

GENERIC DIGITAL DESIGN INCORPORATED

**IMAGE PROCESSING LIBRARY FOR TMS320C6000 DSP
GDD4375 USER'S MANUAL**

2006

GDD4375 Image Processing Library for TMS320C6000 DSP Reference Manual. Release 1.00.00

Copyright (C) 1999-2006 Generic Digital Design Incorporated, All Rights Reserved.

Information in this document is provided in connection with Generic Digital Design Incorporated products. No license, express or implied, to any intellectual property rights is granted by this document. Except as provided in GDDI's Terms and Conditions of Sale or License Agreement for such products, GDDI assumes no liability whatsoever, and disclaims any express or implied warranty, relating to sale and/or use of GDDI products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

Designers must not rely on the absence or characteristics of any features or instructions marked `_reserved_` or `_undefined_`. Generic Digital Design reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Generic Digital Design Incorporated products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available upon request.

GDDI products are not intended for use in medical, life saving, or life sustaining applications. GDDI may make changes to specifications and product descriptions at any time, without notice.

Generic Digital Design Incorporated, GDDI and the GDDI logo, are registered trademarks of Generic Digital Design Incorporated.

Third-party brands and names are the property of their respective owners.

The specification for the Library was developed by Andrew Nesterov <andrew.nesterov@techemail.com>.

Generic Digital Design Incorporated.
email: <andrew.nesterov@techemail.com>
URL: <http://microprocessing.iwarp.com/>

For product marketing information and technical support contact GDDI sales office or authorized distributor. For applications engineering assistance contact

DSP Applications Assistance

Technical Support: <andrew.nesterov@techemail.com>
Custom Design: <andrew.nesterov@techemail.com>

Printed in the USA

Contents

Contents	3
Introduction	5
Installation	6
Image Management	7
Image Copy/Move	7
img_fill8() Fill an 8 bit unsigned image with an 8 bit unsigned constant.....	7
img_fill32() Fill a 32 bit signed image with a 32 bit signed constant.....	8
Subimage extraction	9
Image Arithmetic/Numerical Operations	10
Basic arithmetic operations (add/subtract)	10
img_add8() Add two unsigned 8 bit images.....	10
img_adds8 () Add two unsigned 8 bit images with saturation.....	11
img_sub8() Subtract two unsigned 8 bit images.....	12
img_subs8() Subtract two unsigned 8 bit images with saturation.....	13
Constant add/subtract	14
img_addc8() Add an unsigned 8 bit constant to an unsigned 8 bit image.....	14
img_addcs8() Add an unsigned 8 bit constant to an unsigned 8 bit image with saturation.....	15
img_subc8() Subtract an unsigned 8 bit constant from an unsigned 8 bit image.....	16
img_subcs8() Subtract an unsigned 8 bit constant from an unsigned 8 bit image with saturation.....	18
img_csub8() Subtract an unsigned 8 bit image from an unsigned 8 bit constant.....	19
img_csubcs8() Subtract an unsigned 8 bit image from an unsigned 8 bit constant with saturation.....	20
Image Multiply/Scale	21
img_mpy8() Multiply two unsigned 8 bit images.....	21
img_mpy8s() Multiply two unsigned 8 bit images with saturation.....	22
img_mpyc8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant.....	23
img_mpycs8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant with saturation.....	24
img_scale8() Scale an unsigned 8 bit image by an unsigned fixed point Q8.8 image.....	25
img_scalec8() Scale an unsigned 8 bit image by an unsigned fixed point Q8.8 constant.....	27
Image MAC	28
img_mac8() Multiply two unsigned 8 bit images and accumulate the product.....	28
Image/Constant MAC	29
img_macc8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant and accumulate the product.....	29
img_macsc8() Multiply an unsigned 8 bit image by a signed 8 bit constant and accumulate the product.....	30
Elementary functions	31
Image Limits/Clipping/Thresholds	32
Image Min/Max, absolute Min/Max	32
img_max8() Maximum of two unsigned 8 bit images.....	32
img_min8() Minimum of two unsigned 8 bit images.....	33
Image Clipping/Threshold	34
img_clip8() Clip an unsigned 8 bit image.....	34
img_threshlo8() Apply lower threshold to an unsigned 8 bit image.....	35
img_threshup8() Apply upper threshold to an unsigned 8 bit image.....	36
img_threshld8() Apply lower and upper threshold to an unsigned 8 bit image.....	37

<i>Image Logical Operations/Binarization</i>	39
Image/Image Comparisons	39
img_equ8() Compare two unsigned 8 bit images for equal	39
img_neq8() Compare two unsigned 8 bit images for not equal.....	40
img_ltn8() Compare two unsigned 8 bit images for less than	41
img_gtn8() Compare two unsigned 8 bit images for greater than.....	42
Image/Constant Comparisons	43
img_eqc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for equal	43
img_neqc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for not equal	44
img_ltn8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for less than.....	45
img_gtnc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for greater than	46
<i>Image Bitwise Operations</i>	48
Image/Image bitwise AND/OR/XOR/NOT	48
img_bit_and8() Bitwise AND of two unsigned 8 bit images.....	48
img_bit_or8() Bitwise OR of two unsigned 8 bit images.....	49
img_bit_xor8() Bitwise XOR of two unsigned 8 bit images	50
img_bit_not8() Bitwise NOT of an unsigned 8 bit image	51
Image/Constant bitwise AND/OR/XOR	52
img_bit_andc8() Bitwise AND of an unsigned 8 bit constant with an unsigned 8 bit image	52
img_bit_orc8() Bitwise OR of an unsigned 8 bit constant with an unsigned 8 bit image	53
img_bit_xorc8() Bitwise XOR of an unsigned 8 bit constant with an unsigned 8 bit image	54
<i>Color Transformations</i>	56
<i>Motion Detection/Estimation</i>	57
img_sad8() Sum of absolute differences of unsigned 8 bit images	57
<i>Graphics Primitives</i>	59
<i>Image Correlations/Convolutions</i>	60
<i>Histograms, statistics</i>	61
img_avg8() Average of two unsigned 8 bit images.....	61
img_hist_acc8() Histogram accumulation of an unsigned 8 bit image	62
img_lut8() Lookup table transformation of an unsigned 8 bit image	63
img_neg8() Negative of an unsigned 8 bit image	64
<i>Edge Detection</i>	66
<i>Image Filtering</i>	67
<i>Index</i>	68

Introduction

Installation

Image Management

Image Copy/Move

img_fill8() Fill an 8 bit unsigned image with an 8 bit unsigned constant

Synopsis

```
void img_fill8 (int32 n,           // number of pixels
               uint8 alpha,       // fill constant 8 bit
               uint8 * restrict image) // input/result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image	image initialized with constant value
-------	---------------------------------------

Return value none

Description

The function `img_fill8()` stores an 8 bit unsigned constant into pixels of an unsigned 8 bit input image

$$\text{image}(i) = \text{alpha}$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
for (count = 0; count < nrows; count++)
{
    img_fill8 (rowlen, alpha, ptr);
    ptr += row_stride;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_fill32() Fill a 32 bit signed image with a 32 bit signed constant

Synopsis

```
void img_fill32 (int32 n,           // number of pixels
                int32 alpha,       // fill constant 8 bit
                int32 * restrict image) // input/result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	32 bit signed constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image	image initialized with constant value
-------	---------------------------------------

Return value none

Description

The function `img_fill32()` stores a 32 bit signed constant into pixels of a signed 32 bit input image

$$\text{result_image}(i) = \text{image0}(i) + \text{image1}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
for (count = 0; count < nrows; count++)
{
    img_fill32 (rowlen, alpha, ptr);
    ptr += row_stride;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

Subimage extraction

Image Arithmetic/Numerical Operations

Basic arithmetic operations (add/subtract)

img_add8() Add two unsigned 8 bit images

Synopsis

```
void img_add8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result sum of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_add8()` calculates a sum of pixels of two unsigned 8 bit images without saturation

$$\text{result_image}(i) = \text{image0}(i) + \text{image1}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Addition is performed with no saturation. If the result is above 0xFF boundary, it is wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
```

```

ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_add8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_adds8 () Add two unsigned 8 bit images with saturation

Synopsis

```

void img_adds8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result sum of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_adds8()` calculates a sum of pixels of two unsigned 8 bit images with saturation

$$\text{result_image}(i) = \text{saturate} (\text{image0}(i) + \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Addition is performed with saturation. If the result is above 0xFF boundary it will be automatically saturated to 0xFF. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_adds8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_sub8() Subtract two unsigned 8 bit images

Synopsis

```
void img_sub8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result difference of two input images, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_sub8()` calculates a difference of pixels of two unsigned 8 bit images without saturation

$$\text{result_image}(i) = \text{image0}(i) - \text{image1}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as

internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with no saturation. If the result is below 0x00 boundary, it will be wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_sub8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_subs8() Subtract two unsigned 8 bit images with saturation

Synopsis

```
void img_subs8 (int32 n,           // number of pixels
                uint8 * restrict image0, // input image 0
                uint8 * restrict image1, // input image 1
                uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result difference of two input images, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_subs8()` calculates a difference of pixels of two unsigned 8 bit images with saturation

$$\text{result_image}(i) = \text{saturate} (\text{image0}(i) - \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with saturation. If the result is below 0x00 boundary, it will be automatically saturated to 0x00. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrow; count++)
{
    img_subs8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

Constant add/subtract

img_addc8() Add an unsigned 8 bit constant to an unsigned 8 bit image

Synopsis

```
void img_addc8 (int32 n,           // number of pixels
               uint8 alpha,       // input 8 bit unsigned constant
               uint8 * restrict image, // input image
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

`image_res` pointer to result sum of the input image and constant, aligned on a doubleword boundary

Return value none

Description

The function `img_addc8()` adds a constant to pixels of an input unsigned 8 bit image without saturation

$$\text{result_image}(i) = \text{alpha} + \text{image}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Addition is performed with no saturation. If the result is above 0xFF boundary, it is wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_addc8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_addcs8() Add an unsigned 8 bit constant to an unsigned 8 bit image with saturation

Synopsis

```
void img_addcs8 (int32 n,           // number of pixels
                uint8 alpha,       // input 8 bit unsigned constant
                uint8 * restrict image, // input image
                uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result sum of the input image and constant, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_addcs8()` adds a constant to pixels of an input unsigned 8 bit image with saturation

$$\text{result_image}(i) = \text{saturate} (\text{alpha} + \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Addition is performed with saturation. If the result is above 0xFF boundary it will be automatically saturated to 0xFF. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_addcs8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_subc8() Subtract an unsigned 8 bit constant from an unsigned 8 bit image

Synopsis

```
void img_subc8 (int32 n,           // number of pixels
               uint8 * restrict image, // input image
```

```
uint8 alpha, // input 8 bit unsigned constant
uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary
alpha	input 8 bit unsigned constant

Output Parameters

image_res	pointer to result difference of the input image and constant, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_subc8()` subtracts a constant from pixels of an input unsigned 8 bit image without saturation

$$\text{result_image}(i) = \text{image}(i) - \text{alpha}$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with no saturation. If the result is below 0x00 boundary, it will be wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrow; count++)
{
    img_subc8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_subcs8() Subtract an unsigned 8 bit constant from an unsigned 8 bit image with saturation

Synopsis

```
void img_subcs8 (int32 n,          // number of pixels
                uint8 * restrict image, // input image
                uint8 alpha,        // input 8 bit unsigned constant
                uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary
alpha	input 8 bit unsigned constant

Output Parameters

image_res	pointer to result difference of the input image and constant, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_subcs8()` subtracts a constant from pixels of an input unsigned 8 bit image with saturation

$$\text{result_image}(i) = \text{saturate} (\text{image}(i) - \text{alpha})$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with saturation. If the result is below 0x00 boundary, it will be automatically saturated to 0x00. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_subcs8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_csub8() Subtract an unsigned 8 bit image from an unsigned 8 bit constant

Synopsis

```
void img_csub8 (int32 n,           // number of pixels
               uint8 alpha,       // input 8 bit unsigned constant
               uint8 * restrict image, // input image
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result difference of the input constant and image, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_csub8()` subtracts pixels of an input unsigned 8 bit image from a constant without saturation

$$\text{result_image}(i) = \text{alpha} - \text{image}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with no saturation. If the result is below 0x00 boundary, it will be wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
```

```

        img_csub8 (rowlen, ptr, alpha, ptr_out);
        ptr += row_stride;
        ptr_out += row_stride_res;
    }

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_csubs8() Subtract an unsigned 8 bit image from an unsigned 8 bit constant with saturation

Synopsis

```

void img_csubs8 (int32 n,          // number of pixels
                uint8 alpha,      // input 8 bit unsigned constant
                uint8 * restrict image, // input image
                uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result difference of the input constant and image, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_csubs8()` subtracts a constant from pixels of an input unsigned 8 bit image with saturation

$$\text{result_image}(i) = \text{saturate} (\text{alpha} - \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Subtraction is performed with saturation. If the result is below 0x00 boundary, it will be automatically saturated to 0x00. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```

ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_csubs8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

Image Multiply/Scale

img_mpy8() Multiply two unsigned 8 bit images

Synopsis

```

void img_mpy8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result product of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_mpy8()` calculates a product of pixels of two unsigned 8 bit images without saturation

$$\text{result_image}(i) = \text{image0}(i) \times \text{image1}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication is performed with no saturation. If the result is above 0xFF boundary, it is wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_mpy8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_mpys8() Multiply two unsigned 8 bit images with saturation

Synopsis

```
void img_mpys8 (int32 n,           // number of pixels
                uint8 * restrict image0, // input image 0
                uint8 * restrict image1, // input image 1
                uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result product of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_mpys8()` calculates a product of pixels of two unsigned 8 bit images with saturation

$$\text{result_image}(i) = \text{saturate} (\text{image0}(i) \times \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication is performed with saturation. If the result is above 0xFF boundary it will be automatically saturated to 0xFF. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_mpys8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_mpyc8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant

Synopsis

```
void img_mpyc8 (int32 n,           // number of pixels
               uint8 alpha,       // input 8 bit unsigned constant
               uint8 * restrict image, // input image
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result product of the input image and constant, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_mpyc8()` multiplies pixels of an input unsigned 8 bit image by a constant without saturation

$$\text{result_image}(i) = \text{alpha} \times \text{image}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication is performed with no saturation. If the result is above 0xFF boundary, it is wrapped around to the [0x00,0xFF] interval. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_mpyc8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_mpycs8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant with saturation

Synopsis

```
void img_mpycs8 (int32 n,           // number of pixels
                uint8 alpha,       // input 8 bit unsigned constant
                uint8 * restrict image, // input image
                uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>alpha</code>	input 8 bit unsigned constant
<code>image</code>	pointer to input image, aligned on a doubleword boundary

Output Parameters

<code>image_res</code>	pointer to result product of the input image and constant, aligned on a
------------------------	---

doubleword boundary

Return value none

Description

The function `img_mpycs8()` multiplies pixels of an input unsigned 8 bit image by a constant with saturation

$$\text{result_image}(i) = \text{saturate} (\text{alpha} \times \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication is performed with saturation. If the result is above 0xFF boundary it will be automatically saturated to 0xFF. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_mpycs8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_scale8() Scale an unsigned 8 bit image by an unsigned fixed point Q8.8 image

Synopsis

```
void img_scale8 (int32 n,           // number of pixels
                uint8 * restrict image0, // input image 0 of scale factors
                uint8 * restrict image1, // input image 1
                uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>image0</code>	pointer to input image of scale factors, aligned on a doubleword boundary

`image1` pointer to input image 1, aligned on a doubleword boundary

Output Parameters

`image_res` pointer to result product of input images, aligned on a doubleword boundary

Return value none

Description

The function `img_scale8()` calculates a scaled product of pixels of two unsigned 8 bit images

$$\text{result_image}(i) = (\text{image0}(i) \times \text{image1}(i)) \gg 8$$

assuming that the pixels of one of the input images are fixed point Q8.8 values and the pixels of the other input image are integer numbers. For clarity, it is assumed that the first pointer `image0` points to the array of unsigned Q8.8 fixed point scaling numbers. The unsigned Q8.8 fixed point number format means that there are no sign bit and no integer part:

$$x(\text{Q8.8}) = 0 . b7 b6 \dots b2 b1 b0 < 1$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrow; count++)
{
    img_scale8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_scalec8() Scale an unsigned 8 bit image by an unsigned fixed point Q8.8 constant

Synopsis

```
void img_scalec8 (int32 n,          // number of pixels
                 uint8 alpha,      // input 8 bit unsigned constant
                 uint8 * restrict image, // input image
                 uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result product of the input image and scaling constant, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_scalec8()` scales pixels of an unsigned 8 bit images by an unsigned Q8.8 fixed point 8 bit constant

$$\text{result_image}(i) = (\text{alpha} \times \text{image}(i)) \gg 8$$

The unsigned Q8.8 fixed point number format means that there are no sign bit and no integer part:

$$x(\text{Q8.8}) = 0 . \text{b7 b6} \dots \text{b2 b1 b0} < 1$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input image is assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_scalec8 (rowlen, alpha, ptr, ptr_out);
}
```

```

        ptr += row_stride;
        ptr_out += row_stride_res;
    }

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

Image MAC

img_mac8() Multiply two unsigned 8 bit images and accumulate the product

Synopsis

```

void img_mac8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint32 * restrict image2) // input/output image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary
image2	pointer to input image 2, aligned on a doubleword boundary

Output Parameters

Image2	pointer to a 32 bit accumulated product of input images 0 and 1, aligned on a doubleword boundary
--------	---

Return value none

Description

The function `img_mac8()` multiplies pixels of two unsigned 8 bit images and accumulates the product in pixels of an input/output unsigned 32 bit image pointed to by the `image2` pointer

$$\text{image2}(i) = \text{image2}(i) + \text{image0}(i) \times \text{image1}(i)$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication and addition are performed with no saturation. Accumulator image has 32 bit pixel size. Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr2 = image2;
for (count = 0; count < nrows; count++)
{
    img_mac8 (rowlen, ptr0, ptr1, ptr2);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr2 += row_stride2;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

Image/Constant MAC

img_macc8() Multiply an unsigned 8 bit image by an unsigned 8 bit constant and accumulate the product

Synopsis

```
void img_macc8 (int32 n,           // number of pixels
                uint8 alpha,      // input 8 bit unsigned constant
                uint8 * restrict image, // input image
                uint32 * restrict image2) // input/output image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary
image2	pointer to input image 2, aligned on a doubleword boundary

Output Parameters

image2	pointer to 32 bit accumulated product of input image and constant, aligned on a doubleword boundary
--------	---

Return value none

Description

The function `img_macc8()` multiplies pixels of an unsigned 8 bit image by an unsigned 8 bit constant and accumulates the product in pixels of an input/output unsigned 32 bit image pointed to by the `image2` pointer

$$\text{image2}(i) = \text{image2}(i) + \text{image}(i) \times \text{alpha}$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication and addition are performed with no saturation. Accumulator image has 32 bit pixel size. Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr2 = image2;
for (count = 0; count < nrow; count++)
{
    img_macc8 (rowlen, alpha, ptr0, ptr2);
    ptr0 += row_stride0;
    ptr2 += row_stride2;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_macsc8() Multiply an unsigned 8 bit image by a signed 8 bit constant and accumulate the product

Synopsis

```
void img_macsc8 (int32 n,          // number of pixels
                int8 alpha,       // input 8 bit signed constant
                uint8 * restrict image, // input image
                int32 * restrict image2) // input/output image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>alpha</code>	input 8 bit signed constant
<code>image</code>	pointer to input image, aligned on a doubleword boundary
<code>image2</code>	pointer to input image 2, aligned on a doubleword boundary

Output Parameters

<code>image2</code>	pointer to 32 bit accumulated product of input image and constant,
---------------------	--

aligned on a doubleword boundary

Return value none

Description

The function `img_macsc8()` multiplies pixels of an unsigned 8 bit image with a signed 8 bit constant and accumulates the product in pixels of an input/output signed 32 bit image pointed to by the `image2` pointer

$$\text{image2}(i) = \text{image2}(i) + \text{image}(i) \times \text{alpha}$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Multiplication and addition are performed with no saturation. Accumulator image is signed and has 32 bit pixel size. Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr2 = image2;
for (count = 0; count < nrows; count++)
{
    img_macsc8 (rowlen, alpha, ptr0, ptr2);
    ptr0 += row_stride0;
    ptr2 += row_stride2;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

Elementary functions

Image Limits/Clipping/Thresholds

Image Min/Max, absolute Min/Max

img_max8() Maximum of two unsigned 8 bit images

Synopsis

```
void img_max8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result max of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_max8()` performs comparison of pixels of input images and sets pixels of output image to maximum pixel of input images 0 and 1

```
if (image0(i) > image1(i)) result_image(i) = image0(i)
if (image0(i) = image1(i)) result_image(i) = image1(i)
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
```

```

{
    img_max8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_min8() Minimum of two unsigned 8 bit images

Synopsis

```

void img_min8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result min of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_min8()` performs comparison of pixels of input images and sets pixels of output image to minimum pixel of input images 0 and 1

```

if (image0(i) > image1(i)) result_image(i) = image1(i)
if (image0(i) = image1(i)) result_image(i) = image0(i)

```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```

ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_min8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

Image Clipping/Threshold

img_clip8() Clip an unsigned 8 bit image

Synopsis

```

void img_clip8 (int32 n,           // number of pixels
               uint8 alpha,       // input 8 bit unsigned constant
               uint8 * restrict image, // input image
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned lower threshold
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result clipped image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_clip8()` performs comparison of pixels of an input image and an input threshold and sets pixels of output image to 0x00 if input pixels are less than or equal to threshold or to pixels of input image otherwise

```

if (image(i) = alpha) result_image(i) = 0x00
if (image(i) > alpha) result_image(i) = image(i)

```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as

internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_clip8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_threshlo8() Apply lower threshold to an unsigned 8 bit image

Synopsis

```
void img_thershlo8 (int32 n,          // number of pixels
                   uint8 alpha,      // input 8 bit unsigned constant
                   uint8 * restrict image, // input image
                   uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned lower threshold
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result thresholded image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_thershlo8()` performs comparison of pixels of input image and threshold and set pixels of output image to threshold value if input pixels are less than or equal to threshold or to pixels of input image otherwise

```
if (image(i) = alpha) result_image(i) = alpha
if (image(i) > alpha) result_image(i) = image(i)
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_threshlo8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_threshup8() Apply upper threshold to an unsigned 8 bit image

Synopsis

```
void img_thershup8 (int32 n,          // number of pixels
                   uint8 alpha,      // input 8 bit unsigned constant
                   uint8 * restrict image, // input image
                   uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned upper threshold
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result thresholded input image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_thershup8()` performs comparison of pixels of input image and threshold and set pixels of output image to threshold value if input pixels are greater than threshold or to pixels of input image otherwise

if (image(i) > alpha) result_image(i) = alpha

```
if (image(i) = alpha) result_image(i) = image(i)
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_threshup8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncols)$ can be performed, provided that the total image size is a multiple of 8.

img_threshld8() Apply lower and upper threshold to an unsigned 8 bit image

Synopsis

```
void img_thershld8 (int32 n,          // number of pixels
                   uint8 alfa,       // input 8 bit unsigned constant
                   uint8 beta,       // input 8 bit unsigned constant
                   uint8 * restrict image, // input image
                   uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alfa	input 8 bit unsigned lower threshold
beta	input 8 bit unsigned upper threshold
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result thresholded input image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_threshld8()` performs comparison of pixels of input image and thresholds and set pixels of output image to threshold values if input pixels are outside threshold bounds or to pixels of input image otherwise. Lower threshold value must be strictly less than upper threshold value

```
if (image(i) = alpha) result_image(i) = alpha
if (image(i) = beta) result_image(i) = beta
if (alpha < image(i) < alpha) result_image(i) = image(i)
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_threshld8 (rowlen, alfa, beta, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

Image Logical Operations/Binarization

Image/Image Comparisons

img_equ8() Compare two unsigned 8 bit images for equal

Synopsis

```
void img_equ8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result of comparison of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_equ8()` performs comparison of pixels of input images and set pixels of output image to `0xFF` if input pixels are equal or to `0x00` otherwise

$$\begin{aligned} &\text{if (image0(i) = image1(i)) result_image(i) = 0xFF} \\ &\text{if (image0(i) \neq image1(i)) result_image(i) = 0x00} \end{aligned}$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
```

```

        img_eq8 (rowlen, ptr0, ptr1, ptr_out);
        ptr0 += row_stride0;
        ptr1 += row_stride1;
        ptr_out += row_stride_res;
    }

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_neq8() Compare two unsigned 8 bit images for not equal

Synopsis

```

void img_neq8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result of comparison of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_neq8()` performs comparison of pixels of input images and set pixels of output image to 0xFF if input pixels are not equal or to 0x00 otherwise

```

if (image0(i) != image1(i)) result_image(i) = 0xFF
if (image0(i) == image1(i)) result_image(i) = 0x00

```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
```

```

ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_neq8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_ltn8() Compare two unsigned 8 bit images for less than

Synopsis

```

void img_ltn8 (int32 n,           // number of pixels
              uint8 * restrict image0, // input image 0
              uint8 * restrict image1, // input image 1
              uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result of comparison of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_ltn8()` performs comparison of pixels of input images and set pixels of output image to 0xFF if pixels of image 0 are less than pixels of image 1 or to 0xFF otherwise

```

if (image0(i) < image1(i)) result_image(i) = 0xFF
if (image0(i) = image1(i)) result_image(i) = 0x00

```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_ltn8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_gtn8() Compare two unsigned 8 bit images for greater than

Synopsis

```
void img_gtn8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result of comparison of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_gtn8()` performs comparison of pixels of input images and set pixels of output image to 0xFF if pixels of image 0 are greater than pixels of image 1 or to 0x00 otherwise

```
if (image0(i) > image1(i)) result_image(i) = 0xFF
if (image0(i) = image1(i)) result_image(i) = 0x00
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as

internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_gtn8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

Image/Constant Comparisons

img_equc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for equal

Synopsis

```
void img_equc8 (int32 n,           // number of pixels
               uint8 * restrict image, // input image
               uint8 alpha,        // input 8 bit unsigned constant
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary
alpha	input 8 bit unsigned constant

Output Parameters

image_res	pointer to result of comparison of the constant and input image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_equc8()` performs comparison of pixels of input image and constant and set pixels of output image to `0xFF` if input pixels are equal to constant or to `0x00` otherwise

```
if (image0(i) = alpha) result_image(i) = 0xFF
if (image0(i) ? alpha) result_image(i) = 0x00
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_equc8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_neqc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for not equal

Synopsis

```
void img_neqc8 (int32 n,           // number of pixels
                uint8 * restrict image, // input image
                uint8 alpha,       // input 8 bit unsigned constant
                uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>image</code>	pointer to input image, aligned on a doubleword boundary
<code>alpha</code>	input 8 bit unsigned constant

Output Parameters

<code>image_res</code>	pointer to result of comparison of the constant and input image, aligned on a doubleword boundary
------------------------	---

Return value none

Description

The function `img_neqc8()` performs comparison of pixels of input image and constant and set pixels of output image to `0xFF` if input pixels are not equal to constant or to `0x00` otherwise

```
if (image0(i) ? alpha) result_image(i) = 0xFF
if (image0(i) = alpha) result_image(i) = 0x00
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_neqc8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_ltnc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for less than

Synopsis

```
void img_ltnc8 (int32 n,           // number of pixels
                uint8 * restrict image, // input image
                uint8 alpha,       // input 8 bit unsigned constant
                uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>image</code>	pointer to input image, aligned on a doubleword boundary
<code>alpha</code>	input 8 bit unsigned constant

Output Parameters

<code>image_res</code>	pointer to result of comparison of the constant and input image, aligned on a doubleword boundary
------------------------	---

Return value none

Description

The function `img_ltnc8()` performs comparison of pixels of input image and constant and set pixels of output image to `0xFF` if input pixels are greater than constant or to `0x00` otherwise

```
if (image0(i) < alpha) result_image(i) = 0xFF
if (image0(i) = alpha) result_image(i) = 0x00
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_ltnc8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_gtnc8() Compare an unsigned 8 bit constant with an unsigned 8 bit image for greater than

Synopsis

```
void img_gtnc8 (int32 n,           // number of pixels
               uint8 * restrict image, // input image
               uint8 alpha,       // input 8 bit unsigned constant
               uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>image</code>	pointer to input image, aligned on a doubleword boundary
<code>alpha</code>	input 8 bit unsigned constant

Output Parameters

<code>image_res</code>	pointer to result of comparison of the constant and input image,
------------------------	--

aligned on a doubleword boundary

Return value none

Description

The function `img_gtnc8()` performs comparison of pixels of input image and constant and set pixels of output image to 0xFF if pixels of image 0 are greater than pixels of image 1 or to 0xFF otherwise

```
if (image0(i) > alpha) result_image(i) = 0xFF
if (image0(i) = alpha) result_image(i) = 0x00
```

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_gtnc8 (rowlen, ptr, alpha, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

Image Bitwise Operations

Image/Image bitwise AND/OR/XOR/NOT

img_bit_and8() Bitwise AND of two unsigned 8 bit images

Synopsis

```
void img_bit_and8 (int32 n,          // number of pixels
                  uint8 * restrict image0, // input image 0
                  uint8 * restrict image1, // input image 1
                  uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result bitwise AND of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_bit_and8()` performs bitwise AND operation on pixels of two input images and places results into an output image

$$\text{result_image}(i) = \text{BIT_AND} (\text{image0}(i), \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
```

```

for (count = 0; count < nrows; count++)
{
    img_bit_and8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stridel;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_bit_or8() Bitwise OR of two unsigned 8 bit images

Synopsis

```

void img_bit_or8 (int32 n,          // number of pixels
                 uint8 * restrict image0, // input image 0
                 uint8 * restrict image1, // input image 1
                 uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result bitwise OR of two input images, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_bit_or8()` performs bitwise OR operation on pixels of two input images and places results into an output image

$$\text{result_image}(i) = \text{BIT_OR} (\text{image0}(i), \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_bit_or8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_bit_xor8() Bitwise XOR of two unsigned 8 bit images

Synopsis

```
void img_bit_xor8 (int32 n,          // number of pixels
                  uint8 * restrict image0, // input image 0
                  uint8 * restrict image1, // input image 1
                  uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result bitwise XOR of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_bit_xor8()` performs bitwise XOR operation on pixels of two input images and places results into an output image

$$\text{result_image}(i) = \text{BIT_XOR} (\text{image0}(i), \text{image1}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as

internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_bit_xor8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

img_bit_not8() Bitwise NOT of an unsigned 8 bit image

Synopsis

```
void img_bit_not8 (int32 n,          // number of pixels
                  uint8 * restrict image, // input image
                  uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result NOT of the input image, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_bit_not8()` performs bitwise NOT operation on pixels of the input image and places results into an output image

$$\text{result_image}(i) = \text{BIT_NOT} (\text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_bit_not8 (rowlen, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncol)$ can be performed, provided that the total image size is a multiple of 8.

Image/Constant bitwise AND/OR/XOR

img_bit_andc8() Bitwise AND of an unsigned 8 bit constant with an unsigned 8 bit image

Synopsis

```
void img_bit_andc8 (int32 n,      // number of pixels
                   uint8 alpha,  // input 8 bit unsigned constant
                   uint8 * restrict image, // input image
                   uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
alpha	input 8 bit unsigned constant
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result bitwise AND of the constant and input image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_bit_andc8()` performs bitwise AND of pixels of an input image with an input constant and places results into an output image

$$\text{result_image}(i) = \text{BIT_AND}(\text{alpha}, \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_bit_andc8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_bit_orc8() Bitwise OR of an unsigned 8 bit constant with an unsigned 8 bit image

Synopsis

```
void img_bit_orc8 (int32 n,          // number of pixels
                  uint8 alpha,      // input 8 bit unsigned constant
                  uint8 * restrict image, // input image
                  uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>alpha</code>	input 8 bit unsigned constant
<code>image</code>	pointer to input image, aligned on a doubleword boundary

Output Parameters

<code>image_res</code>	pointer to result bitwise OR of the constant and input image, aligned on a doubleword boundary
------------------------	--

Return value none

Description

The function `img_bit_orc8()` performs bitwise OR of pixels of an input image with an input constant and places results into an output image

$$\text{result_image}(i) = \text{BIT_OR}(\text{alpha}, \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_bit_orc8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(\text{nrow} \times \text{ncols})$ can be performed, provided that the total image size is a multiple of 8.

img_bit_xorc8() Bitwise XOR of an unsigned 8 bit constant with an unsigned 8 bit image

Synopsis

```
void img_bit_xorc8 (int32 n,        // number of pixels
                   uint8 alpha,    // input 8 bit unsigned constant
                   uint8 * restrict image,    // input image
                   uint8 * restrict image_res) // result image
```

Input Parameters

<code>n</code>	number of pixels, multiple of 8
<code>alpha</code>	input 8 bit unsigned constant
<code>image</code>	pointer to input image, aligned on a doubleword boundary

Output Parameters

`image_res` pointer to result bitwise XOR of the constant and input image, aligned on a doubleword boundary

Return value none

Description

The function `img_bit_xor8()` performs bitwise XOR of pixels of an input image with an input constant and places results into an output image

$$\text{result_image}(i) = \text{BIT_XOR}(\text{alpha}, \text{image}(i))$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input/output images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrow; count++)
{
    img_bit_xor8 (rowlen, alpha, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to $(nrow \times ncols)$ can be performed, provided that the total image size is a multiple of 8.

Color Transformations

To be added: Color space transformations

To be added: Color depth transformations

Motion Detection/Estimation

img_sad8() Sum of absolute differences of unsigned 8 bit images

Synopsis

```
uint32 img_sad8 (  
    uint8 * restrict roi0,    // pointer to ROI in image 0  
    uint8 * restrict roi1,    // pointer to ROI in input image 1  
    int32 npixels,           // number of pixels in ROI scanline  
    int32 nlines,            // number of scanlines in ROI  
    int32 stride0,           // number of pixels in full scanline of image 0  
    int32 stride1)           // number of pixels in full scanline of image 1
```

Input Parameters

roi0	pointer to region of interest (ROI) in image 0
roi1	pointer to region of interest (ROI) in image 1
npixels	number of pixels in ROI scanline, multiple of 4
nlines	number of scanlines in ROI, multiple of 2
stride0	number of pixels in full scanline of image 0, multiple of 4
stride1	number of pixels in full scanline of image 1, multiple of 4

Output Parameters

none

Return value The sum of absolute differences.

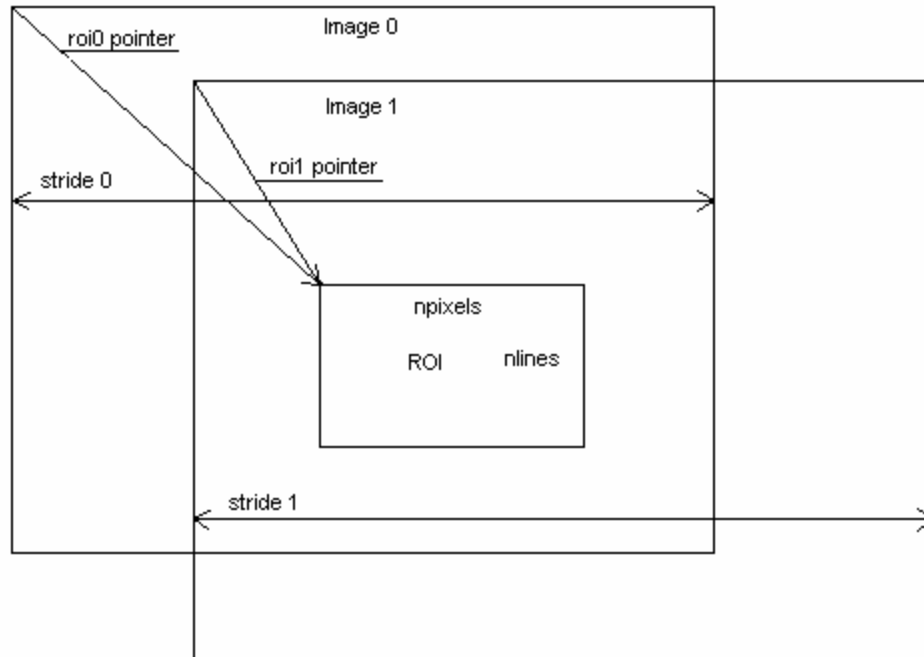
Description

The function `img_sad8()` calculates a sum of absolute difference of pixels in ROI of input images

$$sad = \sum_{i=0, k=0}^{npixels-1, nlines-1} |ROI0(i, k) - ROI1(i, k)|$$

The ROI size is defined by input parameters `npixels`, `nlines`, its location within the total images is pointed by `roi0` and `roi1`. Parameters `stride0` and `stride1` are used to determine locations of the next ROI scanline inside the images.

The ROI, strides and pointers are illustrated on the following picture.



See description of input parameters for explanation of restrictions set on the ROI and source images dimensions. If the parameters do not satisfy to the constrains, the function operation is undefined.

Input images are assumed to contain unsigned data.

Graphics Primitives

To be added: Drawing primitives

To be added: Flood fill

To be added: Coordinate transformations

Image Correlations/Convolutions

To be added: Convolution

To be added: Correlation

Histograms, statistics

img_avg8() Average of two unsigned 8 bit images

Synopsis

```
void img_avg8 (int32 n,           // number of pixels
               uint8 * restrict image0, // input image 0
               uint8 * restrict image1, // input image 1
               uint8 * restrict image_res) // result image
```

Input Parameters

n	number of pixels, multiple of 8
image0	pointer to input image 0, aligned on a doubleword boundary
image1	pointer to input image 1, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result average of two input images, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_avg8()` calculates an average of two unsigned 8 bit images

$$\text{result_image}(i) = (\text{image0}(i) + \text{image1}(i)) / 2$$

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

Input images are assumed to contain unsigned data.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr0 = image0;
ptr1 = image1;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_avg8 (rowlen, ptr0, ptr1, ptr_out);
    ptr0 += row_stride0;
    ptr1 += row_stride1;
}
```

```

        ptr_out += row_stride_res;
    }

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_hist_acc8() Histogram accumulation of an unsigned 8 bit image

Synopsis

```

void img_hist_acc8 (int32 n,      // number of pixels
                   uint8 * restrict image, // input image
                   uint32 * restrict hist) // input/output histogram

```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary
hist	pointer to input image histogram of length N=256, aligned on a doubleword boundary

Output Parameters

hist	pointer to accumulated histogram of length N=256, aligned on a doubleword boundary
------	--

Return value none

Description

The function `img_hist_acc8()` performs histogram accumulation of an unsigned 8 bit image

$$\text{hist}(\text{image}(i)) = \text{hist}(\text{image}(i)) + 1$$

The length of the 32 bit unsigned histogram array pointed to by `hist` is 256 entries. The array is expected to be initialized (filled with zeroes) before the first call to `img_hist_acc8()`.

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```

img_fill132 (256, 0, hist); // initialize histogram
ptr = image;

```

```

for (count = 0; count < nrows; count++)
{
    img_hist_acc8 (rowlen, ptr, hist); // accumulate histogram
    ptr += row_stride;                // of one scanline
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_lut8() Lookup table transformation of an unsigned 8 bit image

Synopsis

```

void img_lut8 (int32 n,           // number of pixels
               uint8 * restrict lut, // input lookup table
               uint8 * restrict image, // input image
               uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
lut	pointer to lookup table of length N=256, aligned on a doubleword boundary
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to transformed image, aligned on a doubleword boundary
-----------	--

Return value none

Description

The function `img_lut8()` performs lookup table transformation of pixels of an unsigned 8 bit image

$$\text{result_image}(i) = \text{LUT}(\text{image}(i))$$

The length of the 8 bit unsigned lookup table array pointed to by `lut` is 256 entries.

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```

ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_lut8 (rowlen, lut, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}

```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

img_neg8() Negative of an unsigned 8 bit image

Synopsis

```

void img_neg8 (int32 n,           // number of pixels
              uint8 * restrict image, // input image
              uint8 * restrict image_res) // result image

```

Input Parameters

n	number of pixels, multiple of 8
image	pointer to input image, aligned on a doubleword boundary

Output Parameters

image_res	pointer to result negative of the input image, aligned on a doubleword boundary
-----------	---

Return value none

Description

The function `img_neg8()` calculates negative of pixels of an unsigned 8 bit image

$$\text{result_image}(i) = \text{BIT_NOT}(\text{image}(i))$$

This operation is equivalent to call to `img_bit_not8()` function, however internally the negative is calculated as `BIT_XOR (0xFF, pixel(i))`.

The operation is performed on linear arrays in memory, the length of the arrays is multiple of 8. If the length is not a multiple of 8, the remainder pixels from 1 to 7 are left unprocessed, as internally the pixel counter is modified to $(n \gg 3)$. It is recommended to pad row or the tail of images with zeros in order to get length a multiple of 8.

The function effectively processes one row of input/output images. The following code can be used to process the whole images

```
ptr = image;
ptr_out = image_res;
for (count = 0; count < nrows; count++)
{
    img_neg8 (rowlen, lut, ptr, ptr_out);
    ptr += row_stride;
    ptr_out += row_stride_res;
}
```

Alternatively, if rows are stored contiguously, a single call to the function with length equal to (nrow x ncols) can be performed, provided that the total image size is a multiple of 8.

To be added: Histogram equalization

To be added: Gamma transformation

Edge Detection

To be added: Sobel mask

To be added: Prewitt mask

To be added: Roberts mask

To be added: Canny edge detector

Image Filtering

To be added: Smoothing
To be added: Sharpening
To be added: FIR/IIR
To be added: Blur/deblur

Index

I

img_add8	10	img_ltn8	41
img_addc8	14	img_ltn8	45
img_addcs8	15	img_lut8	63
img_adds8	11	img_mac8	28
img_avg8	61	img_macc8	29
img_bit_and8	48	img_macsc8	30
img_bit_andc8	52	img_max8	32
img_bit_not8	51	img_min8	33
img_bit_or8	49	img_mpy8	21
img_bit_orc8	53	img_mpyc8	23
img_bit_xor8	50	img_mpycs8	24
img_bit_xorc8	54	img_mpys8	22
img_clip8	34	img_neg8	64
img_csub8	19	img_neq8	40
img_csubs8	20	img_neqc8	44
img_equ8	39	img_sad8	57
img_equc8	43	img_scale8	25
img_fill32	8	img_scalec8	27
img_fill8	7	img_sub8	12
img_gtn8	42	img_subc8	16
img_gtnc8	46	img_subcs8	18
img_hist_acc8	62	img_subs8	13
		img_threshld8	37
		img_threshlo8	35
		img_threshup8	36



Kane Computing Ltd
7 Theatre Court, London Road,
Northwich, Cheshire, CW9 5HB, UK.
Tel: +44(0)1606 351006
Fax: +44(0)1606 351007/8
Email: sales@kanecomputing.com
Web: www.kanecomputing.co.uk