;
            if (pixel) *XY++ = xword;
            if (pixel) *out_data++ = pixel;
            cols_n++;
            ncols--;
            rows_n +=!ncols;
            if (!ncols) cols_n  = 0;
            if (!ncols) ncols = cols;
         }
}
```

Special Requirements

- ‘cols’ is assumed to be a multiple of 4.
- ‘XY’ and ‘out_data’ arrays are separated by two banks and are word aligned.

Implementation Notes

- In the optimized assembly code, two loops are collapsed into one single loop and use is made of an independent down-counting counter to count to zero. This avoids a costly compare against a constant. The loop is unrolled four times. A check for zero is performed after every fourth iteration of the original loop as it is assumed that ‘cols’ is a multiple of four.
- ***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	1.25 * (cols * rows) + 4
	For cols = 128, rows = 3, cycles = 484 For cols = 720, rows = 8, cycles = 7204
Code size	352 bytes

5.2.2 dilate_bin — *3x3 Binary Dilation*

void dilate_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)

Arguments

in_data	pointer to input image array
out_data	pointer to output image array
mask	3x3 binary mask
cols	number of image columns processed in bytes

Description

The routine dilate_bin() implements 3x3 binary dilation. The input image consists of binary valued pixels (0s or 1s) packed 32 to a word, with the function processing 32 pixels per iteration.

Algorithm

The routine dilate_bin() computes output for a target pixel as follows:

A 3x3 mask of binary values is used:

```
mask[3x3]:   m00   m01   m02
             m10   m11   m12
             m20   m21   m22
```

If input image pixels (binary values) are:

```
pix[3x3]:    p00   p01   p02
             p10   p11   p12
             p20   p21   p22
```

The output corresponding to pixel p11 is:

```
o11 = (m00 OR p00) OR (m01 OR p01) OR
      (m02 OR p02) OR
      (m10 OR p10) OR (m11 OR p11) OR
      (m12 OR p12) OR
      (m20 OR p02) OR (m21 OR p21) OR (m22 OR p22)
```

The default mask values are all 0s, which leads to the following output conditions:

```
o11 = 0 if p00 = p01 = p02 = p10 = p11 = p12 = p20
    = p21 = p22 = 0
o11 = 1 for all other cases.
```

The mask can also accommodate “don’t care” values, implemented as –1. To avoid use of predication for “don’t care” cases, all the mask elements are first subjected to XOR with –1, and the resulting mask elements are ANDed with the binary pixel values to achieve the binary dilation operation.

Special Requirements

- ❏ Legal values for mask elements are 0 or ”don’t care” (implemented as value –1). Result of ”don’t care” operation on a pixel is always 0.
- ❏ The input binary image needs to have a multiple of 32 pixels (bits) per row.

Implementation Notes

- ❏ Code size for the optimized assembly code has been reduced by removing the epilog, as well as collapsing the prolog and merging with the setup code.
- ❏ ***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	[(cols/4) * 6] + 34 For cols = 128*8, cycles = 226 For cols = 720*8, cycles = 1114
Code size	480 bytes

5.2.3 erode_bin — *3x3 Binary Erosion*

void erode_bin(unsigned char *in_data, unsigned char *out_data, char *mask, int cols)

Arguments

in_data	pointer to input image array
out_data	pointer to output image array
mask	3x3 binary mask
cols	number of image columns processed in bytes

Description

The routine erode_bin() implements 3x3 binary erosion. The input image consists of binary valued pixels (0s or 1s) packed 32 to a word with the function processing 32 pixels per iteration.

Algorithm

The routine erode_bin() computes output for a target pixel as follows:

A 3x3 mask of binary values is used:

```
mask[3x3]:   m00   m01   m02
             m10   m11   m12
             m20   m21   m22
```

If input image pixels (binary values) are:

```
pix[3x3]:    p00   p01   p02
             p10   p11   p12
             p20   p21   p22
```

The output corresponding to pixel p11 is:

```
o11 = (m00 AND p00) AND (m01 AND p01) AND
      (m02 AND p02) AND
      (m10 AND p10) AND (m11 AND p11) AND
      (m12 AND p12) AND
      (m20 AND p02) AND (m21 AND p21) AND
      (m22 AND p22)
```

The default mask values are all 1s, which leads to the following output conditions:

```
o11 = 1 if p00 = p01 = p02 = p10 = p11 = p12 = p20
    = p21 = p22 = 1
o11 = 0 for all other cases.
```

The mask can also accommodate “don’t care” values, implemented as –1. To avoid use of predication for “don’t care” cases, all the mask elements are first shifted right by 1, and the resulting mask elements are ORed with the binary pixel values to achieve the binary erosion operation.

Special Requirements

- ❏ Legal values for mask elements are 1 or ”don’t care” (implemented as value –1). Result of ”don’t care” operation on a pixel is always 1.
- ❏ The input binary image needs to have a multiple of 32 pixels (bits) per row.

Implementation Notes

- Code size for the optimized assembly code has been reduced by removing the epilog, as well as collapsing the prolog and merging with the setup code.
- ***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	[(cols/4) * 6] + 34 For cols = 128*8, cycles = 226 For cols = 720*8, cycles = 1114
Code size	480 bytes

5.2.4 histogram — *Histogram Computation*

void histogram (unsigned char *in_data, int n, int accumulate, unsigned short *t_hist, unsigned short *hist)

Arguments

in_data	pointer to input image data
n	number of points
accumulate	defines add/sub from existing histogram: has values 1, –1
t_hist	pointer to temporary histogram bins (1024)
hist	pointer to running histogram bins (256)

Description

The routine histogram() computes the histogram of an array of n, 8-bit inputs. It returns the histogram of 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 4 temporary histograms, which are later summed together.

Algorithm

Behavioral C code for the function histogram() is provided below:

```
void 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[data] += accumulate;
          }
}
```

Special Requirements

- ❏ It is assumed that the temporary array of data, t_hist, is initialized to zero.
- ❏ The input array of data, in_data, must be aligned to a 4-byte boundary and 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, and the maximum n is 262143.

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. Both halves operate on the same 4 histogram bins. This introduces a memory dependency which ordinarily would degrade performance. To break the memory dependencies, the two halves forward their results to each other. 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) Goto 1.

- With the ordering used, 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.

- ***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 + 582 For n = 512, cycles = 1158 For n = 1024, cycles = 1734
Code size	960 bytes

5.2.5 perimeter — *Perimeter Structural Operator*

void perimeter (unsigned char *in_data, int cols,
unsigned char *out_data)

Arguments

in_data	pointer to input binary image data
cols	number of input columns
out_data	pointer to output boundary image data

Description

The routine perimeter() 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_up
      pix_lft   pix_cent   pix_rgt
                pix_dn
```

The output pixel at location ‘pix_cent’ is echoed as a boundary pixel if ‘pix_cent’ is non-zero and any one of its four neighbors is zero. The four neighbors are as shown above and stand for the following:

pix_up: top pixel
pix_lft: left pixel
pix_rgt: right pixel
pix_dn: bottom pixel

Algorithm

Behavioral C code for the routine perimeter() is provided below:

```
void 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 every pixel along a given line, examine left, right, top and  */
/* bottom neighbors. Check whether center pixel is non-zero and       */
/* any of the neighbors is zero. The variables are named as:          */
/* pix_lft: left pixel, pix_cent: center pixel, pix_rgt: right pixel, */
/* pix_top: top pixel, pix_bot: bottom pixel                          */
/*********************************************************************/
```

```
        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

- ❑ No specific alignment is expected for the input or output array.
- ❑ 'cols' can be either even or odd.
- ❑ This code expects three input lines each of width 'cols' pixels and produces one output line of width (cols – 1) pixels.

Implementation Notes

- ❑ To decide whether the given pixel at 'pix_cent' is a boundary pixel or not, 5 pixels have to be examined. This leads to a highly conditional code. The conditional code is reduced by performing multiplies to examine whether the four neighboring pixels are zero or not. Conditionally replacing the value of the output pixel based on status flag also helps scheduling. Notice also that the 'pix_cent' variable lives too long in the kernel. This is because its value is not consumed for a long time after it is produced. This can hinder the start of the next iteration. This is avoided by issuing moves and making copies of this variable.
- ❑ ***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	3 * (cols – 2) + 14 For cols = 128, cycles = 392 For cols = 720, cycles = 2168
Code size	358 bytes

5.2.6 sobel *Sobel Edge Detection*

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

Arguments

in_data	pointer to input image data
out_data	pointer to output image data
cols	number of columns in the input image
rows	number of rows in the input image

Description

The routine sobel() applies horizontal and vertical Sobel edge detection masks to the input image to an output image which is two rows shorter than the input image. Within each row of the output, pixels 1 through cols–2 contain filtered outputs. Pixels 0 and cols–1 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

Special Requirements

- ❑ At least eight output pixels must be processed.
- ❑ The image arrays must be half-word aligned.
- ❑ The input image width (value of ‘cols’) must be even.

Implementation Notes

- ❑ The values of the left-most and right-most pixels on each line of the output are not well defined.
- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Endian****:* The code is LITTLE ENDIAN.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	3 * cols * (rows – 2) + 34 For cols = 128, rows = 8, cycles = 2338 For cols = 720, rows = 8, cycles = 12,994
Code size	608 bytes

5.2.7 threshold — *Image Thresholding*

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

Arguments

in_data	pointer to input image data
out_data	pointer to output image data
cols	number of image columns
rows	number of image rows
threshold	threshold value

Description The routine threshold() 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 are exactly the same dimensions.

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

Algorithm Behavioral C code for the routine threshold() is provided below:

```
void threshold (const unsigned char *in_data, unsigned char *out_data, short cols,
short rows, unsigned char threshold)
{
   int i;
/*************************************************************************/
/* Step through input image copying pixels to the output. If the         */
/* pixel value is below threshold, set corresponding output to zero.     */
/*************************************************************************/

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

Special Requirements

- ❏ Both buffers (input and output) must be word aligned.
- ❏ A multiple of 16 pixels must be processed.
- ❏ Stack is aligned to a word boundary.
- ❏ The code requires 12 words of stack space to save Save-On-Entry registers.

Implementation Notes

- ❏ The two loops that were originally in the natural C code have been collapsed into one.
- ❏ For performance, the code outside the loop has been interleaved as much as possible with the prolog and epilog code.
- ❏ Twin stack-pointers are used to accelerate stack accesses.
- ❏ The inner loop is unrolled 16 times and the data is manipulated in packed format for speed.
- ❏ ***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	(cols * rows / 16) * 9 + 50
	For cols = 128, rows = 8, cycles = 626 For cols = 720, rows = 8, cycles = 3290
Code size	576 bytes

5.3 Picture Filtering/Format Conversions

5.3.1 corr_3x3 — *3x3 Correlation with Rounding*

void corr_3x3(const unsigned char *in_data,
unsigned char *out_data, int rows, int cols, unsigned char *mask, short roundval)

Arguments

in_data	pointer to an input array of 8-bit pixels
out_data	pointer to an output array of 8-bit pixels
rows	line number of current output row to be computed
cols	number of columns in the image
mask	pointer to 8-bit mask
roundval	user specified round value

Description

The routine corr_3x3() performs a point by point multiplication of the 3x3 mask with the input image. The result of the nine multiplications are then summed together. The sum is rounded and shifted to produce an 8-bit value which is 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 until the entire row is covered.

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

```
void corr3x3 (unsigned char *in_data, int cols,
int rows, short *mask, unsigned char *out_data)
{
   int i;

   for (i = 0; i < (rows –2); i++)
      {
         corr_3x3 (in_data, out_data, i, cols,
         mask, round);
      }
}
```

Algorithm Behavioral C code for the function corr_3x3() is provided below:

```
void corr_3x3 (const unsigned char *in_data, unsigned char *out_data,
int rows, int cols, unsigned char *mask, short round)
{
      int                   i,j;
      int                   sum00,sum11,sum22;
      int                   sum = 0;
      const unsigned char   *IN1,*IN2,*IN3;
      unsigned char         pix10,pix20,pix30;
      short                 mask10,mask20,mask30;
      unsigned char         *OUT;
      unsigned char         shift = 12;

      IN1 = in_data + (rows*cols);              /* pointer to 1st row */
      IN2 = IN1 + cols;                         /* pointer to 2nd row */
      IN3 = IN2 + cols;                         /* pointer to 3rd row */
      OUT  = out_data + ((rows+1)*cols) + 1;    /* output pointer */

      for (j = 0; j < (cols–2); j++)
      {
            sum  = 0;                 /* initialize correlation to zero */
            for (i = 0; i < 3; i++)
            {
               pix10 = IN1[i];           /* load 1st row pixel value */
               pix20 = IN2[i];           /* load 2nd row pixel value */
               pix30 = IN3[i];           /* load 3rd row pixel value */

               mask10 = mask[i];         /*load 1st row mask value */
               mask20 = mask[i+3];       /* load 2nd row mask value */
               mask30 = mask[i+6];       /* load 3rd row mask value */

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

               sum += sum00+sum11+sum22;
            }

            IN1++;                              /* increment row 1 pointer */
            IN2++;                              /* increment row 2 pointer */
            IN3++;                              /* increment row 3 pointer */
            sum = (sum+round)>>shift;           /* round and shift final sum */
            *OUT++ = sum;                       /* store out the result */
      }
}
```

Special Requirements

- ❑ The array pointed to by out_data should not alias with the array pointed to by in_data.
- ❑ This function only performs 3x3 correlation. A more generalized function for correlation with an arbitrary sized mask is also available as a separate kernel in the benchmark suite.
- ❑ The argument 'row' should be a multiple of two, as two output pixels are processed together in the optimized assembly code.
- ❑ (column – 2) output pixels are produced when three lines, each with a width of 'column' pixels, are given as input.

Implementation Notes

- ❑ Data for the input image pixels is reused by pre-loading it outside the loop and issuing moves to bring it to the appropriate registers once inside the loop. This is done to minimize the loads from nine to six within the loop, for each pair of pixels in the present computation of the correlation.
- ❑ The loop is unrolled once so that eighteen multiplies for the two output pixels can schedule in 9 cycles leading to 4.5 cycles per output pixel.
- ❑ The loop that did the loads three at a time, per row is collapsed to increase parallel operations.
- ❑ ***Bank Conflicts****:* No bank conflicts occur in this function.
- ❑ ***Endian****:* The code is ENDIAN NEUTRAL.
- ❑ ***Interruptibility****:* The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	[(cols – 2) * 4.5) + 21]
	For cols = 256, cycles = 1164 For cols = 720, cycles = 3252
Code size	1120 bytes

5.3.2 corr_gen — *Generalized Correlation*

void corr_gen(short *in_data, short *h, short *out_data, int m, int cols)

Arguments

in_data	pointer to input pixel array
h	pointer to input 1xM mask array
out_data	pointer to output array
m	number of filter taps
cols	number of columns in the image

Description

The routine corr_gen() 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 number of columns in the image (cols) or the number of filter taps (m).

Algorithm

Behavioral C code for the routine corr_gen() is provided below:

```
void corr_gen (short *in_data, short *h, short *out_data, int m, int cols)
{

/*********************************************************************/
/* For all columns compute an M–tap filter. Add                      */
/* correlation sum to value, to allow for a generalized 2–D          */
/* correlation to be built using several 1–D correlations            */
/*********************************************************************/

   inti, j;

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

      for (i = 0; i < m; i++)
      {
         sum += in_data[i + j] * h[i];
      }

      out_data[j] = sum;
   }
}
```

Special Requirements

- Data for input image, filter and output arrays must be word-aligned.
- If the width of the input image is ‘cols’ and the mask is ‘m’ then the output array must have at least a dimension of (cols – m + 4)
- 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 needs to be used and the code should directly start 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 four times, assuming that the loop iteration (cols – m) is a multiple of four. In most typical images ‘cols’ is a multiple of 8 but since ‘m’ is completely general (cols – m) may not be a multiple of 4. If (cols – m) is not a multiple of 4 then the inner loop iterates fewer times than required and certain output pixels may not be computed. This problem is overcome as follows:

  Increment (cols – m) by 4 so that the next higher multiple of 4 is computed. This implies that in certain cases up to four extra pixels may be computed if (cols – m) is an exact multiple of 4. In other cases, 1, 2 or 3 extra pixels may be computed. In order to annul this extra computation, four locations starting at x[cols–m] are zeroed out before returning to the calling function.

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 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 do increase within this loop compared to the natural C code.

- The optimized assembly code tries to balance the computing with the data loads/stores that have to be performed because of the loop order interchange. In order to do this word-wide loads are used. The outer loop that computes one filter tap at a time is unrolled and computes two filter taps at a time. In order to decrease the number of loads the first word is pre-loaded outside the j loop and data is re-used within the j loop. The updating of the present correlation sum with the previous correlation sum is done using the add2 instruction. To get better multiplier utilization the inner loop is unrolled four times and four outputs are computed together.
- ***Bank Conflicts:*** No bank conflicts occur in this function.
- ***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*[15 + (cols – m)/2] For m = 8, cols = 720, cycles = 2968
Cycles (case 2 – odd number of filter taps)	k*[15 + (cols – k)/2] + 10 + cols*3/4 k = m–1 For m = 9, cols = 720, cycles = 3518
Code size	768 bytes

5.3.3 errdif_bin — *Error Diffusion, Binary Output*

void errdif_bin(unsigned char errdif_data[ ], int cols, int rows, short err_buf[ ], unsigned char thresh)

Arguments

errdif_data	pointer to input/output image data
cols	number of columns in the image
rows	number of rows in the image
err_buf	buffer where one row of error values is saved
thresh	threshold value in the range [0x00, 0xFF]

Description

The errdif_bin() 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 propogation 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. Threes 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 errdif_bin() is provided below:

```
void errdif_bin (unsigned char errdif_data[ ], int cols, int rows,
short err_buf[ ], unsigned char thresh)
{
   int x, i, y, F;
   int errA, errB, errC, errE, errF;

   for(y = 0, i = 0; y < h; y++)
   {
/*************************************************************/
/* Start with 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];

            for(x = 0; x < w; x++, i++)
            {
                  errC = err_buf[x+1];
                  F = errdif_data[i];

                  /*****************************************************/
                  /* Calculate the resulting pixel. If we assume       */
                  /* this pixel will be set to zero, it also doubles   */
                  /* as the error term.                                */
                  /*****************************************************/

                  errF = F + ((errE*7 + errA + errB*5 + errC*3) >> 4);

                  /*****************************************************/
                  /* Set pixels that are larger than the threshold     */
                  /* to 255, and pixels that are smaller than or       */
                  /* equal to 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 subtract 255 from the error. Store 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 4.
- ❏ 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)*Half-Word.

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.
- ❏ Twin stack pointers have been used to speed up stack accesses.
- ❏ No special alignment of data arrays is expected.
- ❏ ***Bank Conflicts****:* No bank conflicts occur.
- ❏ ***Endian****:* The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility****:* The code is interrupt tolerant, but not interruptible.

Benchmarks

Cycles	[(cols * 4) + 14] * rows + 21
	For cols = 720, rows = 8, cycles = 23,173
Code size	480 bytes

5.3.4 median_3x3 — *3x3 Median Filter*

void median_3x3(unsigned char *in_data, int cols,
unsigned char *out_data)

Arguments

in_data	pointer to input image array
cols	number of columns in image
out_data	pointer to output image array

Description

The routine median_3x3() performs a 3x3 median filtering algorithm. The gray level at each pixel is replaced by the median of the nine neighborhood values. The median of a set of nine numbers is the middle element so that half of the elements in the list are larger and half are smaller. Median 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 MIN value is retained, call it MIN0.

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

The three MIN values I20, I21, I22 are compared and their MIN value is retained, call it MIN2.

The three values MIN0, MED1, MIN2 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 There are no special requirements for this routine.

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	9 * cols + 55 For cols = 128, cycles = 1207 For cols = 720, cycles = 6535
Code size	544 bytes

5.3.5 pix_expand *Pixel Expand*

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

Arguments

n	number of samples processed
in_data	pointer to input array (unsigned chars)
out_data	pointer to output array (shorts)

Description The routine pix_expand() takes an array of unsigned chars (pixels) and zero extends them upto 16-bits to form shorts.

Algorithm Behavioral C code for the routine pix_expand() is provided below:

```
void 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

- nput and Output arrays must be aligned on at least 4 byte boundaries.
- Input data is unsigned 8-bit format and must be a multiple of 8 elements and be at least 8 elements long.

Implementation Notes

- The optimized assembly code is unrolled 8 times, with 2 LDWs reading in a total of 8 bytes per iteration. The bytes are extracted into registers, and are then re-packed as shorts. The packed shorts are then written out using four STWs.
- The packing is achieved using MPYU and ADD. First, the data is shifted left by 15 with the MPYU by multiplying with (1 << 15). The value is then added to itself to shift it left one more bit. A final ADD merges the shifted quantity with a second quantity, giving the packed result.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	0.5 * n + 26 For n = 256, cycles = 154 For n = 1024, cycles = 538
Code size	288 bytes

5.3.6 pix_sat *Pixel Saturate*

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

Arguments

n	number of samples processed
in_data	pointer to input array (shorts)
out_data	pointer to output array (unsigned chars)

Description

The routine pix_sat() performs the saturation of 16-bit signed numbers to 8-bit unsigned numbers. If the data is over 255 it is saturated to 255, if it is less than 0 it is saturated to 0.

Algorithm

Behavioral C code for the routine pix_sat() is provided below:

```
void pix_sat(int n, short *in_data,
unsigned char *out_data)
{
   int j, pixel, pel;
   for (j = 0; j < n; j++)
   {
         pixel = in_data[j];
         pel = (unsigned char) pixel;
         if (pixel > 0xff) pel = 0xff;
         if (pixel < 0x00) pel = 0x00;
         out_data[j] = pel;
   }
}
```

Special Requirements

The input array must be aligned on an 8 bytes boundary and be a multiple of 8 in length, n % 8 = 0.

Implementation Notes

- The data is loaded in pairs of shorts, the sign bits are detected and the test is done to see if values are over 8 bits. Outputs are packed back together to form words.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	n + 37 For n = 256, cycles = 293 For n = 1024, cycles = 1061
Code size	448 bytes

5.3.7 scale_horz *Horizontal Scaling*

void scale_horz(unsigned short *in_data, unsigned int n_x, short *out_data, unsigned int n_y, short *hh, unsigned int l_hh, unsigned int n_hh, short * patch)

Arguments

in_data	pointer to 16-bit input data
n_x	pixels per line in input (un-scaled) data
out_data	pointer to 16-bit output data
n_y	pixels per line in output (scaled) data
hh	pointer to filter coefficients
l_hh	length of each scaling filter (number of filter taps)
n_hh	number of scaling sub-filters
patch	pointer to decrement pattern

Description

The routine scale_horz() performs a resizing function on a horizontal line of image data, using a polyphase FIR filter approach. Scale factors are indicated as m:n where m input samples correspond to n output samples. The code is flexible (within certain restrictions explained below) in terms of scale factors possible as well as number of filter taps.

Algorithm

Behavioral C code for the function histogram() is provided below:

```
void scale_horz (short *in_data, int n_x,
short *out_data, int n_y, short *hh, int l_hh,
int n_hh, short * patch)
{
      int      filter_count;
      int      i,j,k,ka;
      int      jump;
      int      y0;
      short    h0;
      short    *ptr_hh;
      short    x00;
      short    *line0_x, *line0_y;
      short    *patch0;
```

```
                    patch0 = patch + 1;
                    filter_count = n_hh;
                    ka = 0;
                    line0_x = in_data;
                    line0_y = out_data;
                    ptr_hh = hh;

                    for ( i = 0; i < n_y; i++)
                    {
                       y0 = 1 << 5;
                       for ( j = 0; j < l_hh; j+=4)
                       {
                          jump = (int) (*patch0++);
                          for (k=0; k < 4; k++)
                          {
                             h0 = *ptr_hh++;
                             x00 = *(line0_x + ka + k);
                             y0 += ( x00 * h0 );
                          }
                          ka = ka + jump;
                       }

                       *line0_y++ = (short) (y0 >> 6) ;

                       filter_count –= 1;
                       if (!filter_count)
                       {
                          patch0 = patch + 1;
                          ptr_hh = hh;
                          filter_count = n_hh;
                       }
                    }
}
```

Special Requirements

- ❑ Two lines are scaled simultaneously. Data for each line must be aligned on a double word boundary and be multiples of 8 bytes.
- ❑ Filters must be multiples of 4-taps, maximum number of filters possible per scale factor is 16.
- ❑ Input and output data must be 16-bit signed shorts.
- ❑ Filter coefficients must be 16-bit signed shorts.

- ❏ The data is assumed to be in Q6 precision format, 10.6 form.
- ❏ The sub-filters must be all of the same length and must be concatenated in a single linear array.

Implementation Notes

- ❏ Ratio n_y/n_x is the scale factor.
- ❏ ***Bank Conflicts****:* Different filter lengths can produce different bank conflicts but almost zero effect.
- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility****:* The code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	(l_hh * (1+k) * sf * n_x) +15 where k = 1/(4* l_hh) when l_hh%8 = 0, k = 0 otherwise For l_hh = 8, n_x = 640, sf = 0.1875, cycles = 1005 For l_hh = 16, n_x = 1024, sf = 1.3333, cycles = 22,201
Code size	416 bytes

5.3.8 scale_vert *Vertical Scaling*

void scale_vert(short *in_data, short *out_data, int cols, short *ptr_hh, short *mod_hh, int l_hh, int start_line)

Arguments

in_data	pointer to 16-bit input data
cols	width of the input image in pixels
out_data	pointer to 16-bit output data
ptr_hh	pointer to filter taps
mod_hh	pointer to rotated filter taps
l_hh	number of taps in each filter, l_hh % 2 = 0
start_line	position in the buffer for the filter to start

Description

The routine scale_vert() performs a vertical scaling function on a block of data from a frame store. It uses a scaling filter decided upon by the calling function. The filter is rotated depending on the starting line of the circular buffer. One line of length 'cols' is produced per function call. The same filter runs along the entire length of buffer and performs the filter vertically along each set of parallel data points.

Algorithm

Behavioral C code for the routine scale_vert() is provided below:

```
void scale_vert(short *in_data, short *out_data,
int cols, short *ptr_hh, short *mod_hh, int l_hh,
int start_line)
{
      int i, j, k, y;
      k = 0;
      for (j = start_line; j < l_hh; j++)
         mod_hh[j] = ptr_hh[k++];
      if (start_line)
      {
         for (j = 0; j < start_line; j++)
               mod_hh[j] = ptr_hh[k++];
      }
      for ( i = 0; i < cols; i++)
      {
            y = 0;
            for ( j = 0; j < l_hh; j++)
            {
               y += (in_data[j*cols+i] *
               mod_hh[j]);
            }
            *out_data++ = (y >> 16);
      }
}
```

Special Requirements

- ❏ l_hh must be divisible by 2, pad with zeros for non % 2
- ❏ All data must be double word aligned.
- ❏ Input and output array sizes, mod_hh, ptr_hh, cols must be multiples of 8.
- ❏ Scaling filter coefficients must be of 12-bit precision.

Implementation Notes

- Inner and outer loops are collapsed into one loop.
- The inner loop is unrolled twice with l_hh % 2 = 0.
- The output data half stores are packed together into words and the outer loop is unrolled 4 times to allow 4 parallel filters to be convolved at once.
- ***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	0.75 * l_hh * cols + 6 * l_hh + 37 For cols = 128, l_hh = 4, cycles = 445 For cols = 720, l_hh = 16, cycles = 8773
Code size	544 bytes

Appendix A

Performance/ Warranty and Support

This appendix describes performance considerations related to the ’62x IMGLIB and provides information about warranty, software updates, and customer support issues.

Topic **Page**

A.1 Performance Considerations

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 internal data memory. Any extra cycles due to placement of code or data in external data 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.

Table A–1 provides a listing of the routines provided in this software package as well as 'C62x performance data for each:

Table A–1. 'C62x Routines Performance Data

Function	Description	Cycles	Code Size
boundary()	Boundary Structural Operator	1.25 * (cols * rows) + 4 cycs 'cols' is number of image columns 'rows' is number of image rows For cols = 128, rows = 3, cycs = 484 For cols = 720, rows = 8, cycs = 7204	352 bytes
corr_3x3()	3x3 Correlation with Rounding	[(cols – 2) * 4.5) + 21] cycs 'cols' is number of image columns For cols = 256, cycs = 1164 For cols = 720, cycs = 3252	1120 bytes
corr_gen()	Generalized Correlation	Case 1 – Even number of filter taps m*[15 + (cols – m)/2] cycs 'm' is number of filter taps 'cols' is number of image columns For m = 8, cols = 720, cycs = 2968 Case 2 – Odd number of filter taps k*[15 + (cols – k)/2] + 10 + cols*3/4 cycs k = m–1, 'm' is number of filter taps 'cols' is number of image columns For m = 9, cols = 720, cycs = 3518	768 bytes

Table A–1. ’C62x Routines Performance Data (Continued)

Function	Description	Cycles	Code Size
dilate_bin()	3x3 Binary Dilation	[(cols/4) * 6] + 34 cycs ‘cols’ is number of image cols in bytes For cols = 128*8, cycs = 226 For cols = 720,*8 cycs = 1114	480 bytes
erode_bin()	3x3 Binary Erosion	[(cols/4) * 6] + 34 cycs ‘cols’ is number of image cols in bytes For cols = 128*8, cycs = 226 For cols = 720*8, cycs = 1114	480 bytes
errdif_bin()	Error Diffusion, Binary Output	[(cols * 4) + 14] * rows + 21 cycs ‘cols’ is number of image columns ‘rows’ is number of image rows For cols = 720, rows = 8, cycs = 23,173	480 bytes
fdct_8x8()	Forward Discrete Cosine Transform (FDCT)	160 * num_fdcts + 48 cycs ‘num_fdcts’ is number of fdcts For num_fdcts = 6, cycs = 1008 For num_fdcts = 24, cycs = 3888	1216 bytes
histogram()	Histogram Computation	9/8 * n + 582 cycs ‘n’ is number of points processed For n = 512, cycs = 1158 For n = 1024, cycs = 1734	960 bytes
idct_8x8()	Inverse Discrete Cosine Transform (IDCT)	168 * num_idcts + 62 cycs ‘num_idcts’ is number of idcts For num_idcts = 6, cycs = 1070 For num_idcts = 24, cycs = 4094	1344 bytes
mad_8x8()	8x8 Minimum Absolute Difference	62 * H * V + 21 cycs ‘H’ = columns in search area ‘V’ = rows in search area For H = 4, V = 4, cycs = 1013 For H = 64, V = 32, cycs = 126,997	768 bytes

Table A–1. ’C62x Routines Performance Data (Continued)

Function	Description	Cycles	Code Size
mad_16x16()	16x16 Minimum Absolute Difference	231 * H * V + 21 cycs ‘H’ = columns in search area ‘V’ = rows in search area For H = 4, V = 4, cycs = 3717 For H = 64, V = 32, cycs = 473,109	768 bytes
median_3x3()	3x3 Median Filter	9 * cols + 55 cycs ‘cols’ is number of image columns For cols = 128, cycs = 1207 For cols = 720, cycs = 6535	544 bytes
perimeter()	Perimeter Structural Operator	3 * (cols –2) + 14 cycs ‘cols’ is number of image columns For cols = 128, cycs = 392 For cols = 720, cycs = 2168	358 bytes
pix_expand()	Pixel Expand	0.5 * n + 26 cycs ‘n’ is number of data samples For n = 256, cycs = 154 For n = 1024, cycs = 538	288 bytes
pix_sat()	Pixel Saturate	n + 37 cycs ‘n’ is number of data samples For n = 256, cycs = 293 For n = 1024, cycs = 1061	448 bytes
quantize()	Matrix Quantization with Rounding	(blk_size/16) * (4 + num_blks * 12) + 26 cycs ‘blk_size’ is block size ‘num_blks’ is number of blocks For blk_size=64, num_blks=8, cycs=426 For blk_size=256, num_blks=24, cycs=4696	1024 bytes

Table A–1. ’C62x Routines Performance Data (Continued)

Function	Description	Cycles	Code Size
scale_horz()	Horizontal Scaling	(l_hh*(1+k)*sf*n_x)+15 cycs where k=1/(4*l_hh) when l_hh%8=0, k=0 otherwise ‘l_hh’ is number of filter taps per output ‘sf’ is scale factor ‘n_x’ is pixels per line in input For l_hh=8, n_x=640, sf=0.1875, cycs=1005 For l_hh=16, n_x=1024, sf=1.3333, cycs=22,201	416 bytes
scale_vert()	Vertical Scaling	0.75*l_hh*cols+6*l_hh+37 cycs ‘l_hh’ is number of filter taps per output ‘cols’ is number of image columns For cols = 128, l_hh = 4, cycs = 445 For cols = 720, l_hh = 16, cycs = 8773	544 bytes
sobel()	Sobel Edge Detection	3 * cols * (rows –2) + 34 cycs ‘cols’ is number of image columns ‘rows’ is number of image rows For cols = 128, rows = 8, cycs = 2338 For cols = 720, rows = 8, cycs = 12,994	608 bytes
threshold()	Image Thresholding	(cols * rows/16) * 9 + 50 cycs ‘cols’ is number of image columns ‘rows’ is number of image rows For cols = 128, rows = 8, cycs = 626 For cols = 720, rows = 8, cycs = 3290	576 bytes
wave_horz()	Horizontal Wavelet Transform	(4 * cols) + 5 cycs ‘cols’ is number of image columns For cols = 256, cycs = 1029 For cols = 512, cycs = 2058	640 bytes
wave_vert()	Vertical Wavelet Transform	(8 * cols) + 48 cycs ‘cols’ is number of image columns For cols = 256, cycs = 2096 For cols = 512, cycs = 4144	736 bytes

A.2 Warranty

The ’C62x IMGLIB is distributed free of charge.

BETA RELEASE SPECIAL DISCLAIMER: This IMGLIB software release is preliminary (Beta). It is intended for evaluation only. Testing and characterization has not been fully completed. Production release typically follows the Beta release but there are no explicit guarantees.

A.3 IMGLIB Software Updates

’C62x 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.4 IMGLIB Customer Support

If you have questions or want to report problems or suggestions regarding the ’C62x 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): The BSL is 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 μ-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

μ-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’s 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 micro-programmed 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

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W