VICP Computation Unit Library and VICP Scheduling Unit Library for DM6446, DM6441, DM647, and DM648

User's Guide

Literature Number: SPRUGN1C
November 2009

TEXAS INSTRUMENTS

Contents

List of Figures

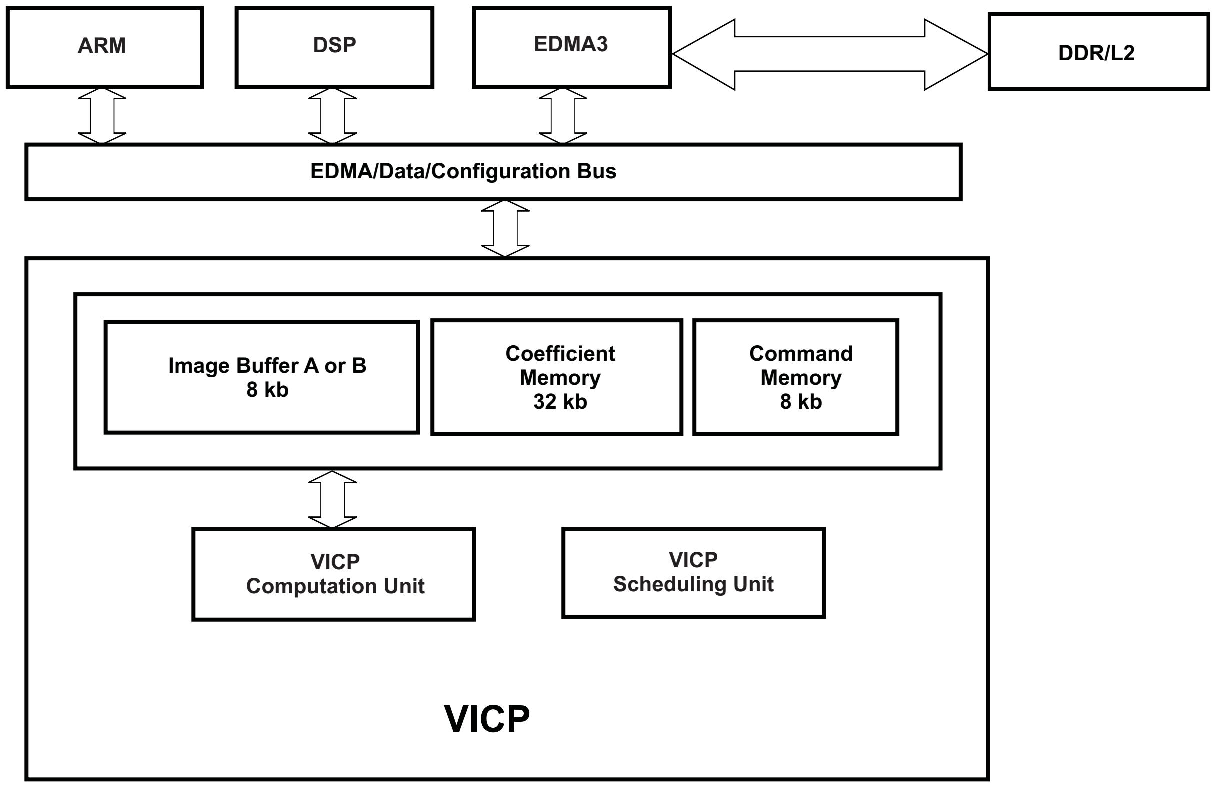
List of Tables

Introduction to the VICP Computation Unit and Scheduling Unit Libraries

1.1 VICP Overview

Figure 1-1 illustrates the VICP, which includes the VICP's processing and scheduling units, as well the attached memories.

Figure 1-1. VICP Block Diagram



The VICP computation unit is based on a SIMD architecture, capable of doing 8 multiple-and-accumulate operations per cycle. It is the ideal compute engine for signal processing and other regular vector processing. To ensure fast data access and instruction decoding time, it can only access its dedicated memory: image buffers A and B, Coefficient and Command memories.

Here is a brief description of each of these memories:

- Command memory: used to store VICP's computation unit's command sequences. A VICP command is made of 18 to 54 bytes. In general simple computation task such as filtering, matrix multiplication can be implemented by a single VICP command. More complex computation task are implemented by a chain of simpler tasks, which results in a sequence of several VICP commands. A *sleep* command at the end of a sequence indicates the termination of a computation chain.

- Image buffers A and B: used to store data to be processed. Data residing in external memory such as DDR or L2 must be brought by EDMA3 into these buffers in order to be processed by the VICP computation unit. After being processed, the data is sent back to L2 memory or DDR. Ping-pong buffer scheme between image buffer A and image buffer B must be implemented for concurrent processing: while VICP processes data in one buffer, EDMA3 writes data from/to the other buffer. An image buffer allows one access per cycle from the VICP. An access being defined as a read or write of up to 8 bytes or 8 16-bits word.
- Coefficient memory: this memory is larger than the image buffers and allows two simultaneous accesses per cycle from the VICP. It can be partitioned by the algorithms into different regions to store constants or scratch data.

Apart from the computation unit, the VICP is equipped with a scheduling unit, which is a hardware controller whose purpose is to offload the DSP from servicing the computation unit and the EDMA3. While both processing and scheduling units are running, the DSP can perform useful operations. The programming of the scheduling unit is abstracted through the VICP scheduling unit's library so no firmware knowledge of the scheduling hardware controller is necessary.

The VICP interfaces with the external memory through the EDMA3. The DSP can also access the VICP memories but very slowly. The normal procedure is to have the scheduling unit control the EDMA3 to bring data into image buffers for processing.

The image buffers, command and coefficient memories are all cached by the DSP.

1.2 Block-Based Processing

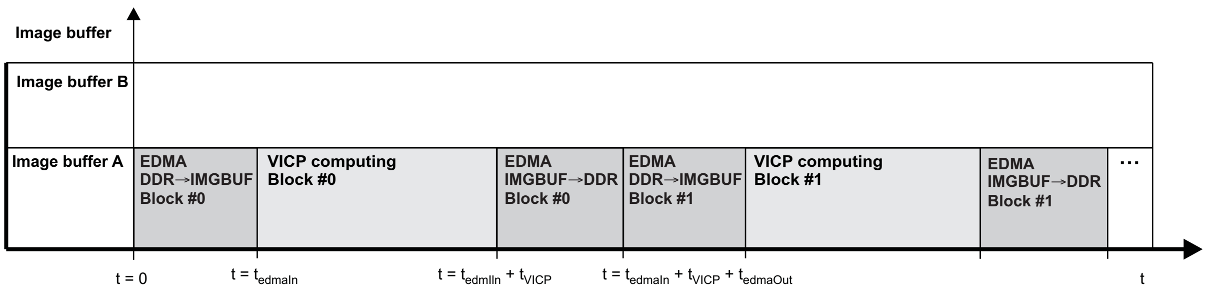
Due to the size of an image buffer, the VICP computation unit can only operate on 8kb of data at a time. To cope with this, if the data to be processed is a 2-D frame, it must be divided into 2-D blocks. To process the entire frame, every block is transferred from external memory to image buffer, processed and then transferred back to external memory. Blocks' shape can be various, not necessarily square. The only restriction is that they must fit within the 8kb of an image buffer.

1.3 Concurrent VICP Processing and EDMA3 Transfer

The block-based nature of the processing implies that EDMA3 transfers must occur as many times as there are blocks in the frame. In order to minimize memory transfer overhead, parallelism between the EDMA3 and the VICP computation unit must be implemented. The VICP scheduling unit's library takes care of that, in addition to abstracting the firmware details of the scheduling unit.

The inefficient approach would be to run the processing sequentially with the EDMA3 transfers on a single image buffer as shown in Figure 1-2:

Figure 1-2. Sequential Processing Schedule (Inefficient)

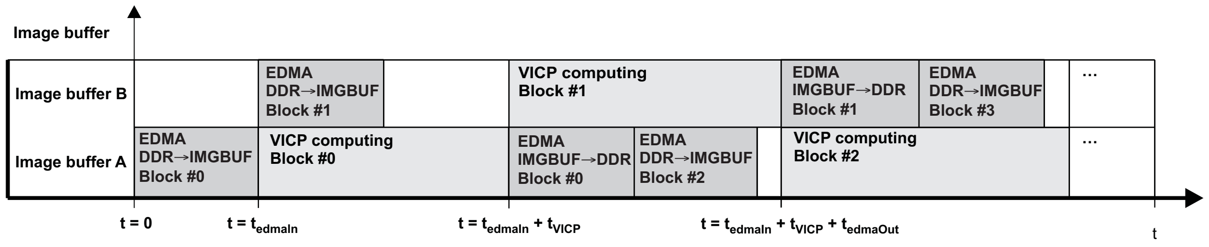


In sequential processing, the execution time for one block is t= t_{edmaIn} + t_{VICP} + $t_{edmaOut}$.

If the VICP scheduling unit's library is used, the same algorithm can execute in a concurrent manner by making use of both image buffers A and B.

In the scenario shown in Figure 1-3, VICP processing time is greater than transfer time.

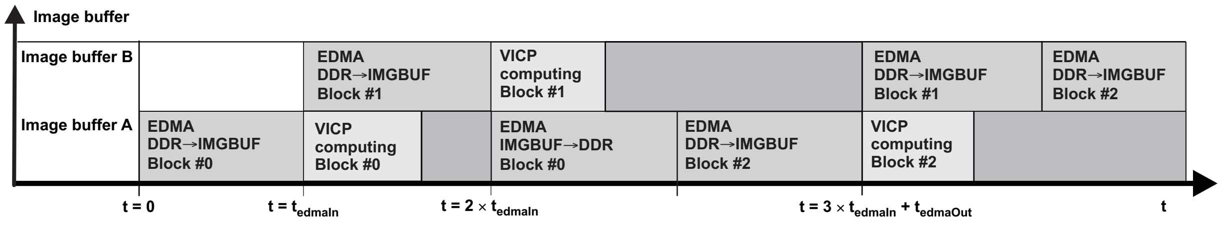
Figure 1-3. Concurrent Processing Schedule Example 1



You can observe that the EDMA3 transfers are completely hidden behind the VICP processing, resulting in an execution time for one block of t = t_{VICP}. This is generally the case since VICP processing is often more computational intensive than EDMA3 transfer. However, for a very simple algorithm, which executes only one or two VICP commands, the EDMA3 transfer time can exceed the processing time. In such scenario the VICP resource is under utilized, which leaves room for extra feature to be implemented into the processing chain without increasing the total processing time.

Figure 1-4 illustrates a case where VICP processing time is smaller than transfer time.

Figure 1-4. Concurrent Processing Schedule Example 2



This document explains how to assess whether the VICP resource is under-utilized respective to the EDMA transfer and how to chain multiple computation tasks to improve the VICP utilization.

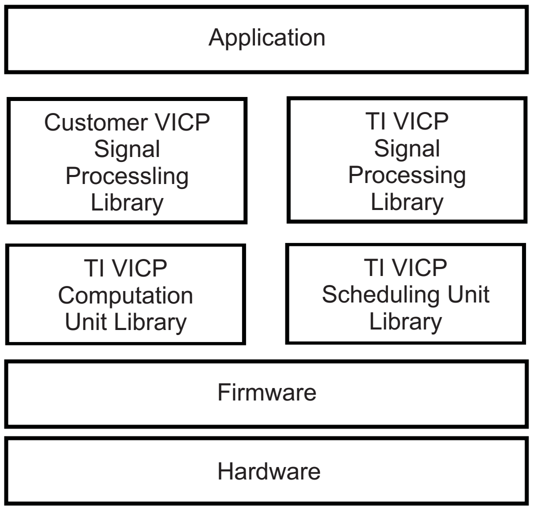
Between each VICP computation and transfer, there exists some small overhead, not represented on the graphs. The source of overhead comes from triggering the VICP computation unit and the EDMA3 transfers, keeping track of block counters, updating the EDMA3 transfer parameters at each block or at the end of the row. However, this overhead is limited and does not exceed 10% of the total processing time.

In conclusion, the programmer can rely on the VICP scheduling unit's library to parallelize the actual algorithm processing with data transfers while keeping control overhead to the minimum.

1.4 Software Infrastructure

Most of the programming and controlling of the VICP's hardware components is abstracted from the application by software libraries provided by TI. The software stack is represented in Figure 1-5.

Figure 1-5. Software Stack


Application
Customer VICP Signal Processling Library
TI VICP Signal Processing Library
TI VICP Computation Unit Library
TI VICP Scheduling Unit Library
Firmware
Hardware

As one moves from the application layer to the hardware layer, the level of firmware abstraction decreases:

- **Application Layer**

  The application layer implements an algorithm through calls to functions belonging to TI's or a custom's VICP signal processing library. At this level, the user of the functions does not need to have any knowledge of the VICP accelerator. Each function processes an entire frame. Usage of the VICP signal processing library is documented at: http://focus.ti.com/lit/ug/sprugj3a/sprugj3a.pdf
- **VICP signal processing library layer**

  This layer implements typical signal processing/mathematical functions on the VICP. In release v3.0 of the library, 23 functions have been implemented. TI will continue adding new functions based on customer's needs. This layer makes use of both the VICP computation unit and scheduling unit libraries to implement functions that process full data frames.
- **VICP computation unit library**

  It is a collection of functions that implement a variety of computation tasks on the VICP computation unit: filtering, matrix arithmetic, table lookups, etc . Each computation task is designed to operate on data residing in the VICP memories. Currently it provides more than 30 functions that the programmer can chain together to produce customized algorithms.
- **VICP scheduling unit libraries layer**

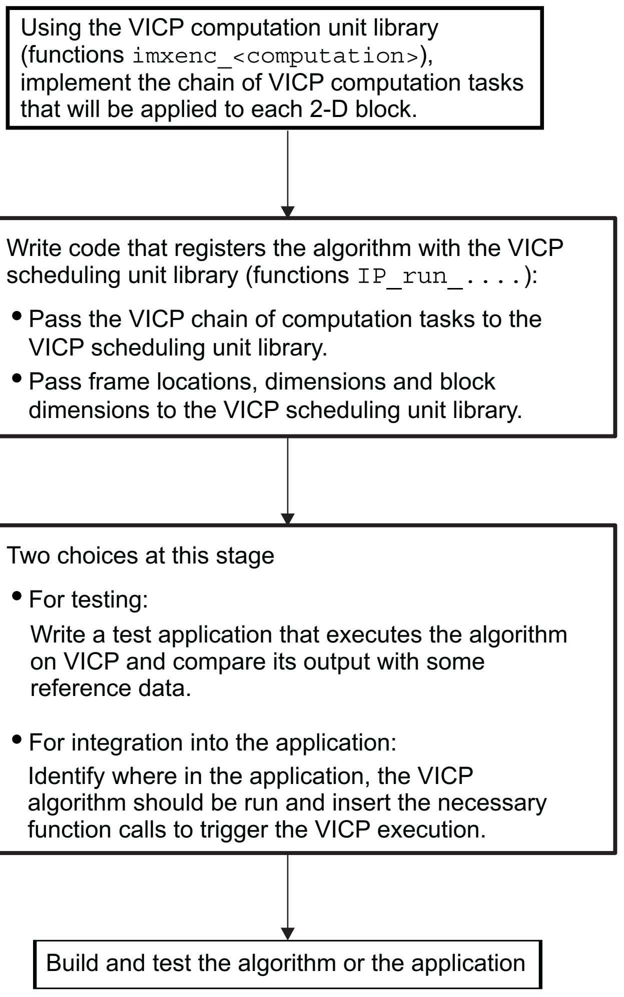
  It provides the infrastructure necessary to apply the VICP chain of computation tasks to each block of the frame. The VICP scheduling unit is programmed such that the DSP is offloaded during the time the VICP chain of computation tasks is applied to each block of the frame.

The VICP computation unit and scheduling unit libraries directly access the VICP firmware. At present, implementation details of these libraries are closed to the broad audience. Their usage will be fully described in the present document in order to allow the creation of custom functions complementing TI' s VICP signal processing library.

1.5 Programming Flow

The programming flow chart for the VICP is depicted in Figure 1-6.

Figure 1-6. Programming Flow Chart


Using the VICP computation unit library (functions imxenc_<computation>), implement the chain of VICP computation tasks that will be applied to each 2-D block.
Write code that registers the algorithm with the VICP scheduling unit library (functions IP_run_....):
• Pass the VICP chain of computation tasks to the VICP scheduling unit library.
• Pass frame locations, dimensions and block dimensions to the VICP scheduling unit library.
Two choices at this stage
• For testing:
Write a test application that executes the algorithm on VICP and compare its output with some reference data.
• For integration into the application:
Identify where in the application, the VICP algorithm should be run and insert the necessary function calls to trigger the VICP execution.
Build and test the algorithm or the application

1.6 Where to Obtain the Software Components

Download the VICP signal processing library at: http://focus.ti.com/docs/toolsw/folders/print/sprc831.html

It will contain all the source code and libraries that are necessary to create custom functions. After installation, the libraries will be located in install_dir/lib/dm648 and install_dir/lib/dm6446. The source files to implement the VICP signal processing library are located in install_dir/src/src_hw and can be used as sample code or as a starting point for custom functions. Header files are located in install_dir/src/src_hw/inc.

All libraries were built using code gen tools v6.1.8 with optimization –o3 enabled, without debug symbols, and tested in applications using DSP/BIOS v5.33.

VICP Computation Unit Library

The VICP computation unit library is a collection of functions that generate sequence of one or several VICP commands. Each command describes a computation task that will be executed on the 8-MACs VICP computation unit. The data being processed must reside locally in the VICP attached memories. The data movement between external memory such as DDR/L2 and local memories is handled by the VICP scheduling unit's library and will be discussed in a separate chapter.

2.1 Library and Header Files

The library filename is imx648.lib for DM647/8 platforms or imx644x.lib for DM6441/6 platforms. The associated header files are vicp_comp.h and vicp_support.h .

2.2 Dependencies With Other Libraries

The library requires another library dmcsl648_bios.lib or dmcsl644x_bios.lib to be linked with the application.

2.3 Commands Encoding

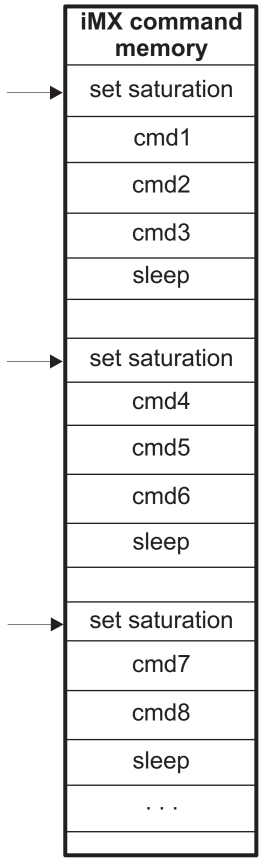
VICP commands are generated at run time by functions of the VICP computation unit library. Each function encodes/generates one or more commands associated to a particular computation task. A function name is in the form imxenc_<computation> where the <computation> substitutes the description of the particular task whose command is generated. For instance imxenc_filter() will encode a command that will perform filter operation if executed by the VICP.

Commands are written into the region pointed by the command pointer which is an input argument of imxenc_<computation>. The number of command words written by the function is returned to the application. Each command can be made of up to 27 16-bits words and a function can generate multiple commands. Once the VICP computation unit is started, it will decode these commands and perform the desired computation task.

A chain of computation tasks is created by calling several imxenc_<computation> functions one after another in order to generate a sequence of VICP commands.

The execution of a VICP command sequence results in the execution of the associated computation tasks. A VICP command sequence usually starts with a saturation command generated by imxenc_set_saturation(). It sets the saturation bounds of all subsequent computation commands. A VICP command sequence always finishes with a sleep command encoded by imxenc_sleep(). Figure 2-1 illustrates the case where three sequences have been encoded into the command memory.

Figure 2-1. Example of VICP Command Memory Filled with Multiple Command Sequences



Each function of the library is described in Chapter 6. This chapter provides a general description of the functions.

2.4 General Syntax of imxenc_<computation> Function

All the imxenc_<computation> functions share a more or less similar interface:

```
cmdlen = imxenc_<computation>(
        input1_ptr,
        input2_ptr,
        output_ptr,
        input1_width,
        input1_height,
        input2_width,
        input2_height,
        output_width,
        output_height,
        computation_width,
        computation_height,
        input1_type,
        input2_type,
        output_type,
        rightshift,
        operation,
        cmdptr,
        );
```

Here is a description of the input arguments:

- *input1_ptr*, *input2_ptr*, *output_ptr*:

  The input pointers specify where the two input and output operands reside in memory. They can either point in image buffer or coefficient memory. The symbols for the base addresses of the image buffer and coefficient memory are IMGBUF_BASE and COEFFBUF_BASE and are defined in file vicp_support.h. When data is in an image buffer, the programmer does not need to specify which one (A or B) is used. The addresses `ptr= (IMGBUF_BASE + ofst)`, `ptr= (IMGBUFA_BASE + ofst)`, `ptr= (IMGBUFB_BASE + ofst)` are all treated equally by the imxenc_<computation> function: they all mean that the pointed data is located at *ofst* bytes from the beginning of an image buffer. Indeed, some pointer manipulation in the imxenc_<computation> function actually discards the base address part of ptr and only retains the offset part ofst.

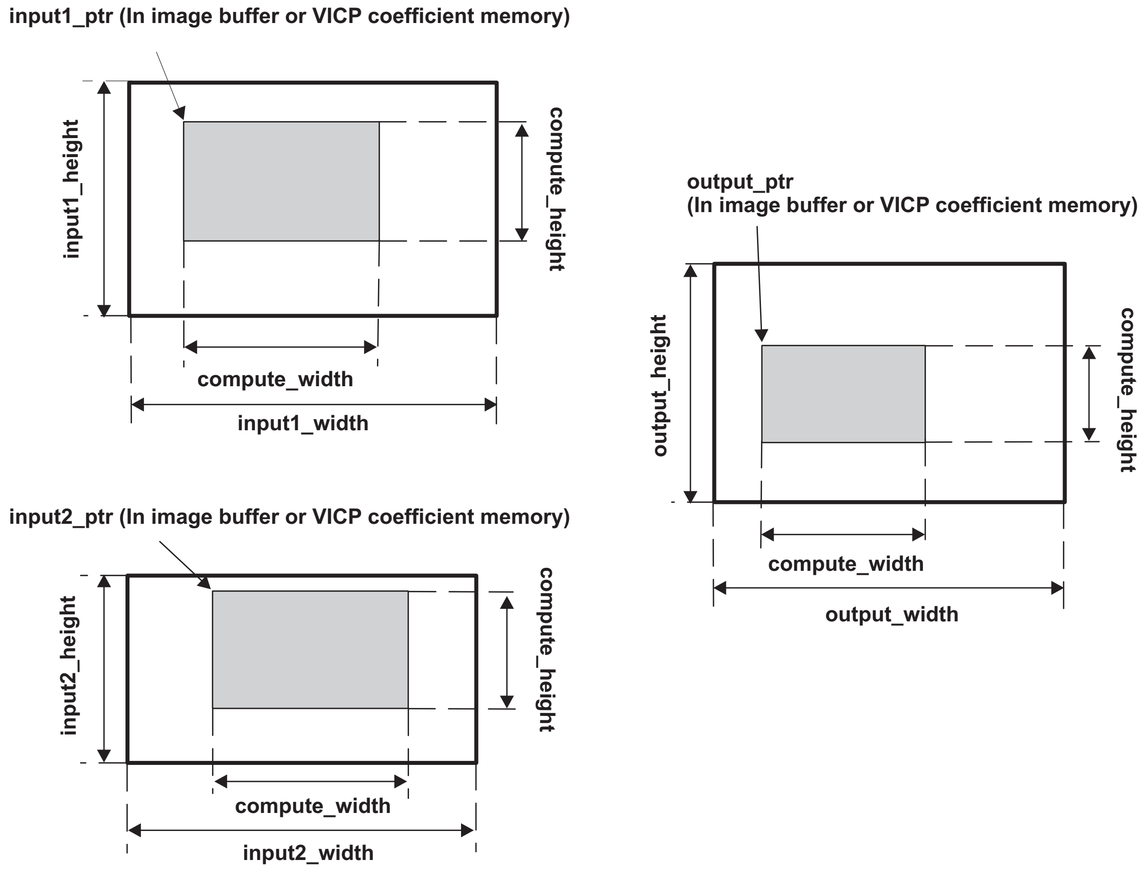
  Consequently nowhere in the code, one needs to specify which image buffer the command sequence is supposed to operate on. Then how can the programmer implement ping-pong buffering ? Actually ping-pong buffering between image buffer A and B is handled by the VICP scheduling unit's library during the processing.

- *input1_width*, *input1_height*, *input2_width*, *input2_height*, *output_width*, *output_height*, *computation_width*, *computation_height*:

  The width and height arguments describe the dimensions of the block of data to be processed. The unit is in number of elements. An element can be a 8-bits byte or 16-bits word, depending on the values of the input1_type, input2_type, output_type arguments.

  Figure 2-2 depicts the layout of both input block and output block labeled with the different arguments passed to the imxenc_<computation> function.

Figure 2-2. Layout of Input and Output Data



The arguments input1_height, input2_height, output_height are actually useless most of the time since the height of the region to be processed can by characterized with the parameter compute_height alone. They are listed as arguments for consistency purpose. However, input1_width, input2_with, output_width are necessary since they characterize the stride for each category of data.

- *input1_type*, *input2_type*:

  The input types describe the types of the elements making up the input data blocks. Use these symbols defined in vicp_comp.h as argument values:

  - IMXTYPE_SHORT (signed 16 bit)
  - IMXTYPE_USHORT (unsigned 16 bit)
  - IMXTYPE_BYTE (signed 8 bit)
  - IMXTYPE_UBYTE (unsigned 8 bit)

  VICP treats IMXTYPE_SHORT and IMXTYPE_USHORT in little endian format. For example 0x0E71, 0xA145 will be represented internally as the sequence of bytes 0x71, 0x0E, 0x45, 0xA1 with increasing addresses.

  There is no performance speed up in processing byte vs 16-bits elements. Computationally-wise, the VICP computation unit treats both types equally. However, storage and EDMA bandwidth is halved whenever byte type is used. For algorithms whose performance is bounded by EDMA transfers, it is better for the VICP to process byte types whenever possible.

- *output_type* can be:
  - IMXOTYPE_SHORT
  - IMXOTYPE_BYTE

  The symbols for output_type differ from those used for input_type by one letter O. Do not try to use IMXTYPE_<…> symbol for an output_type, it will lead to unexpected behavior during the execution of the command. Use IMXOTYPE_<…> instead. Observe also that signed/unsigned distinction is not needed to specify an output type. The VICP computation unit only needs to know the size of the element in order to store it in memory under the correct format. Signed/unsigned distinction is only needed for the input elements for computation purpose.
- *rightshift*:

  Number of bits to right shift before outputting the value. Internally the computation unit has a 32-bits accumulator but only the 16 LSBs are written out into the memory. If inputs are IMXTYPE_SHORT or IMXTYPE_USHORT and multiply operations are carried out then right shift may be required to obtain the most significant bits of the output. Right shift can be up to 31bits. One advantage is that right shift is a free operation that does not incur any cycles on the VICP computation unit, in addition to the computation cycles. The results of the right shift is also rounded, see rounding section later.
- *operation*:

  This parameter is not available for all computation functions. When used *operation* can have the following values:
  - IMXOP_MPY: multiply
  - IMXOP_ABDF: absolute difference
  - IMXOP_ADD: addition
  - IMXOP_SUB: subtraction
  - IMXOP_AND: logical and
  - IMXOP_OR: logical or
  - IMXOP_XOR: logical XOR
  - IMXOP_MIN, IMXOP_MAX: min, max.
- *cmdptr*:

  Points to the location in memory where the command words will be generated by the function. In general it points to the command memory. The programmer can set it to point to other memory but then the generated command sequence will have to be copied to the command memory before it is executed.

The function returns the length in 16-bit words of the generated command sequence.

Some computation function may require pre-initializing the VICP coefficient memory with some constants. For instance, if the operation is filtering, the filter coefficients must be written by the application into the coefficient memory and one of the input pointers must point to them. The application should never put constants in an image buffer since it is used for I/O and its content is getting overwritten every data block transferred by the EDMA3.

2.5 Set Saturation Function

The set saturation function generates the command that sets the saturation boundaries for all computation commands that are subsequently encoded.

The prototype of the function is:

```
(Int16) imxenc_set_saturation(
    Int32 sat_high,
    Int32 sat_high_set,
    Int32 sat_low,
    Int32 sat_low_set,
    (Int16*)cmdptr);
```

For each output value generated by a command, it implements:

```
If (output >= sat_high)
         output= sat_high_set;
else if (output < sat_low)
         output= sat_low_set;
```

Example: Set saturation bounds between 0 and 65535:

```
imxenc_set_saturation(65535, 65535, 0, 0, cmdptr);
```

Set saturation bounds between -128 and 127:

```
imxenc_set_saturation(127, 127, -128, -128, cmdptr);
```

NOTE: For table lookup types of operation implemented by the functions imxenc_table_lookup(), imxenc_tables_lookup(), imxenc_table_lookup2D(), imxenc_3d_table_lookup(), imxenc_table_lookup_int(), imxenc_table_lookup_32bit(), the saturation is applied to the input data before the lookup. In case of lookup operation, input data represents the index of the elements to lookup.

This saturation operation comes for free and does not add any extra cycles to the computation operation.

It is possible to disable saturation by calling the function IMX_setSat(IMX_SAT_NO). After such a call, all subsequent functions encoded will have saturation disabled.

2.6 Sleep Command

The sleep command must be inserted at the end of a command sequence to terminate the execution of the chain of computation tasks.

The function call is:

```
cmdlen = imxenc_sleep(cmdptr);
```

2.7 Rounding

When the *rightshift* argument of a function is non zero, rounding is performed. For instance if the output of the computation is 15 and rightshift parameter is 2 then the result will be round((15>>2))= round(3.75)= 4. If the output is -15 then the result after rounding will be -4 . To disable rounding, call the function IMX_setRound(IMX_ROUND_NO). After such a call, rounding will be disabled for all subsequent functions encoded and instead truncating will be in effect.

2.8 Memory Switching and Cache

When the application chooses to generate the commands directly into the VICP command memory, the buffer switch must be configured correctly to grant access to the DSP by calling IMGBUF_switch() as follow: IMGBUF_switch(SELCMDBUF, CMDBUFAUTO) .

Likewise, to initialize the coefficient memory with constant values such as filter coefficients, its access must be granted to DSP with the call: IMGBUF_switch(SELCOEFBUF, COEFBUFAUTO) .

A shortcut is to call the function to switch both buffers to DSP as follow:

```
IMGBUF_switch(SELCMDBUF | SELCOEFBUF, CMDBUFAUTO | COEFBUFAUTO)
```

See Chapter 5 for details on the IMGBUF_switch() function.

All VICP memories are cached by the DSP so after writing into the VICP coefficient and command memories, cache write back functions must be called in order to flush the cached data into the physical memory. If the cache is not properly flushed then there is a possibility that the VICP runs the incorrect command sequence with incorrect data.

2.9 Constraints

The VICP computation unit design poses some constraints on the buffer and data organization:

- The width of the computation area should be strictly inferior to 2048 elements.
- The height of the computation area should be strictly inferior to 256 rows. Some functions such as imxenc_transpose() have this constraint lifted. In this case mention is made in the function's documentation.
- APIs involving matrix multiplication (such as DCT) can not operate in place; input and output must be at different locations. All other APIs may or may not work in place depending on the size of input and output array.

2.10 Performance Estimation

The amount of VICP cycles to execute a computation task generated by an imxenc_<computation> function is given by:

amount_of_work × *memory_conflict_factor* / *speedup_factor*

The amount of work and memory conflict factor are found in tables for each documented API. The amount of work is a function of the complexity of the computation task carried out by the VICP. The memory conflict factor is a function of the locations of the inputs and output pointers.

The speedup factor depends generally on the value of the compute_width argument passed to the API imxenc_<computation> function. If compute_width is a multiple of 8 then the speedup factor is the maximum value of 8. In each API description, a table listing the different speed up factors for different values of compute_width is given.

As a rule of thumb, to obtain optimum performance, at least two of the pointers should point to the coefficient memory. For instance, if all pointers point to image buffers then the conflict factor becomes 3 for imxenc_array_op(). If two pointers are in coefficient memory and one pointer in image buffer then the conflict factor becomes 1.

The total execution cycles of a chain of VICP computation task can be obtained by summing up all computation task's execution times and adding an overhead of 30 VICP cycles for pipelining. In general real-world benchmarks track pretty close to the theoretical estimates obtained with this method.

There is a special execution mode called ASAP (as soon as possible) mode, in which each back-to-back computation task starts a little bit earlier than the completion of the previous task. If that mode is enabled, the real-world benchmarks actually beat the theoretical estimates produced by the method described above. However, in some cases, results can be wrong in ASAP mode. In particular when a computation task needs to first process the last outputs of the previous task, the results can be wrong. In other words, when the read direction of the computation task is opposite the write direction of the previous task, results can be wrong.

The ASAP mode is disabled by default and can be enabled by calling the function IMX_setASAP(IMX_ASAP_ENABLE) causing subsequent computation tasks encoded to run in ASAP mode. It is advised to leave the ASAP mode disabled when developing VICP code. Only when the algorithm is fully tested and if the performance needs a further boost, one can enable it.

2.11 Example

This example contains several imxenc_<computation> functions to encode two sequences directly into the VICP command memory:

```
#include "vicp_comp.h"
#include "vicp_support.h"


/* Make sure the command and coefficient memories
are switched to DSP before writing to them. */
IMGBUF_switch(SELCMDBUF|SELCOEFBUF, CMDBUFAUTO|COEFFBUFAUTO);

/* ----- sequence 1 ----- */
cmd_ptr = (short *)CMDBUF_BASE;

cmd_len  = imxenc_set_saturation(...., cmd_ptr);
cmd_len += imxenc_filter(....,cmd_ptr+ cmdlen);
cmd_len += imxenc_color_spc_conv(....,cmd_ptr+ cmdlen);
cmd_len += imxenc_sleep(....,cmd_ptr+ cmdlen);

/* ----- sequence 2 ----- */
cmd_len += imxenc_set_saturation(...., cmd_ptr+cmdlen);
cmd_len += imxenc_table_lookup(...., cmd_ptr+cmdlen);
cmd_len += imxenc_filter(...., cmd_ptr+cmdlen);
cmd_len += imxenc_sleep(...., cmd_ptr+cmdlen);
```

2.12 Applying a VICP Command Sequence to an Entire Data Frame Using the TI VICP Scheduling Unit's Library

The recommended way to apply a VICP command sequence over a data frame is to use the VICP scheduling unit's library. The advantages are two folds:

- Performance advantage: the library uses a hardware scheduling unit different than the DSP to control the VICP computation unit and trigger the EDMA3. This totally frees up the DSP.
- Easier code development and better portability: since the library abstracts the controlling of the EDMA3, the scheduling and the computation units, the implementer of the algorithm just needs to focus on designing the per-block chain of computation tasks. The library automatically transfers the data and applies the same sequence of VICP commands to each block of the frame.

2.13 Applying a VICP Command Sequence to an Entire Data Frame Using DSP Code

In some rare cases, the programmer may want to write his/her own DSP code to control the VICP computation unit. To achieve that, he/she first needs to know how to apply a VICP command sequence to only a block of data in image buffer A or B. The different steps are: write input data into the desired image buffer, configure the buffer switch crossbar to grant VICP access to the targeted image buffer, start the VICP computation unit and wait for its completion. When one image buffer is switched to VICP, the other buffer is automatically switched to the DSP/EDMA3 data bus. This is how ping-pong buffering can be implemented between VICP and DSP/EDMA3.

The following C code illustrates the usage of the functions IMGBUF_switch(), IMX_start(), IMX_wait() in order to apply VICP computation to all data blocks constituting a frame:

```
/* Switch command and coefficient buffers to automatic switching mode in which the access is
   automatically switched whenever VICP becomes active. If VICP idle, DSP has access. */
IMGBUF_switch(SELCMDBUF|SELCOEFFBUF, CMDBUFAUTO|COEFFBUFAUTO);

/* Call the application function that encodes the VICP commmands */
encodeVICPcommands(cmd_ptr);

/* Switch image buffer A to DSP/EDMA3 */
IMGBUF_switch(SELIMGBUFA, IMGBUFADSP);

/* Fill image buffer A with first block of data */
dspEdmaFillImgA();

/* With this loop, remaining blocks are covered */
for(blockCount= 0; blockCount < totalNumBlocks; blockCount+=2) {

        /********   VICP processes image buffer A content   *********/

        /* Switch image buffer A to VICP, note image buffer B automatically
           switched to DSP/EDMA3 */
        IMGBUF_switch(SELIMGBUFA, IMGBUFAVICP);

        /* Start VICP command sequence pointed by cmd_ptr */
        IMX_start(cmd_ptr);

        /* In the meantime, DSP can do further processing in previous block residing
           in image buffer B, empty image buffer B and then refill with new data */
        if (blockCount)
          dspProcAndEdmaEmptyImgB();

        dspProcAndEdmaFillImgB();

        /* Wait for VICP completion */
        IMX_wait();

        /********   VICP processes image buffer B content   *********/

        /* Switch image buffer B to VICP, note image buffer A
        automatically switched to DSP/EDMA3 */
        IMGBUF_switch(SELIMGBUFB, IMGBUFBVICP);

        /* Start VICP command sequence pointed by cmd_ptr */
        IMX_start(cmd_ptr);

        /* In the meantime, DSP can do further processing in previous block residing
           in image buffer A, empty image buffer A and then refill with new data */
        dspProcAndEdmaEmptyImgA();
        dspProcAndEdmaFillImgA();

        /* Wait for VICP completion */
        IMX_wait();
}

/* flush processed data in image buffer B */
IMGBUF_switch(SELIMGBUFB, IMGBUFBVICP);
dspProcAndEdmaEmptyImgB();
```

VICP Scheduling Unit Library

The VICP scheduling unit library provides simple functions that enable an entire frame to be processed by the chain of computation tasks previously implemented on the VICP. The same chain of computation tasks is applied to each block of the frame.

Under the hood, the library programs the VICP scheduling unit to control the computation unit and the EDMA3 hardware. The major performance benefit is that the DSP is offloaded during the entire time the frame is being processed by the VICP.

3.1 Library and Header files

The libraries are IP_run648 or IP_run644x.lib . The associated header file is vicp_sch.lib.

All the names of the functions in the library start with the prefix IP_run.

3.2 Dependencies With Other Libraries/Modules

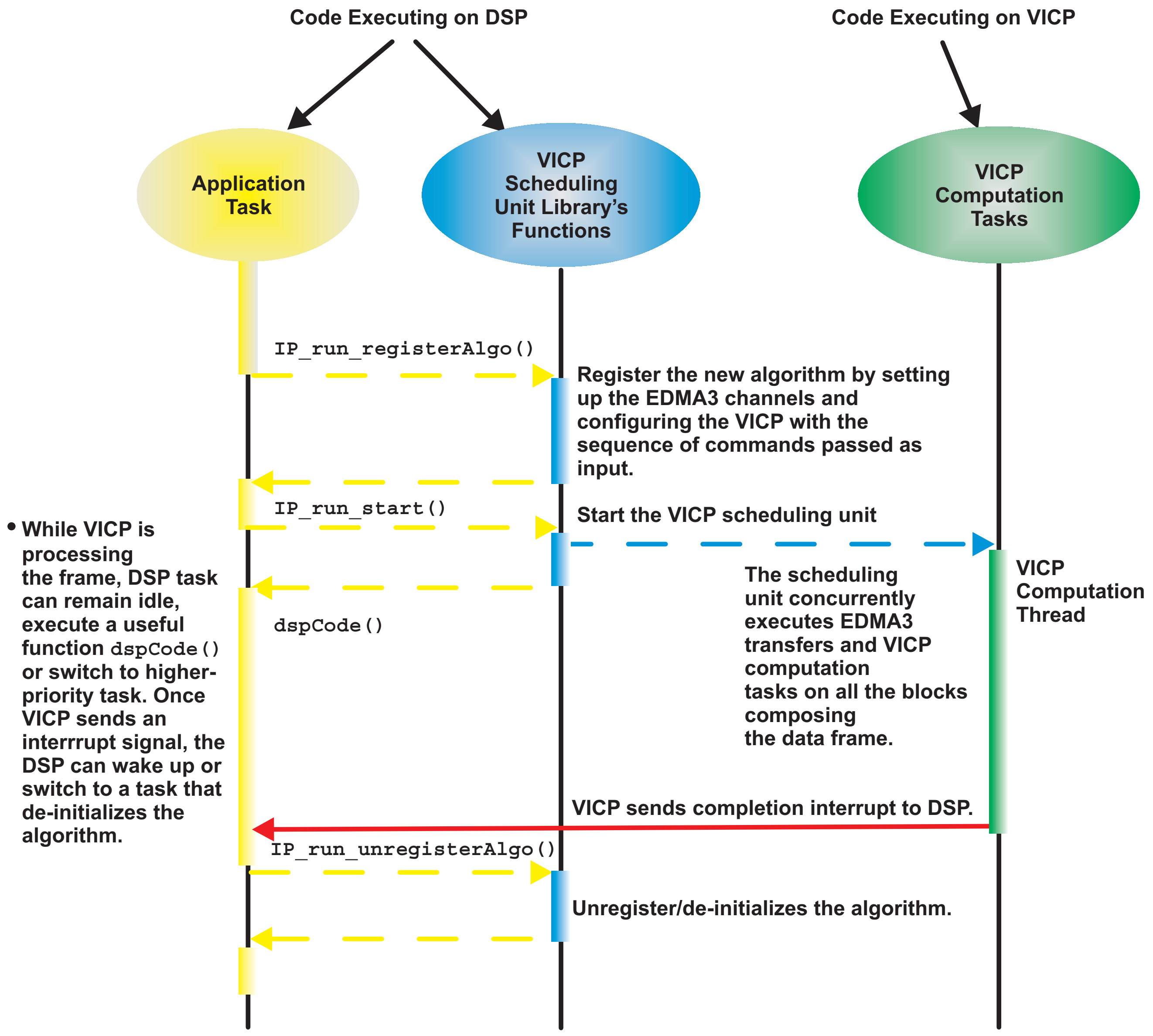
The library requires dmcsl648_bios.lib or dmcsl644x_bios.lib be linked with the application.

The library requires that the EDMA3 LLD (low-level driver) be installed. The release was tested with the release version 1_05_00_01, which can be downloaded at: https://www-a.ti.com/downloads/sds_support/targetcontent/psp/edma3_lld/edma3_lld_1_05/index.html. Add the system environment variable TI_EDMA3LLD to point to [*INSTALL_DIR*]\edma3_lld_1_05_00\packages. Where *INSTALL_DIR* is the directory where the EDMA3 LLD software is installed.

3.3 Usage Scenarios

Figure 3-1 shows how the DSP application task, the VICP scheduling unit's library's functions and the VICP computation tasks are interleaved when an algorithm is executed.

Figure 3-1. VICP Processing Diagram


Code Executing on DSP
Code Executing on VICP
Application Task
VICP Scheduling Unit Library's Functions
VICP Computation Tasks
IP_run_registerAlgo()
Register the new algorithm by setting up the EDMA3 channels and configuring the VICP with the sequence of commands passed as input.
IP_run_start()
Start the VICP scheduling unit
• While VICP is processing the frame, DSP task can remain idle, execute a useful function dspCode() or switch to higher-priority task. Once VICP sends an interrrupt signal, the DSP can wake up or switch to a task that de-initializes the algorithm.
dspCode()
The scheduling unit concurrently executes EDMA3 transfers and VICP computation tasks on all the blocks composing the data frame.
VICP Computation Thread
VICP sends completion interrupt to DSP.
IP_run_unregisterAlgo()
Unregister/de-initializes the algorithm.

The algorithm developer is responsible of writing a DSP task that will make calls to the VICP scheduling library to: register the algorithm, execute the algorithm and unregister the algorithm.

Since none of the VICP scheduling unit library's functions is re-entrant, one and only one DSP task should make a call to the library at any given time. If that is not the case, a resource sharing method such as semaphore must be used.

Registering an algorithm is done by calling IP_run_registerAlgo(). Input arguments to this function include the VICP command sequence location and EDMA3 transfer parameters.

After registration, the application calls IP_run_start() to apply the computation command sequence to every block of the frame. The scheduling unit controls the EDMA3 transfers and the VICP so the whole frame gets processed. From now on, the term VICP computation thread is used to characterize the execution of the VICP computation tasks on every block until the end of the frame.

While the VICP computation thread is executing, the DSP is free and can do other useful processing. Once the whole frame is processed, the VICP sends an interrupt to the DSP.

In Figure 3-1, the DSP synchronizes with the VICP by way of an interrupt. This is an asynchronous type of synchronization. Busy polling or synchronous type can also be achieved by calling IP_RUN_wait() but is less efficient since it ties up DSP resource. Only exception for busy polling usage, is when the execution time of dspCode() is longer than the VICP processing. In this case, IP_RUN_wait() returns immediately.

The sequence of actions depicted in Figure 3-1 could be implemented by the following pseudo C code:

```
IP_RUN_registerAlgo ()
IP_RUN_start() /* start VICP computation thread of algorithm*/
dspCode()
IP_RUN_wait() /* OK to use if dspCode() exec time > VICP exec time */
IP_RUN_unregisterAlgo ()
```

Another usage is that once the algorithm is registered, the associated computation thread can be executed an infinite number of times as long as IP_RUN_resetAlgo() is called before IP_RUN_start():

```
IP_RUN_registerAlgo()

/* Execute the VICP thread indefinitely until a certain event happens */
while(!stopEvent){
  IP_RUN_resetAlgo()
  IP_RUN_start()
  dspCode()
  IP_RUN_wait()
}

IP_RUN_unregisterAlgo()
```

IP_RUN_resetAlgo() can accept new values for the addresses of the input and output frames as well for the VICP command sequence location. Its cycles count is between 6,500 and 9,000 DSP cycles depending on the number of input and output channels required by the algorithm.

Now, how is the case where the application needs to execute multiple algorithms back to back implemented?

If the algorithms do not differ in frame dimensions and block processing sizes then IP_RUN_resetAlgo() should be used since it has the capability of resetting the VICP command sequence location.

For other cases, the previous algorithm must be unregistered and the new algorithm registered, before the new algorithm can be executed.

The following pseudo-code illustrates such as situation where N algorithms are ran back to back:

```
while (!stopEvent) {
      IP_RUN_registerAlgo() for algo #0
      IP_RUN_start() for algo #0
      dspCode0()
      IP_RUN_wait()for algo #0
      IP_RUN_unregisterAlgo() for algo #0
      IP_RUN_registerAlgo () for algo #1
      IP_RUN_start() for algo #1
      dspCode1()
      IP_RUN_wait()for algo #1
      IP_RUN_unregisterAlgo() for algo #1
      ....
      IP_RUN_registerAlgo() for algo #N
      IP_RUN_start() for algo #N
      dspCodeN()
      IP_RUN_wait()for algo #N
      IP_RUN_unregisterAlgo() for algo #N
}
```

Since each IP_RUN_registerAlgo() call can consume over 40,000 DSP cycles, it can become the bottleneck in case the computation threads are short in execution time relative to the execution time of IP_RUN_registerAlgo(). That could really lead to some inefficiency in processing time.

One should try to minimize the number of computation threads that an application must execute. The goal of the algorithm designer should be to implement computation threads that apply as many computation tasks as possible on every block of the frame. If that goal is kept in mind then the application will execute long VICP computation threads just a few times instead of fragmented VICP computation threads many times.

For instance if the application needs to execute a 5x5 filter followed by a 3x3 filter. Instead of implementing two separate computation threads, the algorithm designer should create one computation thread that applies both filtering on every block of the frame.

Of course sometimes, it is just not possible to merge two computation threads into one. For instance imagine an algorithm that needs to normalize a set of values, using the maximum value as normalization factor. This algorithm can only be implemented in two computation threads: the first computation thread has to look for the maximum value over the entire set of values and the second thread multiplies each value of the set with the reciprocal of the maximum value found by the first thread.

Still, the VICP scheduling unit library provides a mechanism to reduce the overhead between two computation threads. The library actually allows having up to two algorithms registered in the system at the same time. IP_RUN_registerAlgo() just needs to be called two times in a row. Another feature that comes along is that if two computation threads are registered in the system, the second thread is executed automatically after the first thread without being explicitly started with IP_RUN_start().

The pseudo-code for handling two computation threads can be as follow:

```
IP_RUN_registerAlgo(0) /* register algo #0 */
IP_RUN_registerAlgo(1) /* register algo #1 */


while (!stopEvent) {
      IP_RUN_resetAlgo()
      IP_RUN_start() /* start threads of both algo #0 and algo #1 */
      dspCode0()
      dspCode1()
      IP_RUN_wait() /* wait for algo #0 and algo #1 */
}

IP_RUN_unRegisterAlgo(1) /* unregister algo #1 */
IP_RUN_unRegisterAlgo(0) /* unregister algo #0 */
```

3.4 Performance

The execution time of a VICP computation thread can be calculated by the following formula:

$$T_{vicp\_thread} = num\_blocks \times max(T_{vicp\_tasks}, T_{data\_transfer}) + 10\%\ overhead \quad (1)$$

Where:

- T_{vicp_tasks} is obtained using the method described in Section 2.10.
- $T_{data\_transfer}$ is the number of VICP cycles spent in data transfers per block. Sometimes the algorithm requires reading the equivalent of 2 input data blocks because 2 input operands are needed (ex: array operation). Table 3-1 gives both read and write throughput for DDR2↔IMGBUF case on DM648. Measurements were made for a VICP at 445 Mhz and DDR at 266 Mhz. If the system being used has a different speed ratio, the table must be adjusted accordingly. Also L2↔IMGBUF might lead to better data throughput.

Table 3-1. Data Transfer Throughput on DM648

SRC→DEST	Amount of Bytes Transferred	VICP Cycles/Byte
DDR→IMGBUF	4096	0.32
IMGBUF→DDR	4096	0.48
DDR→IMGBUF	8192	0.29
IMGBUF→DDR	8192	0.43

3.5 Data Cache Handling

The VICP scheduling unit needs to touch some undocumented VICP memories for its internal functioning. Since the DSP caches all VICP memories, the cache must be written back whenever a library's function writes into them. To make the implementation of the VICP scheduling unit library platform independent, some cache management functions must be passed by the application.

To pass such functions, the following code must be inserted before calling IP_RUN_init():

```
CACHE_InitPrm_t cachePrm;

cachePrm.wbCb= (CACHE_Cb)Custom_WriteBack;
cachePrm.invCb= (CACHE_Cb)Custom_Invalidate;
cachePrm.wbInvCb= (CACHE_Cb)Custom_WbInv;

CACHE_init(&cachePrm);
```

Where Custom_WriteBack(), Custom_Invalidate(), Custom_WbInv() are implemented by the application with prototype:

```
void CACHE_writeBack(CACHE_Addr_t addr, CACHE_Size_t size);
void CACHE_invalidate(CACHE_Addr_t addr, CACHE_Size_t size);
void CACHE_wbInv(CACHE_Addr_t addr, CACHE_Size_t size);
```

If the application does not do that then the library will not function properly.

3.6 EDMA3 Requirements

The library uses functions from the EDMA3 LLD to request, program, release EDMA3 channels.

The library requirement for EDMA3 resources is device dependent:

- For DM648: 11 channels and 31 param entries
- For DM644x: 9 channels and 29 param entries

The library uses EDMA3 region 2 to allocate the EDMA3 resource. Configuration files provided along with the EDMA3 LLD were customized and compiled into the libraries dmcsl648_bios.lib or dmcsl644x_bios.lib. These files named vicp_edma3_dm648_cfg.c, vicp_edma3_dm644x_cfg.c are also provided in source at [VICP_LIBRARY_INSTALLATION_DIR]\test\src. The modifications were for region 2 usage. The structure vicpInstInitConfig present in the file vicp_edma3_[DEVICE_NAME]_cfg.c contains EDMA configuration for region 2 only and reserves EDMA channels 8-11, 36-63 on DM644x and channels 7-10, 36-63 on DM648. The algorithm integrator is free to add another configuration structure for region 1 (DSP) and change the EDMA partition between region 1 and 2. However, region 2 should have an allocation of at least 11 channels and 31 param entries on DM648 or 9 channels and 29 param entries on DM6446 to ensure full functionality of the VICP signal processing library's functions. If the modified files are included in the application project, they will override the EDMA3 configuration set by the dmcsl648_bios.lib or dmcsl644x_bios.lib .

The function VICP_EDMA3_init() must be called by the application before any call to the VICP scheduling unit library. It basically configures the EDMA3 with the information provided in the configuration files. This function is implemented in dmcsl648_bios.lib or dmcsl644x_bios.lib but its implementation is also provided in source at [VICP_LIBRARY_INSTALLATION_DIR]\test\src\vicp_edma3_support.c . The algorithm integrator is free to modify the file and add it to his/her application project in order to override the original implementation.

3.7 Functions Usage

This paragraph gives a general description of the functions used in the different phases of algorithm registration, execution and unregistration.

3.7.1 VICP Scheduling Unit Initialization

The VICP scheduling unit is initialized by calling IP_RUN_init(). Prior to that, the application must have called VICP_EDMA3_init() and CACHE_init().

This function accepts a structure of type IP_RUN_InitParams as input argument. The description of the structure is given in Section 4.1.1 but it is advised to pass the default settings defined by the global variable IP_RUN_DEFAULT_INIT.

3.7.2 Algorithm Registration

IP_RUN_registerAlgo() must be called in order to register an algorithm implemented in form of a VICP command sequence.

The function accepts two input arguments. The first argument is a pointer to the structure IP_run and the second argument is the index of the computation thread (0 or 1). The definition of IP_run is as follow:

```
typedef struct IP_run{

  DmaTferStruct *dmaIn;
  Uint16 numDmaIn ;

  DmaTferStruct *dmaOut;
  Uint16 numDmaOut ;

  Uint16 numVertBlocks ;
  Uint16 numHorzBlocks;

  Uint32 chunksize ;
  Uint16 compIntEna;
  Uint16 compCode;
  Uint16 cmdptr_ofst ;
  void *extension ;
  void *customerExtension;

} IP_run;
```

dmaIn	Pointer to array of structures DmaTferStruct used to initialize input DMA transfers from DDR to image buffer. The size of the array should match the member numDmaIn.
numDmaIn	Number of input transfers. Maximum value is 4 in this release.
dmaOut	Pointer to array of structures DmaTferStruct used to initialize output DMA transfers from image buffer to DDR. The size of the array should match the member numDmaOut.
numDmaOut	Number of output transfers. Maximum value is 4 in this release.
numHorzBlocks	Number of processing blocks in the horizontal direction.
numVertBlocks	Number of processing blocks in the vertical direction
chunksize	Ignored in the current release.
compIntEna	Completion interrupt enabled. If set to 1 then sequencer sends an interrupt at the completion of the computation thread.
compCode	Completion code. Code returned by IP_RUN_getCompCode() after a computation thread finishes. Has to be non zero. In case of multi thread scenario, this can be used by the application to find out which thread has just completed.
cmdptr_ofst	Number of 16-bits words separating the beginning of the VICP command memory and the starting point of the VICP command sequence. The VICP command sequence is generated by calling imxenc_<computation> functions from the VICP computation unit library.

extension	For future extensions. Ignored for now
customerExtension	For future customer extensions. Ignored for now.

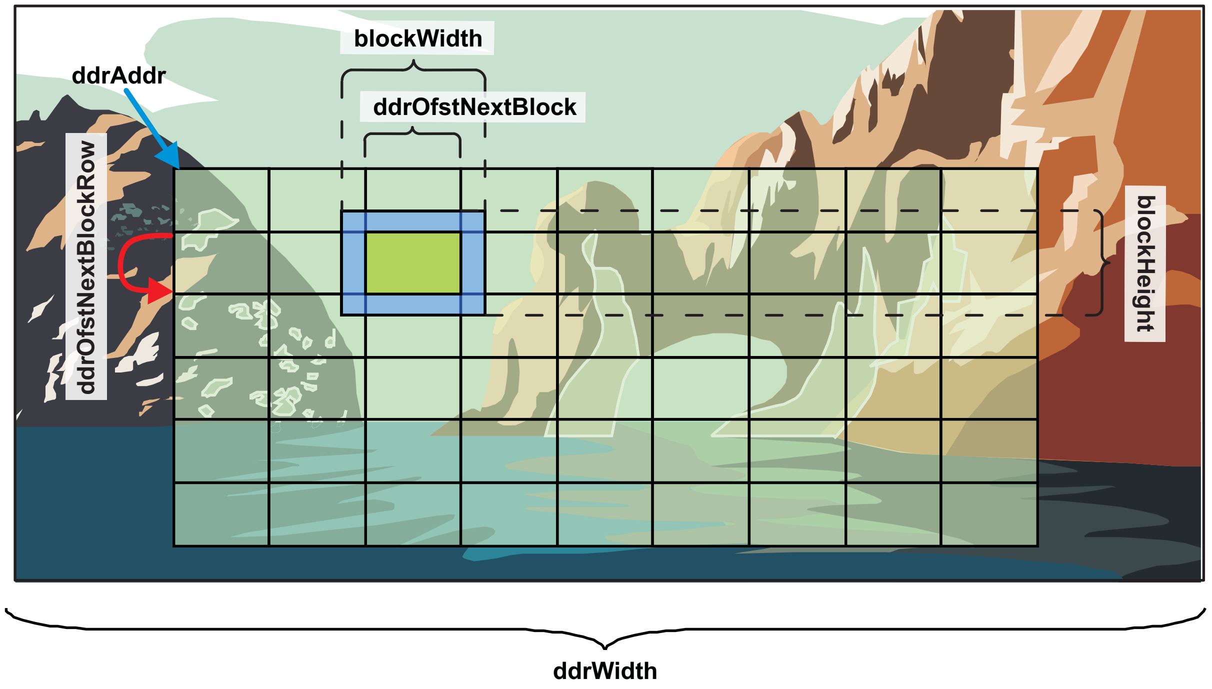
The IP_run structure has two fields of type DmaTferStruct , one to describe input data transfers and one for output. This structure is shown in detail here:

```
typedef struct {

  Uint32 ddrAddr ;
  Int16  ddrWidth ;
  Int16  reserved0 ;
  Int32  ddrOfstNextBlock ;
  Int32  ddrOfstNextBlockRow ;

  Uint32 imgBufAddr ;
  Int16  imgBufWidth ;

  Int16  reserved1 ;

  Uint16 blockWidth ;
  Uint16 blockHeight ;
  Uint16 reserved2 ;

  Uint16 dmaChNo;

}  DmaTferStruct ;
```

ddrAddr	Starting address in DDR or L2 of the input or output data.
ddrWidth	Width in bytes of the data in DDR or L2.
ddrOfstNextBlock	Offset to next block, in bytes
ddrOfstNextBlockRow	Offset to next row of blocks, in bytes. See Figure 3-2.
imgBufAddr	Starting address in image buffer of the block transferred to/from DDR
imgBufWidth	Width in bytes of the data in image buffer
blockWidth	Block width in bytes
blockHeight	Block height in number of pixel rows
dmaChNo	Usually set it to DMAC_CHAN_ANY

Figure 3-2 shows the meaning of each of the above fields in a graphical way. In this example, the green shaded area represents the processing area which is a subset of the 2-D data frame. The argument ddrAddr is set to point to the upper left corner of that area. The argument ddrWidth is generally equal to the whole frame width. The example also assumes that border pixels around a processing block are needed. These border pixels overlap with the adjacent processing blocks. That is always the case when the algorithm implements some FIR filtering operations. If border pixels are present then blockWidth is larger than ddrOfstNextBlock. Otherwise they are equal. Likewise if border pixels are present then blockHeight*ddrWidth is larger then ddrOfstNextBlockRow.

Figure 3-2. Depiction of DmaTferStruct's Members (Figure 10)



In case the chain of computation tasks contain N filtering operations, with filter size being $H_i \times V_i$ then blockWidth must be equal to $procBlockWidth + \sum_{i=0}^{N-1}(H_i - 1)$, where procBlockWidth is the width of the processing data block (the green block in Figure 3-2) and is equal to ddrOfstNextBlock. Likewise blockHeight should be $procBlockHeight + \sum_{i=0}^{N-1}(V_i - 1)$.

For instance if 3x3 filter is applied followed by a 5x5 filter on 8x4 blocks, then blockWidth should be 8 + (2 + 4) = 14 and blockHeight should be 4 + (2 + 4)= 10 . The argument ddrOfstNextBlock would be equal to 8.

3.7.3 Algorithm Execution

3.7.3.1 Starting the Processing

VICP computation threads are started by calling IP_RUN_start(). The function returns to the caller right after starting the VICP scheduling unit, leaving it in an execution state. This frees up the DSP who can spend its MIPS in other tasks than controlling the VICP.

3.7.3.2 Waiting for Completion

There are two ways the DSP can synchronize with the VICP after IP_RUN_start() is called:

- Busy waiting: by calling IP_RUN_wait(), the DSP waits for the completion of the computation thread. This is of course the least efficient way of synchronization since the DSP is wasting MIPS pending for other processors to finish. The effect can be mitigated by calling IP_RUN_wait() in a low priority task inside a multi-task OS environment but this is not the ideal solution. One case the usage of IP_RUN_wait() is acceptable, is when between IP_RUN_start() and IP_RUN_wait(), DSP is doing some useful processing, whose execution time is longer than the VICP thread's.

- Use of interrupt: the most efficient synchronization method is to enable the interrupt notification feature of the VICP scheduling unit by setting the member compIntEna of the algorithm's IP_run structure to 1. This allows the VICP scheduling unit to send an interrupt to the DSP once the frame computation ends. The DSP must implement a specific ISR to intercept the interrupt. The implementation of the ISR is left up to the application. Usually it sets some semaphore on which another task is pending on.

3.7.3.3 Restarting Processing

If needed, the application can restart the processing as many times it wants before de-initializing the algorithm instance. The requirement is that the application must call IP_RUN_resetAlgo() before calling IP_RUN_start() again.

3.7.4 Algorithm De-Initialization

Once the application is done with the algorithm, it can de-initialize the instance by calling IP_RUN_unregisterAlgo ().

3.7.5 IP_RUN De-Initialization

Call IP_RUN_deInit() to release the allocated EDMA resources by the VICP scheduling unit. The number of channels and param entries released to the system is specified in Section 3.6. The application must also call VICP_EDMA3_deinit() to close the EDMA3 driver instance tied to region 2.

3.7.6 Multiple Threads Usage Scenario

Back to the multiple threads usage scenario described in Section 3.3:

```
IP_RUN_registerAlgo(0) /* register algo #0 */
IP_RUN_registerAlgo(1) /* register algo #1 */


while (!stopEvent) {
      IP_RUN_resetAlgo()
      IP_RUN_start() /* start threads of both algo #0 and algo #1 */
      dspCode0()
      dspCode1()
      IP_RUN_wait() /* wait for algo #0 and algo #1 */
}

IP_RUN_unRegisterAlgo(1) /* unregister algo #1 */
IP_RUN_unRegisterAlgo(0) /* unregister algo #0 */
```

In the example above, the busy wait synchronization scheme is used by calling IP_RUN_wait(). In a multi-task environment, it is preferable to use the interrupt model. Besides the interrupt model allows the VICP to send an interrupt to the DSP after each thread's completion whereas the IP_RUN_wait() functions only returns once all the threads are completed. The application decides which thread whose completion triggers an interrupt by setting the member compIntEna of the structure IP_run. Once the DSP's interrupt service routine receives such an interrupt, it can find out which thread completed by calling the function IP_RUN_getCompCode(). This function returns the completion code of the last thread that completed its processing. Completion code is set by the application at algorithm initialization time through the member compCode of the structure IP_run. If IP_RUN_getCompCode() returns 0, it means the first thread has not yet completed.

3.8 Debugging Infrastructure

To assist in algorithm debugging, the application can implement a debug callback function that will be called after each block is processed by the VICP computation unit. To enable this, the following two requirements must be met:

- The pointer to the debug callback function must be passed to the VICP scheduling unit's library by using the function IP_RUN_setDebugCB().
- Busy waiting with IP_RUN_wait() must be used as synchronization technique instead of interrupt method.

The debug callback function must have the IP_RUN_DebugFunc type defined in vicp_sch.h:

```
typedef void (*IP_RUN_DebugFunc) (IP_RUN_DebugStruct *debug, void*arg);
```

When the VICP scheduling unit calls this function, it passes a structure of type IP_RUNDebugStruct:

```
    typedef struct {
            Int16 blkX;
            Int16 blkY;
            Int16 numBlocksX;
            Int16 numBlocksY;
            Int8 *imBufPtr;
            Int8 *coefBufPtr;
    } IP_RUN_DebugStruct;
```

This structure is first filled by the VICP scheduling unit library before being passed to the application's callback function. The callback function can in turn extract the desired information for further debugging. The member blkX and blkY provide the (X,Y) indexes of the block that has just been processed by the computation unit. The total number of X and Y blocks in the frame are provided by the member numBlocksX and numBlocksY. Consequently, blkX runs from 0 to numBlockX-1 and blkY runs from 0 to numBlockY-1.

The pointers imBufPtr and coefBufPtr provide means for the debug callback function to inspect the data that has just been processed. Indeed imBufPtr points to the image buffer that contains data just processed by the VICP. coefBufPtr points to the coefficient memory. At each block, imBufPtr toggles between the base address of image buffer A and image buffer B. The content of the image buffer at each call is the result of the computation commands sequence applied to the input data present in the image buffer. To debug a computation command sequence, one can insert in the middle of the encoding sequence an imxenc_sleep() command, recompile the code and run the algorithm in order to see results of the compute operations up to the sleep command.

A second input parameter passed by the VICP scheduling unit library to the debug callback function is the generic argument pointer arg. This pointer is actually passed by the application when calling IP_RUN_setDebugCB() and provides a way to pass application specific information to the debug callback function.

To pass the debug callback function's pointer to the library, the application calls IP_RUN_setDebugCB()which accepts as input parameters a pointer to the callback function and a generic argument pointer. Subsequently, IP_RUN_wait() calls the callback function after each block processing. If the application does not use busy waiting through IP_RUN_wait() then the debug callback function is never called. The implementation of the debug callback can be as simple as an empty function, just to allow putting a breakpoint in it. The breakpoint will be hit each time a block is processed and the content of the image buffer can be checked for correctness through Code Composer's memory window.

Also when passing a NULL pointer to IP_RUN_setDebugCB(), the library then disables the debug callback feature.

As one can expect, enabling the debug callback function and using busy waiting slow down the frame processing. Hence these features should only be used when debugging the code.

3.9 Example Code

Since the VICP signal processing library uses the VICP scheduling unit library to implement functions operating at frame level, its source code provides a good starting point towards a more customized library. It can also offer some good example codes on how the VICP scheduling unit library's functions are called.

For sample code using IP_RUN_init(), refer to the implementation of function CPIS_init() in [*VICP_LIBRARY_INSTALLATION_DIR*]\ src\src_hw\imgproclib.c .

For sample code using IP_RUN_registerAlgo(), refer to the file [*VICP_LIBRARY_INSTALLATION_DIR*]\ src\src_hw_imgproclib.c . The functions that set the structures DmaTferStruct are _CPIS_setDmaInTransfers() and _CPIS_setDmaOutTransfers().

VICP Scheduling Unit Functions

This chapter lists all the APIs provided by the VICP scheduling unit library. The declarations can be found in file vicp_sch.h , which must be included by any source files using the library.

4.1 Data Types

The following structure types are used in the VICP scheduling unit library.

4.1.1 IP_RUN_InitParams

This structure is used as input argument to IP_RUN_init().

Table 4-1. Input Argument Structure for IP_RUN_init()

Field	Data Type	Input/Output	Description
magicString	char*	Input	Used by IP_RUN_init() to check for validity of input parameter structure. Application must set it to “IP_run”
versionMajor	Uint16	Input	Major number of version which the application is compatible with. For example if version is 1.2, set versionMajor to 1.
versionMinor	Uint16	Input	Minor number of version which the application is compatible with. For example if version is 1.2, set versionMinor to 2.
staticDmaAlloc	Uint16	Input	Enable static allocation of EDMA3 channels and param entries.
numStaticDmaIn	Uint16	Input	Number of input EDMA3 channels that must be statically allocated inside IP_RUN_init() and will be later used. If staticDmaAlloc= 0, dynamic allocation will be used for all input channels.
numStaticDmaOut	Uint16	Input	Number of output EDMA channels that must be statically allocated inside IP_RUN_init() and will be later used. If staticDmaAlloc= 0, dynamic allocation will be used for all output channels.

4.1.2 IP_run

This structure is used as the input argument to IP_RUN_registerAlgo() and provides information on the location and layout of the frame data to be processed by the VICP.

Table 4-2. IP_run Structure

Field	Data Type	Input/Output	Description
dmaIn	DMATferStruct *	Input	Pointer to an array of structures DmaTferStruct used to initialize input DMA transfers from DDR to image buffer. The size of this array should match the member numDmaIn.
numDmaIn	Uint16	Input	Number of input transfers. Maximum value is limited by the symbol IP_RUN_MAX_NUM_DMAIN_CHAN.
dmaOut	DMATferStruct *	Input	Pointer to array of structures DmaTferStruct used to initialize output DMA transfers DMA transfers from image buffer to DDR. The size of this array should match the member numDmaOut.
numDmaOut	Uint16	Input	Number of output transfers. Maximum value is limited by the symbol IP_RUN_MAX_NUM_DMAOUT_CHAN.
numHorzBlocks	Uint16	Input	Number of blocks in the horizontal direction
numVertBlocks	Uint16	Input	Number of blocks in the vertical direction
chunksize	Uint16	Input	Ignored
compIntEna	Uint16	Input	Completion interrupt enabled. If set to 1 then VICP sends an interrupt at the completion of the computation thread.
compCode	Uint16	Input	Completion code. Code returned by IP_RUN_getCompCode() after the VICP computation thread finishes. Has to be non zero. In case of multi thread scenario, this can be used by the application to find out which thread has just completed.
cmdptr_ofst	Uint16	Input	Number of 16-bits words separating the beginning of the VICP command memory and the starting point of the VICP command sequence. The VICP command sequence is usually generated by calling imxenc_<computation> functions from the VICP computation unit library. The value of cmdptr_ofst should never exceed the total number of words in the command memory. The application should also take care not to generate a VICP command sequence beyond the end of the command memory.
extension	void *	Input	For future expansion
customExtension	void *	Input	For future expansion

4.1.3 DmaTferStruct

Members dmaIn and dmaOut of the IP_run structure are of type DmaTferStruct defined in Table 4-3.

Table 4-3. DmaTferStruct Structure

Field	Data Type	Input/Output	Description
ddrAddr	Uint32	Input	Starting address in DDR of the input or output data
ddrWidth	Uint16	Input	Width in bytes of the data in DDR
reserved0	Int16	Input	Not used
ddrOfstNextBlock	Int32	Input	Offset to next block, in bytes
ddrOfstNextBlockRow	Int32	Input	Offset to next row of blocks, in bytes. See Figure 3-2.
imgBufAddr	Uint32	Input	Starting address in image buffer of the block transferred to/from DDR.
imgBufWidth	Int16	Input	Width in bytes of the data in image buffer. Does not have to match blockWidth.
reserved1	Int16	Input	Not used
blockWidth	Uint16	Input	Block width in bytes
blockHeight	Uint16	Input	Block height in number of pixel rows
reserved2	Int16	Input	Not used
dmaChNo	Uint16	Input	Usually set it to DMAC_CHAN_ANY to get the next available channel unless a particular EDMA3 channel is to be allocated.

4.1.4 IP_RUN_DebugStruct

This structure is passed to the debug callback function of type IP_RUN_DebugFunc .

Table 4-4. IP_RUN_DebugStruct Structure

Field	Data Type	Input/Output	Description
blkX	Int16	Input	X-axis coordinate of the block just processed by the VICP
blkY	Int16	Input	Y-axis coordinate of the block just processed by the VICP
numBlocksX	Int16	Input	Total number of blocks in one row
numBlocksY	Int16	Input	Total number of rows of blocks
imgBufPtr	Int8 *	Input	Pointer to the image buffer where the block of data has just been processed
coefBufPtr	Int8 *	Input	Pointer to coefficient memory

4.2 Scheduling Unit Functions

Recommend you add introductory text here.

IP_RUN_init

Initialize the IP_RUN framework

Syntax

```
Int32 IP_RUN_init(IP_RUN_InitParams* init);
```

Arguments

`IP_RUN_InitParams* init;`	Pointer to IP_run initialization structure

Return Value

Int32	Returns 0 if success, ERROR_CHAN_ALLOC if EDMA channel allocation error.

Description

IP_RUN_init() initializes the internal structures used by the VICP scheduling unit library and loads specialized code and data into the scheduling unit's program and data memories (undocumented memories). This has as effect of overwriting any previous content in these memories.

The application can pass to IP_RUN_init() the default settings defined by the global variable IP_RUN_DEFAULT_INIT.

The member staticDmaAlloc in the initialization structure specifies whether some EDMA channels need to be statically allocated.

If set to 0, the EDMA channel allocation will be handled dynamically: each time IP_RUN_registerAlgo() is called, channels are allocated through the EDMA3 driver and each time IP_RUN_unregisterAlgo()is called, these same channels are released.

If set to 1, the application must also initialize the other members numStaticDmaIn and numStaticDmaOut. These members will represent how many input or output EDMA channels must be statically allocated by IP_RUN_init() to be used by IP_RUN_registerAlgo().

In the default settings staticDmaAlloc is set to 1 and numStaticDmaIn= numStaticDmaOut= 4.

This call is required prior to using any other function of the VICP scheduling unit library.

IP_RUN_registerAlgo *Register an algorithm*

Syntax

```
Int32 IP_RUN_registerAlgo (IP_run *handle, Int32 threadId);
```

Arguments

`IP_run *handle`	Pointer to IP_run structure, the member dmaChNo of the dmaIn and dmaOut structures will be modified by IP_RUN_registerAlgo
`Int32 threadId`	Index of the thread 0 or 1

Return Value

Int32	Returns 0 if success, ERROR_CHAN_ALLOC if EDMA channel allocation error

Description

For each computation thread, IP_RUN_registerAlgo () must be called to register it into the system.

IP_RUN_registerAlgo () also sets up and allocates the EDMA3 channels if they have not been yet statically allocated at initialization time. It returns an error ERROR_CHAN_ALLOC if it was unable to allocate all the EDMA channels.

The application should not free the IP_run structure after calling IP_RUN_registerAlgo() because it will be re-used when calling IP_RUN_resetAlgo() and IP_RUN_unregisterAlgo().

See Also

Structure IP_run

IP_RUN_start *Start the VICP processing*

Syntax

```
IP_RUN_start();
```

Arguments

`void`

Return Value

void

Description

IP_RUN_start() kicks off the VICP scheduling unit.

The function returns to the caller right after starting the VICP scheduling unit. The scheduling unit is responsible of controlling the VICP computation unit and the EDMA3 transfers in the most efficient manner so the transfers occur in parallel with the VICP computation. This frees up the DSP who can spend its MIPS in other tasks than controlling the VICP.

In the current version of the library, if several computation threads have been registered, then they are executed all at once. An interrupt can be sent to the DSP at completion of each thread if the member compIntEna in structure IP_run is set. The VICP does not wait for any acknowledgment from the DSP between computation threads, it just executes them at once without interruption.

See Also

IP_RUN_wait()

IP_RUN_wait *Wait for VICP completion*

Syntax `Void IP_RUN_wait();`

Arguments `void`

Return Value void

Description By calling IP_RUN_wait(), the DSP waits for the completion of the VICP computation threads. This is of course the least efficient way of synchronization since the DSP is wasting MIPS pending for other processors to finish. The effect can be mitigated by calling IP_RUN_wait() in a low priority task inside a multi-task OS environment but this is not the ideal solution.

Once IP_RUN_wait() exits, all the threads registered will have completed. This function does not interfere with the interrupt generation feature enabled by the member compIntEna in structure IP_run . The DSP can still call IP_RUN_wait() and receives interrupts at completion of each thread.

See Also IP_RUN_start()

IP_RUN_isBusy *Check whether VICP processing still ongoing.*

Syntax `Int32 IP_RUN_isBusy();`

Arguments `void`

Return Value Int32 1 if busy, 0 if done

Description Return 1 if VICP still processing, 0 otherwise.

See Also IP_RUN_start()

IP_RUN_getCompCode *Get completion code of the VICP thread last completed.*

Syntax `Int32 IP_RUN_getCompCode()`

Arguments `void`

Return Value Int32 Completion code value

Description For each computation thread, a completion code has been provided to the library by setting compCode member in structure IP_run. It is this completion code that IP_RUN_getCompcode() is returning.

See Also Structure IP_run, IP_RUN_start()

IP_RUN_resetAlgo *Reset the algorithm so it is ready to be re-executed by calling IP_RUN_start()*

Syntax

```
Int32 IP_RUN_resetAlgo(IP_run* handle, Int32 threadId, Int32 resetCmd)
```

Arguments

`IP_run *handle`	Handle to IP_run structure
`Int32 threadId`	Index of the thread, 0 or 1
`Int32 resetCmd`	For future use

Return Value

Int32 Returns 0 if no error

Description

Reset internal variables of IP_RUN so the computation thread can be re-run on a new image. The IP_run structure can be the same as the original one passed to IP_RUN_registerAlgo (), or have the following members with modified values:

```
dmaIn[ ].ddrAddr, dmaOut[ ].ddrAddr
cmdptr_ofst
```

IP_RUN_resetAlgo() allows the reconfiguration of the addresses of the input and output frames, as well as the sequence of computation tasks by modifying the command pointer offset.

The last feature actually allows the swapping of algorithm as long as the frame dimensions and processing block sizes do not change.

Its cycles count is between 6,000 and 9,000 DSP cycles depending on the number of input and output channels required by the algorithm.

See Also

Structure IP_run, IP_RUN_start()

IP_RUN_setDebugCB *Set the debug callback function*

Syntax

```
Int32 IP_RUN_setDebugCB(IP_RUN_DebugFunc debugCB, void *arg)
```

Arguments

`IP_RUN_DebugFunc debugCB`	Pointer to the debug function implemented by the application. If NULL then previously set debug callback function is disabled
`void *arg`	Pointer to the generic argument that will be passed to the debug function

Return Value

Int32 Returns 0 if no error

Description

This function is used to pass a debug callback function's pointer to the VICP computation unit library. The debug callback function has type IP_RUN_DebugFunc and is called after each block processing from within IP_RUN_wait():

```
typedef void (*IP_RUN_DebugFunc) (IP_RUN_DebugStruct *debug, void*arg);
```

The implementation of the debug callback can be as simple as an empty function, just to allow putting a breakpoint in it. The breakpoint will be hit each time a block is processed and the content of the image buffer can be checked for correctness through Code Composer's memory window.

See Section 3.8 for further information.

See Also

Structure IP_RUN_DebugStruct

IP_RUN_unregisterAlgo *Unregister an algorithm*

Syntax `Int32 IP_RUN_unregisterAlgo(IP_run* handle, Int32 threadId)`

Arguments

`IP_run *handle`	Handle to IP_run structure that was used in IP_RUN_registerAlgo
`Int32 threadId`	Index of the computation thread 0 or 1

Return Value Int32 Returns 0 if no error

Description Calling IP_RUN_unregisterAlgo() frees up only the EDMA3 resources that were dynamically allocated during IP_RUN_registerAlgo(). It needs the same structure IP_run passed to IP_RUN_registerAlgo() because it needs the dmaChNo information in dmaIn and dmaOut of that structure.

See Also Structure IP_run, IP_RUN_registerAlgo()

IP_RUN_deInit *Initialize the IP_RUN framework*

Syntax `Int32 IP_RUN_deInit();`

Arguments `None`

Return Value Int32 Returns 0 if success,
ERROR_CHAN_DEALLOC if EDMA channel de-allocation error

Description De-initialize IP_run by freeing all EDMA channels and param entries

Chapter 5
SPRUGN1C–November 2009

VICP Support Functions

This chapter provides the descriptions of some support functions that could be useful in the course of developing VICP code. All functions' interfaces and symbols are defined in vicp_support.h . The functions themselves are provided in dmcsl648_bios.lib or dmcsl644x_bios.lib libraries.

5.1 Functions

The CACHE functions are wrapper functions and do not actually implement any cache functionality. The implementation is provided by the application when calling CACHE_init(). Afterwards, the wrapper functions can be invoked by the VICP Scheduling Unit library.

CACHE_init — *Init CACHE Wrapper Functions*

Function

```
void CACHE_init(CACHE_InitPrm_t *pPrm)
```

Arguments

`CACHE_InitPrm_t`	Data structure which contains pointers to the actual implementations of cache functions

Return Value None

Description The CACHE_init function is used to pass the actual implementation of data cache functions to the wrapper functions. CACHE_init must be called before IP_RUN_init().

CACHE_writeBack — *Wrapper function for data cache write-back function*

Function

```
void CACHE_writeBack(CACHE_Addr_t addr, CACHE_Size_t size)
```

Arguments

`CACHE_Addr_t addr`	Buffer address
`CACHE_Size_t size`	Buffer size

Return Value None

Description Wrapper function that calls the corresponding function in the structure CACHE_InitPrm_t passed to CACHE_init().

CACHE_invalidate *Wrapper function for data cache invalidate function.*

Function

```
void CACHE_invalidate(CACHE_Addr_t addr, CACHE_Size_t size)
```

Arguments

`CACHE_Addr_t`	`addr`	Buffer address
`CACHE_Size_t`	`size`	Buffer size

Return Value

None

Description

Wrapper function that calls the corresponding function in the structure CACHE_InitPrm_t passed to CACHE_init().

CACHE_wbInv *Wrapper function for data cache write back and invalidate function.*

Function

```
void CACHE_wbInv(CACHE_Addr_t addr, CACHE_Size_t size)
```

Arguments

`CACHE_Addr_t`	`addr`	Buffer address
`CACHE_Size_t`	`size`	Buffer size

Return Value

None

Description

Wrapper function that calls the corresponding function in the structure CACHE_InitPrm_t passed to CACHE_init().

IMGBUF_switch *Switch buffers between DSP/EDMA and VICP*

Function

```
Uint16 IMGBUF_switch(Uint16 buffers, Uint16 connections);
```

Arguments

`Uint16 buffers`	Bit flags value to select switches. Use the predefined macro symbols defined in vicp_support.h : SELIMGBUFA, SELIMGBUFB, SELCOEFBUF, SELCMDBUF, SELALLBUF. Any combination of these symbols can be ORed together to produce the value buffers.
`Uint16 connections`	Bit flags value to set data path switch. Use the predefined macro symbols defined in vicp_support.h : IMGBUFADSP, IMGBUFAVICP, IMGBUFBDSP, IMGBUFBVICP, COEFFBUFDSP, COEFFBUFVICP, COEFFBUFAUTO, CMDBUFDSP, CMDBUFVICP, CMDBUFAUTO Any combination of these symbols can be ORed together to produce the value connections.

Return Value Previous connections values

Description

The function IMGBUF_switch() is used to grant access of the image buffer A, image buffer B, coefficient memory, command memory to either DSP or VICP computation unit. The value returned can be used to restore the switch setting later on.

A few remarks: Both coefficient and command memories have an automatic access mode, very convenient since it allows these buffers to be automatically switched to the VICP computation unit when it becomes active. Also when image buffer A's access is granted to VICP then automatically image buffer B is granted access to DSP, and vice versa. The access switch for image buffer A or B does not need to be changed if the VICP scheduling unit library is used. This is handled automatically by the scheduling unit.

Example

```
#include "vicp_support.h"

/* switch coef and cmd memories to auto mode */
connections= IMGBUF_switch(SELCOEFBUF|SELCMDBUFF, COEFFBUFAUTO|CMDBUFAUTO);

/* switch image buffer A to DSP, image buffer B automatically switched to VICP */
IMGBUF_switch(SELIMGBUFA, IMGBUFADSP);

/* Restore previous settings */
IMGBUF_switch(SELALLBUF, connections);
```

IMX_start — *Start VICP computation unit execution*

Function `void IMX_start(Uint16 *startaddr)`

Arguments

`Uint16 *startaddr`	Absolute address of the sequence of VICP commands to execute. Can fall anywhere between (Int16*)CMDBUF_BASE and (Int16*)CMDBUF_BASE + CMDBUF_SIZE

Return Value None

Description Start VICP computation unit execution. The IMX_start function is not needed if the VICP scheduling unit library is used. IMX_start is needed only if custom control/scheduling code is to be written on the DSP.

IMX_wait — *Wait for VICP computation unit to finish execution*

Function `void IMX_wait()`

Arguments None

Return Value None

Description Busy polls some register until VICP computation unit stops. The IMX_wait function is not needed if the VICP scheduling unit library is used. IMX_wait is needed only if custom control/scheduling code is to be written on the DSP.

5.2 Data Types

CACHE_InitPrm_t — *Structure for CACHE_init*

Members

`CACHE_Cb wbCb`	Callback for data-cache write back function
`CACHE_Cb invCb`	Callback for data-cache invalidate function
CACHE_Cb wbInvCb	Callback for data-cache write back and invalidate function

Description This data structure is used to pass pointer of the callbacks functions that will be used by the wrapper CACHE_writeBack(), CACHE_invalidate(), CACHE_wbInv().This structure is passed during CACHE_init.

VICP Computation Unit Library's Functions

This chapter details all the functions making up the VICP computation unit library.

6.1 Functions That Encode Computation Tasks

These functions of the form imxenc_<computation> encode the VICP commands that correspond to a particular mathematical or image processing task.

6.1.1 *imxenc_alphablend*

imxenc_alphablend ***Perform alphablend algorithm between 32 bpp αRGB source data and 16 bpp RGB555 or RGB565 background data. The output is written in unpacked R,G,B planes.***

Syntax

```
cmdlen = imxenc_alphablend(
    src,              /* Int16*, Input to αRGB 32 bpp data */
    bkg,              /* Int16*, starting address of background rgb565 or rgb555 */
    tempScratch1,     /* Int16*, starting address of temporary scratch buffer 1 of
                         size 2*compute_width*compute_height+4 words */
    tempScratch2,     /* Int16*, starting address of temporary scratch buffer 2 of
                         size 1.5*compute_width*compute_height words */
    permScratch,      /* Int16*, starting address of permanent scratch buffer of
                         size 7 words in iMX coefficient memory */
    dst,              /* Int16*, pointer to destination */
    src_width,        /* Int16, src width in number of pixels */
    src_height,       /* Int16, src height/rows of in number of pixels */
    bkg_width,        /* Int16, bkg width in number pixels */
    bkg_height,       /* Int16, bkg height/rows of in number of pixels */
    dst_width,        /* Int16, dst width/columns of in number of pixels */
    dst_height,       /* Int16, dst height/rows of in number of pixels */
    compute_width,    /* Int16, computation width in number of pixels */
    compute_height,   /* Int16, computation height in number of pixels */
    dst_type,         /* Int16, IMXOTYPE_SHORT or IMXOTYPE_BYTE */
    colorformat,      /* Int16, RGB565=0 or RGB555=1 */
    cmdptr);          /* Int16*, cmdptr */
```

Description

This function takes the source bitmap and blends it with the background bitmap into the destination buffer. In the source bitmap, each pixel is coded on 32 bits αRGB colorformat.

Table 6-1 and Table 6-2 describe how two consecutive pixels α0R0G0B0, α1R1G1B1 are expected to be organized in ARM or DSP memory.

Table 6-1. Organization of Two Consecutive Pixels in ARM Memory

Byte Address	Value
0	B_0
1	G_0
2	R_0
3	α_0
4	B_1
5	G_1
6	R_1
7	α_1

Table 6-2. Organization of Two Consecutive Pixels in DSP Memory

Word Address	Value	
	8 msb	8 lsb
0	G_0	B_0
1	α_0	R_0
2	G_1	B_1
3	α_1	R_1

The background format is 16bpp RGB565 or RGB555 as specified by the parameter colorformat. The function actually unpacks the output and writes it into R,G,B planes. Elements in each plane can be either 8 or 16 bits wide with only 8 significant bits since R,G,B are in the [0…255] range. Planes are arranged sequentially in the output: if dst_type is IMXOTYPE_SHORT then R plane occupies the first dst_width x dst_height 16-bits word, G occupies the next dst_width x dst_height 16-bits word and B occupies the last dst_width x dst_height 16-bits word.

Three scratch buffers must be allocated in advance:

- Two temporary scratch buffers: one of size `(2 × compute_width × compute_height + 4)` words and the other one of size `(1.5 × compute_width × compute_height)` words.
- One permanent scratch buffer of size 7 words in VICP coefficient memory.

The temporary scratch buffers can be re-used by other VICP functions needing a temporary scratch buffer and its content can be overwritten after the corresponding alphablending VICP sequence is executed. Re-use of these temporary scratch buffers by other VICP functions is highly recommended in order to optimize memory allocation in the image buffer or coefficient buffers.

In contrast, the permanent scratch must never be altered by the application. The imxenc_alphablend() function initializes the 7 words contained in the scratch and imxUpdate_alphablend() update those when it is called. This permanent scratch must be allocated in the VICP coefficient memory.

The locations, dimensions of source, background, destination bitmaps and the background colorformat are fixed for the encoded VICP command. If the program wants to execute the VICP commands that performs the alphablend algorithm for a different background colorformat, it can call the function imxUpdate_alphablend() to update an existing VICP command sequence. If any other parameters must be changed, then the application must call imxenc_alphablend() all over again since the imxUpdate_alphablend() only allows for changing the colorformat value.

Constraints

Compute_width must be a multiple of 8.

Performance

Top performance is 5.6 cycles/pixel. This top performance can be achieved if all the scratch buffers are in VICP coefficient memory and the source, destination are in image buffer and the background in VICP coefficient memory.

6.1.2 imxenc_alphablendYUV422I

imxenc_alphablendYUV422I ***Perform alphablend algorithm between 16 bpp YUV422 interleaved foreground and 16 bpp YUV422 interleaved background data, using alpha plane.***

Syntax

```
cmdlen = imxenc_alphablendYUV422I(
    Int16 *foreground,    /* point to yuv422 foreground, 2 bytes/pixel */
    Int16 *background,    /* point to yuv422 background, 2 bytes/pixel */
    Int16 *alpha,         /* point to alpha plane, 1 byte/pixel */
    Int16 *output,        /* point to output, 2 bytes/pixel */
    Int16 *tempScratch,   /* point to temporary scratch of size 1.5*compute_width*
                             compute_height bytes */
    Int16 *permScratch,   /* point to permanent scratch in coefficient memory, of
                             size 13*compute_width*compute_height bytes+6 bytes */
    Int16 input_width,    /* width of the input in number of pixels */
    Int16 output_width,   /* width of the output in number of pixels */
    Int16 compute_width,  /* computation width in number of pixels */
    Int16 compute_height, /* computation height in number of pixels */
    Int16 *cmdptr);      /* Pointer to command memory */
```

Description

This function takes the foreground and blends it with the background into the destination buffer. The formula used is:

$$\mathit{output}[i,j] = \alpha[i,j] \times \mathit{foreground}[i,j] + (1- \alpha[i,j]) \times \mathit{background}[i,j] \quad (2)$$

Constraints

compute_width must be a multiple of 4.

Performance

Top performance is 2.5 cycles/pixel. This top performance can be achieved if alpha plane and scratch buffer are in VICP coef memory and foreground, background in image buffers.

6.1.3 *imxenc_accumulate2d_array_op*

imxenc_accumulate2d_array_op ***Perform point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) on arrays, with accumulation.***

Syntax

```
cmdlen = imxenc_accumulate2d_array_op(
    Int16 *input1_ptr,              /* starting address of 1st input */
    Int16 *input2_ptr,              /* starting address of 2nd input */
    Int16 *output_ptr,              /* starting address of output */
    Int16 input1_width,             /* width/columns of 1st input */
    Int16 input1_block_height,      /* height/rows of a single block of the 1st input */
    Int16 input2_width,             /* width/columns of 2nd input */
    Int16 input2_block_height,      /* height/rows of a single block of the 2nd input */
    Int16 output_width,             /* width/columns of output */
    Int16 output_block_height,      /* height/rows of single block of the output */
    Int16 number_blocks,            /* number of blocks */
    Int16 acc_depth,                /* depth of accumulate i.e number of repetition sets over which
                                       accumulation occurs */
    Int16 compute_units_1,          /* number of units processed in the first set of data repetitions */
    Int16 compute_units_2,          /* number of units processed in the second set of data repetitions*/
    Int16 input1_compute_offset_1,  /* offset between input1 for the first set of data repetitions  */
    Int16 input1_compute_offset_2,  /* offset between input1 for the second set of data repetitions  */
    Int16 input2_compute_offset_1,  /* offset between input2 for the first set of data repetitions  */
    Int16 input2_compute_offset_2,  /* offset between input2 for the second set of data repetitions  */
    Int16 output_compute_offset_1,  /* offset between output for the first set of data repetitions  */
    Int16 output_compute_offset_2,  /* offset between output for the second set of data repetitions  */
    Int16 operation,
    Int16 round_shift_off,          /* rounding off: 1: off, 0: on */
    Int16 round_shift,              /* shifting parameter */
    Int16 asap,                     /*       1: asap mode, 0: non-asap mode */
    Int16 input1_type,     /* IMXTYPE_UBYTE,IMXTYPE_BYTE,IMXTYPE_USHORT,IMXTYPE_SHORT*/
    Int16 input2_type,     /* IMXTYPE_UBYTE,IMXTYPE_BYTE,IMXTYPE_USHORT,IMXTYPE_SHORT*/
    Int16 output_type,     /* IMXOTYPE_BYTE,IMXOTYPE_SHORT */
    Int16 *cmdptr);        /* starting point of command sequence in memory */

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs add, subtract, multiply, absolute difference, pack bytes, and, or, mask, mask-not, minimum, maximum, packed-bytes absolute differences between two matrices of dimension [compute_width x compute_height]. The operation is performed on blocks of data, each block having width equal to the width of the particular array, and height equal to the block height.

The API allows for the operations to be repeated over sets of data within each block. compute_units_1 is the number of first set of repetitions, and compute_units_2 is the number of repetitions of the first repetition. So, the total number of data operation repetitions is compute_units_1* compute_units_2. The step sizes for the two sets of data operation repetitions is given by XXXX_compute_offset_1 and XXXX_compute_offset_2. These step sizes are absolute displacements, and not x and y displacements. The second data operation repetition offset is applied to the starting point of the previous occurring first data repetition, and not the end point of the first data set repetition. The command allows the results of the data operations repetitions to be accumulated over one or both of the data sets.

The operations are performed along the entire width of the inputs and for each block. Within each block, the operating region is specified by XXXX_compute_offset_1 and XXXX_compute_offset_2. The pointers, input1_ptr, input2_ptr, and output_ptr, specify the first element, or upper-left corner of actual operands and output.

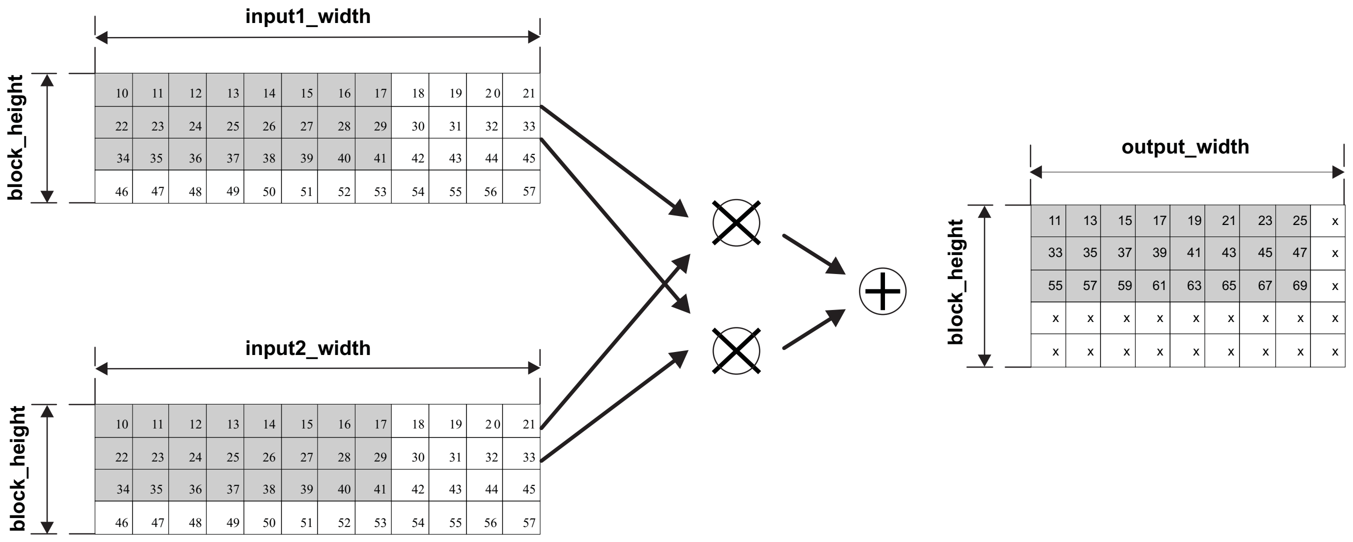
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider input1 and input 2 → 16x16 matrix, consisting of 4 blocks of dimension 16(W)x4(H), and output → 20(W)x16(H) matrix; that is, 4 blocks of dimension 20x4. The goal is to perform an element-wise multiply of the first two rows of each block, and add the resulting rows. Translating that to the functioning of this API: The multiple operation is repeated over two pairs of elements and the result accumulated. So, there is one set of data operation repetitions over two pairs of elements with a step size between elements equal to the width of the array. There is no second set of data repetitions, so the corresponding offsets will be set to 0. Also, since the output over the first set of data repetitions is accumulated, output_compute_offset1 is also set to zero. The following illustrates what the API call would look like and a descriptive figure is given below.

```
cmdlen = imxenc_accumulate2d_array_op(
    Int16 *input1_ptr,  /* starting address of 1st input */
    Int16 *input2_ptr,  /* starting address of 2nd input */
    Int16 *output_ptr,  /* starting address of output */
    16,                 /* width/columns of 1st input */
    4,                  /* height/rows of a single block of the 1st input */
    16,                 /* width/columns of 2nd input */
    4,                  /* height/rows of a single block of the 2nd input */
    20                  /* width/columns of output */
    4                   /* height/rows of single block of the output */
    4                   /* number of blocks */
    1,                  /* depth of accumulate i.e number of
                           repetition sets over which accumulation occurs */
    2,         /* number of units processed in the first set of data repetitions */
    1,         /* number of units processed in the second set of data repetitions */
    16         /* offset between input1 for the first set of data repetitions  */
    0          /* offset between input1 for the second set of data repetitions  */
    16         /* offset between input2 for the first set of data repetitions  */
    0          /* offset between input2 for the second set of data repetitions  */
    0          /* offset between output for the first set of data repetitions  */
    0          /* offset between output for the second set of data repetitions  */
    0,
    0,                  /* rounding off: 1: off,  0: on */
    0,                  /* shifting parameter */
    0,                  /*     1: asap mode, 0: non-asap mode */
    IMXTYPE_SHORT,
    IMXTYPE_SHORT ,
    IMXOTYPE_SHORT,
    Int16 *cmdptr);
```

Figure 6-1. imxenc_accumulate2d_array_op



Constraints

- The input1 width and input2 width must be equal, and a multiple of 8. Output width can be greater than the input widths.
- Offsets for each of the dimensions must be less than or equal to 32768.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (3)$$

- amount_of_work =

$$\textit{compute\_units\_1} \times \textit{compute\_units\_2} \times \max(\textit{input1\_width}, \textit{input2\_width}) \times \textit{number\_blocks} \quad (4)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	2
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	2
COEFF	COEFF	IMGBUF	2
COEFF	COEFF	COEFF	3

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [1]	1	256

[1] That is, compute_width is odd.

6.1.4 imxenc_accumulate2d_array_scalar_op

imxenc_accumulate2d_array_scalar_op ***Perform point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) on arrays, with accumulation.***

Syntax

```
cmdlen = imxenc_accumulate2d_array_scalar_op(
    Int16 *input1_ptr,              /* starting address of 1st input */
    Int16 *input2_ptr,              /* starting address of 2nd input */
    Int16 *output_ptr,              /* starting address of output */
    Int16 input1_width,             /* width/columns of 1st input */
    Int16 input1_block_height,      /* height/rows of a single block of the 1st input */
    Int16 input2_width,             /* width/columns of 2nd input */
    Int16 input2_block_height,      /* height/rows of a single block of the 2nd input */
    Int16 output_width,             /* width/columns of output */
    Int16 output_block_height,      /* height/rows of single block of the output */
    Int16 number_blocks,            /* number of blocks */
    Int16 acc_depth,                /* depth of accumulate i.e number of
                                       repetition sets over which accumulation occurs */
    Int16 compute_units_1,          /* number of units processed in the first set of data repetitions */
    Int16 compute_units_2,          /* number of units processed in the second set of data repetitions*/
    Int16 input1_compute_offset_1,  /* offset between input1 for the first set of data repetitions  */
    Int16 input1_compute_offset_2,  /* offset between input1 for the second set of data repetitions  */
    Int16 input2_compute_offset_1,  /* offset between input2 for the first set of data repetitions  */
    Int16 input2_compute_offset_2,  /* offset between input2 for the second set of data repetitions  */
    Int16 output_compute_offset_1,  /* offset between output for the first set of data repetitions  */
    Int16 output_compute_offset_2,  /* offset between output for the second set of data repetitions  */
    Int16 operation,                /* IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB, IMXOP_AND, IMXOP_OR,
                                       IMXOP_XOR, IMXOP_MIN, IMXOP_MAX */
    Int16 round_shift_off,          /* rounding off: 1: off 0: on */
    Int16 round_shift,              /* shifting parameter */
    Int16 asap,                     /* 1: asap mode, 0: non-asap mode */
    Int16 input1_type,              /* IMXTYPE_UBYTE,IMXTYPE_BYTE,IMXTYPE_USHORT,IMXTYPE_SHORT*/
    Int16 input2_type,              /* IMXTYPE_UBYTE,IMXTYPE_BYTE,IMXTYPE_USHORT,IMXTYPE_SHORT*/
    Int16 output_type,              /* IMXOTYPE_BYTE,IMXOTYPE_SHORT */
    Int16 *cmdptr);                 /* starting point of command sequence in memory */

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

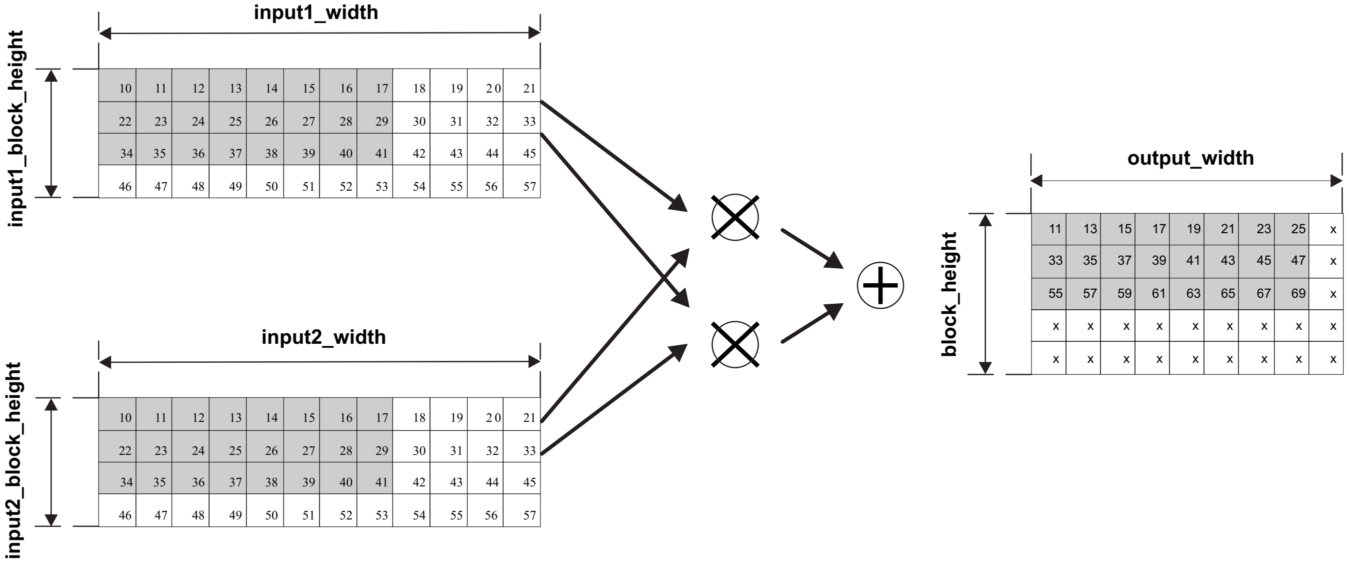
This function performs add, subtract, multiply, absolute difference, pack bytes, and, or, mask, mask-not, minimum, maximum, packed-bytes absolute differences between two matrices of dimension [compute_width x compute_height]. The operation is performed on blocks of the first input, each block having width equal to the input width, and height equal to the block height. Operation is performed on only the first block of the second input – this block is operated on all the blocks of the first input Hence this API is a “scalar” version of imxenc_accumulate2d_array_op. Apart from this “scalar” type of operation, the API is identical in operation to imxenc_accumulate2d_array_op

Example

Consider input1 and input 2 → 16x16 matrix, consisting of 4 blocks of dimension 16(W)x4(H), and output → 20(W)x16(H) matrix; that is, 4 blocks of dimension 20x4. The goal is to perform an element-wise multiply of the first two rows of each block of input1 with the first two rows of the first block of input2, and add the resulting rows. Translating that to the functioning of this API: The multiple operation is repeated over two pairs of elements and the result accumulated. So, there is one set of data operation repetitions over two pairs of elements with a step size between elements equal to the width of the array. There is no second set of data repetitions, so the corresponding offsets will be set to 0. Also, since the output over the first set of data repetitions is accumulated, output_compute_offset1 is also set to zero. The following illustrates what the API call would look like and Figure 6-2 provides an illustration.

```
cmdlen = imxenc_accumulate2d_array_scalar_op(
    Int16 *input1_ptr,  /* starting address of 1st input */
    Int16 *input2_ptr,  /* starting address of 2nd input */
    Int16 *output_ptr,  /* starting address of output */
    16,                 /* width/columns of 1st input */
    4,                  /* height/rows of a single block of the 1st input */
    16,                 /* width/columns of 2nd input */
    4,                  /* height/rows of a single block of the 2nd input */
    20                  /* width/columns of output */
    4                   /* height/rows of single block of the output */
    4                   /* number of blocks */
    1,                  /* depth of accumulate i.e number of
                           repetition sets over which accumulation occurs */
    2,       /* number of units processed in the first set of data repetitions */
    1,       /* number of units processed in the second set of data repetitions */
    16       /* offset between input1 for the first set of data repetitions  */
    0        /* offset between input1 for the second set of data repetitions  */
    16       /* offset between input2 for the first set of data repetitions  */
    0        /* offset between input2 for the second set of data repetitions  */
    0        /* offset between output for the first set of data repetitions  */
    0        /* offset between output for the second set of data repetitions  */
    0,
    0,                  /* rounding off: 1: off
                                         0: on  */
    0,                  /* shifting parameter */
    0,                  /*               1: asap mode
                                         0: non-asap mode */
    IMXTYPE_SHORT,
    IMXTYPE_SHORT ,
    IMXOTYPE_SHORT,
    Int16 *cmdptr);
```

Figure 6-2. imxenc_accumulate2d_array_scalar_op



Constraints

- The input1 width and input2 width must be equal, and a multiple of 8. Output width can be greater than the input widths.
- Offsets for each of the dimensions must be less than or equal to 32768.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (5)$$

- amount_of_work =

$$compute\_units\_1 \times compute\_units\_2 \times max(input1\_width, input2\_width) \times number\_blocks \quad (6)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	2
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	2
COEFF	COEFF	IMGBUF	2
COEFF	COEFF	COEFF	3

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [(2)]	1	256

(2) That is, compute_width is odd.

6.1.5 imxenc_argb2argbPlanar()

imxenc_argb2argbPlanar() ***Unpack 32 bpp αRGB source data into alpha, R,G,B planes, in which each element is byte wide.***

Syntax

```
cmdlen = imxenc_argb2argbPlanar(
    input,            /* Int16*, Input to *RGB 32 bpp data */
    scratch,          /* Int16*, pointer to permanent scratch buffer of 2 words */
    output,           /* Int16*, pointer to output */
    input_width,      /* Int16*, input width in number of pixels */
    input_height,     /* Int16*, input height in number of pixels */
    output_width,     /* Int16*, output width in number of pixels */
    output_height,    /* Int16*, output height in number of pixels */
    compute_width,    /* Int16, computation width in number of pixels */
    compute_height,   /* Int16, computation height in number of pixels */
    endian,           /* Int16 endianess, 0 = little endian and 1 = big endian */
    cmdptr);          /* Int16*, cmdptr */

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

Table 6-3 describes how two consecutive pixels α0R0G0B0, α1R1G1B1 are expected to be organized in ARM or DSP memory, depending on the input argument endian value.

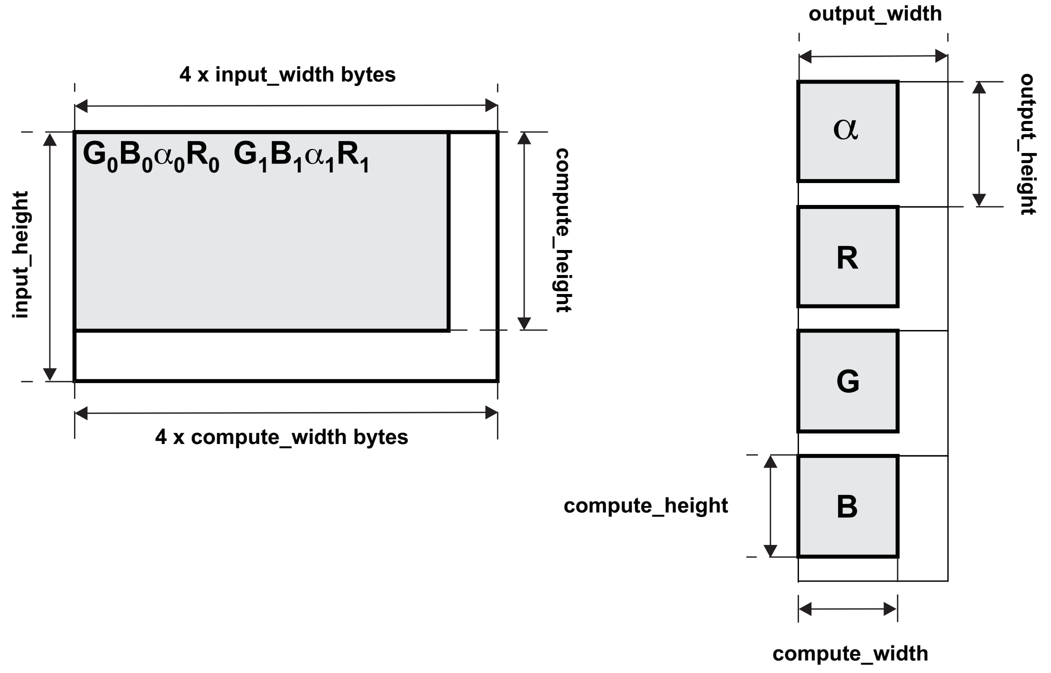
Table 6-3. Organization of Two Consecutive Pixels in Memory

Byte Address	Value When Endian = 0	Value When Endian = 1
0	α_0	B_0
1	R_0	G_0
2	G_0	R_0
3	B_0	α_0
4	α_1	B_1
5	R_1	R_1
6	G_1	G_1
7	B_1	α_1

Planes are arranged sequentially in the output: α plane occupies the first output_width × output_height bytes, R plane occupies the second output_width × output_height bytes, G occupies the next output_width × output_height bytes and B occupies the output_width × output_height bytes.

A permanent scratch buffer of two 16-bits words must be allocated. The permanent scratch must never be altered by the application and is usually allocated in the VICP coefficient memory. Each element of the output planes is in byte format.

Figure 6-3. imxenc_argb2argbPlanar()



Constraints	compute_width must be a multiple of 8.
Performance	Performance is between 2 and 4 cycles/pixel. Top performance can be achieved if at least 2 of the buffers (input, scratch or output) are in VICP coefficient memory.

6.1.6 imxenc_array_cond_write

imxenc_array_cond_write ***Perform point-by-point conditional write of input1.***

Syntax

```
cmdlen = imxenc_array_cond_write(
    input1_ptr,          /* Int16*, starting address of 1st input */
    input2_ptr,          /* Int16*, starting address of 2nd input */
    output_ptr,          /* Int16*, starting address of the output array */
    input1_width,        /* Int16, width of 1st input */
    input1_height,       /* Int16, height of 1st input */
    input2_width,        /* Int16, width of 2nd input */
    input2_height,       /* Int16, height of 2nd input */
    output_width,        /* Int16, width of output */
    output_height,       /* Int16, height of output */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input1_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    input2_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    cond_write_mode,     /* Int16 (0) -> write upon zero
                                  (1) -> write upon non-zero
                                  (2) -> write upon saturation
                                  (3) -> write upon not saturate
                         */
    cmdptr,              /* Int16*, starting point of command sequence in memory*/
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs a conditional write between two matrices of dimension [compute_width x compute_height]. The output element is set equal to the input1 element if the input2 element satisfies the condition specified by the cond_write_mode. Actual operand size of [compute_width x compute_height] resides within [input1_width x input1_height] 1st input and within [input2_width x input2_height] 2nd input. Actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The pointers, input1_ptr, input2_ptr, and output_ptr, specify the first element, or upper-left corner of actual operands and output.

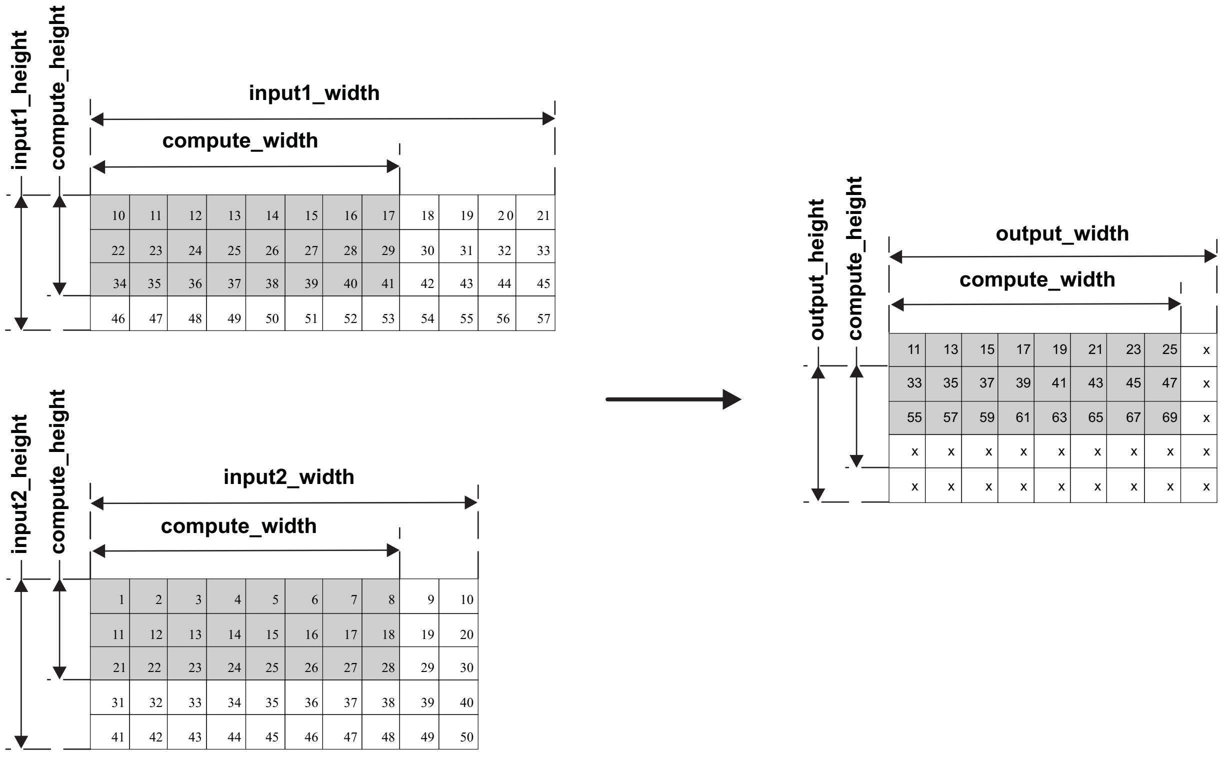
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider input1 → 16x16 matrix, input2 → 12(W)x20(H) matrix and output → 20(W)x13(H) matrix. Choose a matrix addition of 8(W)x10(H) as dimensions. The following code illustrates what the API call would look like and Figure 6-4 provides a visual description.

```
cmdlen = imxenc_array_cond_write(
   input1_ptr,          /* starting address of 1st input */
   input2_ptr,          /* starting address of 2nd input */
   output_ptr,          /* starting address of output */
   12,                  /* width of 1st input */
   4,                   /* height of 1st input */
   10,                  /* width of 2nd input */
   5,                   /* height of 2nd input */
   9,                   /* width of output */
   5,                   /* height of output */
   8,                   /* computation width */
   3,                   /* computation height */
   IMXTYPE_SHORT,       /* signed Int16/unsigned byte*/
   IMXTYPE_SHORT,       /* byte, Int16 */
   IMXOTYPE_SHORT,      /* byte, Int16 */
   0,                   /* number of bits to downshift before output */
   0,                   /* write upon zero */
   cmdptr               /* starting point for command sequence in memory */
  );
```

Figure 6-4. imxenc_array_cond_write



Constraints

- input1_width, input2_width, output_width >= compute_width
- input1_height, input2_height, output_height >= compute_height
- compute_height and input_height must be < 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \tag{7}$$

- amount_of_work =

$$compute\_width \times compute\_height \tag{8}$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	2
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	2
COEFF	COEFF	IMGBUF	2
COEFF	COEFF	COEFF	3

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [3]	1	256

[3] That is, compute_width is odd.

6.1.7 imxenc_array_inner_product

imxenc_array_inner_product *Perform inner-product operation between a 4-D data array and a 4-D coefficient array, producing a 2-D output array.*

Syntax

```
cmdlen = imxenc_array_inner_product(
    input_ptr,          /* Int16*, starting address of input */
    coeff_ptr,          /* Int16*, starting address of coefficients */
    output_ptr,         /* Int16*, starting address of output */
    input_width,        /* Int16, width of input array */
    input_height,       /* Int16, height of input array */
    coeff_width,        /* Int16, width of coefficient array */
    coeff_height,       /* Int16, height of coefficient array */
    output_width,       /* Int16, width of output array */
    output_height,      /* Int16, height of output array */
    compute_width,      /* Int16, computed width */
    compute_height,     /* Int16, computed height */
    num_terms_horz,     /* Int16, number of arrays horizontally to combine */
    num_terms_vert,     /* Int16, number of arrays vertically to combine */
    input_offset_horz,  /* Int16, horizontal offset of input arrays in data pts */
    input_offset_vert,  /* Int16, vertical offset of input arrays in data pts */
    coeff_offset_horz,  /* Int16, horizontal offset of coeff arrays in data pts */
    coeff_offset_vert,  /* Int16, vertical offset of coeff arrays in data pts */
    input_type,         /* Int16, Int16/byte, signed/unsigned */
    coeff_type,         /* Int16, Int16/byte, signed/unsigned */
    output_type,        /* Int16, Int16/byte */
    round_shift,        /* Int16, number of bits to downshift before output */
    cmdptr              /* Int16*, starting point of command sequence in memory */
);
```

Description

This function performs inner product between a 4-D data array and a 4-D coefficient array, resulting in a 2-D output array. Each pair of data inner array and coefficient inner array are multiplied point-by-point, and then these product arrays are summed together on the outer 2 dimensions.

For the inner 2-D arrays, the actual operand size of [compute_width x compute_height] resides within [input_width x input_height] of data input and within [coeff_width x coeff_height] of coefficient input, and actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The outer two dimensions are indexed with horizontal and vertical offsets, and the offsets are in data points (not the address offsets). The pointers, input_ptr, coeff_ptr, and output_ptr, specify the first element, or upper-left corner of operands and output. For example, logical data item input[m, n, i, j] is assumed to reside at input_ptr[m * input_offset_vert + n * input_offset_horz + i * input_width + j]. The xxx_height parameters are included for modularity, but are not used in any address calculation.

The outer dimensions of input and/or coefficient can optionally be used to index overlapping sub-arrays in an input array. For example, using input_offset_horz = input_offset_vert = 1, and num_distribute_horz = num_distribute_vert = 3 addresses a 3x3 neighborhood for each inner-array data point.

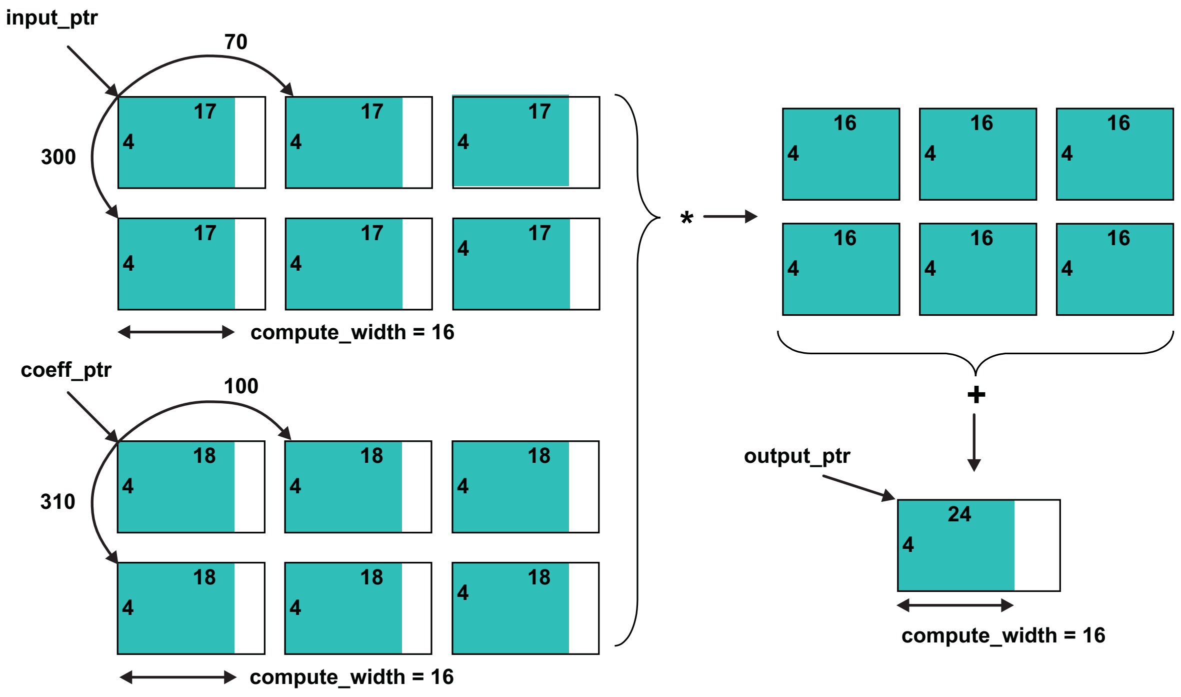
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider data input → 2(outerH) x 3(outerW) x 4(H) x 17(H) matrix, coefficient input → 2(outerH) x 3(outerW) x 4(W) x 18(H) matrix and output → 4(H) x 24(W) matrix. Compute 4(H) x 16(W) in the inner dimensions. The following code illustrates what the API call would look like and Figure 6-5 provides a visual description.

```
cmdlen = imxenc_array_inner_product(
    input_ptr,        /* starting address of 1st input */
    coeff_ptr,        /* starting address of 2nd input */
    output_ptr,       /* starting address of output */
    17,               /* width of data input */
    4,                /* height of data input */
    18,               /* width of coefficient input */
    4,                /* height of coefficient input */
    24,               /* width of output */
    4,                /* height of output */
    16,               /* computation width */
    4,                /* computation height */
    3,                /* outer width */
    2,                /* outer height */
    70,               /* input horizontal offset */
    300,              /* input vertical offset */
    100,              /* coefficient horizontal offset */
    310,              /* coefficient vertical offset */
    IMXTYPE_SHORT,    /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXOTYPE_SHORT,   /* byte, Int16 */
    0,                /* number of bits to downshift before output */
    cmdptr            /* starting point for command sequence in memory */
  );
```

Figure 6-5. imxenc_array_inner_product



Constraints

- input_width, coeff_width, output_width >= compute_width.
- input_height, coeff_height, output_height >= compute_height.
- compute_height, compute_width, num_terms_horz, num_terms_vert <= 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (9)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} \times \textit{num\_terms\_horiz} \times \textit{num\_terms\_vert} \quad (10)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	1 + 2 / (num_terms_horiz × num_terms_vert)
IMGBUF	IMGBUF	COEFF	1 + 1 / (num_terms_horiz × num_terms_vert)
IMGBUF	COEFF	IMGBUF	1 + 1 / (num_terms_horiz × num_terms_vert)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1 / (num_terms_horiz × num_terms_vert)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1 / (num_terms_horiz × num_terms_vert)

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048

6.1.8 *imxenc_array_op*

imxenc_array_op ***Perform point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) on arrays.***

Syntax

```
cmdlen = imxenc_array_op(
    input1_ptr,          /* Int16*, starting address of  1st input */
    input2_ptr,          /* Int16*, starting address of 2nd input */
    output_ptr,          /* Int16*, starting address of the output array */
    input1_width,        /* Int16, width of 1st input */
    input1_height,       /* Int16, height of 1st input */
    input2_width,        /* Int16, width of 2nd input */
    input2_height,       /* Int16, height of 2nd input */
    output_width,        /* Int16, width of output */
    output_height,       /* Int16, height of output */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input1_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    input2_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    operation,           /* Int16, IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB,
                            IMXOP_AND, IMXOP_OR, IMXOP_XOR,
                            IMXOP_MIN, IMXOP_MAX */
    cmdptr,              /* Int16*, starting point of command sequence in memory*/
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs add, subtract, multiply, absolute difference, and, or, minimum, maximum, between two matrices of dimension [compute_width x compute_height]. The operation is performed on two input matrices resulting in an output matrix.

Actual operand size of [compute_width x compute_height] resides within [input1_width x input1_height] of the first input and within [input2_width x input2_height] of the second input. Actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The pointers, input1_ptr, input2_ptr, and output_ptr, specify the first element, or upper-left corner of actual operands and output.

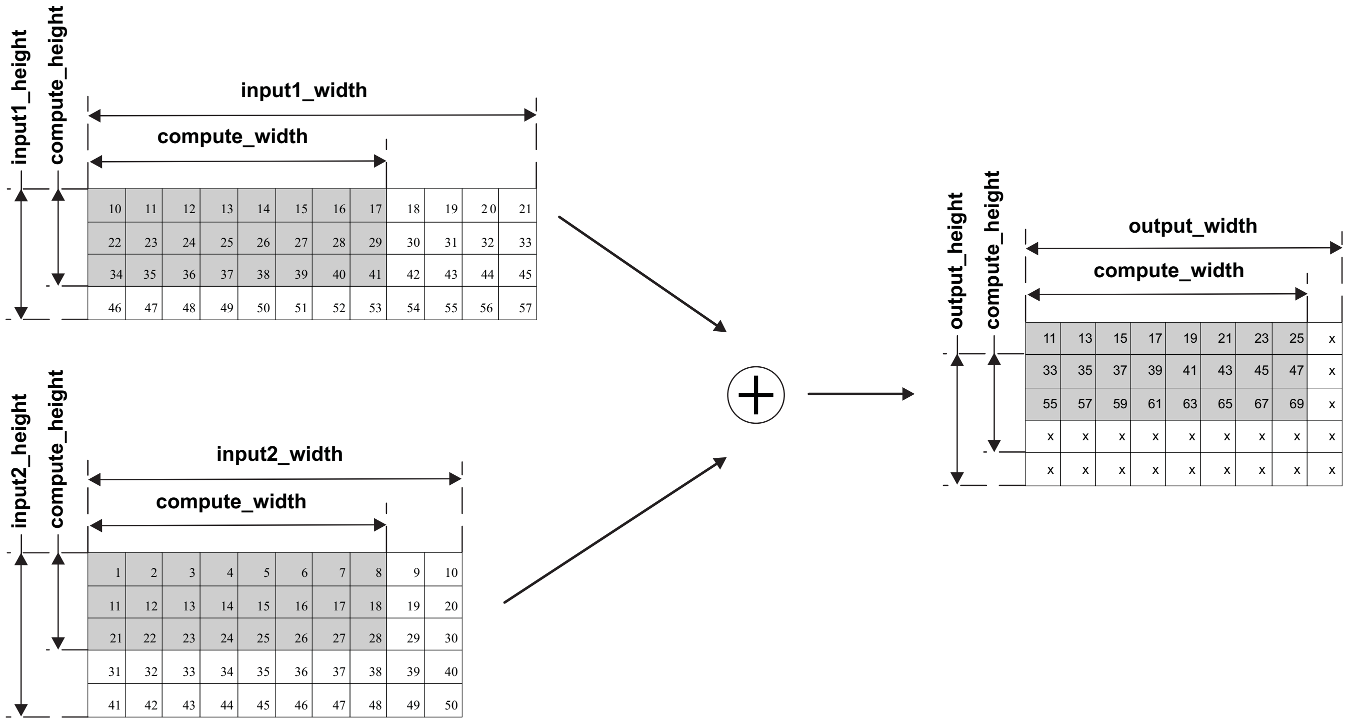
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider input1 → 16x16 matrix, input2 → 12(W)x20(H) matrix and output → 20(W)x13(H) matrix. Choose a matrix addition of 8(W)x10(H) as dimensions. The combination stage performs a max operation per group of 8. The following code illustrates what the API call would look like and Figure 6-6 provides a visual description.

```
cmdlen = imxenc_array_op(
    input1_ptr,      /* starting address of 1st input */
    input2_ptr,      /* starting address of 2nd input */
    output_ptr,      /* starting address of output */
    12,              /* width of 1st input */
    4,               /* height of 1st input */
    10,              /* width of 2nd input */
    5,               /* height of 2nd input */
    9,               /* width of output */
    5,               /* height of output */
    8,               /* computation width */
    3,               /* computation height */
    IMXTYPE_SHORT,   /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    0,               /* number of bits to downshift before output */
    IMXOP_ADD,       /* addition between two matrices */
    *cmdptr          /* starting point for command sequence in memory */
    );
```

Figure 6-6. imxenc_array_op



Constraints

- input1_width, input2_width, output_width >= compute_width
- input1_height, input2_height, output_height >= compute_height
- compute_height and input_height must be < 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \tag{11}$$

- amount_of_work =

$$compute\_width \times compute\_height \tag{12}$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [4]	1	256

[4] That is, compute_width is odd.

6.1.9 imxenc_array_op_distribute

imxenc_array_op_distribute ***Perform point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) between a 4-D data array and a 2-D coefficient array, producing a 4-D output array.***

Syntax

```
cmdlen = imxenc_array_op_distribute(
    input_ptr,             /* Int16, starting address of input */
    coeff_ptr,             /* Int16, starting address of coefficients */
    output_ptr,            /* Int16, starting address of output */
    input_width,           /* Int16, width of input array */
    input_height,          /* Int16, height of input array */
    coeff_width,           /* Int16, width of coefficient array */
    coeff_height,          /* Int16, height of coefficient array */
    output_width,          /* Int16, width of output array */
    output_height,         /* Int16, height of output array */
    compute_width,         /* Int16, computed width */
    compute_height,        /* Int16, computed height */
    operation,             /* IMXOP_ADD, IMXOP_SUB,IMXOP_MPY, IMXOP_ABDF,
                           /* IMXOP_AND, IMXOP_OR, IMXOP_XOR, */
                           /* IMXOP_MIN, IMXOP_MAX */
    num_distribute_horz,   /* Int16, number of inner arrays horizontally */
    num_distribute_vert,   /* Int16, number of inner arrays vertically */
    input_offset_horz,     /* Int16, horizontal offset of input arrays in data pts */
    input_offset_vert,     /* Int16, vertical offset of input arrays in data pts */
    output_offset_horz,    /* Int16, horizontal offset of output arrays in data pts */
    output_offset_vert,    /* Int16, vertical offset of output arrays in data pts */
    input_type,            /* Int16, Int16/byte, signed/unsigned */
    coeff_type,            /* Int16, Int16/byte, signed/unsigned */
    output_type,           /* Int16, Int16/byte */
    round_shift,           /* Int16, number of bits to downshift before output */
    cmdptr                 /* Int16*, starting point of command sequence in memory */
    );
```

Description

This function performs add, subtract, multiply, absolute difference, pack bytes, and, or, mask, mask-not, minimum, maximum, packed-bytes absolute differences between a 4-D data array and a 2-D coefficient array. The coefficient array is replicated and distributed to make up the outer 2 dimensions. A 4-D output array is produced.

For the inner 2-D arrays, the actual operand size of [compute_width x compute_height] resides within [input_width x input_height] of data input and within [coeff_width x coeff_height] of coefficient input, and actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The outer two dimensions are indexed with horizontal and vertical offsets, and the offsets are in data points (not the address offsets). The pointers, input_ptr, coeff_ptr, and output_ptr, specify the first element, or upper-left corner of operands and output. For example, logical data item input[m, n, i, j] is assumed to reside at input_ptr[m * input_offset_vert + n * input_offset_horz + i * input_width + j]. The xxx_height parameters are included for consistency, but are not used in any address calculation.

The outer dimensions of input can optionally be used to index overlapping sub-arrays in an input array. For example, using input_offset_horz = input_offset_vert = 1, and num_distribute_horz = num_distribute_vert = 3 addresses a 3x3 neighborhood for each inner-array data point.

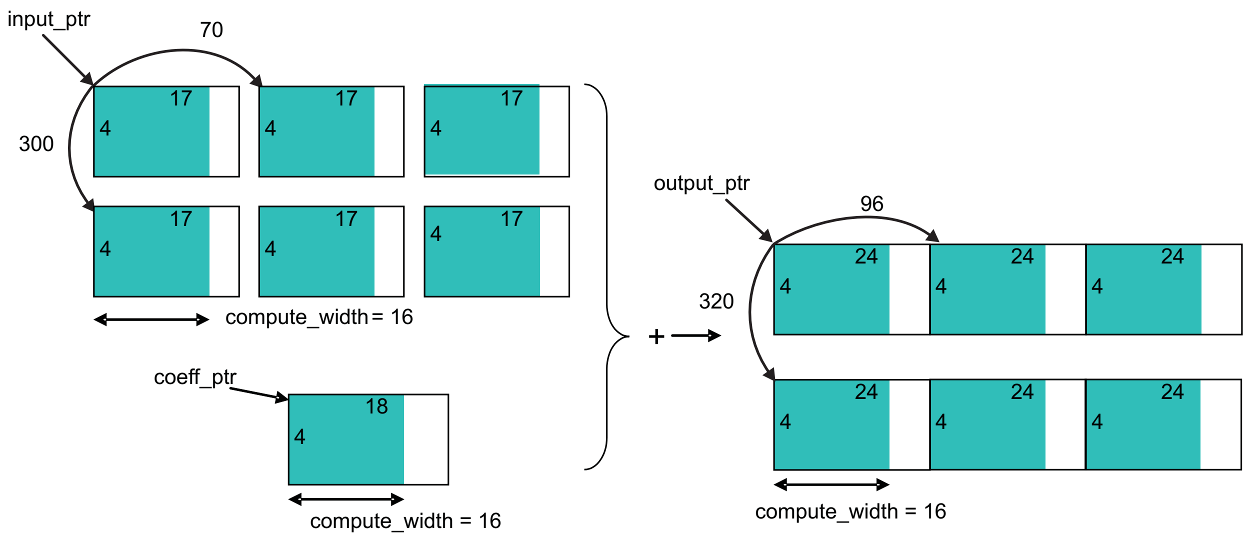
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider data input → 2(outerH) x 3(outerW) x 4(H) x 17(H) matrix, coefficient input → 4(W) x 18(H) matrix and output → 2(outerH) x 3(outerW) x 4(H) x 24(W) matrix. Chose to have a matrix addition of 2(outerH) x 3(outerW) x 4(H) x 16(W) as dimensions. The following code illustrates what the API call would look like and Figure 6-7 provides a visual description.

```
cmdlen = imxenc_array_op_distribute(
    input_ptr,        /* starting address of 1st input */
    coeff_ptr,        /* starting address of 2nd input */
    output_ptr,       /* starting address of output */
    17,               /* width of data input */
    4,                /* height of data input */
    18,               /* width of coefficient input */
    4,                /* height of coefficient input */
    24,               /* width of output */
    4,                /* height of output */
    16,               /* computation width */
    4,                /* computation height */
    IMXOP_ADD,        /* addition between two matrices */
    3,                /* outer width */
    2,                /* outer height */
    70,               /* input horizontal offset */
    300,              /* input vertical offset */
    96,               /* output horizontal offset */
    320,              /* output vertical offset */
    IMXTYPE_SHORT,    /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXOTYPE_SHORT,   /* byte, Int16 */
    0,                /* number of bits to downshift before output */
    cmdptr            /* starting point for command sequence in memory */
    );
```

Figure 6-7. imxenc_array_op_distribute



Constraints

- input1_width, coeff_width, output_width >= compute_width
- input1_height, coeff_height, output_height >= compute_height
- compute_height, compute_width, num_distribute_horz, num_distribute_vert <= 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

amount_of_work × *memory_conflict_factor* / *speedup_factor* (13)

- amount_of_work =

compute_width × *compute_height* × *num_distribute_horz* × *num_distribute_vert* (14)

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048

6.1.10 imxenc_array_scalar_op

imxenc_array_scalar_op ***Perform each point-by-common point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) between an array an a scalar.***

Syntax

```
cmdlen = imxenc_array_scalar_op(
    input1_ptr,          /* Int16*, starting address of  1st input */
    input2_ptr,          /* Int16*, starting address of 2nd input */
    output_ptr,          /* Int16*, starting address of the output array */
    input1_width,        /* Int16, width of 1st input */
    input1_height,       /* Int16, height of 1st input */
    input2_width,        /* Int16, width of 2nd input */
    input2_height,       /* Int16, height of 2nd input */
    output_width,        /* Int16, width of output */
    output_height,       /* Int16, height of output */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input1_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    input2_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    operation,           /* Int16, IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB, */
                         /* IMXOP_AND, IMXOP_OR, IMXOP_XOR*/
                         /* IMXOP_MIN, IMXOP_MAX*/
    cmdptr,             /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs operations (add, subtract, multiply, absolute difference, pack bytes, and, or, mask, mask-not, minimum, maximum, packed-bytes absolute differences) between a matrix of dimension [compute_width x compute_height] and a scalar. The scalar operand can be as large as a 2 x 2 matrix. Whatever its size, it is replicated and tiled horizontally and vertically to the same size as the first operand before applying the point-by-point operation.

Actual operand size of [compute_width x compute_height] resides within [input1_width x input1_height] 1st input. The 2nd input is a single value, 1x2, 2x1, or 2x2 matrix pointed by input2_ptr. Actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The pointers, input1_ptr and output_ptr, specify the first element, or upper-left corner of actual operand and output.

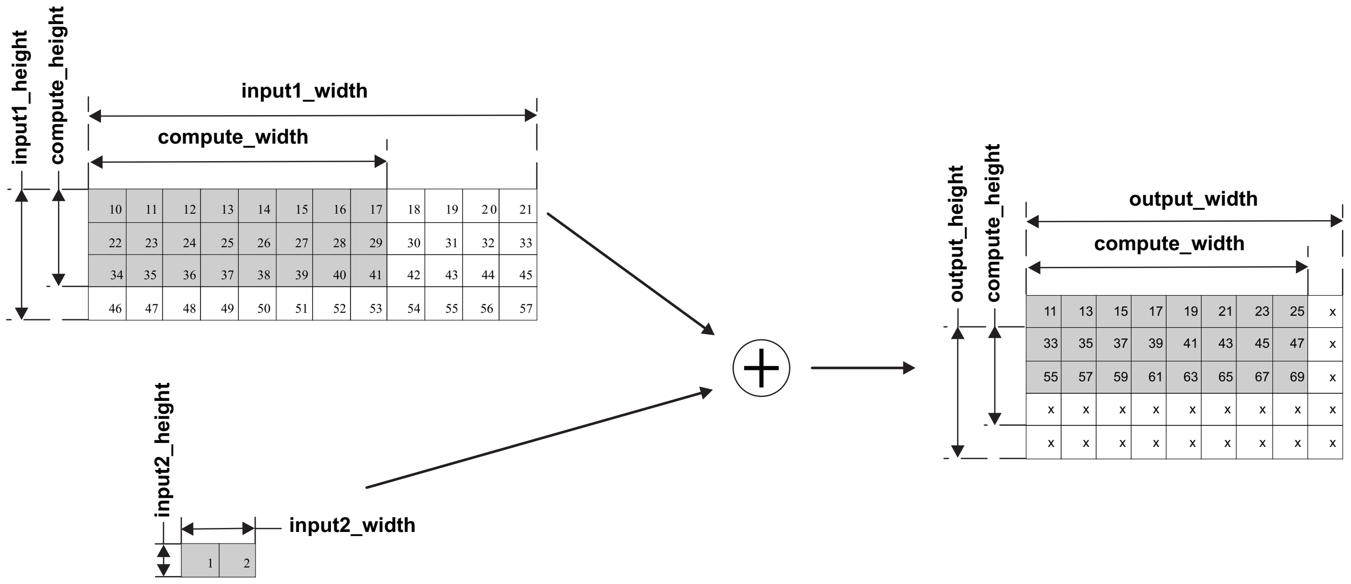
Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Some limited signed/unsigned combinations are also supported. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider input1 → 16x16 matrix and output → 20(W)x13(H) matrix. Choose to multiply a submatrix of size 8(W)x10(H) within input1 with a scalar. Round down by 8 bits. The following code illustrates what the API call would look like and Figure 6-8 provides a visual description.

```
cmdlen = imxenc_array_scalar_op(
    input1_ptr,      /* starting address of 1st input */
    input2_ptr,      /* starting address of 2nd input */
    output_ptr,      /* starting address of output */
    12,              /* width of 1st input */
    4,               /* height of 1st input */
    2,               /* width of 2nd input */
    1,               /* height of 2nd input */
    9,               /* width of output */
    5,               /* height of output */
    8,               /* computation width */
    3,               /* computation height */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    0,               /* number of bits to downshift before output */
    IMXOP_ADD        /* IMXOP_ADD */
    cmdptr           /* starting address for the command sequence */
);
```

Figure 6-8. imxenc_array_scalar_op



Constraints

- input1_width, output_width >= compute_width
- input1_height, output_height >= compute_height
- compute_width multiple of input2_width
- compute_height multiple of input2_height
- compute_height and input_height must be < 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \quad (15)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (16)$$

- memory_conflict_factor if input2_height=1:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 16 / compute_width × compute_height
IMGBUF	IMGBUF	COEFF	1 + 8 / compute_width × compute_height
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / compute_width × compute_height
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 / compute_width × compute_height

- memory_conflict_factor if input2_height=2:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 16 / compute_width
IMGBUF	IMGBUF	COEFF	1 + 8 / compute_width
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / compute_width
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 / compute_width

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [5]	1	256

[5] That is, compute_width is odd.

6.1.11 imxenc_average2x2

imxenc_average2x2 ***This function partitions the input array into sub-blocks of 2x2 data points and produces the sum of the 4 points in each sub-block.***

Syntax

```
cmdlen = imxenc_average2x2(
    input_ptr,           /* Int16*, starting address of input */
    output_ptr,          /* Int16*, starting address of the output array */
    input_width,         /* Int16, width of input */
    output_width,        /* Int16, width of output */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    cmdptr,            /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function calculates the sum of every sub-blocks of 2x2 data points. If round_shift is set to 2 then the average of every 2x2 data points is produced. An input array of MxN elements produces an output array of size (M/2)x(N/2). computation_width and computation_height correspond to the original width and height of the region of interest in the input array that is going to be reduced.

The input_height and output_height information does not need to be passed to imxenc_average2x2() since this information is not needed for the computation.

This function can be used for downscaling a 2-D image by a factor of 2 along each dimension, resulting in a factor of 4 downscaling. It can be used inside a simple implementation of image pyramid.

Example

Consider an input array of 19x20 unsigned shorts, in which a region of interest of 16x18 elements is going to be downscaled into 8x9 elements fitting in a larger 11x10 output array.

```
cmdlen = imxenc_average2x2(
    input_ptr,         /* starting address of input */
    output_ptr,        /* starting address of output */
    19,                /* width of input */
    11,                /* width of output */
    16,                /* computation width */
    18,                /* computation height */
    IMXTYPE_USHORT,    /* Int16 */
    IMXOTYPE_SHORT,    /* Int16 */
    2                  /* divide by 4 to produce average */
    cmdptr             /* starting address for the command sequence */
);
```

Constraints

- input_width, output_width/2 >= compute_width
- compute_width must be an even number
- compute_height/2 must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \tag{17}$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} / 2 \tag{18}$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	2	512
2	2	512

6.1.12 imxenc_bin_log

imxenc_bin_log ***Calculate a binary logarithm of the input data.***

Syntax

```
cmdlen =  imxenc_bin_log(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* width/columns of 1st input */
    Int16 input_height,    /* height/rows of 1st input */
    Int16 output_width,    /* width/columns of output */
    Int16 output_height,   /* height/rows of output */
    Int16 compute_width,   /* computation width */
    Int16 compute_height,  /* computation height */
    Int16 output_type,     /* Int16/byte */
    Int16 weighting,       /* 0=bottom-weighted, 1=top-and-bottom-weighted */
    Int16 output_offset,   /* 0=output binary log, 1=output bin log offset */
    Int16 rnd_shift,       /* Shifting parameter */
    Int16 *cmdptr),
);
/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

This function computes an approximate binary logarithm of the input data. The logarithm can be bottom weighted or top and bottom weighted. The input is always unsigned data. The input data is first saturated to get rid of the top 4 bits. The next six bits are detected for the most significant bit. This produces a four-bit index, 0...6 for bottom-weighted and 0...11 for top-and-bottom weighted. Table 6-4 shows how the index is created.

Table 6-4. Creation of Index for imxenc_bin_log

Bits 11...6	Bottom-Weighted	Top-and-Bottom-Weighted
000000	0	0
000001	1	1
00001F	2	2
0001FF	3	3
001FFF	4	4
01FFFF	5	5
10FFFF	6	6
110FFF	6	7
1110FF	6	8
11110F	6	9
111110	6	10
111111	6	11

After the index is computed, the fractional bits are extracted. The fractional bits (up to 11 bits) are the bits to the right of the most significant bit. The fraction bits are padded to 11 bits by left shifting and then appended to the four-bit index.

If output_offset is 0, the output is computed by right shifting by the rnd_shift parameter. If output_offset is 1 then the output is the rnd_shift LSBs of the index and fraction bits.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / 8 \quad (19)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (20)$$

- memory_conflict_factor:

Location of input1	Location of zero	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 8 / (compute_height × compute_width)
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / (compute_height × compute_width)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 /(compute_height × compute_width)

6.1.13 imxenc_blkAverage

imxenc_blkAverage ***Calculate the average***

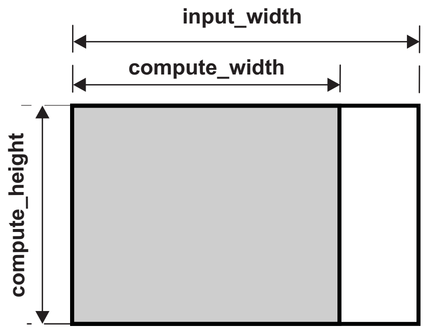
Syntax

```
cmdlen = imxenc_blkAverage(
    input_ptr,       /* Int16*, starting address of current block */
    zero_ptr,        /* Int16*, points to a Int16 0 */
    output_ptr,      /* Int16*, address where result will be stored */
    input_width,     /* Int16, input width of entire block */
    compute_width,   /* Int16, width of subsection being operated on */
    compute_height,  /* Int16, height of subsection being operated on */
    shift,           /* Int16, amount to shift for averaging */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */

    cmd_ptr,         /* Int16*, starting point of command sequence in memory */
);
/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function calculates the average value in an area of size compute_width*compute_height, which is within of an area of bigger size input_width*compute_height.

Figure 6-9. imxenc_blkAverage



Shift must be selected using the following equation:

$$shift = \log_2(block\_width \times sub\_height) \quad (21)$$

The result will be a single number of type Int16 or byte (depends on output_type).

Example

The current block is stored in data memory. The total block width is 32, and total block height is 16. The sub-block has dimensions 16x16. The resulting shift value is 8. The result will be stored in data memory. The VICP command would be:

```
cmdlen = imxenc_blkAverage (
    data_ptr,
    zero_ptr,
    result_ptr,
    32,
    16,
    16,
    8,
    IMXTYPE_UBYTE,
    IMXTYPE_SHORT,
    cmd_ptr
);
```

Constraints

compute_width must be multiple of 8 and inferior or equal to 2048.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} / 8 \tag{22}$$

- amount_of_work =

$$\mathit{compute\_width} \times \mathit{compute\_height} \tag{23}$$

- memory_conflict_factor:

Location of input1	**Location of zero**	**Location of output**	**memory_conflict_factor**
IMGBUF	IMGBUF	IMGBUF	2 + 8 / (compute_height × compute_width)
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / (compute_height × compute_width)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 /(compute_height × compute_width)

6.1.14 imxenc_blkVariance

imxenc_blkVariance ***Calculates the variance of block.***

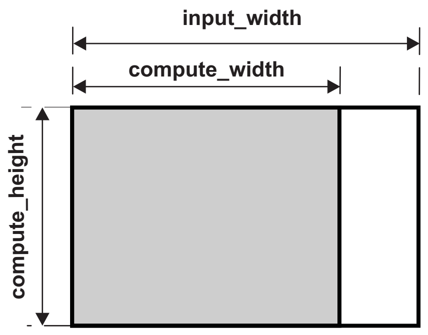
Syntax

```
cmdlen = imxenc_blkVariance(
    input_ptr,       /* Int16*, starting address of current block */
    avg_ptr,         /* Int16*, points to address where average value of block
                        is stored */
    output_ptr,      /* Int16*, address where result will be stored */
    input_width,     /* Int16, input width of entire block */
    compute_width,   /* Int16, width of subsection being operated on */
    compute_height,  /* Int16, height of subsection being operated on */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    average_type,    /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */

    cmd_ptr,         /* Int16*, starting point of command sequence in memory */
       );
/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function calculates the variance vin an area of size compute_width*compute_height, which is within of an area of bigger size input_width*compute_height. It requires that the average be calculated prior to calling this function. It sums the absolute differences between each value of the area and the average value pointed by avg_ptr.

Figure 6-10. imxenc_blkVariance



Example

The current block is stored in data memory. The average value is stored at the first location in coefficient memory. The total block width is 32, and total block height is 16. The sub-block has dimensions 16x16. The result will be stored in data memory. The VICP command would be:

```
cmdlen = imxenc_blkVariance (
    data_ptr,
    avg_ptr,
    result_ptr,
    32,
    16,
    16,
    IMXTYPE_BYTE,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,
    cmdptr
);
```

Constraints

compute_width must be multiple of 8 and inferior or equal to 2048.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} / 8 \qquad (24)$$

- amount_of_work =

$$\mathit{compute\_width} \times \mathit{compute\_height} \qquad (25)$$

- memory_conflict_factor:

Location of input1	Location of zero	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 8 / (compute_height × compute_width)
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / (compute_height × compute_width)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 /(compute_height × compute_width)

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [6]	1	256

[6] That is, compute_width is odd.

6.1.15 imxenc_blkSeq2Array

imxenc_blkSeq2Array ***Reorganize a sequence of NxN blocks into an array (almost complementary function to imxenc_y2blkseq())***

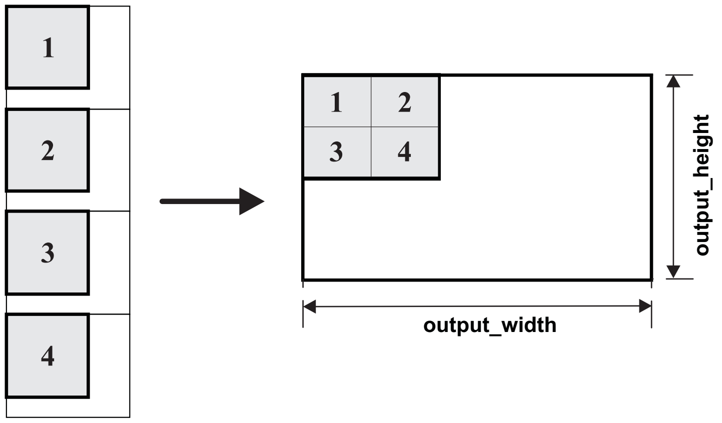
Syntax

```
cmdlen = imxenc_blkSeq2Array(
    data_ptr,          /* Int16*, starting address of data */
    coeff_ptr,         /* Int16*, starting address of coefficient */
    output_ptr,        /* Int16*, starting address of the output */
    input_blksize,     /* Int16, input size of the square blocks */
    compute_blksize,   /* Int16, compute size of the square blocks */
    output_width,      /* Int16, width of the output array */
    output_height,     /* Int16, height of the output array */
    no_blks_x,         /* Int16, Number of processed input blocks horizontally */
    no_blks_y,         /* Int16, Number of processed input blocks vertically */
    input_type,        /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                       /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,        /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                       /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,       /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,       /* Int16, number of bits to downshift before output */
    cmdptr             /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function reorganizes the data from a block sequential fashion into an array:

Figure 6-11. imxenc_blkSeq2Array



The gray blocks at the input side are square blocks of size compute_blksize × compute_blksize and the plain blocks surrounding them are of size input_blksize × input_blksize.

At output side, the gray area is the effective output array whose width is no_blks_x*compute_blks_size and height is no_blks_y*compute_blks_size.

Normally, only the data is reorganized and the values are not changed, so the Int16 word pointed by coeff_ptr contains 1, and round_shift = 0. Otherwise, during the reorganization, each element of the output array is scaled by (*coeff_ptr / 2^round_shift).

This function behaves as the inverse of imxenc_y2blkseq() if input_blksize=compute_blksize=8.

For backward compatibility with old API imxenc_blkseq2y(), a macro is provided in vicp_comp.h.

Example

Mapping a 8x(8x8) block into 16x16 block.

```
cmdlen = imxenc_blkSeq2Array(
    data_ptr,        /* point to input data array */
    coeff_ptr,       /* point to scalar scaling factor */
    output_ptr,      /* point to output array */
    8,               /* input block */
    8,               /* computation block */
    16,              /* width of output data array */
    16,              /* height of output data array */
    2,               /* Number of processed input blocks horizontally */
    2,                  /* Number of processed input blocks vertically  */
    IMXTYPE_SHORT,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,
    0,                  /* number of bits to downshift  */
    cmdptr           /* starting point for command sequence in memory */
 );
```

Performance

The overhead time for this VICP API is ~ 30 cycles. The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (26)$$

- amount_of_work =

$$compute\_blksize \times compute\_blksize \times no\_blks\_x \times no\_blks\_y \quad (27)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	(2 + 8 / (no_blks_x × no_blks_y × compute_blksize × compute_blksize))
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	(1 + 8 / (no_blks_x × no_blks_y × compute_blksize × compute_blksize))
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	(1 + 8 /(no_blks_x × no_blks_y × compute_blksize × compute_blksize))

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [7]	1	256

[7] That is, compute_width is odd.

6.1.16 imxenc_cfa_fast

imxenc_cfa_fast ***Perform fast CFA interpolation using 3x3 bilinear filter***

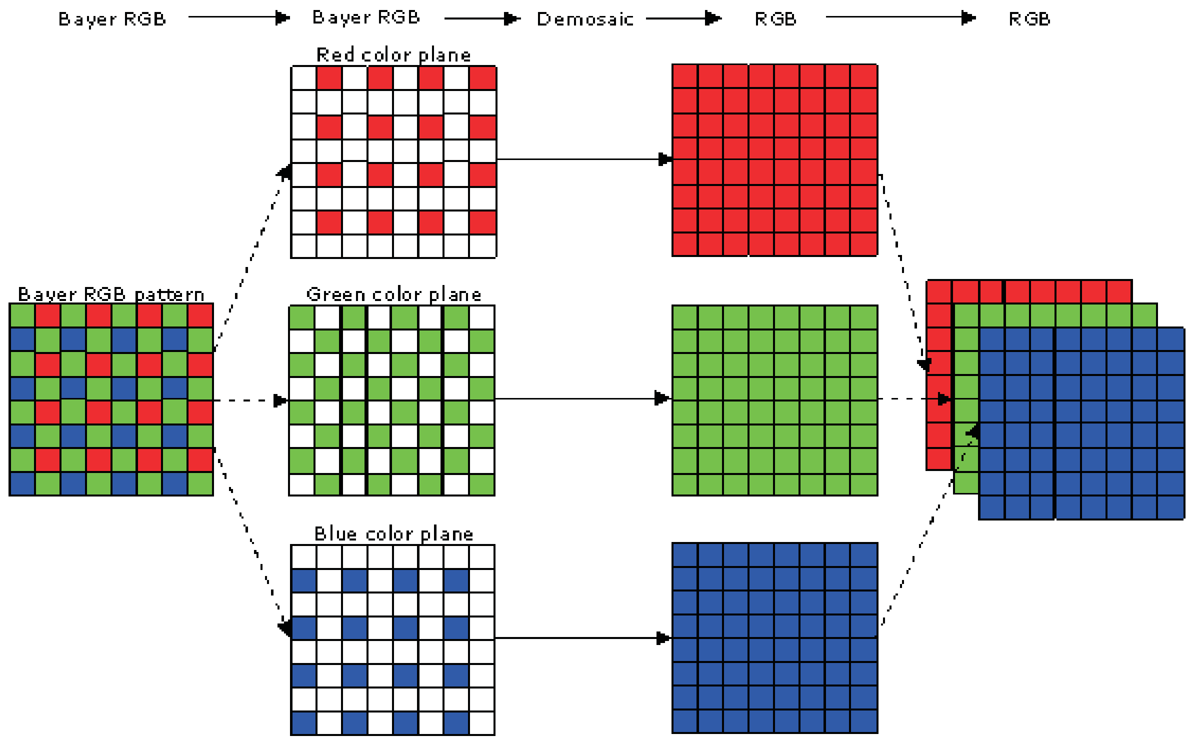
Syntax

```
cmdlen = imxenc_cfa_fast(
    input_p,        /* Int16*, starting address of  1st input */
    outputr_p,      /* Int16*, starting address of the output array,red */
    outputg_p,      /* Int16*, starting address of the output array,green */
    outputb_p,      /* Int16*, starting address of the output array,blue */
    scratch_ptr,    /* Int16*, starting address of scratch buffer of size */
                    /* (compute_width+2) * compute_height elements of type output_type */
    coeff_ptr,      /* Int16*, Points to 4 values [coef_r, coef_g, coef_g, coef_b] of
                    /*   type IMXTYPE_SHORT */
    zero_ptr,       /* Int16*, Point to a zero value of type IMXTYPE_SHORT */
    input_width,    /* Int16, width of the input array */
    input_height,   /* Int16, height of the input array */
    output_width,   /* Int16, width of the output array */
    output_height,  /* Int16, height of the output array */
    compute_width   /* Int16, computation width */
    compute_height, /* Int16, computation height */
    input_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,    /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,    /* Int16, number of bits to downshift before output, recommended to be 1 */
    phase,          /* Phase of the Bayer pattern */
    cmdptr          /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

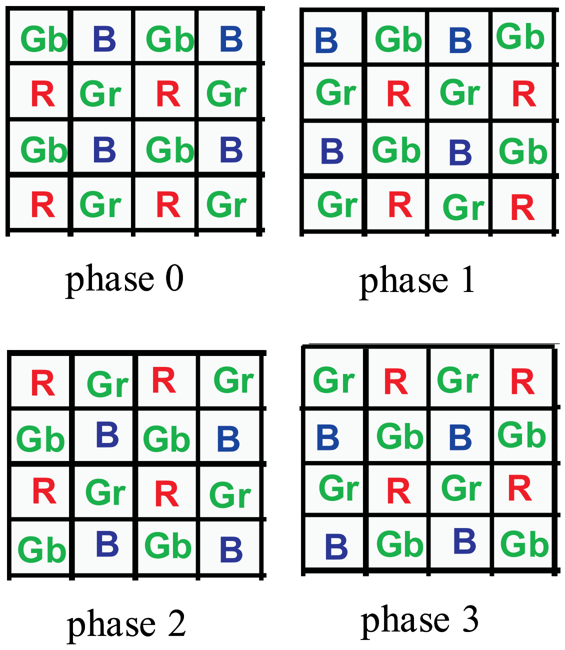
This function performs an interpolation of a CFA image into a full-resolution RGB-image. The CFA-pattern is assumed to be a Bayer pattern. The output is composed of three planes, r, g, b of dimensions output_width x output_height elements. Each plane r, g, b is scaled by coef_r, coef_g, coef_g, respectively. Typically coef_r = coef_g= coef_b= 1 but in case luminance value needs to be obtained, the following values can be used: coef_r= $0.2126^{\text{round\_shift}}$, coef_g= $0.7152^{\text{round\_shift}}$, coef_b= $0.0722^{\text{round\_shift}}$.

Figure 6-12. imxenc_cfa_fast



The meaning of the phase argument is illustrated in Figure 6-13.

Figure 6-13. imxenc_cfa_fast phase Argument



Constraints

- compute_width must be a multiple of 8.
- input_width >= compute_width + 2, output_width >= compute_width
- input_height >= compute_ height + 2, output_ height >= compute_ height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed.
- compute_height and input_height must be < 256

The current implementation only supports input_width = compute_width + 2, input_height = compute_height + 2, output_width = compute_width, output_height = input_height.

Performance

Performance is around 3 cycles per input pixels for input and scratch in image buffer, output and coefficients in coefficient memory.

6.1.17 *imxenc_cfa_hq_interpolation*

imxenc_cfa_hq_interpolation *Perform high-quality CFA interpolation*

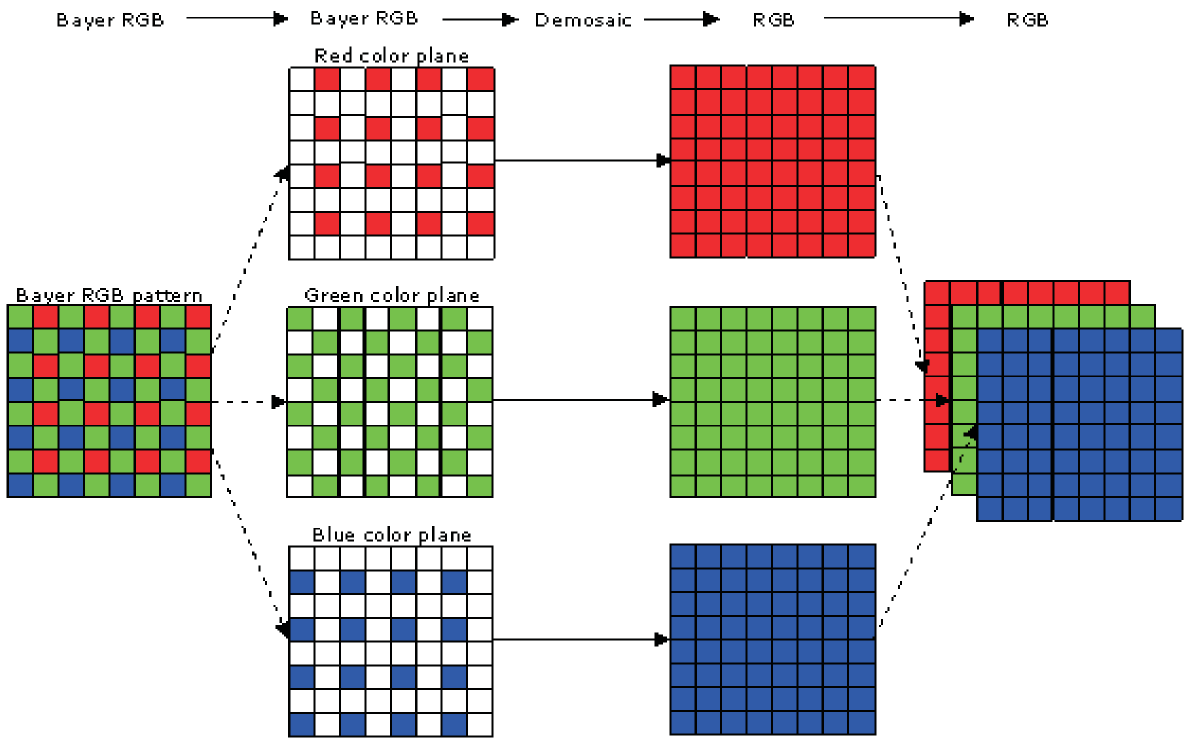
Syntax

```
cmdlen = imxenc_cfa_hq_interpolation(
    input_p,          /* Int16*, starting address of  1st input */
    coeff_p,          /* Int16*, starting address of 2nd input */
    outputr_p,        /* Int16*, starting address of the output array,red */
    outputg_p,        /* Int16*, starting address of the output array,green */
    outputb_p,        /* Int16*, starting address of the output array,blue */
    input_width,      /* Int16, width of the input array */
    input_height,     /* Int16, height of the input array */
    coeff_width,      /* Int16, width of the filter kernel */
    coeff_height,     /* Int16, height of the filter kernel */
    output_width,     /* Int16, width of the output array */
    output_height,    /* Int16, height of the output array */
    compute_width,    /* Int16, computation width */
    compute_height,   /* Int16, computation height */
    input_type,       /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                      /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,       /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                      /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,      /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,      /* Int16, number of bits to downshift before output */
    cmdptr            /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr *
```

Description

This function performs an interpolation of a CFA image into a full-resolution RGB-image. The CFA-pattern is assumed to be a Bayer pattern.

Figure 6-14. imxenc_cfa_hq_interpolation



The coefficients used in the filtering must be setup by calls to imx_cfa_hq_setup().

Example

```
qShift= 9;
phase= 0;

imx_cfa_hq_setup(coeff_ptr, VICP_CFA_HQ_5x5_COEFS, phase, qShift);

cmdlen = imxenc_set_saturation(255, 255, 0, 0, cmdptr);

cmdlen += imxenc_cfa_hq_interpolation(  in,
                                        coeff_ptr,
                                        rgb_p[0],
                                        rgb_p[1],
                                        rgb_p[2],
                                         inWidth,
                                         inHeight,
                                         5,
                                         5,
                                         outWidth,
                                         outHeight,
                                         dataWidth,
                                         dataHeight,
                                         IMXTYPE_SHORT, IMXTYPE_SHORT,
                                         IMXOTYPE_SHORT,
                                         qShift,
                                         cmdptr + cmdlen);
```

Constraints

- compute_width must be a multiple of 8.
- input_width >= compute_width + coeff_width – 1, output_width >= compute_width
- input_height >= compute_ height + coeff_ height – 1, output_ height >= compute_ height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed.
- compute_height and input_height must be < 256

Performance

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (28)$$

- amount_of_work=

$$3 \times coeff\_width \times coeff\_height \times compute\_width \times compute\_height \quad (29)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1 / (coeff_width × coeff_height)
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1 / (coeff_width × coeff_height)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1 / (coeff_width × coeff_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1 / (coeff_width × coeff_height)

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [8]	1	256

[8] That is, compute_width is odd.

6.1.18 imxenc_cfa_upsmpl_horz

imxenc_cfa_upsmpl_horz *Perform horizontal upsampling of CFA data*

Syntax

```
cmdlen = imxenc_cfa_upsmpl_horz(
    input_ptr,       /* Int16*, starting address of input */
    coeff_ptr,       /* Int16*, starting address of coefficients */
    output_ptr,      /* Int16*, starting address of output */
    input_width,     /* Int16, width of input array */
    input_height,    /* Int16, height of input array */
    output_width,    /* Int16, width of output array */
    output_height,   /* Int16, height of output array */
    compute_width,   /* Int16, computed width */
    compute_height,  /* Int16, computed height */
    upsmpl_horz,     /* Int16, number of arrays horizontally to combine */
    input_type,      /* Int16, Int16/byte, signed/unsigned */
    coeff_type,      /* Int16, Int16/byte, signed/unsigned */
    output_type,     /* Int16, Int16/byte */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr );        /* Int16*, starting point of command sequence in memory */
```

Description

This function performs horizontal upsampling of CFA data. It upsamples by any multiple-of-4 integer factor. Input/output are assumed to be in CFA pattern. The 4 phases (even/odd pixel on even/odd row) are processed independently and using the same set of filter coefficients. The CFA pattern can be any color pattern as long as it's organized in 2x2 tiling. Simple 2-tap bilinear interpolation is assumed.

The function takes (compute_width + 2) x compute_height of input data, and upsamples into (compute_width * upsmpl_horz) x compute_height output. Wider input/output arrays can be specified to skip over garbage data. Since CFA pattern is assumed, compute_width and compute_height must each be a multiple of 2. The extra two columns of input are needed to provide anchors so that the program always upsamples by interpolating between two input points.

Coefficient array is organized output-phase-first, and input-tap on the outer dimension. Normally linear interpolation is used, and thus the following coefficients can be used, where U = horizontal upsampling factor, and L is the number of bits coefficients are quantized to:

```
coeff[i] = 2^L*(U-i)/U,
coeff[U + i] = 2^L*i/U,  i = 0..U-1
```

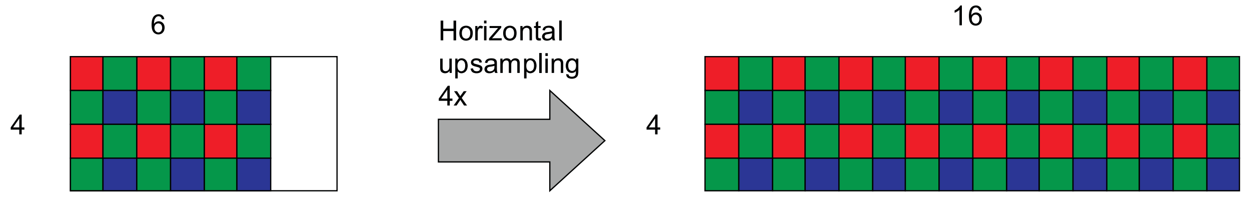
The input array and the coefficient array can each be signed or unsigned, 16-bit (Int16) or 8-bit (byte) per element. The output array can be either 16-bit or 8-bit per element.

Example

Consider data input = 6 (wide) x 4 (tall), and the goal to upsample 4x horizontally to obtain 16 (wide) x 4 (tall). Further assume that the input array resides in a 2-D array of 8 elements in width. The following code illustrates the coefficient array and the API call. Figure 6-15 provides a visual description.

```
Int16 coeff_ptr[ ] = {4, 3, 2, 1, 0, 1, 2, 3};  /* U=4, L=2 */
cmdlen = imxenc_cfa_upsmpl_horz(
    input_ptr,      /* starting address of 1st input */
    coeff_ptr,      /* starting address of 2nd input */
    output_ptr,     /* starting address of output */
    8,              /* width of data input */
    4,              /* height of data input */
    16,             /* width of output */
    4,              /* height of output */
    4,              /* computation width */
    4,              /* computation height */
    4,              /* upsampling factor */
    IMXTYPE_SHORT,  /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,  /* byte, Int16 */
    IMXOTYPE_SHORT, /* byte, Int16 */
    2,              /* number of bits to downshift before output */
    cmdptr          /* starting point for command sequence in memory */
);
```

Figure 6-15. imxenc_cfa_upsmpl_horz



Constraints

- compute_width, compute_height are multiples of 2
- input_width >= compute_width + 2
- output_width >= compute_width * upsmpl_horz
- input_height, output_height >= compute_height
- compute_height, compute_width, upsmpl_horz <= 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (30)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} \times \textit{upsmpl\_horz} \times 2 \quad (31)$$

- speedup_factor = 8

6.1.19 imxenc_cfa_upsmpl_vert

imxenc_cfa_upsmpl_vert *Perform vertical upsampling of CFA data*

Syntax

```
cmdlen = imxenc_cfa_upsmpl_vert(
    input_ptr,       /* Int16*, starting address of input */
    coeff_ptr,       /* Int16*, starting address of coefficients */
    output_ptr,      /* Int16*, starting address of output */
    input_width,     /* Int16, width of input array */
    input_height,    /* Int16, height of input array */
    output_width,    /* Int16, width of output array */
    output_height,   /* Int16, height of output array */
    compute_width,   /* Int16, computed width */
    compute_height,  /* Int16, computed height */
    upsmpl_vert,     /* Int16, number of arrays horizontally to combine */
    input_type,      /* Int16, Int16/byte, signed/unsigned */
    coeff_type,      /* Int16, Int16/byte, signed/unsigned */
    output_type,     /* Int16, Int16/byte */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);
```

Description

This function performs vertical upsampling of CFA data. It upsamples by any integer factor. Input/output are assumed to be in CFA pattern. The 4 phases (even/odd pixel on even/odd row) are processed independently and using the same set of filter coefficients. The CFA pattern can be any color pattern as long as it's organized in 2x2 tiling. Simple 2-tap bilinear interpolation is assumed.

The function takes compute_width x (compute_height + 2) of input data, and upsamples into compute_width x (upsmpl_vert * compute_height) output. Wider input/output arrays can be specified to skip over garbage data. Since CFA pattern is assumed, compute_width and compute_height must each be a multiple of 2. In addition, compute_width must be a multiple of 8 to work with the parallelism of VICP. The extra two rows are needed to provide anchors so that the program always upsamples by interpolating between two input points.

Coefficient array is organized input-tap-first, and output-phase on the outer dimension. Normally linear interpolation is used, and thus the following coefficients can be used, where U = vertical upsampling factor, and L is the number of bits coefficients are quantized to:

```
coeff[2*i] = 2^L*(U-i)/U,
coeff[2*i+1] = 2^L*i/U,    i = 0..U-1
```

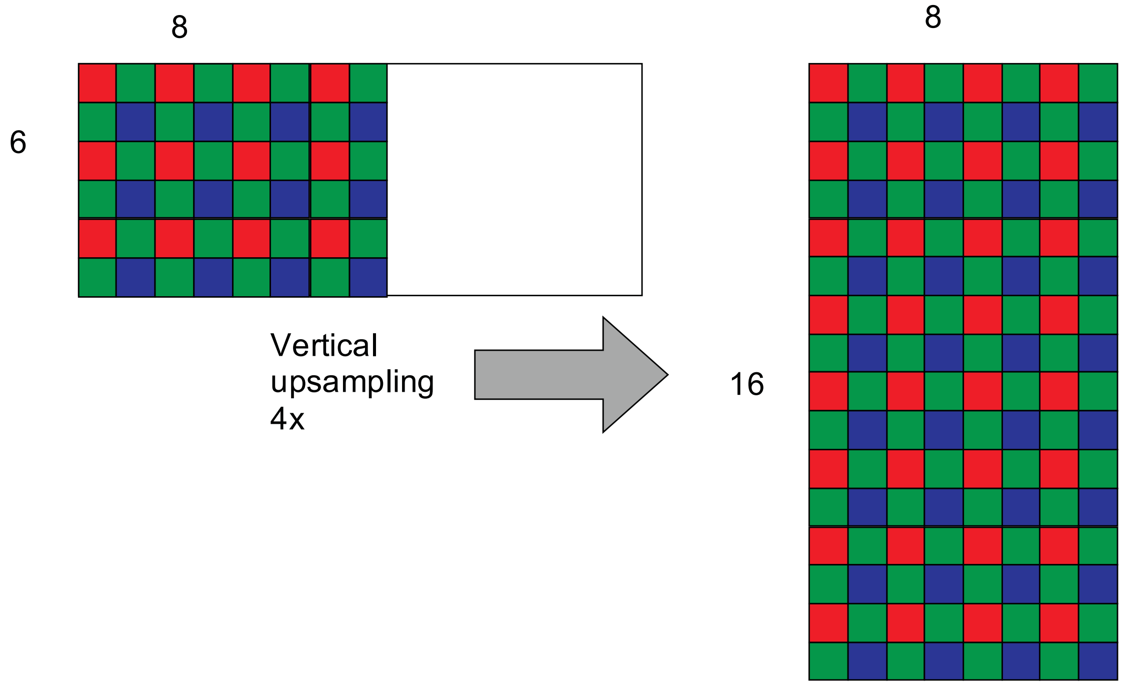
The input array and the coefficient array can each be signed or unsigned, 16-bit (Int16) or 8-bit (byte) per element. The output array can be either 16-bit or 8-bit per element. Rounding shift (normally set to L) is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider data input = 8 (wide) x 6 (tall), and the goal to upsample 4x vertically to obtain 8 (wide) x 16 (tall). Further assume that the input array resides in a 2-D array of 16 elements in width. The following code illustrates the coefficient array and the API call. Figure 6-16 provides a visual description.

```
Int16 coeff_ptr[ ] = {4, 3, 2, 1, 0, 1, 2, 3};  /* U=4, L=2 */
cmdlen = imxenc_cfa_upsmpl_vert(
    input_ptr,       /* starting address of 1st input */
    coeff_ptr,       /* starting address of 2nd input */
    output_ptr,      /* starting address of output */
    16,              /* width of data input */
    6,               /* height of data input */
    8,               /* width of output */
    16,              /* height of output */
    8,               /* computation width */
    4,               /* computation height */
    4,               /* upsampling factor */
    IMXTYPE_SHORT,   /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    2,               /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Figure 6-16. imxenc_cfa_upsmpl_vert



Constraints

- compute_width is a multiple of 8, compute_height is a multiple of 2
- input_width, output_width >= compute_width
- input_height >= compute_height + 2
- output_height >= compute_height × upsmpl_vert
- compute_height, compute_width, upsmpl_vert <= 256

Performance The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (32)$$

- amount_of_work =

$$compute\_width \times compute\_height \times upsmpl\_vert \times 2 \quad (33)$$

- speedup_factor = 8

6.1.20 imxenc_color_spc_conv

imxenc_color_spc_conv *Perform color space conversion on array*

Syntax

```
cmdlen = imxenc_color_spc_conv(
    input_ptr,          /* Int16*, starting address of  1st input */
    coeff_ptr,          /* Int16*, starting address of 2nd input */
    output_ptr,         /* Int16*, starting address of the output array */
    input_width,        /* Int16, width of the input array */
    input_height,       /* Int16, height of the input array */
    input_depth,        /* Int16, depth of the input array */
    output_width,       /* Int16, width of the output array */
    output_height,      /* Int16, height of the output array */
    output_depth,       /* Int16, number of output color planes */
    compute_width,      /* Int16, number of pixels processed horizontally */
    compute_height,     /* Int16, number of pixels processed vertically */
    input_step_color,   /* Int16, offset between input colors */
    input_step_row,     /* Int16, offset between input rows */
    output_step_color,  /* Int16, offset between output colors */
    output_step_row,    /* Int16, offset between output rows */
    input_type,         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                        /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,            /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                        /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,        /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,        /* Int16, number of bits to downshift before output */
    cmdptr              /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs matrix multiplication for color space conversion. Logically, input is a 3-D array, of input_width x input_height x input_depth. Output is also a 3-D array, of output_width x output_height x output_depth. Coefficient matrix is output_depth (height) x input_depth (width). compute_width x compute_height pixels are processed. There is flexibility in input/output storage organization. Input/output data can be stored row-interleaved or completely color-separate. This is controlled by *_step_color and *_step_row parameters. For example, with input color row-interleaved, set input_step_color = input_width, input_step_row = input_width * input_depth, and with input color separate, this results in input_step_color = input_height * input_width, input_step_row = input_width.

Coefficients are stored in transposed, or column-first, format.

Example

Consider 16x16 block color space conversion, from RGB to YUV (3x3). Input row-interleaved and output color-separate. All data are Int16 (coefficients 12-bit). Round down by 10 bits before output.

```
cmdlen = imxenc_color_spc_conv(
    input_ptr,       /* point to input data */
    coeff_ptr,       /* point to coef array */
    output_ptr,      /* point to output array */
    16,              /* width of input */
    16,              /* height of input */
    3,               /* number of input color planes */
    16,              /* width of output */
    16,              /* height of output */
    3,               /* number of output color planes */
    16,              /* number of pixels processed horizontally */
    16,              /* number of pixels processed vertically */
    16,              /* offset between input colors */
    48,              /* offset between input rows */
    256,             /* offset between output colors */
    16,              /* offset between output rows */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    10,              /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- input_width, output_width >= compute_width
- input_height, output_height >= compute_height
- output_depth must be between 1 and 3. When there are more then 3 output planes, user needs to break it up into multiple function calls.
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (34)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} \times \textit{input\_depth} \times \textit{output\_depth} \quad (35)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/3
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/3
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/3

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [9]	1	256

[9] That is, compute_width is odd.

6.1.21 imxenc_cumulativeSumCol32bits

imxenc_cumulativeSumCol32bits ***Produce cumulative sum over each column of the input array. Can be used for integral image computation.***

Syntax

```
cmdlen = imxenc_cumulativeSumCol32bits(
    inputLsb_ptr,        /* Int16*, starting address of input's 8 or 16 least significant bits */
    inputMsb_ptr,        /* Int16*, starting address of input's 8 or 16 most significant bits, */
                         /* can be 0 */
    outputLsb_ptr,       /* Int16*, starting address of output's 16 least significant */
    outputMsb_ptr,       /* Int16*, starting address of output's 16 most significant */
    initialLsb_ptr,      /* Int16*, starting address of initial values' 16 LSBs, can be 0 */
    initialMsb_ptr,      /* Int16*, starting address of initial values' 16 MSBs, can be 0 */
    scratchCoefPtr_ptr,  /* Int16*, starting address of scratch buffer in coef memory */
                         /* of size 1 + compute_width 16-bits words */
    compute_width,       /* Int16, width of the input and output arrays */
    compute_height,      /* Int16, height of the input and output arrays */
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_USHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    cmdptr               /* Int16*, starting point of command sequence in memory */
);
```

Description

This function cumulatively sums each column of the input array and writes each partial sum into the output array. Values up to 32 bits are supported by the function, which accepts pointers to least significant bits or most significant bits arrays.

Mathematically the operation can be expressed as follow: if we let x[i, j] be the input array and y[i,j] the output array where i is the column index and j the row index, then for a given column C, each term y[C, j] is computed using the previous term y[C, j-1] and the present input x[C, j] as shown in Equation 36:

$$y[C, j]= x[C, j] + y[C, j - 1] \text{ , for } j > 0 \tag{36}$$

For j= 0, the function can either accept a 1-D array of initial values if initialLsb_ptr≠ 0 as in Equation 37 or treat all the initial values as zeros if initiaLsb_ptr= 0 as in Equation 38:

$$y[C, 0]= x[C, 0] + \text{initial}[C] \text{ , for } j =0 \text{ and if initialLsb\_ptr} \neq 0 \tag{37}$$

$$y[C, 0]= x[C, 0], \text{ for } j =0 \text{ and if initialLsb\_ptr}= 0 \tag{38}$$

The generated output values are always 32 bits unsigned integer, which are rearranged into two arrays of size compute_width × compute_height unsigned shorts. One array contains the 16 LSBs and the other array contains the 16 MSBs.

Likewise the initial values are always 32 bits unsigned integers as they are very likely to come from an execution of the function on a previous array.

The input array however can be made of either, 8 bits, 16 bits or 32 bits unsigned integer.

- For 8-bits unsigned integers, set inputLsb_ptr to point to unsigned bytes and inputMsb_ptr= 0.
- For 16-bits unsigned integers, set inputLsb_ptr to point to unsigned shorts and inputMsb_ptr= 0.
- For 32-bits unsigned integers, set inputLsb_ptr to point to the 16 LSBs and inputMsb_ptr point to the 16 MSBs

This function can be used to implement integral image computation in two stages. In the first stage, use imxenc_cumulativeCol32bits() to generate the cumulative sum over each column of the input. In the second stage, apply imxenc_cumulativeCol32bits() again, but to the transposed result of the first stage. The results must be transposed back again to match the original dimensions of the frame.

Constraints

compute_height must be < 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (39)$$

- amount_of_work =

$$4 \times compute\_width \times compute\_height \quad (40)$$

- memory_conflict_factor:

Location of input and initial values	Location of output values	memory_conflict_factor
IMGBUF	IMGBUF	3
IMGBUF	COEFF	3 / 2
COEFF	IMGBUF	3 / 2
COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [10]	1	256

[10] That is, compute_width is odd.

6.1.22 imxenc_dct8x8col

imxenc_dct8x8col ***1-D Column DCT on 8x8 blocks of data with 8x8 coefficient matrix***

Syntax

```
cmdlen = imxenc_dct8x8col (
    input_ptr,       /* Int16*, starting address of  1st input */
    coeff_ptr,       /* Int16*, starting address of 2nd input */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    calc_Hblks,      /* Int16, number of horizontal blocks */
    calc_Vblks,      /* Int16, number of vertical blocks */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs the 1-D column DCT on an 8x8 data blocks with the DCT coefficients stored in another 8x8 coefficient matrix.

The 1-D column DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area.

The data array is stored row-by-row, input_width data points per row. Coefficients are stored in transposed form when view as an 8-by-8 matrix. See Section 6.1.23 for details on imxenc_dct8x8row.

Example

Consider a 1-D column DCT on 8x8 data blocks within an image of 20x20. Number of horizontal blocks and vertical blocks are 2 each. Output is written to a 16x16 memory. This is second dimension transform, round down by 14 bits.

```
cmdlen = imxenc_dct8x8col(
    input_ptr,        /* starting address of the input matrix */
    coeff_ptr,        /* starting address of the coefficient matrix */
    output_ptr,       /* starting address of the output matrix */
    20,               /* width of the input  */
    20,               /* height of the input  */
    16,               /* width of the output */
    16,               /* height of the output */
    2,                /* number of horizontal blocks */
    2,                /* number of vertical blocks */
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXOTYPE_SHORT,   /* byte, Int16 */
    14,               /* number of bits to downshift before output */
    cmdptr            /* starting point for command sequence in memory */
);
```

Constraints

- input_width should be greater than or equal to 8 × calc_Hblks
- input_height should be greater than or equal to 8 × calc_Vblks
- calc_Hblks ≤ 256
- calc_Vblks ≤ 256
- Image data should be arranged in a regular fashion
- DCT coefficients can be either transposed or regular. See discussion above.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \quad (41)$$

- amount_of_work =

$$64 \times calc\_Hblks \times calc\_Vblks \quad (42)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/8
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/8
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/8
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/8

6.1.23 imxenc_dct8x8row

imxenc_dct8x8row ***1-D row DCT on 8x8 blocks of data with 8x8 coefficient matrix***

Syntax

```
cmdlen = imxenc_dct8x8row (
    input_ptr,      /* Int16*, starting address of  1st input */
    coeff_ptr,      /* Int16*, starting address of 2nd input */
    output_ptr,     /* Int16*, starting address of the output array */
    input_width,    /* Int16, width of the input array */
    input_height,   /* Int16, height of the input array */
    output_width,   /* Int16, width of the output array */
    output_height,  /* Int16, height of the output array */
    calc_Hblks,     /* Int16, number of horizontal blocks */
    calc_Vblks,     /* Int16, number of vertical blocks */
    input_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,    /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,    /* Int16, number of bits to downshift before output */
    cmdptr          /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs the 1-D row DCT on an 8x8 data blocks with the DCT coefficients stored in another 8x8 coefficient matrix.

The 1-D column DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area.

The input data are stored row by row (regular C array format), with input_width data points per row. Coefficients are stored in transposed format when viewed as an 8-by-8 matrix. The transform can be expressed mathematically as:

$$X(m) = \frac{C(m)}{2} \sum_{i=0}^{7} x(i)\cos(\frac{(2i+1)m\pi}{16}) = \sum_{i=0}^{7} C(m,i)x(i) \qquad (43)$$

The coefficient array contains C(0,0), C(1,0),..., C(7,0), C(0,1), C(1,1), ..., C(7, 7) scaled up by a two's power. Programmer has control over the precision and dynamic range of the intermediate result (in between row and column transforms) via the round_shift parameter.

Example

Consider a 1-D row DCT on 8x8 data blocks within an image of 20x20. Number of horizontal blocks and vertical blocks are 2 each. Output is written to a 16x16 memory. This is first dimension transform, round down by 8 bits.

```
cmdlen = imxenc_dct8x8row(
    input_ptr,       /* starting address of the input matrix */
    coeff_ptr,       /* starting address of the coefficient matrix */
    output_ptr,      /* starting address of the output matrix */
    20,              /* width of the input */
    20,              /* height of the input  */
    16,              /* width of the output */
    16,              /* height of the output */
    2,               /* number of horizontal blocks */
    2,               /* number of vertical blocks */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* signed Int16/unsigned byte  */
    8                /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- input_width should be greater than or equal to 8 × calc_Hblks
- input_height should be greater than or equal to 8 × calc_Vblks
- calc_Hblks ≤ 256
- calc_Vblks ≤ 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \quad (44)$$

- amount_of_work =

$$64 \times calc\_Hblks \times calc\_Vblks \quad (45)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/8
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/8
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/8
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/8

6.1.24 imxenc_deinterleaveData

imxenc_deinterleaveData ***Deinterleave data contained in one input arrays into two output arrays. Optionally scalar operation can be applied to the interleaved outputs, without any added performance loss.***

Syntax

```
cmdlen = imxenc_deinterleaveData(
    input_ptr,          /* Int16*, starting address of input */
    scalar1_ptr,        /* Int16*, pointer to scalar used for operation with first output */
    scalar2_ptr,        /* Int16*, pointer to scalar used for operation with second output */
    output1_ptr,        /* Int16*, starting address of the first deinterleaved output array */
    output2_ptr,        /* Int16*, starting address of the second deinterleaved output array */
    input_width,        /* Int16, width of input in number of elements */
    output1_width,      /* Int16, width of first deinterleaved output array in number of elements */
    output2_width,      /* Int16, width of second deinterleaved output array in number of elements*/
    computation_width,  /* Int16, computation width */
    computation_height, /* Int16, computation_height */
    input_type,            /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                        /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    round_shift,    /* Int16, number of bits to downshift before output */
    operation,          /* Int16, operation IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB,
                           IMXOP_AND, IMXOP_OR, IMXOP_XOR,
                           IMXOP_MIN, IMXOP_MAX */
    cmdptr,             /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

This function first deinterleaves the input array into two output arrays and then performs a point-to-point scalar operation between each data point of output1, output2 arrays and scalar1, scalar2 values. The parameter output1_ptr points to the output array that will contain elements from the input array that have even indexes and the parameter output2_ptr points to the output array that will contain elements from the input array that have odd indexes.

output1[0]= scalar1*input[0], output1[1]= scalar1*input[2], etc and output2[0]= scalar2*input[1], output2[1]= scalar2*input[3], etc.

Example

To deinterleave an input array of 16 x 8 bytes into two arrays of 8 x 8 bytes:

```
*scalar1_ptr= *scalar2_ptr= 0;  /* operation will be set to IMXOP_ADD so we set the scalars to 0
    as we don’t want any value to be changed */

cmdlen = imxenc_deinterleaveData(
    input_ptr,       /* Int16*, starting address of input */
    scalar1_ptr,  /* Int16*, pointer to scalar used for operation with first output */
    scalar2_ptr,  /* Int16*, pointer to scalar used for operation with second output */
    output1_ptr,  /* Int16*, starting address of the first deinterleaved output array */
    output2_ptr,  /* Int16*, starting address of the second deinterleaved output array */
    16,       /* Int16, width of input in number of elements */
    8,            /* Int16, width of first deinterleaved output array in number of elements */
    8,            /* Int16, width of second deinterleaved output array in number of elements*/
    16,           /* Int16, computation width */
    8,            /* Int16, computation_height */
    IMXTYPE_BYTE, /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                  /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    0,            /* Int16, number of bits to downshift before output */
    IMXOP_ADD,    /* Int16, operation IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB,
                     IMXOP_AND, IMXOP_OR, IMXOP_XOR, IMXOP_MIN, IMXOP_MAX */
    cmdptr,       /* Int16*, starting point of command sequence in memory */
    );
```

Constraints

- input_width >= compute_width
- output1_width >= compute_width/2 and output2_width >= compute_width/2
- compute_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \quad (46)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (47)$$

- memory_conflict_factor:

Location of input	Location of output1	Location of output2	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	IMGBUF	1
IMGBUF	IMGBUF	COEFF	1.5
COEFF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	1.5
COEFF	COEFF	IMGBUF	1
IMGBUF	COEFF	COEFF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	1024

6.1.25 imxenc_fillMem

imxenc_fillMem ***Fill memory with one value***

Syntax

```
cmdlen = imxenc_fillMem (
    val,            /* Int16*, starting address of constant value */
    coef,           /* Int16*, starting address of coefficient value */
    output_ptr,     /* Int16*, starting address of the output array */
    byteOfst,       /* Int16, offset of byte */
    output_width,   /* Int16, width of the output array */
    output_height,  /* Int16, height of the output array */
    nbHsteps,       /* Int16, number of horizontal steps */
    nbVsteps,       /* Int16, number of vertical steps */
    hStepSize,      /* Int16, size of horizontal steps */
    vStepSize,      /* Int16, size of vertical steps */
    val_type,       /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coef_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,    /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,    /* Int16, number of bits to downshift before output */
    operation,      /* Int16, IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB */
                    /* IMXOP_AND, IMXOP_OR, IMXOP_XOR, */
                    /* IMXOP_MIN, IMXOP_MAX */
    cmdptr          /* Int16*, starting point of command sequence in memory */
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

Fill memory pointed by output_ptr with the result of the binary operation between the value pointed by the val pointer and the value pointed by the coef pointer.

To fill the memory with the constant value pointed by val pointer, put 0 at the location pointed by coef pointer and use the operation IMXOP_ADD.

Current implementation ignores hStepSize and vStepSize so nbHsteps and nbVsteps can be interpreted as computation width and computation height.

The byte_ofst is also ignored by the current implementation.

Example

Consider output → 20(W)x13(H) matrix and the goal to fill a subregion of size 8(W)x10(H) with the value pointed by val_ptr. The following code illustrates what the API call would look like.

```
cmdlen = imxenc_fillMem(
    val_ptr,         /* pointer to val, fill value */
    coeff_ptr,       /* pointer to coeff */
    output_ptr,      /* starting address of output */
    0,               /* no byte offset */
    20,              /* output_width */
    13,              /* output_height */
    8,               /* computation width */
    10,              /* computation height */
    0,0,0,0,         /* ignored */
    IMXTYPE_SHORT,   /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    0,               /* number of bits to downshift before output */
    IMXOP_ADD,       /* addition */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- output_width >= nbHsteps
- output_height >= nbVsteps
- nbVsteps must be ≤256
- When operation is IMXOP_SADBY (packed-bytes absolute difference), val_type and coeff_type must be IMXTYPE_USHORT.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \tag{48}$$

- amount_of_work =

$$nbHsteps \times nbVsteps \tag{49}$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + (16 / compute_width × compute_height)
IMGBUF	IMGBUF	COEFF	1 + (8 / compute_width × compute_height)
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + (8 / compute_width × compute_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + (8 / compute_width × compute_height)

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [11]	1	256

[11] That is, compute_width is odd.

6.1.26 imxenc_filter

imxenc_filter ***Perform 2-D FIR filtering, 1-D column and row FIR filtering***

Syntax

```
cmdlen = imxenc_filter (
    input_ptr,       /* Int16*, starting address of  1st input */
    coeff_ptr,       /* Int16*, starting address of 2nd input */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    coeff_width,     /* Int16, width of the filter kernel */
    coeff_height,    /* Int16, height of the filter kernel */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    dnsmpl_horz,     /* Int16, horizontal downsampling factor: 1, 2, 4 or 8 */
    dnsmpl_vert,     /* Int16, vertical  downsampling factor, any value */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function can perform one of the following FIR filtering operations on a block of data:

- 2-D filtering
- 1-D row filtering
- 1-D column filtering

The result is in an output block of dimension [compute_width x compute_height]. Some horizontal and vertical downsampling factors are supported.

Kernel size is [coeff_width x coeff_height].

For the case of 1-D row filtering, set coeff_height to 1, and for 1-D column filtering, set coeff_width to 1. It is assumed that the input block of data is appropriately zero-filled and/or history-managed at appropriate places. Width of input accessed for this operation is (compute_width + coeff_width – 1), and height of input accessed is (compute_height + coeff_height – 1).

The filtering is performed by using correlation instead of convolution operation. 2-D correlation is related to 2-D convolution by a 180 degrees rotation of the coefficient matrix. The mathematical formula would be:

$$output(i,j) = \sum_{i,j} \sum_{k,l} input(i+k, j+l) * coeff(k,l) \tag{50}$$

Example

Consider an input block of 20(W) x 12(H), output block of 20(W) x 8(H), coefficient block of 5(W) x 3(H) and n_cols x n_rows = 16x8, round down by 10 bits before output.

```
cmdlen = imxenc_filter(
    input_ptr,       /* starting address of input array */
    coeff_ptr,       /* starting address of coefficient array */
    output_ptr,      /* starting address of output array */
    20,              /* width of input block */
    12,              /* height of input block */
    5,               /* width of filter kernel */
    3,               /* height of filter kernel */
    20,              /* width of output block */
    8,               /* height of output block */
    16,              /* computation width */
    8,               /* computation height */
    1,               /* horizontal downsampling */
    1,               /* vertical downsampling */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    10               /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- compute_width must be a multiple of 8
- input_width ≥ compute_width + coeff_width – 1, output_width ≥ compute_width
- input_height ≥ compute_ height + coeff_ height – 1, output_ height ≥ compute_ height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are complete.
- dnsmpl_horz can only be 1, 2, 4, or 8 (dnsampl_vert can be any positive integer)
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

amount_of_work × *memory_conflict_factor* / *speedup_factor* (51)

- amount_of_work =

coeff_width × *coeff_height* × *dnsmpl_horz* × *compute_width* × *compute_height* (52)

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/ (coeff_width × coeff_height)
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/ (coeff_width × coeff_height)

- speedup_factor and maximum value for compute_width × dnsmpl_horz:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [12]	1	256

[12] That is, compute_width is odd.

6.1.27 imxenc_filter_op

imxenc_filter_op ***Generic 2-D filtering using +, -, | - |, min, max, and logical operators***

Syntax

```
cmdlen = imxenc_filter_op(
    input_ptr,       /* Int16*, starting address of  1st input */
    coeff_ptr,       /* Int16*, starting address of 2nd input */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    num_coeff_horz,  /* Int16, number of horizontal coefficients */
    num_coeff_vert,  /* Int16, number of vertical coefficients */
    coeff_width,     /* Int16, width of the array containing the coefficients */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    dnsmpl_horz,     /* Int16, horizontal downsampling factor */
    dnsmpl_vert,     /* Int16, vertical  downsampling factor */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
   coeff_type,       /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
   output_type,      /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
   operation,        /* Int16, IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB, */
                     /* IMXOP_AND, IMXOP_OR, IMXOP_XOR, */
                     /* IMXOP_MIN, IMXOP_MAX */
   round_shift,      /* Int16, number of bits to downshift before output */
   cmdptr            /* Int16*, starting point of command sequence in memory */
   );

/* cmdlen is the number of words written to cmd memory starting at cmdptr *
```

Description

Similar to imxenc_filter() except:

- The 2-D coefficient matrix of size num_coeff_horz x num_coeff_vert can be embedded in a bigger array of width coeff_width
- In typical filtering, the operation between the input and the coefficient is performed as a product and the accumulation is performed as an addition. In this particular filtering operation, the input/coefficient operation and accumulation can change upon the value of input parameter operation:

Operation Parameter	Operation Between Input a and Coefficient b	Operation for Accumulation
IMXOP_MPY	a × c	+
IMXOP_ABDF	|a - b|	+
IMXOP_ADD	a + b	+
IMXOP_SUB	a - b	+
IMXOP_AND	a & b	|
IMXOP_OR	a | b	|
IMXOP_XOR	a ^ b	^
IMXOP_MIN	min(a,b)	min
IMXOP_MAX	mac(a,b)	max

Example

Calculate sum of absolute difference between a matching block and a reference array, for different positions. The position of the matching block is incremented by 1 data point from left to right. The reference array is of size 66x5 and the matching block of size 3x3. The matching block is matched 64 times against the reference array horizontally from left to right. So the output fits in an array of 64x1 elements. This function should be:

```
cmdlen = imxenc_filter_op(
    input_ptr,       /* starting address of input array */
    coeff_ptr,       /* starting address of matching block */
    output_ptr,      /* starting address of output array */
    66,              /* width of input array */
    5,               /* height of input array */
    3,               /* width of matching block */
    3,               /* height of matching block */
    3,               /* matching block's stride */
    64,              /* width of output block */
    1,               /* height of output block */
    64,              /* computation width */
    1,               /* computation height */
    1,               /* horizontal downsampling */
    1,               /* vertical downsampling */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    IMXOP_ABSDIFF,   /* operation */
    0                /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- compute_width must be a multiple of 8
- input_width ≥ compute_width + num_coeff_horz – 1, output_width ≥ compute_width
- input_height ≥ compute_ height + num_coeff_vert – 1
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed
- dnsmpl_horz can only be 1, 2, or 4 (dnsampl_vert can be any positive integer)
- compute_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (53)$$

- amount_of_work =

$$coeff\_width \times coeff\_height \times dnsmpl\_horz \times compute\_width \times compute\_height \quad (54)$$

- memory_conflict_factor:

Location of input_ptr	Location of coeff_ptr	Location of output_ptr	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/ (coeff_width × coeff_height)
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/ (coeff_width × coeff_height)

- speedup_factor and maximum value for compute_width × dnsmpl_horz:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [13]	1	256

[13] That is, compute_width is odd.

6.1.28 imxenc_filter_distribute

imxenc_filter_distribute ***Perform filtering on a 4-D data array with a 2-D coefficient array, producing a 4-D output array.***

Syntax

```
cmdlen = imxenc_filter_distribute(
    input_ptr,        /* Int16, starting address of input */
    coeff_ptr,        /* Int16, starting address of coefficients */
    output_ptr,       /* Int16, starting address of output */
    input_width,      /* Int16, width of input array */
    input_height,     /* Int16, height of input array */
    coeff_width,      /* Int16, width of coefficient array */
    coeff_height,     /* Int16, height of coefficient array */
    output_width,     /* Int16, width of output array */
    output_height,    /* Int16, height of output array */
    compute_width,    /* Int16, computed width */
    compute_height,   /* Int16, computed height */
    dnsmpl_horz,      /* Int16, horizontal downsample factor:  1, 2, 4, or 8 */
    dnsmpl_vert,      /* Int16, vertical downsample factor */
    num_blks_x,       /* Int16, Number of horizontal blocks */
    num_blks_y,       /* Int16, Number of vertical blocks */
    input_offset_x,   /* Int16, Offset between horizontal blocks */
    input_offset_y,   /* Int16, Offset between vertical blocks */
    output_offset_x,  /* Int16, Offset between horizontal output blocks */
    output_offset_y,  /* Int16, Offset between vertical output blocks */
    input_type,       /* Int16, Int16/byte, signed/unsigned */
    coeff_type,       /* Int16, Int16/byte, signed/unsigned */
    output_type,      /* Int16, Int16/byte */
    round_shift,      /* Int16, number of bits to downshift before output */
    cmdptr            /* Int16*, starting point of command sequence in memory */
);
```

Description

This function performs filtering on multiple, regularly-spaced, 2D arrays of data.

For the inner 2-D arrays, the actual operand size of [compute_width x compute_height] resides within [input_width x input_height] of data input and within [coeff_width x coeff_height] of coefficient input, and actual output of size [compute_width x compute_height] resides within [output_width x output_height] of the output. The outer two dimensions are indexed with horizontal and vertical offsets, and the offsets are in data points (not the address offsets). The pointers, input_ptr, coeff_ptr, and output_ptr, specify the first element, or upper-left corner of operands and output. For example, logical data item input[m, n, i, j] is assumed to reside at input_ptr[m * input_offset_vert + n * input_offset_horz + i * input_width + j]. The xxx_height parameters are included for modularity, but are not used in any address calculation.

The outer dimensions of input can optionally be used to index overlapping sub-arrays in an input array. For example, using input_offset_horz = input_offset_vert = 1, and num_distribute_horz = num_distribute_vert = 3 addresses a 3x3 neighborhood for each inner-array data point.

Each of the two inputs and output can be operated on either 16-bit (Int16) data or 8-bit (byte) data. Rounding shifts is specified in the command, and saturation parameters should be appropriately set in the imxenc_set_saturation command.

Example

Consider data input → 2(outerH) x 3(outerW) x 4(H) x 17(H) matrix, coefficient input → 2(W) x 2(H) matrix and output → 2(outerH) x 3(outerW) x 3(H) x 16(W) matrix. The following illustrates what the API call would look like and a descriptive figure is given below.

```
cmdlen = imxenc_filter_distribute(
    input_ptr,        /* starting address of 1st input */
    coeff_ptr,        /* starting address of 2nd input */
    output_ptr,       /* starting address of output */
    17,               /* width of data input */
    4,                /* height of data input */
    2,                /* width of coefficient input */
    2,                /* height of coefficient input */
    16,               /* width of output */
    3,                /* height of output */
    16,               /* computation width */
    3,                /* computation height */
    1,                /* horizontal downsample factor */
    1,                /* vertical downsample factor */
    3,                /* outer width */
    2,                /* outer height */
    70,               /* input horizontal offset */
    300,              /* input vertical offset */
    96,               /* output horizontal offset */
    320,              /* output vertical offset */
    IMXTYPE_SHORT,    /* signed Int16/unsigned byte*/
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXOTYPE_SHORT,   /* byte, Int16 */
    0,                /* number of bits to downshift before output */
    cmdptr            /* starting point for command sequence in memory */
);
```

Constraints

- input_width, coeff_width, output_width ≥ compute_width
- input_height, coeff_height, output_height ≥ compute_height
- compute_height, compute_width, num_distribute_horz, num_distribute_vert ≤ 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} \,/\, \mathit{speedup\_factor} \quad (55)$$

- amount_of_work =

$$\mathit{compute\_width} \times \mathit{compute\_height} \times \mathit{num\_distribute\_horz} \times \mathit{num\_distribute\_vert} \quad (56)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/ (coeff_width × coeff_height)
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/ (coeff_width × coeff_height)

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048

6.1.29 imxenc_filter_ds

imxenc_filter_ds ***Perform 2-D FIR filtering, 1-D column and row FIR filtering with heavy downsampling and wide filtering kernel***

Syntax

```
cmdlen = imxenc_filter_ds(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *coeff_ptr,      /* starting address of coefficients */
    Int16 *output_ptr,     /* starting address of output */
    Int16 *temp_ptr,       /* starting address of temp buffer */
    Int16 input_width,     /* width/columns of input */
    Int16 input_height,    /* height/rows of input */
    Int16 coeff_width,     /* width/columns of coefficients */
    Int16 coeff_height,    /* height/rows of coefficients */
    Int16 output_width,    /* width/columns of output */
    Int16 output_height,   /* height/rows of output */
    Int16 compute_width,   /* computed output number of columns */
    Int16 compute_height,  /* computed output number of rows */
    Int16 dnsmpl_horz,     /* horizontal downsampling factor */
    Int16 dnsmpl_vert,     /* vertical downsampling factor */
    Int16 input_type,      /* Int16/byte, signed/unsigned */
    Int16 coeff_type,      /* Int16/byte, signed/unsigned */
    Int16 output_type,     /* Int16/byte */
    Int16 round_shift,     /* shifting parameter */
    Int16 *cmdptr)

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function can perform one of the following FIR filtering operations on a block of data:

- 2-D filtering
- 1-D row filtering
- 1-D column filtering

The result is in an output block of dimension [compute_width x compute_height]. Horizontal and vertical downsampling is supported. Contrary to imxenc_filter(), this function requires that the coeff_width be multiple of 8. Hence zero padding may be required for narrow coefficient kernel. Due to this requirement, it is only worth using this function if the horizontal downsample is heavy. Otherwise use imxenc_filter().

The filtering is performed by using correlation instead of convolution operation. 2-D correlation is related to 2-D convolution by a 180 degrees rotation of the coefficient matrix.

Constraints

- coeff_width must be a multiple of 8; zero-padding can be used to meet the requirement
- input_width ≥ compute_width + coeff_width – 1, output_width ≥ compute_width
- input_height ≥ compute_ height + coeff_ height – 1, output_ height ≥ compute_ height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (57)$$

- amount_of_work =

$$coeff\_width \times coeff\_height \times compute\_width \times compute\_height + compute\_height \times compute\_width \quad (58)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/ (coeff_width × coeff_height)
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/ (coeff_width × coeff_height)
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/ (coeff_width × coeff_height)

- speedup_factor and maximum value for coeff_width:

coeff_width multiple only of	speedup_factor	Maximum value of compute_width × dnsmpl_horz
8	8	2048
4	4	1024
2	2	512
1 (14)	1	256

(14) That is, compute_width is odd.

6.1.30 imxenc_interleaveData

imxenc_interleaveData ***Interleave data contained in two input arrays. Optionally scalar operation can be applied to the inputs before interleaving.***

Syntax

```
cmdlen = imxenc_interleaveData(
    input1_ptr,          /* Int16*, starting address of first input */
    input2_ptr,          /* Int16*, starting address of second input */
    scalar1_ptr,         /* Int16*, pointer to scalar used for operation with first input */
    scalar2_ptr,         /* Int16*, pointer to scalar used for operation with second input */
    output_ptr,          /* Int16*, starting address of the output array */
    input1_width,        /* Int16, width of 1st input in number of elements */
    input2_width,        /* Int16, width of 2nd input in number of elements */
    output_width,        /* Int16, width of output in number of elements */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input_type,          /* Int16, type of input1, scalar1, input2, scalar2 */
                         /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    operation,           /* Int16, operation IMXOP_MPY, IMXOP_ABDF, IMXOP_ADD, IMXOP_SUB,
                            IMXOP_AND, IMXOP_OR, IMXOP_XOR,
                            IMXOP_MIN, IMXOP_MAX */
    cmdptr,              /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

This function first performs a point-to-point scalar operation between each data point of input1, input2 arrays and scalar1, scalar2 values. Afterwards it writes out the results in an interleaved fashion, with output[0]= scalar1×input1[0], output[1]= scalar2×input2[0], output[2]= scalar1×input1[1], output[3]= scalar2×input2[1], … . If no scalar operation is desired, set operation to IMXOP_ADD and scalar1= scalar2= 0 . Since the output is written in an interleaved fashion, one row of the output is at least twice the number of elements computed per input row.

This function can be used to interleave three planes of Y, U, V data into YUV422 interleaved format and has better performance than imxenc_YCbCrPack() if the input type is byte.

Example

To produce YUV422 interleaved format from of a 16x16 block of Y, 8x16 block of U and V, first call imxenc_interleaveData() to produce VU interleaved:

```
cmdlen= imxenc_interleaveData(inputV_ptr, inputU_ptr, scalar1_ptr,
scalar2_ptr, outputVU_ptr, 8, 8, 16, 8, 16, IMXTYPE_BYTE, 0, IMXOP_ADD, cmdptr);
```

Then call again to interleave VU with Y in order to produce YUV422 interleaved:

```
cmdlen+= imxenc_interleaveData(inputVU_ptr, inputY_ptr, scalar1_ptr, scalar2_ptr,
outputYUV422_ptr, 16, 16, 32, 16, 16, IMXTYPE_BYTE, 0, IMXOP_ADD, cmdptr +
cmdlen);
```

Constraints

- input1_width, input2_width ≥ compute_ width
- output_width ≥ 2 × compute_width
- compute_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (59)$$

- amount_of_work = Equation 60, where S=2 for short input type or 1 for byte input type:

$$S \times compute\_width \times compute\_height \quad (60)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [15]	1	256

[15] That is, compute_width is odd.

6.1.31 *imxenc_fir_poly_col*

imxenc_fir_poly_col ***1D polyphase filtering along columns, including sampling by rational factors (up/down)***

Syntax

```
cmdlen = imxenc_fir_poly_col(
    input_ptr,       /* Int16*, starting address of  1st input */
    coeff_ptr,       /* Int16*, starting address of 2nd input */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    coeff_taps,      /* Int16, number of filter coefficients */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    smpl_nom         /* Int16, Nominator of rational sampling factor
                        (upsampling) */
    smpl_denom,      /* Int16, Denominator of rational sampling factor
                        (downsampling) */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function performs filtering of a block of data along the columns (vertical filtering). For filtering a polyphase filter structure is employed to provide sampling by rational factors (N/M). Almost all sampling ratios are supported, limitations are coming from memory restrictions and output restrictions (see constraints). Filtering including sampling results in an output block of dimension [compute_width x compute_height]. The filter coefficients must be set up in a particular way => imx_fir_poly_setup_coeff().

It is assumed that the input block of data is appropriately zero-filled and/or history-managed at appropriate places.

Height of input accessed is smpl_nom × (compute_height/smpl_denom -1) + smpl_denom + coeff_taps / smpl_nom .

Example

Consider an input block of 20(W) x 12(H), output block of 20(W) x 8(H), A 5-tap FIR filter is used, subsampling ratio is ¾, and n_cols x n_rows = 16x8 elements are computes (after sampling), which are rounded down by 10 bits before output.

```
cmdlen = imxenc_fir_poly_col(
    input_ptr,       /* starting address of input array */
    coeff_ptr,       /* starting address of coefficient array */
    output_ptr,      /* starting address of output array */
    20,              /* width of input block */
    13,              /* height of input block */
    5,               /* taps of filter kernel */
    20,              /* width of output block */
    9,               /* height of output block */
    16,              /* computation width */
    9,               /* computation height */
    3,               /* nominator (upsampling) */
    4,               /* denominator (downsampling) */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    10               /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- input_width ≥ compute_ width
- input_height ≥ smpl_nom × (compute_height / smpl_denom -1) + smpl_denom + coeff_taps / smpl_nom
- output_width ≥ compute_width
- output_ height ≥ compute_ height
- compute_height must be multiple of smpl_nom
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} / \mathit{speedup\_factor} \quad (61)$$

- amount_of_work =

$$(\mathit{smpl\_denom} + \mathit{coeff\_taps} / \mathit{smpl\_nom}) \times \mathit{compute\_width} \times \mathit{compute\_height} \quad (62)$$

- memory_conflict_factor, where
  $a = (\mathit{smpl\_nom} \times \mathit{smpl\_denom} + \mathit{coeff\_taps}) / \mathit{smpl\_nom}$:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	(1 + 2 × a) / a
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	(1 + a) / a
IMGBUF	COEFF	COEFF	(1 + a) / a
COEFF	IMGBUF	IMGBUF	(1 + a) / a
COEFF	IMGBUF	COEFF	(1 + a) / a
COEFF	COEFF	IMGBUF	2
COEFF	COEFF	COEFF	(1 + 2 × a) / a

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [16]	1	256

[16] That is, compute_width is odd.

6.1.32 imxenc_mat_mul

imxenc_mat_mul ***Computes the product of two matrices, A and B, using normal matrix multiplication.***

Syntax

```
cmdlen = imxenc_mat_mul(
    input1_ptr,     /* Int16*, starting address of  1st input */
    input2_ptr,     /* Int16*, starting address of 2nd input */
    output_ptr,     /* Int16*, starting address of the output array */
    input1_width,   /* Int16, width of 1st input */
    input1_height,  /* Int16, height of 1st input */
    input2_width,   /* Int16, width of 2nd input */
    input2_height,  /* Int16, height of 2nd input */
    output_width,   /* Int16, width of output */
    output_height,  /* Int16, height of output */
    mat1_width,     /* Int16, Matrix 1 width */
    mat1_height,    /* Int16, Matrix 1 height */
    mat2_width,     /* Int16, Matrix 2 width */
    mat2_height,    /* Int16, Matrix 2 height */
    input1_type,    /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    input2_type,    /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,    /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT, IMXOTYPE_LONG */
    round_shift,    /* Int16, number of bits to downshift before output */
    cmdptr,         /* Int16*, starting point of command sequence in memory */
    );

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function multiplies two matrices, using normal matrix multiplication.

Width of 1st matrix (mat1_width) should be equal to the height of the 2nd matrix (mat2_height). [mat1_width x mat1_height] 1st matrix is multiplied with [mat2_width x mat2_height] 2nd matrix to yield result matrix [mat2_width x mat1_height]. The matrices can be blocks in a larger input block, thus input1_width and input1_height do not have to be the same as mat1_width and mat1_height. Similarly, input2_width and input2_height do not have to be the same as mat2_width and mat2_height. The constraints are that input1_height >= mat1_height, input2_height >= mat2_height, input1_width >= mat1_width, and input2_width >= mat2_width.

The output type can be set to IMXOTYPE_LONG to output 32-bit results. When this mode is enabled, output_ptr must be 128-bit aligned and output_width must be a multiple of 4. The saturation should also be disabled with a call to IMX_setSat(IMX_SAT_OFF), if you want to get the full 32-bit range.

Constraints

- mat1_width == mat2_height
- input1_width ≥ mat1_width, input2_width ≥ mat2_width
- input1_height ≥ mat1_height, input2_height ≥ mat2_height
- output_width ≥mat2_width, output_height ≥ mat1_height
- mat2_width must be a multiple of 8 (number of MACs)
- compute_height and input_height must be < 256.
- If output_type== IMXOTYPE_LONG, then output_width must be a multiple of 4 and output_ptr must be 128-bits aligned.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (63)$$

- amount_of_work =

$$mat1\_height \times mat1\_width \times mat2\_width \quad (64)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 1/ mat2_height
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	1 + 1/ mat2_height
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 1/ mat2_height
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 1/ mat2_height

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 (17)	1	256

(17) That is, compute_width is odd.

6.1.33 imxenc_median_filter_row

imxenc_median_filter_row ***Perform a 1-D median filter (3 or 5 tap) along the rows of an input matrix.***

Syntax

```
cmdlen = imxenc_median_filter_row(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* height of input array */
    Int16 input_height,    /* width of input array */
    Int16 output_width,    /* height of output array */
    Int16 output_height,   /* width of output array */
    Int16 compute_width,   /* height of compute block */
    Int16 compute_height,  /* width of compute block */
    Int16 median_size,     /* 3 or 5-tap median filter */
    Int16 input_type,      /* Int16/byte */
    Int16 output_type,     /* Int16/byte */
    Int16 *cmdptr);
```

Description

This function performs a median filtering operation along the rows of an input matrix. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a window of size three or five (depending on filter size) is written to the output. Like other filtering operations, a border of one pixel for a 3-tap filter and a border of two pixels for a 5-tap filter must be present on the input data to obtain the correct output size.

A block of compute_width × compute_height is computed using the median filter and written to the output buffer. The output buffer can be larger the compute window.

Example

Consider data input → 8(H) x 18(W) matrix and output → 24(H) x 8(W) matrix. The goal is compute 8(H) × 16(W) with a 3-tap median filter. The following code illustrates what the API call would look like.

```
cmdlen = imxenc_median_filter_row(
    input_ptr,      /* starting address of input */
    output_ptr,     /* starting address of output */
    18,             /* width of input array */
    8,              /* height of input array */
    24,             /* width of output array */
    8,              /* height of output array */
    16,             /* width of compute block */
    8,              /* height of compute block */
    IMXTYPE_SHORT,  /* input data type - Int16/byte */
    IMXOTYPE_SHORT  /* output data type - Int16/byte */
    cmdptr);
```

Constraints

- output_width ≥ compute_width
- input_width ≥ compute_width + 2 (3-tap filter), input_width ≥ compute_width + 4 (5-tap filter)
- input_height, output_height ≥ compute_height

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \quad (65)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (66)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1

- speedup_factor:
  - If compute_width is multiple of 8, speedup_factor = 8
  - If compute_width is multiple of 4, speedup_factor = 4
  - If compute_width is multiple of 2, speedup_factor = 2
  - Else, speedup_factor = 1.

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 [18]	1	256

[18] That is, compute_width is odd.

6.1.34 imxenc_median_filter_col

imxenc_median_filter_col ***Perform a 1-D median filter (3 or 5 tap) along the columns of an input matrix.***

Syntax

```
cmdlen = imxenc_median_filter_col(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* height of input array */
    Int16 input_height,    /* width of input array */
    Int16 output_width,    /* height of output array */
    Int16 output_height,   /* width of output array */
    Int16 compute_width,   /* height of compute block */
    Int16 compute_height,  /* width of compute block */
    Int16 median_size,     /* 3 or 5-tap median filter */
    Int16 input_type,      /* Int16/byte */
    Int16 output_type,     /* Int16/byte */
    Int16 *cmdptr);
```

Description

This function performs a median filtering operation along the columns of an input matrix. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a vertical window of size three or five (depending on filter size) is written to the output. Like other filtering operations, a border of one pixel for a 3-tap filter and a border of two pixels for a 5-tap filter must be present on the input data to obtain the correct output size.

A block of compute_width x compute_height is computed using the median filter and written to the output buffer. The output buffer can be larger the compute window.

Input data can be byte or Int16, signed or unsigned. Output data can be byte or Int16.

Example

Consider data input → 10(H) x 16(W) matrix and output → 24(H) x 8(W) matrix. The goal is to compute 8(H) x 16(W) with a 3-tap median filter. The following code illustrates what the API call would look like.

```
cmdlen = imxenc_median_filter_col(
    input_ptr,       /* starting address of input */
    output_ptr,      /* starting address of output */
    16,              /* width of input array */
    10,              /* height of input array */
    24,              /* width of output array */
    8,               /* height of output array */
    16,              /* width of compute block */
    8,               /* height of compute block */
    IMXTYPE_SHORT,   /* input data type - Int16/byte */
    IMXOTYPE_SHORT,  /* output data type - Int16/byte */
    cmdptr);
```

Constraints

- input_width, output_width ≥ compute_width
- output_height ≥ compute_height
- input_height ≥ compute_height + 2 (3-tap filter)
- input_height ≥ compute_height + 4 (5-tap filter)

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (67)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (68)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1

- speedup_factor and maximum value of compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1 (19)	1	256

(19) That is, compute_width is odd.

6.1.35 imxenc_median3x3

imxenc_median3x3 ***This function performs median filtering around a 3x3 neighborhood of each data point passed as input***

Syntax

```
cmdlen = imxenc_median3x3(
    input_ptr,           /* Int16*, starting address of input */
    output_ptr,          /* Int16*, starting address of the output array */
    scratch1_ptr,        /* Int16*, starting address of scratch1 of size */
                         /* 3*(compute_width*compute_height) elements */
    scratch2_ptr,        /* Int16*, starting address of scratch2 of size */
                         /* 3*(compute_width*(compute_height+1)) elements */
    input_width,         /* Int16, width of input */
    output_width,        /* Int16, width of output */
    computation_width,   /* Int16, computation width */
    computation_height,  /* Int16, computation_height */
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    cmdptr,              /* Int16*, starting point of command sequence in memory*/
);

/* cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

Perform median filtering around a 3x3 neighborhood of each data point passed as input. If input[i,j] is the input array and output[i,j] is the output array with i indexing the row and j indexing the column, then:

for (i,j) € [0, *compute_height* + 2 -1] × [0, *compute_width* + 2 -1]: *output*[i,j]= *median*(*input*[i, j], *input*[i, j+1], *input*[i, j+2], *input*[i+1, j], *input*[i+1, j+1], *input*[i+1, j+2], *input*[i+2, j], *input*[i+2, j+1], *input*[i+2, j+2]) (69)

In other words, each output[i,j] is the median of the 3x3 neighborhood around input[i+1, j+1].

input_ptr must point to an array of at least (compute_width+2) x (compute_height+2) elements; otherwise, the output points on the boundary will not be correct.

For better performance, scratch1_ptr should be in a different memory than input_ptr and scratch2_ptr should be in a different memory than scratch1_ptr.

The transform can be in-place by setting input_ptr= output_ptr .

Example

Consider an input array of 33x17 unsigned shorts, in which 3x3 median filtering is going to be applied to a region of interest of 32x16 elements.

```
cmdlen = imxenc_median3x3(
    input_ptr,       /* starting address of input */
    output_ptr,      /* starting address of output */
    scratch1_ptr,    /* pointer to scratch1 buffer */
    scratch2_ptr,    /* pointer to scratch2 buffer */
    33,              /* width of input */
    32,              /* width of output */
    32,              /* computation width */
    16,              /* computation height */
    IMXTYPE_USHORT,  /* Int16 */
    IMXOTYPE_SHORT,  /* Int16 */
    cmdptr           /* starting address for the command sequence */
);
```

Constraints

- input_width ≥ compute_width + 2
- Number of input rows must be ≥ compute_height + 2
- compute_height must be < 254

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles per point to perform the operation (except overhead time) is:

VICP_cycles_per_point / *speedup_factor* (70)

- VICP_cycles_per_point:

Location of input	Location of scratch1	Location of scratch2	Location of output	VICP cycles/point
IMGBUF	COEFF	IMGBUF	IMGBUF	20.5
IMGBUF	COEFF	IMGBUF	COEFF	20.5
COEFF	COEFF	IMGBUF	IMGBUF	18.6
COEFF	COEFF	IMGBUF	COEFF	18.6

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1	1	256

6.1.36 imxenc_recursiveFilterVert1stOrder

imxenc_recursiveFilterVert1stOrder ***Apply 1st order recursive filter (IIR) on 2-D data***

Syntax

```
cmdlen =  imxenc_recursiveFilterVert1stOrder(
    Int16 verticalDir,      /* 0: top to bottom, 1: bottom to top */
    Int16 *input_ptr,       /* Point to input data's top row */
    Uint16 alphaVal,        /* alpha value */
    Int16 *output_ptr,      /* Point to output data, must be in same physical */
                            /* memory as input_ptr */
    Int16 *initial_ptr,     /* Point to compute_width initial values */
                            /* must be in same physical memory as input_ptr */
    Int16 *scratchPtr,      /* Point to scratch buffer of size */
                            /* 2 16-bit words in image buffer or coef memory */
    Uint16 compute_width,   /* Number of horizontal elements in the 2-D block */
                            /* to be processed */
    Uint16 compute_height,  /* Number of vertical elements in the 2-D block */
                            /* to be processed */
    Uint16 input_type,      /* IMXTYPE_UBYTE, IMXTYPE_BYTE */
                            /* IMXTYPE_USHORT, IMXTYPE_SHORT */
    Uint16 rnd_shift,       /* Shifting parameter */
    Int16 *cmdptr)          /* Starting point of command sequence */

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr.
   If compute_width <104, cmdlen = 26 + 28*(compute_height-1) otherwise
   cmdlen = 26 + 32 */
```

Description

This function applies 1st order recursive filtering on a 2-D data set. The propagation direction verticalDir can be set to: top to bottom or bottom to top.

Mathematically the operation can be expressed as follow: if x[i, j] is the input array and y[i,j] is the output array, where i is the column index and j the row index, then for a given column C, each term y[C, j] is computed using the previous term y[C, j-1] and the present input x[C, j]:

$$y[C, j] = (1 - \alpha) \times x[C, j] + \alpha \times y[C, j - 1] \text{ , for } j > 0 \qquad (71)$$

For top to bottom propagation, j would index the rows in downward fashion, meaning j=0 would represent the top row and j=1, the row just below it. For bottom to top propagation, j would index the rows in upward fashion, meaning j=0 would represent the bottom row and j=1, the row just above it.

To calculate the terms of the first row j=0, since previous values belonging to the previous row are not available, initial values pointed by initial_ptr are used.

Example

```
/* The example code below initializes the different input argument
   of the recursive filter function */
// top to bottom filtering
verticalDir= 0 ;
// 2-D data set is made of 320x10 bytes
blockWidth= 320;
blockheight= 10;
inputType= IMXTYPE_UBYTE;

// Set input pointer to image buffer A base address
input_ptr= (Int16*)IMGBUF_A_BASE;
// Here, initial values coincides with 1st row
initial_ptr= input_ptr;
// Transform is in-place
output_ptr= input_ptr;

// scratchCoefPtr is set base address of coefficient memory
scratchCoefPtr= (Int16*)IMXCOEFFBUF_BASE;

// Set alpha Value and qShift
alphaValue= 32;
qShift= 7;

// Set saturation
cmdlen= imxenc_set_saturation(127, 127, -128, -128, cmdptr);

// Encode recursive filtering command sequence
cmdlen += imxenc_recursiveFilterVert1stOrder(
                verticalDir,
                input_ptr,
                alphaValue,
                output_ptr,
                initial_ptr,
                scratchPtr,
                blockWidth,
                blockHeight,
                inputType,
                qShift,
                cmdptr + cmdlen);
```

Constraints

- input_ptr, initial_ptr, output_ptr must reside in the same physical memory, either in the image buffer or VICP coefficient memory.
- If verticalDir= 0, the number of bytes separating initial_ptr and the first row pointed by input_ptr must not exceed 4095 bytes. To ensure this constraint is always met, place the initial values right before the input data set.
- If verticalDir= 1, the number of bytes separating initial_ptr and the last row pointed by input_ptr + compute_width*(compute_height-1) must not exceed 4095 bytes. To ensure this constraint is always met, place the initial values right after the input data set.
- scratchPtr must point to an area in VICP coef memory or image buffer for which two 16-bits words are reserved as scratch memory.
- The number of 16-bits words generated in the VICP command memory depends on the compute_width and compute_height dimensions and is equal to 26 + (compute_height-1)×28 if compute_width < 104. Otherwise it is equal to 26 + 32 words. So make sure that there is enough space left in the VICP command memory before encoding the sequence or reduce the vertical dimension of the 2-D block to process.

Tips

- The filtering can operate in-place so output_ptr= input_ptr is allowed.
- For filtering along horizontal direction, where the computation propagates column-wise, first transpose the input data using imxenc_transpose(), then call imxenc_recursiveFilterVert1stOrder() and finally transpose back.

Performance

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (72)$$

- amount_of_work =

$$2 \times \textit{compute\_width} \times \textit{compute\_height} \quad (73)$$

- memory_conflict_factor:

Location of input_ptr, initial_ptr, output_ptr	Location of scratch_ptr	memory_conflict_factor
IMGBUF	IMGBUF	5/2
IMGBUF	COEFF	3/2
COEFF	IMGBUF	1
COEFF	COEFF	3/2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1(20)	1	256

(20) That is, compute_width is odd.

6.1.37 imxenc_rgbpack

imxenc_rgbpack ***Pack R,G,B separated planes into 16 bpp RGB555 or RGB565 data***

Syntax

```
cmdlen = imxenc_rgbpack(
    input,           /* Int16 **, array of pointers to R,G,B planes. */
    tempScratch,     /* Int16* temporary scratch 2*compute_width*compute_height
                        words*/
    permScratch,     /* Int16* permanent scratch buffer of size 5 words, in VICP
                        coef mem */
    output,          /* Int16* pointer to output buffer */
    input_width,     /* Int16 width of input in number of pixels/elements */
    input_height,    /* Int16 height/rows of in number of pixels */
    output_width,    /* Int16 width/columns of output in number of pixels*/
    output_height,   /* Int16 height/rows of output in number of pixels */
    compute_width,   /* Int16 computation width in number of pixels*/
    compute_height,  /* Int16 computation height in number of pixels*/
    input_type,      /* input can be IMXTYPE_UBYTE or IMXTYPE_USHORT */
    colorformat,     /* output colorformat RGB565=0 or RGB555=1 */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr*/
```

Description

This function takes each of the 8-bit R,G,B planes of a bitmap image and pack the components together. Each element in the input R,G,B, planes are either 16-bits or 8 bits wide as indicated by the input_type parameter. In the case they are 16-bits wide, the 8 most significant bits are always zero since R,G,B values are comprised in the [0-255] range. The output format is 16 bit per pixel and is either RGB555 or RGB565. The first pixel in the output is made of the first R,G,B values of the input planes, the second is pixel in the output is made of the second R,G,B value of the input planes, etc.

Two types of scratch memory must be allocated in advance: a temporary scratch memory of size ‘2*compute_width*compute_height’ words and a permanent scratch buffer of size 5 words. The temporary scratch buffer can be re-used by other VICP functions needing a temporary scratch buffer and its content can be overwritten after the corresponding rgbpack VICP sequence is executed. Re-use of this temporary scratch buffer by other VICP functions is highly recommended in order to optimize memory allocation in the image buffer or coefficient buffers. In the other hand, the permanent scratch must never be altered by the application. The imxenc_rgbpack() function initializes the 5 words contained in the scratch and imxUpdate_rgbpack() update those when it is called. This permanent scratch must be allocated in the VICP coefficient memory.

The locations, dimensions of input and output bitmaps and the color format are fixed for the encoded VICP command. If the program wants to execute the VICP commands that performs the rgbpack algorithm for a different colorformat, it can call the function imxUpdate_rgbpack() to update an existing VICP command sequence. If any other parameters must be changed, then the application must call imxenc_rgbpack() all over again since the imxUpdate_rgbpack() only allows for changing the color format.

Constraints

compute_width must be a multiple of 8.

Performance

Performance is between 1.125 and 1.77 cycles/pixel.

The best performance can be achieved if all the scratch buffers are in VICP coefficient memory, the input in image buffer and the output in VICP coefficient memory.

6.1.38 imxenc_rgbunpack

imxenc_rgbunpack ***Unpack 16 bpp RGB555 or RGB565 data into R,G,B separated planes***

Syntax

```
cmdlen = imxenc_rgbunpack(
    input,           /* Int16*, pointer to RGB555 or RGB565 packed data */
    permScratch,     /* Int16*, 4-words permanent scratch buffer, in VICP coef
                        mem */
    output,          /* Int16**, array of pointers to R,G,B unpacked output */
    input_width,     /* Int16, width/columns of input in number pixels */
    input_height,    /* Int16, height/rows of input in number of pixels */
    output_width,    /* Int16, width/columns of output in number of pixels*/
    output_height,   /* Int16, height/rows of output in number of pixels */
    compute_width,   /* Int16, computation width in number of pixels*/
    compute_height,  /* Int16, computation height in number of pixels*/
    output_type,     /* Int16, IMXOTYPE_BYTE or IMXOTYPE_SHORT  */
    colorformat,     /* Int16, color format of the input, RGB565=0 or RGB555=1 */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*
```

Description

This function takes 16 bits-per-pixel RGB format RGB555 or RGB565 as input and unpacks it into 3 color planes: R,G,B. The elements of the output color planes can be either 16-bits or 8-bits wide depending on the value of output_type. In both cases, only 8 bits are used to encode the R,G,B values which are in the [0, 255] range.

The 4 words permanent scratch must never be altered by the application. The imxenc_rgbunpack() function initializes the 4 words contained in the scratch and imxUpdate_rgbunpack() update those when it is called. This permanent scratch must be allocated in the VICP coefficient memory.

The locations, dimensions of input and output bitmaps and the color format are fixed for the encoded VICP command. If the program wants to execute the VICP commands that performs the rgbunpack algorithm for a different colorformat, it can call the function imxUpdate_rgbunpack() to update an existing VICP command sequence. If any other parameters must be changed, then the application must call imxenc_rgbunpack() all over again since the imxUpdate_rgbunpack() only allows for changing the color format.

Constraints

compute_width must be a multiple of 8.

Performance

Performance is around 1.14 cycles/pixel.

The best performance can be achieved if the scratch buffer is in VICP coefficient memory, the input and output in image buffer.

6.1.39 imxenc_rotate

imxenc_rotate ***Rotation of a data matrix by 90, 180, or 270 degrees.***

Syntax

```
cmdlen = imxenc_rotate(
    input_ptr;       /* Int16*, starting address of input array */
    coeff_ptr,       /* Int16*, starting address of scaling coefficient */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    angle,           /* Int16,  Rotation angle 90, 180, or 270 */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function rotates a submatrix of size compute_width x compute_height of the input data matrix (size input_width x input_height), and writes the rotated submatrix into the output matrix of size output_width x output_height, aligned to the top left corner.

Each input data point can be scaled by the scaling factor defined in coef_ptr, and being rounded and saturated prior to write out. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Input and output are each selectable to be byte or Int16 type.

Example

Rotating a 12x6 block out of an array of size 18x11 bu 90 degree and write into an array of size 8x16.

```
cmdlen = imxenc_rotate(
    data_ptr,          /* point to input data array */
    coef_ptr,          /* point to scaling coefficient */
    output_ptr,        /* point to output array */
    18,                /* width of input data array */
    11,                /* height of input data array */
    8,                 /* width of output data array */
    16,                /* height of output data array */
    12,                /* computation width */
    6,                 /* computation height */
    IMXTYPE_SHORT,     /* input signed Int16 */
    IMXTYPE_SHORT,     /* coeff signed Int16 */
    IMXOTYPE_SHORT,    /* output signed Int16 */
    90,                /* rotation by 90 degrees */
    0,                 /* number of bits to downshift */
    cmdptr             /* starting point for command sequence in memory */
);
```

Constraints

- Rotation by 90 / 270 degrees:
  - output_width ≥ compute_height ≥ input_height
  - output_height ≥ compute_width ≥ input_width
- Operand types are IMXTYPE_UBYTE (unsigned byte), IMXTYPE_SHORT, IMXTYPE_USHORT (unsigned Int16).
- output_type can only be IMXOTYPE_SHORT
- compute_width, compute_height and input_height must be <= 256.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \quad (74)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (75)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

6.1.40 imxenc_set_saturation

imxenc_set_saturation ***Set saturation parameters***

Syntax

```
cmdlen = imxenc_set_saturation(
    sat_high,      /* Int32 saturation upper bound compare value */
    sat_high_set,  /* Int32 value set if >= sat_high */
    sat_low,       /* Int32 saturation lower bound compare value */
    sat_low_set,   /* Int32 value set if < sat_low */
    cmdptr         /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr*/
```

Description

This function sets saturation upper and lower bounds for subsequent VICP computation. sat_high_set and sat_low_set must both fall at the same time within the signed range S= [-32768, 32767] or within the unsigned range U= [0, 65535]. For instance sat_high_set=65535 and sat_low_set= -32768 would be an illegal combination since one falls in the range S and the other one falls in the range U. Generally if the command sequence that follows imxenc_set_saturation() contains computation commands that involve the input type IMXTYPE_SHORT then the range S should be used.

For normal saturation, sat_high = sat_high_set = upper bound, and sat_low = sat_low_set = lower bound

This API must be called at the beginning of each commands sequence encoding since the saturation values are lost each time the imxenc_sleep opcode is encountered.

6.1.41 imxenc_save_sat_parameters

imxenc_save_sat_parameters ***Save saturation parameters previously set by imxenc_set_saturation()***

Syntax

```
imxenc_save_sat_parameters(IMX_SatParamsStruct *imxsatparams);
```

Arguments

imxsatparams	IMX_SatParamsStruct*	Pointer to structure that will hold the values of the saturation parameters passed to imxenc_set_saturation(), the last time it was called.

The structure IMX_SatParamsStruct is defined as follow:

```
typedef struct IMX_SatParams{
    Int16 sat_unsigned;
    Int16 sat_high;
    Int16 sat_high_set;
    Int16 sat_low;
    Int16 sat_low_set;
} IMX_SatParamsStruct;
```

Description

This function is useful in the case of function A calling imxenc_set_saturation() and a function B. Function B calls imxenc_set_saturation() and sets its own saturation parameters but needs to restore the saturation parameters set by function A before returning. To achieve that, function B calls imxenc_save_sat_parameters() before calling imxenc_set_saturation() and calls imxenc_restore_sat_parameters() before returning.

6.1.42 imxenc_restore_sat_parameters

imxenc_restore_sat_parameters ***Restore saturation parameters saved by imxenc_save_sat_parameters()***

Syntax

```
cmdlen = imxenc_restore_sat_parameters(IMX_SatParamsStruct *imxsatparams, Int16 * cmd_ptr);
```

Arguments

imxsatparams	IMX_SatParamsStruct*	Pointer to structure that will hold the values of the saturation parameters.
cmd_ptr	Int16*	Pointer to location in memory where the set saturation command is generated.
cmdlen	Int16	Number of words generated

The structure IMX_SatParamsStruct is defined as follow:

```
typedef struct IMX_SatParams{
    Int16 sat_unsigned;
    Int16 sat_high;
    Int16 sat_high_set;
    Int16 sat_low;
    Int16 sat_low_set;
} IMX_SatParamsStruct;
```

Description

The behavior of the function is equivalent to calling imxenc_set_saturation() with input arguments equal to the different members of IMX_SatParamsStruct. This function is useful in the case of function A calling imxenc_set_saturation() and a function B. Function B calls imxenc_set_saturation() and sets its own saturation parameters but needs to restore the saturation parameters set by function A before returning. To achieve that, function B calls imxenc_save_sat_parameters() before calling imxenc_set_saturation() and calls imxenc_restore_sat_parameters() before returning.

6.1.43 imxenc_sleep

imxenc_sleep ***Puts VICP to sleep***

Syntax

```
cmdlen = imxenc_sleep(
    cmdptr   /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

Write the sleep command into the command sequence. The VICP module will stop the execution of the commands sequence when this particular command is reached.

6.1.44 imxenc_sobelx

imxenc_sobelx ***Perform the Sobel filtering in horizontal (x) direction***

Syntax

```
cmdlen = imxenc_sobelx(
    input_ptr,       /* Int16*, starting address of 1st input */
    coeff_ptr,       /* Int16*, starting address of coefficients, reserve 3 16-bits words */
    temp_ptr,        /* Int16*, starting address of the temporary storage array of size */
                     /* compute_width x (compute_height+2) 16-bits words*/
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output (3 is recommended) */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );
/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

This function calculates the horizontal gradient approximation on a block of data. The result is in the output block of dimension [compute_width × compute_height]. Kernel size is [3 × 3] and has the following coefficients:

$$\text{coeff} = \begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix}$$

Width of input accessed for this operation is (compute_width + 2), and height of input accessed is (compute_height + 2). If both input and output data have the same bit-width, the recommended value for round_shift is 3, because the absolute sum of filter coefficients is 8. Choosing the value of 3 guarantees that saturation will not occur, while rounding is minimal.

coeff_ptr needs to point to three free 16-bits words in coefficient memory. The function will initialize these locations with the separable sobel filter coefficients. Do not share the region pointed by this coeff_ptr with another imxenc_<computation> function such as imxenc_sobely(). If you do share, then the coefficients set by imxenc_sobelx() are at risk to be overwritten by the other imxenc_<computation> function.

Example

Consider an input block of 20(W) × 12(H), output block of 20(W) × 8(H), and the computed block size of 16(W) × 8(H), round down by 3 bits before output.

```
cmdlen= imxenc_sobelx(
    input_ptr,       /* starting address of input array */
    coeff_ptr,       /* starting address of coefficient array */
    temp_ptr,        /* starting address of temporary storage array */
    output_ptr,      /* starting address of output array */
    20,              /* width of input block */
    12,              /* height of input block */
    20,              /* width of output block */
    8,               /* height of output block */
    16,              /* computation width */
    8,               /* computation height */
    IMXTYPE_SHORT,   /* byte, short */
    IMXOTYPE_SHORT,  /* byte, short */
    3                 /* number of bits to downshift before output */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- compute_width must be a multiple of 8.
- input_width ≥ compute_width + 2, output_width ≥ compute_width
- input_height ≥ compute_height + 2, output_height ≥ compute_height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed.
- coeff_ptr must point to three free 16-bits words located in coefficient memory.
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} \,/\, \textit{speedup\_factor} \qquad (76)$$

- amount_of_work =

$$4 \times \textit{compute\_width} \times \textit{compute\_height} \qquad (77)$$

- memory_conflict_factor:

Location of input_ptr	Location of temp	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	8/3
IMGBUF	IMGBUF	COEFF	7/6
IMGBUF	COEFF	IMGBUF	1
IMGBUF	COEFF	COEFF	7/6
COEFF	IMGBUF	IMGBUF	7/6
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	7/6
COEFF	COEFF	COEFF	4/3

- speedup_factor and maximum value for compute_width:

compute_width × dnsmpl_horz multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1[1]	1	256

[1] That is, compute_width is odd.

6.1.45 imxenc_sobely

imxenc_sobely ***Perform the Sobel filtering in vertical (y) direction***

Syntax

```
cmdlen = imxenc_sobely(
    input_ptr,       /* Int16*, starting address of 1st input */
    coeff_ptr,       /* Int16*, starting address of coefficients, reserve 3 16-bits words */
    temp_ptr,        /* Int16*, starting address of the temporary storage array of size */
                     /* compute_width x (compute_height+2) 16-bits words*/
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output (3 is recommended) */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );
/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr */
```

Description

This function calculates the vertical gradient approximation on a block of data. The result is in the output block of dimension [compute_width × compute_height]. Kernel size is [3 × 3] and has the following coefficients:

$$\text{coeff} = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

Width of input accessed for this operation is (compute_width + 2), and height of input accessed is (compute_height + 2). If both input and output data have the same bit-width, the recommended value for round_shift is 3, because the absolute sum of filter coefficients is 8. Choosing the value of 3 guarantees that saturation will not occur, while rounding is minimal.

coeff_ptr needs to point to three free 16-bits words in coefficient memory. The function will initialize these locations with the separable sobel filter coefficients. Do not share the region pointed by this coeff_ptr with another imxenc_<computation> function such as imxenc_sobelx(). If you do share, then the coefficients set by imxenc_sobely() are at risk to be overwritten by the other imxenc_<computation> function.

Example

Consider an input block of 20(W) × 12(H), output block of 20(W) × 8(H), and the computed block size of 16(W) × 8(H), round down by 3 bits before output.

```
cmdlen= imxenc_sobely(
    input_ptr,       /* starting address of input array */
    coeff_ptr,       /* starting address of coefficient array */
    temp_ptr,        /* starting address of temporary storage array */
    output_ptr,      /* starting address of output array */
    20,              /* width of input block */
    12,              /* height of input block */
    20,              /* width of output block */
    8,               /* height of output block */
    16,              /* computation width */
    8,               /* computation height */
    IMXTYPE_SHORT,   /* byte, short */
    IMXOTYPE_SHORT,  /* byte, short */
    3                 /* number of bits to downshift before output */
    *cmdptr          /* starting point for command sequence in memory */
);
```

Constraints

- compute_width must be a multiple of 8.
- input_width ≥ compute_width + 2, output_width ≥ compute_width
- input_height ≥ compute_height + 2, output_height ≥ compute_height
- To prevent write after read (WAR) hazards, input data should not be overwritten by output unless all computations involving that input location are completed.
- coeff_ptr must point to three free 16-bits words located in coefficient memory.
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} \;/\; \mathit{speedup\_factor} \qquad (78)$$

- amount_of_work =

$$4 \times \mathit{compute\_width} \times \mathit{compute\_height} \qquad (79)$$

- memory_conflict_factor:

Location of input_ptr	Location of temp	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	8/3
IMGBUF	IMGBUF	COEFF	7/6
IMGBUF	COEFF	IMGBUF	1
IMGBUF	COEFF	COEFF	7/6
COEFF	IMGBUF	IMGBUF	7/6
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	7/6
COEFF	COEFF	COEFF	4/3

- speedup_factor and maximum value for compute_width:

compute_width × dnsmpl_horz multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1[2]	1	256

[2] That is, compute_width is odd.

6.1.46 imxenc_sum

imxenc_sum ***Perform summation of an array***

Syntax

```
cmdlen = imxenc_sum(
    input_ptr;       /* Int16*, starting address of input array */
    scaler_ptr,      /* Int16*, starting address of scaling coefficient */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    scaler_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    sum_mode,        /* Int16, Number of sum per cycles. Ignored here */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function sums up the elements of a scalar times a matrix of size [compute_width x compute_height]. Sum_mode is ignored. The resulting sum is written to the location pointed to by output_ptr.

Example

Consider summing up an input matrix of size 16x16. one_ptr points to a Int16 1.

```
cmdlen = imxenc_sum(
    input_ptr,       /* starting address of input array */
    one_ptr,         /* address of a scaler, normally containing 1*/
    temp,            /* address of output */
    16,              /* width of input array */
    16,              /* height of input array */
    16,              /* computation height */
    16,              /* computation width */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXTYPE_SHORT,   /* byte, Int16 */
    IMXOTYPE_SHORT,  /* byte, Int16 */
    0,               /* number of bits to downshift, should be 0*/
    0                /* 0 */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- input_width ≥ compute_width
- input_height ≥ compute_height
- compute_height and input_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (80)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} \quad (81)$$

- memory_conflict_factor:

Location of input_ptr	Location of scaler_ptr	Location of output_ptr	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	2 + 16 / compute_width × compute_height
IMGBUF	IMGBUF	COEFF	1 + 8 / compute_width × compute_height
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / compute_width × compute_height
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 / compute_width × compute_height

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	2048
4	4	1024
2	2	512
1[3]	1	256

[3] That is, compute_width is odd.

6.1.47 imxenc_sum_cfa

imxenc_sum_cfa ***Perform summation of an array where a partial sum is obtained for each element of a 2x2 tile (CFA pattern)***

Syntax

```
cmdlen = imxenc_sum_cfa(
    input_ptr;        /* Int16*, starting address of input array */
    scaler_ptr,       /* Int16*, starting address of scaling coefficient */
    output_ptr,       /* Int16*, starting address of the output array */
    input_width,      /* Int16, width of the input array */
    input_height,     /* Int16, height of the input array */
    compute_width     /* Int16, computation width */
    compute_height,   /* Int16, computation height */
    input_type,       /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                      /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    scaler_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                      /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,      /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,      /* Int16, number of bits to downshift before output */
    cmdptr            /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

The input matrix is composed of 2x2 tiles. The four elements of each tile are 4 independent CFA components. Imxenc_sum_cfa sums each of those components independently across the input matrix, producing 4 partial sums in the output. The order of appearance in the output is:

1. 1st output: sum of the top left component of the tiles
2. 2nd output: sum of the top right component of the tiles
3. 3rd output: sum of the bottom left component of the tiles
4. 4th output: sum of the bottom right component of the tiles

Example

Consider summing up an input matrix of size 16x16. one_ptr points to a Int16 1. sum_mode = 8 is used to produce 8 partial sums into a temp array. Host DSP then adds up the 4 partial sums.

```
cmdlen = imxenc_sum_cfa(
    input_ptr,        /* starting address of input array */
    one_ptr,          /* address of a scaler, normally containing 1*/
    temp,             /* address of output */
    16,               /* width of input array */
    16,               /* height of input array */
    16,               /* computation height */
    16,               /* computation width */
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXTYPE_SHORT,    /* byte, Int16 */
    IMXOTYPE_SHORT,   /* byte, Int16 */
    0,                /* number of bits to downshift, should be 0*/
    cmdptr            /* starting point for command sequence in memory */
);
```

Constraints

- compute_width must be multiple of 2 and ≤ 512
- compute_height and input_height must be ≤ 512
- input_width ≥ compute_width
- input_height ≥ compute_height

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \,/\, speedup\_factor \quad (82)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (83)$$

- memory_conflict_factor:

Location of data	**Location of coeff**	**Location of output**	**memory_conflict_factor**
IMGBUF	IMGBUF	IMGBUF	2 + 16 / compute_width × compute_height
IMGBUF	IMGBUF	COEFF	1 + 8 / compute_width × compute_height
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	1 + 8 / compute_width × compute_height
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	1 + 8 / compute_width × compute_height

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	**speedup_factor**	**Maximum value of compute_width**
2	2	512

6.1.48 imxenc_sum_abs_diff

imxenc_sum_abs_diff *Perform sum of absolute differences on 2-D arrays*

Syntax

```
cmdlen = imxenc_sum_abs_diff(
    target_ptr;      /* Int16*, starting address of target array */
    ref_ptr,         /* Int16*, starting address of reference array */
    output_ptr,      /* Int16*, starting address of the output array */
    interm_ptr,      /* Not used, for backward compatibility */
    zero_ptr,        /* Not used, for backward compatibility */
    block_width,     /* Int16, width of matching block */
    block_height,    /* Int16, height of matching block */
    target_width,    /* Int16, width of the target array */
    target_height,   /* Int16, height of the target array */
    ref_width        /* Int16, width of the reference array */
    ref_height,      /* Int16, height of the reference array */
    step_horz,       /* Int16, horizontal step between matches */
    step_vert,       /* Int16, vertical step between matches */
    nsteps_horz,     /* Int16, number of steps horizontally */
    nsteps_vert,     /* Int16, number of steps vertically */
    target_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    ref_type,        /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function takes a target 2-D array, target_width x target_height, and matches it against MULTIPLE 2-D windows each of block_width x block_height, inside a 2-D reference array of ref_width x ref_height. Sums of absolute differences are computed.

Matches are done nsteps_horz x nsteps_vert times, each time stepping to the right by step_horz points until reaching nsteps_horz steps, then recalibrating to the first column and stepping down by step_vert points.

nsteps_horz x nsteps_vert output elements are written to output array of outp_width x outp_height.

Target, reference, and output array elements are each selectable to be byte or Int16 type.

Example

Consider 16x16 block matching, between a 16x16 Int16 target and a 32x32 byte reference, for 2x2 steps with step size of 8x8. Integer output, no rounding down.

```
cmdlen = sum_abs_diff(
    target_ptr,       /* point to target array */
    ref_ptr,          /* point to reference array */
    output_ptr,       /* point to output array */
    interm_ptr,       /* Not used, for backward compatibility */
    zero_ptr,         /* Not used, for backward compatibility */
    16,               /* width of matching block */
    16,               /* height of matching block */
    16,               /* width of target array */
    16,               /* height of target array */
    32,               /* width of reference array */
    32,               /* height of reference array */
    8,                /* horizontal offset between matches */
    8,                /* vertical offset between matches */
    2,                /* number of steps horizontally */
    2,                /* number of steps vertically */
    IMXTYPE_SHORT,    /* target signed Int16/unsigned byte */
    IMXTYPE_UBYTE,    /* ref signed Int16/unsigned byte */
    IMXOTYPE_SHORT,   /* output signed Int16/unsigned byte */
    0,                /* number of bits to downshift before output */
    cmdptr            /* starting point for command sequence in memory */
);
```

Constraints

- block_width must be a multiple of 4 or multiple of 8 (optimum performance)
- target_width ≥ block_width, target_height ≥ block_height
- ref_width ≥ block_width + ((nsteps_horz - 1) × step_horz)
- ref_height ≥ block_height + ((nsteps_vert - 1) × step_vert)
- output_width ≥ nsteps_horz, outp_height ≥ nsteps_vert
- ref_height and block_height must be < 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \tag{84}$$

- amount_of_work =

$$\textit{block\_width} \times \textit{block\_height} \times \textit{nsteps\_horz} \times \textit{nsteps\_vert} \tag{85}$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for block_width:

block_width multiple only of	speedup_factor	Maximum value of block_width
8	8	2048
4	4	1024

6.1.49 imxenc_table_lookup

imxenc_table_lookup ***Generic table lookup operation.***

Syntax

```
cmdlen = imxenc_table_lookup(
    input1_ptr;          /* Int16*, starting address of input array */
    input2_ptr,          /* Int16*, starting address of table lookup, 128-bits
                            aligned */
    output_ptr,          /* Int16*, starting address of the output array */
    compute_elements,    /* Int16, number of elements to lookup*/
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    table_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    thread,              /* Int16, 1, 2, 4 */
    cmdptr               /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

The input array contains the indexes of the output elements in the lookup table. This API will simply fetch the corresponding elements in the lookup table and write them in the output array. It is possible to perform thread=1, 2 or 4 independent lookups per clock cycle. Here is the layout of the input array and table array for different values of thread:

Thread=1:

There is a single table in the lookup table array and all elements of the input array are indexes in this unique table.

Thread=2:

There are two tables interleaved in the lookup table array. If the tables are called T_1 and T_2, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

$t_1{}^1$ $t_1{}^2$ $t_1{}^3$ $t_1{}^4$ $t_1{}^5$ $t_1{}^6$ $t_1{}^7$ $t_1{}^8$

$t_2{}^1$ $t_2{}^2$ $t_2{}^3$ $t_2{}^4$ $t_2{}^5$ $t_2{}^6$ $t_2{}^7$ $t_2{}^8$

$t_1{}^9$ $t_1{}^{10}$ $t_1{}^{11}$ $t_1{}^{12}$ $t_1{}^{13}$ $t_1{}^{14}$ $t_1{}^{15}$ $t_1{}^{16}$

$t_2{}^9$ $t_2{}^{10}$ $t_2{}^{11}$ $t_2{}^{12}$ $t_2{}^{13}$ $t_2{}^{14}$ $t_2{}^{15}$ $t_2{}^{16}$

...

}

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

$t_1{}^1$ $t_1{}^2$ $t_1{}^3$ $t_1{}^4$

$t_2{}^1$ $t_2{}^2$ $t_2{}^3$ $t_2{}^4$

$t_1{}^5$ $t_1{}^6$ $t_1{}^7$ $t_1{}^8$

$t_2{}^5$ $t_2{}^6$ $t_2{}^7$ $t_2{}^8$

...

}

where $t_1{}^k$ is the kth element of table T_1 and $t_2{}^k$ is the kth element of table T_2.

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={$I_1{}^1$, $I_2{}^2$, $I_1{}^3$, $I_2{}^4$, $I_1{}^5$, $I_2{}^6$, $I_1{}^7$, …}

where $I_1{}^k$ is an index of an element in T_1 and $I_2{}^k$ is an index of an element of table T_2.

Thread=4:

There are four tables interleaved in the lookup table array. If the tables are called T_1, T_2, T_3, and T_4, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

$t_1^{\ 1}$ $t_1^{\ 2}$ $t_1^{\ 3}$ $t_1^{\ 4}$

$t_2^{\ 1}$ $t_2^{\ 2}$ $t_2^{\ 3}$ $t_2^{\ 4}$

$t_3^{\ 1}$ $t_3^{\ 2}$ $t_3^{\ 3}$ $t_3^{\ 4}$

$t_4^{\ 1}$ $t_4^{\ 2}$ $t_4^{\ 3}$ $t_4^{\ 4}$

$t_1^{\ 5}$ $t_1^{\ 6}$ $t_1^{\ 7}$ $t_1^{\ 8}$

$t_2^{\ 5}$ $t_2^{\ 6}$ $t_2^{\ 7}$ $t_2^{\ 8}$

...

}

where $t_1^{\ k}$ is the kth element of table T_1, $t_2^{\ k}$ is the kth element of table T_2, $t_3^{\ k}$ is the kth element of table T_3, and $t_4^{\ k}$ is the kth element of table T_4.

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

$t_1^{\ 1}$ $t_1^{\ 2}$

$t_2^{\ 1}$ $t_2^{\ 2}$

$t_3^{\ 1}$ $t_3^{\ 2}$

$t_4^{\ 1}$ $t_4^{\ 2}$

...

}

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={$I_1^{\ 1}$, $I_2^{\ 2}$, $I_3^{\ 3}$, $I_4^{\ 4}$, $I_1^{\ 5}$, $I_2^{\ 6}$, $I_3^{\ 7}$, …}

where $I_1^{\ k}$ is an index of an element in T_1, $I_2^{\ k}$ is an index of an element of table T_2, and $I_3^{\ k}$ is an index of an element of table T_3, and $I_4^{\ k}$ is an index of an element of table T_4.

The function imx_formatTLU() can be used to generate the array table_base[ ] for all these multi-thread use cases.

The input array and the lookup table cannot both be in an image buffer.

Each input data point is rounded and saturated prior to lookup. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Negative indexing is permitted when the input data array's elements are signed. In this case, the lookup table's base address points to the element whose index is 0.

Example

Single table lookup on 512 entries.

```
cmdlen = imxenc_table_lookup(
    data_ptr,        /* point to input data array */
    table_base,      /* point to lookup table */
    output_ptr,      /* point to output array */
    512,             /* width of input data array
    IMXTYPE_USHORT,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,  /* output Int16/byte */
    0,               /* number of bits to downshift input before lookup */
    1,               /* single table */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- The input data array and lookup table cannot both be in an image buffer (A or B).
- The lookup table must be 128-bits aligned.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

amount_of_work × *memory_conflict_factor* / *speedup_factor* (86)

- amount_of_work = *compute_elements*
- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor:

thread	speedup_factor
4	4
2	2
1	1

6.1.50 imxenc_tables_lookup

imxenc_tables_lookup ***Generic table lookup operation, allowing multiple lookups per input.***

Syntax

```
cmdlen = imxenc_tables_lookup(
    input1_ptr,          /* Int16*, starting address of input array */
    input2_ptr,          /* Int16*, starting address of table lookup, 128-bits
                            aligned*/
    output_ptr,          /* Int16*, starting address of the output array */
    numPerLookup,        /* Int16, number of input points per lookup cycle,
                            1 2 or 4 */
    computePoints,       /* Int16, total number of input points per table */
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    table_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    numThreadsPerTable,  /* Int16, 1, 2, 4 */
    numTables,           /* Int16, number of tables */
    data_offset,         /* Int16, offset between input sets */
    table_offset,        /* Int16, offset between tables */
    out_offset,          /* Int16, offset between output sets */
    cmdptr               /* Int16*, starting point of command sequence in memory*/
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

The difference between the function imxenc_table_lookup() is that each input element pointed by input1_ptr can produce up to 4 outputs, as specified by the input argument numPerLookup.

Table 6-5 summarizes what the first four outputs would look like for different combinations of numPerLookup and numThreadsPerTable input arguments. Highlighted in bold are the most useful use cases.

Table 6-5. Initial Four Outputs for imxenc_tables_lookup Function

numPer Lookup	numThreads Per Table	Output
1	**1**	**output[0]=T1[input[0]], output[1]=T1[input[1]], output[2]=T1[input[2]], output[3]=T1[input[3]]**
2	1	output[0]=T1[input[0]], output[1]=T1[input[2]], output[2]=T1[input[4]], output[3]=T1[input[6]]
4	1	output[0]=T1[input[0]], output[1]=T1[input[4]], output[2]=T1[input[8]], output[3]=T1[input[12]]
1	**2**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T1[input[1]], output[3]=T2[input[1]]**
2	**2**	**output[0]=T1[input[0]], output[1]=T2[input[1]], output[2]=T1[input[2]], output[3]=T2[input[3]]**
4	2	output[0]=T1[input[0]], output[1]=T2[input[2]], output[2]=T1[input[4]], output[3]=T2[input[6]]
1	**4**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T3[input[0]], output[3]=T4[input[0]]**
2	**4**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T3[input[1]], output[3]=T4[input[1]]**
4	**4**	**output[0]=T1[input[0]], output[1]=T2[input[1]], output[2]=T3[input[2]], output[3]=T4[input[3]]**

Also the function is able to iterate the lookup process several times over the same input data set but with different tables if numTables > 1.

Here is the layout of the input array and table array for different values of numThreadsPerTable:

Thread=1:

There is a single table in the lookup table array and all elements of the input array are indexes in this unique table.

Thread=2:

There are two tables interleaved in the lookup table array. If the tables are called T_1 and T_2, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

t_1^{1} t_1^{2} t_1^{3} t_1^{4} t_1^{5} t_1^{6} t_1^{7} t_1^{8}

t_2^{1} t_2^{2} t_2^{3} t_2^{4} t_2^{5} t_2^{6} t_2^{7} t_2^{8}

t_1^{9} t_1^{10} t_1^{11} t_1^{12} t_1^{13} t_1^{14} t_1^{15} t_1^{16}

t_2^{9} t_2^{10} t_2^{11} t_2^{12} t_2^{13} t_2^{14} t_2^{15} t_2^{16}

...

}

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

t_1^{1} t_1^{2} t_1^{3} t_1^{4}

t_2^{1} t_2^{2} t_2^{3} t_2^{4}

t_1^{5} t_1^{6} t_1^{7} t_1^{8}

t_2^{5} t_2^{6} t_2^{7} t_2^{8}

...

}

where t_1^{k} is the kth element of table T_1 and t_2^{k} is the kth element of table T_2.

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={I_1^{1}, I_2^{2}, I_1^{3}, I_2^{4}, I_1^{5}, I_2^{6}, I_1^{7}, …}

where I_1^{k} is an index of an element in T_1 and I_2^{k} is an index of an element of table T_2.

Thread=4:

There are four tables interleaved in the lookup table array. If the tables are called T_1, T_2, T_3, and T_4, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

t_1^{1} t_1^{2} t_1^{3} t_1^{4}

t_2^{1} t_2^{2} t_2^{3} t_2^{4}

t_3^{1} t_3^{2} t_3^{3} t_3^{4}

t_4^{1} t_4^{2} t_4^{3} t_4^{4}

t_1^{5} t_1^{6} t_1^{7} t_1^{8}

t_2^{5} t_2^{6} t_2^{7} t_2^{8}

...

}

where t_1^{k} is the kth element of table T_1, t_2^{k} is the kth element of table T_2, t_3^{k} is the kth element of table T_3, and t_4^{k} is the kth element of table T_4.

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

$t_1{}^1$ $t_1{}^2$

$t_2{}^1$ $t_2{}^2$

$t_3{}^1$ $t_3{}^2$

$t_4{}^1$ $t_4{}^2$

...

}

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={$I_1{}^1$, $I_2{}^2$, $I_3{}^3$, $I_4{}^4$, $I_1{}^5$, $I_2{}^6$, $I_3{}^7$, …}

The function imx_formatTLU() can be used to generate the array table_base[ ] for all these multi-thread use cases.

The input array and the lookup table cannot both be in an image buffer.

Each input data point is rounded and saturated prior to lookup. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Negative indexing is permitted when the input data array's elements are signed. In this case, the lookup table's base address points to the element whose index is 0.

Example

Single table lookup on 512 entries.

```
cmdlen = imxenc_tables_lookup(
    data_ptr,        /* point to input data array */
    table_base,      /* point to lookup table */
    output_ptr,      /* point to output array */
    512,             /* width of input data array
    IMXTYPE_USHORT,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,  /* output Int16/byte */
    0,               /* number of bits to downshift input before lookup */
    1,               /* single table */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- The input data array and lookup table cannot both be in an image buffer (A or B).
- The lookup table must be 128-bits aligned.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

amount_of_work × *memory_conflict_factor* / *speedup_factor* (87)

- amount_of_work = *compute_elements*
- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor:

thread	speedup_factor
4	4
2	2
1	1

6.1.51 imxenc_table_lookup2D

imxenc_table_lookup2D ***Generic table lookup operation.***

Syntax

```
cmdlen = imxenc_tables_lookup(
    input1_ptr,          /* Int16*, starting address of input array */
    input2_ptr,          /* Int16*, starting address of table lookup,
                            128-bits aligned */
    output_ptr,          /* Int16*, starting address of the output array */
    numPerLookup,        /* Int16, number of input points per lookup cycle,
                            1 2 or 4 */
    computePoints,       /* Int16, total number of input points per table */
    input_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    table_type,          /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                         /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,         /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,         /* Int16, number of bits to downshift before output */
    numThreadsPerTable,  /* Int16, 1, 2, 4 */
    numTables,           /* Int16, number of tables */
    data_offset,         /* Int16, offset between input sets */
    table_offset,        /* Int16, offset between tables */
    out_offset,          /* Int16, offset between output sets */
    cmdptr               /* Int16*, starting point of command sequence in memory*/
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

The difference between the function imxenc_table_lookup() is that each input element pointed by input1_ptr can produce up to 4 outputs, as specified by the input argument numPerLookup.

Table 6-6 summarizes what the first four outputs would look like for different combinations of numPerLookup and numThreadsPerTable input arguments. Highlighted in bold are the most useful use cases.

Table 6-6. Initial Four Outputs for imxenc_tables_lookup Function

numPer Lookup	numThreads Per Table	Output
1	**1**	**output[0]=T1[input[0]], output[1]=T1[input[1]], output[2]=T1[input[2]], output[3]=T1[input[3]]**
2	1	output[0]=T1[input[0]], output[1]=T1[input[2]], output[2]=T1[input[4]], output[3]=T1[input[6]]
4	1	output[0]=T1[input[0]], output[1]=T1[input[4]], output[2]=T1[input[8]], output[3]=T1[input[12]]
1	**2**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T1[input[1]], output[3]=T2[input[1]]**
2	**2**	**output[0]=T1[input[0]], output[1]=T2[input[1]], output[2]=T1[input[2]], output[3]=T2[input[3]]**
4	2	output[0]=T1[input[0]], output[1]=T2[input[2]], output[2]=T1[input[4]], output[3]=T2[input[6]]
1	**4**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T3[input[0]], output[3]=T4[input[0]]**
2	**4**	**output[0]=T1[input[0]], output[1]=T2[input[0]], output[2]=T3[input[1]], output[3]=T4[input[1]]**
4	**4**	**output[0]=T1[input[0]], output[1]=T2[input[1]], output[2]=T3[input[2]], output[3]=T4[input[3]]**

Also the function is able to iterate the lookup process several times over the same input data set but with different tables if numTables > 1.

Here is the layout of the input array and table array for different values of numThreadsPerTable:

Thread=1:

There is a single table in the lookup table array and all elements of the input array are indexes in this unique table.

Thread=2:

There are two tables interleaved in the lookup table array. If the tables are called T_1 and T_2, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

$t_1^{\ 1}$	$t_1^{\ 2}$	$t_1^{\ 3}$	$t_1^{\ 4}$	$t_1^{\ 5}$	$t_1^{\ 6}$	$t_1^{\ 7}$	$t_1^{\ 8}$
$t_2^{\ 1}$	$t_2^{\ 2}$	$t_2^{\ 3}$	$t_2^{\ 4}$	$t_2^{\ 5}$	$t_2^{\ 6}$	$t_2^{\ 7}$	$t_2^{\ 8}$
$t_1^{\ 9}$	$t_1^{\ 10}$	$t_1^{\ 11}$	$t_1^{\ 12}$	$t_1^{\ 13}$	$t_1^{\ 14}$	$t_1^{\ 15}$	$t_1^{\ 16}$
$t_2^{\ 9}$	$t_2^{\ 10}$	$t_2^{\ 11}$	$t_2^{\ 12}$	$t_2^{\ 13}$	$t_2^{\ 14}$	$t_2^{\ 15}$	$t_2^{\ 16}$

...

}

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

$t_1^{\ 1}$	$t_1^{\ 2}$	$t_1^{\ 3}$	$t_1^{\ 4}$
$t_2^{\ 1}$	$t_2^{\ 2}$	$t_2^{\ 3}$	$t_2^{\ 4}$
$t_1^{\ 5}$	$t_1^{\ 6}$	$t_1^{\ 7}$	$t_1^{\ 8}$
$t_2^{\ 5}$	$t_2^{\ 6}$	$t_2^{\ 7}$	$t_2^{\ 8}$

...

}

where $t_1^{\ k}$ is the kth element of table T_1 and $t_2^{\ k}$ is the kth element of table T_2.

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={$I_1^{\ 1}$, $I_2^{\ 2}$, $I_1^{\ 3}$, $I_2^{\ 4}$, $I_1^{\ 5}$, $I_2^{\ 6}$, $I_1^{\ 7}$, …}

where $I_1^{\ k}$ is an index of an element in T_1 and $I_2^{\ k}$ is an index of an element of table T_2.

Thread=4:

There are four tables interleaved in the lookup table array. If the tables are called T_1, T_2, T_3, and T_4, then the memory layout of the lookup table array pointed by table_base must look like the following:

If table_type=IMXTYPE_UBYTE or IMXTYPE_BYTE

table_base[ ]={

$t_1^{\ 1}$	$t_1^{\ 2}$	$t_1^{\ 3}$	$t_1^{\ 4}$
$t_2^{\ 1}$	$t_2^{\ 2}$	$t_2^{\ 3}$	$t_2^{\ 4}$
$t_3^{\ 1}$	$t_3^{\ 2}$	$t_3^{\ 3}$	$t_3^{\ 4}$
$t_4^{\ 1}$	$t_4^{\ 2}$	$t_4^{\ 3}$	$t_4^{\ 4}$
$t_1^{\ 5}$	$t_1^{\ 6}$	$t_1^{\ 7}$	$t_1^{\ 8}$
$t_2^{\ 5}$	$t_2^{\ 6}$	$t_2^{\ 7}$	$t_2^{\ 8}$

...

}

where $t_1^{\ k}$ is the kth element of table T_1, $t_2^{\ k}$ is the kth element of table T_2, $t_3^{\ k}$ is the kth element of table T_3, and $t_4^{\ k}$ is the kth element of table T_4.

If table_type=IMXTYPE_USHORT or IMXTYPE_SHORT

table_base[ ]={

$t_1{}^1$ $t_1{}^2$

$t_2{}^1$ $t_2{}^2$

$t_3{}^1$ $t_3{}^2$

$t_4{}^1$ $t_4{}^2$

...

}

The input array contains indexes into the lookup tables, arranged in a cyclic manner:

Input_tab[ ]={$I_1{}^1$, $I_2{}^2$, $I_3{}^3$, $I_4{}^4$, $I_1{}^5$, $I_2{}^6$, $I_3{}^7$, …}

where $I_1{}^k$ is an index of an element in T_1, $I_2{}^k$ is an index of an element of table T_2, and $I_3{}^k$ is an index of an element of table T_3, and $I_4{}^k$ is an index of an element of table T_4.

The function imx_formatTLU() can be used to generate the table lookups pointed to by input2_ptr for all these multi-thread use cases.

Input array and the lookup table cannot both be in an image buffer.

Each input data point is rounded and saturated prior to lookup. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Negative indexing is permitted when the input data array's elements are signed. In this case, the lookup table's base address points to the element whose index is 0.

Example

Single table lookup on 512 entries.

```
void imxenc_table_lookup(
    data_ptr,         /* point to input data array */
    table_base,       /* point to lookup table */
    output_ptr,       /* point to output array */
    512,              /* width of input data array
    IMXTYPE_USHORT,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,   /* output Int16/byte */
    0,                /* number of bits to downshift input before lookup */
    1,                /* single table */
    *cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- The input data array and lookup table cannot both be in an image buffer (A or B).
- The lookup table must be 128-bits aligned.

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (88)$$

- amount_of_work = *compute_elements*
- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor:

thread	**speedup_factor**
4	4
2	2
1	1

6.1.52 imxenc_3d_table_lookup

imxenc_3d_table_lookup ***Perform a tetrahedral interpolation using a 3D table lookup for a 9x9x9 table or a 17x17x17 table. Lookup is performed on a 2D array of input data, with the three inputs interleaved.***

Syntax

```
cmdlen = imxenc_3d_table_lookup(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *table_ptr,      /* starting address of table */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* height of input array */
    Int16 input_height,    /* width of input array */
    Int16 table_size,      /* 9/17, 9x9x9 table or 17x17x17 table */
    Int16 output_width,    /* height of output array */
    Int16 output_height,   /* width of output array */
    Int16 compute_width,   /* height of compute block */
    Int16 compute_height,  /* width of compute block */
    Int16 input_type,      /* unsigned Int16/unsigned byte */
    Int16 table_type,      /* Int16/unsigned Int16 */
    Int16 output_type,     /* Int16/byte */
    Int16 round_shift,     /* shifting parameter (0..8) */
    Int16 *cmdptr
);
```

Description

This function performs a tetrahedral interpolation on a 2D array of data organized such that the three inputs are interleaved. The tetrahedral interpolation is performed using a 9x9x9 lookup table or a 17x17x17 lookup table.

Input data can be byte or Int16, but is always unsigned. Further, input data should be saturated correctly such that after shifting the input data by round_shift, the data is clipped to the range [0..8] for a 9x9x9 table and [0..16] for a 17x17x17 table. Input data is arranged in triples such that input_width refers to the number of triples in a row of the data buffer. The actual row width is three times larger. Since the output data is also presented in sets of three, the output_width also refers to the number of triples in each row and the actual buffer width is three times larger. Compute_width tells the function how many output triples to calculate.

The output buffer of size 3*output_width x output_height and the input buffer of size 3*input_width x input_height should be larger than the compute block of size 3*compute_width x compute_height.

The parameter table_size determines the size of the table. If table_size = 9, the lookup table is a 9x9x9 table (729 32-bit elements). If table_size = 17, the lookup table is a 17x17x17 table (4913 32-bit elements). Each 32-bit element contains three packed values; bits 21-31 give the first color, bits 10-20 give the second color and bits 0-9 give the third color. If the table contains signed data, the MSB of each packed element is the sign bit. Table type must be IMXTYPE_SHORT or IMXTYPE_USHORT.

Example

Consider data input → 8(H) x 16(W) x 3 matrix, lookup table 9x9x9 x 32-bit elements, and output → 24(H) x 8(W) x 3 matrix. Compute 8(H) x 16(W) x 3 outputs. The following illustrates what the API call would look like.

```
cmdlen = imxenc_3d_table_lookup(
    input_ptr,       /* starting address of input */
    table_ptr,       /* starting address of table */
    output_ptr,      /* starting address of output */
    16,              /* height of input array */
    8,               /* width of input array */
    9,               /* 9/17, 9x9x9 table or 17x17x17 table */
    24,              /* height of output array */
    8,               /* width of output array */
    16,              /* height of compute block */
    8,               /* width of compute block */
    IMXTYPE_SHORT,   /* unsigned Int16/unsigned byte */
    IMXTYPE_SHORT,   /* Int16/unsigned Int16 */
    IMXOTYPE_SHORT,  /* Int16/byte */
    0,               /* shifting parameter (0..8) */
    cmdptr );
```

Constraints

- input_width, output_width ≥ compute_width
- input_height, output_height ≥ compute_height
- table_size = 9 or table_size = 17

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \tag{89}$$

- amount_of_work =

$$compute\_width \times compute\_height \times 3 \tag{90}$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

6.1.53 imxenc_table_lookup_int

imxenc_table_lookup_int ***Perform a 2D table lookup with interpolation. Lookup is performed on a 2D array of input data. Input data that lie between two points in the lookup table are computed using a linear interpolation between these two points.***

Syntax

```
cmdlen = imxenc_table_lookup_int(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *table_ptr,      /* starting address of table */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* height of input array */
    Int16 input_height,    /* width of input array */
    Int16 output_width,    /* height of output array */
    Int16 output_height,   /* width of output array */
    Int16 compute_width,   /* height of compute block */
    Int16 compute_height,  /* width of compute block */
    Int16 input_type,      /* Int16/byte */
    Int16 table_type,      /* Int16/byte */
    Int16 output_type,     /* Int16/byte */
    Int16 round_shift,     /* shifting parameter (0..8) */
    Int16 thread,      /* single thread, two-thread, or four-thread (1, 2, 4) */

    Int16 *cmdptr
);
```

Description

This function performs a table lookup with interpolation on the input data. After shifting the input data by round_shift bits, the integer part and the fractional part of the number is computed using:

ip = (input >> round_shift)
fp = input – ip

The fractional part is used to interpolate between two points in the lookup table, table_ptr[ip] and table_ptr[ip+1]. The interpolation is a linear interpolation, so the final output value is computed using:

output = (1 – fp / (2^round_shift)) × table_ptr[ip] + (fp / (2^round_shift)) × table_ptr[ip+1]
output = output >> round_shift

The bit shift operation applied to the input data is always truncation. The shift operation performed on the output data is rounding. The amount of round_shift is constrained to be less than or equal to 8.

Input data and table data can be byte or 16-bits word, signed or unsigned. The output data can be byte or 16-bits word.

An output block of compute_width × compute_height is computed inside the output buffer of size output_width × output_height. Thus, both output_width and input_width should be greater than or equal to compute_width. Also, output_height and input_height should be greater than or equal to compute_height.

Tables need to be aligned on 128-bit boundaries.

Multi-threaded table lookup allows VICP to perform parallel lookup operations. For this operation, the lookup table needs to be organized in a certain manner. Also, the tables are interleaved within the table storage buffer. For multi-threaded lookup, each cycle of this interleaved table is 128-bit aligned. So for 16-bit table data in two-thread mode, the table is stored as:

t0[0]	t0[1]	t0[2]	t0[3]	t1[0]	t1[1]	t1[2]	t1[3]	t0[4]	t0[5]	t0[6]	t0[7]	t1[4]	t1[5]	t1[6]	t1[7]	...

After eight 16-bit table values (128 bits), the cycle repeats. For four-thread lookup, only two-values from each table are in each cycle. This also gives 128-bit alignment, since two values are taken from four threads. For 8-bit data and two-thread mode, 8 values from each table are in each cycle. For four-thread mode with 8-bit data, 4 values from each table are in each cycle. Each of these cases gives 128-bit alignment for each cycle of the interleaved tables.

Example

Consider data input → 8(H) x 16(W) matrix, lookup table contains Int16 elements, and output → 24(H) x 8(W) matrix. Compute 8(H) x 16(W). The following illustrates what the API call would look like.

```
cmdlen = imxenc_table_lookup_int(
    input_ptr,       /* starting address of input */
    table_ptr,       /* starting address of table */
    output_ptr,      /* starting address of output */
    16,              /* height of input array */
    8,               /* width of input array */
    24,              /* height of output array */
    8,               /* width of output array */
    16,              /* height of compute block */
    8,               /* width of compute block */
    IMXTYPE_SHORT,   /* Int16/byte */
    IMXTYPE_SHORT,   /* Int16/byte */
    IMXOTYPE_SHORT,  /* Int16/byte */
    4,               /* shifting parameter (0..8) */
    2,               /* 2-thread lookup  (can be 1, 2, or 4)*/
    cmdptr);
```

Constraints

- input_width, output_width ≥ compute_width
- input_height, output_height ≥ compute_height
- round_shift is between 0 and 8
- thread can be 1, 2, or 4
- table data must be aligned on 128-bit boundary in memory (i.e. 4 LSB of address are 0).

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / \textit{speedup\_factor} \quad (91)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} \quad (92)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor = thread

6.1.54 imxenc_table_lookup_32bit

imxenc_table_lookup_32bit ***Perform a 2D table lookup with 32-bit table data.***

Syntax

```
cmdlen = imxenc_table_lookup_32bit(
    Int16 *input_ptr,      /* starting address of input */
    Int16 *table_ptr,      /* starting address of table */
    Int16 *output_ptr,     /* starting address of output */
    Int16 input_width,     /* height of input array */
    Int16 input_height,    /* width of input array */
    Int16 output_width,    /* height of output array */
    Int16 output_height,   /* width of output array */
    Int16 compute_width,   /* height of compute block */
    Int16 compute_height,  /* width of compute block */
    Int16 input_type,      /* Int16/ byte */
    Int16 round_shift,     /* shifting parameter (0..8) */
    Int16 thread,      /* single thread, two-thread, or four-thread (1, 2, 4) */

    Int16 *cmdptr
);
```

Description

This function performs a table lookup for a table with 32-bit data. The table index is formed by shifting the input data by round_shift bits and rounding the result. This index is used to retrieve the 32-bit data from the lookup table.

A block of compute_width × compute_height is taken from the input block and generates an output block of compute_width × compute_height in the output buffer. Input data can be IMXTYPE_BYTE or IMXTYPE_SHORT. Table data is forced to be IMXTYPE_USHORT and output data is forced to be IMXOTYPE_SHORT.

Table data must be aligned on 128-bit boundaries. The table contains 32-bit data (two 16-bit unsigned shorts).

Multi-threaded lookup can also be performed to reduce computation time. In this case, the tables must be arranged in a certain manner. The tables are interleaved. Each cycle of the table must be aligned on a 128-bit boundary. For two-thread lookup, two 32-bit values from each table are in each cycle. The table arrangement for this case is:

t0[0]	t0[1]	t1[0]	t1[1]	t0[2]	t0[3]	t1[2]	t1[3]	...

Each cycle contains four 32-bit values to ensure 128-bit alignment. For four-thread lookup, each of the four tables has one 32-bit value in each cycle. This also ensures 128-bit alignment.

Example

Consider data input → 8(H) x 16(W), lookup table contains 32-bit elements, and output → 24(H) x 8(W) matrix. Compute 8(H) x 16(W) outputs. The following illustrates what the API call would look like.

```
cmdlen = imxenc_table_lookup_32bit(
    input_ptr,      /* starting address of input */
    table_ptr,      /* starting address of table */
    output_ptr,     /* starting address of output */
    16,             /* height of input array */
    8,              /* width of input array */
    24,             /* height of output array */
    8,              /* width of output array */
    16,             /* height of compute block */
    8,              /* width of compute block */
    IMXTYPE_SHORT,  /* unsigned Int16/unsigned byte */
    4,              /* shifting parameter (0..8) */
    2,              /* 2-thread lookup  (can be 1, 2, or 4)*/
    cmdptr);
```

Constraints

- input_width, output_width ≥ compute_width
- input_height, output_height ≥ compute_height
- round_shift is between 0 and 8
- thread can be 1, 2, or 4
- table data must be aligned on 128-bit boundary in memory (i.e. 4 LSB of address are 0).

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\mathit{amount\_of\_work} \times \mathit{memory\_conflict\_factor} / \mathit{speedup\_factor} \quad (93)$$

- amount_of_work =

$$\mathit{compute\_width} \times \mathit{compute\_height} \quad (94)$$

- memory_conflict_factor:

Location of input1	Location of input2	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor = thread

6.1.55 imxenc_transparentblt

imxenc_transparentblt *16 bits per pixel transparent blts*

Syntax

```
cmdlen = imxenc_transparentblt(
    src,             /* Int16*, source bitmap with transparent color. 16 bpp */
    bkg,             /* Int16*, background bitmap. 16 bpp */
    tempScratch,     /* Int16*, temporary memory of size 2*compute_width*compute_height words */
    permScratch,     /* Int16*, permanent memory of size 2 words, must be in iMX coef mem */
    dest,            /* Int16*, destination bitmap. Can be equal to bkg */
    src_width,       /* Int16, src width */
    src_height,      /* Int16, src height */
    bkg_with,        /* Int16, background width */
    bkg_height,      /* Int16, background height */
    dest_width,      /* Int16, destination width */
    dest_height,     /* Int16, destination height */
    compute_width,   /* Int16, compute width */
    compute_height,  /* Int16, compute height */
    transparentclr,  /* Int16, Value of transparent color */
    cmdptr           /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr */
```

Description

This function takes the source bitmap and makes one of the colors in the bitmap transparent, then blts it to the background so that the background can be seen *through* the bitmap's transparent color. The dest pointer can point to the same location as the bkg pointer. In this case the background bitmap gets overwritten after execution of the encoded VICP command.

Two types of scratch memory must be allocated in: a temporary scratch memory of size 2*compute_width*compute_height words and a permanent scratch buffer of size 2 words. The temporary scratch buffer can be re-used by other VICP functions needing a temporary scratch buffer and its content can be overwritten after the corresponding transparentblt VICP sequence is executed. Re-use of this temporary scratch buffer by other VICP functions is highly recommended in order to optimize memory allocation in the image buffer or coefficient buffers. In the other hand, the permanent scratch must never be altered by the application. The imxenc_transparentblt() function initializes the 2 words contained in the scratch and imxUpdate_transparentblt() update those when it is called. This permanent scratch must be allocated in the VICP coefficient memory.

The bitmaps involved are 16 bits per pixel bitmaps, which are usually RGB55 or RGB565. The value of transparentclr specifies which color in the bitmap is treated as the transparent color.

Be aware that the locations, dimensions of source, background, destination bitmaps and the transparent color are fixed for the encoded VICP command. If the program wants to execute the VICP commands that performs the transparentblt algorithm for a different transparent color, it can call the function imxUpdate_transparentblt() to update an existing VICP command sequence. If any other parameters must be changed, then the application must call imxenc_transparentblt() all over again with the new parameters since the imxUpdate_transparentblt() only allows for changing the transparent color value.

Constraints

compute_width must be a multiple of 8.

Performance

Top performance is between 1.375 and 1.6 cycles/pixel, each pixel being a 16-bits word. This top performance can be achieved if all the scratch buffers are in VICP coefficient memory and the source, destination, background buffers are not in the same memories.

6.1.56 imxenc_transpose

imxenc_transpose ***Transposes a data matrix***

Syntax

```
cmdlen = imxenc_transpose(
    input_ptr;       /* Int16*, starting address of input array */
    coeff_ptr,       /* Int16*, starting address of scaling coefficient */
    output_ptr,      /* Int16*, starting address of the output array */
    input_width,     /* Int16, width of the input array */
    input_height,    /* Int16, height of the input array */
    output_width,    /* Int16, width of the output array */
    output_height,   /* Int16, height of the output array */
    compute_width    /* Int16, computation width */
    compute_height,  /* Int16, computation height */
    input_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,      /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                     /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,     /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,     /* Int16, number of bits to downshift before output */
    cmdptr           /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function transposes a submatrix of size compute_width x compute_height of the input data matrix (size input_width x input_height), and writes the transposed submatrix into the output matrix of size output_width x output_height, aligned to the top left corner.

Each input data point can be scaled by the scaling factor defined in coef_ptr, and being rounded and saturated prior to write out. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Input and output are each selectable to be byte or short type.

Example

12x10 block transpose out of an array of size 16x16

```
cmdlen = imxenc_transpose(
    data_ptr,        /* point to input data array */
    coef_ptr,        /* point to scaling coefficient */
    output_ptr,      /* point to output array */
    16,              /* width of input data array */
    16,              /* height of input data array */
    16,              /* width of output data array */
    16,              /* height of output data array */
    12,              /* computation width */
    10,              /* computation height */
    IMXTYPE_SHORT,
    IMXTYPE_USHORT,
    IMXOTYPE_SHORT,
    0                /* number of bits to downshift */
    cmdptr           /* starting point for command sequence in memory */
);
```

Constraints

- input_width, output_width ≥ compute_width
- input_height, output_height ≥ compute_height

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \quad (95)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (96)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

6.1.57 imxenc_transpose_interleave

imxenc_transpose_interleave ***Transposes a data matrix with the option to read or write data in an interleaved fashion***

Syntax

```
cmdlen = imxenc_transpose_interleave(
    input_ptr;      /* Int16*, starting address of input array */
    coeff_ptr,      /* Int16*, starting address of scaling coefficient */
    output_ptr,     /* Int16*, starting address of the output array */
    input_width,    /* Int16, width of the input array */
    input_height,   /* Int16, height of the input array */
    input_step_x,   /* Int16, horizontal step between each input data point */
    input_step_y,   /* Int16, vertical step between each input data point */
    output_width,   /* Int16, width of the output array */
    output_height,  /* Int16, height of the output array */
    output_step_x,  /* Int16, horizontal step between each output data point */
    output_step_y,  /* Int16, vertical step between each output data point */
    compute_width   /* Int16, computation width or number of data elements to
                       transpose horizontally */
    compute_height, /* Int16, computation height or number of data elements to
                       transpose vertically */
    input_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,     /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                    /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,    /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,    /* Int16, number of bits to downshift before output */
    cmdptr          /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of word written to cmd memory starting at cmdptr*/
```

Description

This function extracts a submatrix of size compute_width*compute_height from the input data matrix (size input_width × input_height), and writes the transposed submatrix into the output matrix of size output_width × output_height in an interleaved fashion dictated by output_step_x and output_step_y. The extraction of compute_width × compute_height data points from the input matrix is also performed in an interleaved fashion using the parameters input_step_x and input_step_y .

Each input data point can be scaled by the scaling factor defined in coef_ptr, and being rounded and saturated prior to write out. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Input and output are each selectable to be byte or short type.

Example

6x10 block transpose out of an array of size 16x16, with horizontal interleaving enabled at the input

```
cmdlen= imxenc_transpose_interleave(
    data_ptr,        /* point to input data array */
    coef_ptr,        /* point to scaling coefficient */
    output_ptr,      /* point to output array */
    16,              /* width of input data array */
    16,              /* height of input data array */
    2,               /* Extract every other points in the same row */
    1,               /* Extract every row */
    6,               /* width of output data array */
    10,              /* height of output data array */
    1,               /* No horizontal interleaving at the output */
    1,               /* No vertical interleaving at the output */
    6,               /* computation width */
    10,              /* computation height */
    IMXTYPE_SHORT,
    IMXTYPE_USHORT,
    IMXOTYPE_SHORT,
    0               /* number of bits to downshift */
    cmdptr          /* starting point for command sequence in memory */
);
```

Constraints

- input_width, output_width ≥ compute_width
- input_height, output_height ≥ compute_height

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \quad (97)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (98)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

6.1.58 imxenc_YCbCrPack

imxenc_YCbCrPack ***Packing YcbCr color data stored in separate components into either a YCbCr422 or a YCbCr444 packed format pixel array.***

Syntax

```
cmdlen = imxenc_YCbCrPack(
    input_ptr;      /* Int16**, Starting address of the input array;
                       pointer to an array with the addresses of the 3 color
                       components */
    coeff_ptr,      /* Int16*, starting address of scaling coefficient */
    output_ptr,     /* Int16*, starting address of the output array */
    input_width,    /* Int16, width of the input array */
    input_height,   /* Int16, height of the input array */
    output_width,   /* Int16, width of the output array */
    output_height,  /* Int16, height of the output array */
    calc_width      /* Int16, computation width */
    calc_height,    /* Int16, computation height */
    colorspace,     /* Int16, Color format of the input & output data: */
                    /* 0:444->422, 1:422->422, 2:420->422 */
                    /* 3:422*->422,4:444->444, 5:422->444, 6:420->444 */
                    /* 7:422*->444 */
                       /* If most significant bit is set */
                       /* then each output color component occupies 16 bits */
                       /* instead of 8 bits */
                       /* If bit 14 is set */
                       /* then each input component occupies 8 bits */
                       /* instead of 16 bits */
    round_shift,    /* Int16, number of bits to downshift before output */
    cmdptr          /* Int16*, starting point of command sequence in memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function packs the components of a color image into a single image. The output image can be in 422 format or 444 format, while the input can be in 444, 422, 420 format or a modified 422 format with vertical downsampling of the chroma samples instead of the more common horizontal downsampling (422* in the table below). For 422 output, the pixels are composed as: YCb YCr YCb YCr and for 444 output, the pixels are composed as: YCb CrY CbCr YCb CrY etc. Each output component is assumed to be 8 bits or 16 bits depending on the value of the most significant bit of the parameter colorformat. The size of each of the input elements is either 8 bits or 16 bits.

The routines takes a submatrix of size compute_width x compute_height of the input data matrix (size input_width x input_height), and writes the submatrix into the output matrix of size output_width x output_height, aligned to the top left corner. It is assumed that the size of the color components is scaled according to the color format.

The meaning of output_width can be different depending on the colorformat parameter. Table 6-7 summarizes the different units of output_width:

Table 6-7. Units for output_width Depending on colorformat

colorformat	In format→out format	Size of Input Element	Size of Each Output Color Component	output_width unit
0x0000	444→422	16 bits	8 bits	16 bits
0x0001	422→422	16 bits	8 bits	16 bits
0x0002	420→422	16 bits	8 bits	16 bits
0x0003	422*→422	16 bits	8 bits	16 bits
0x0004	444→444	16 bits	8 bits	8 bits
0x0005	422→444	16 bits	8 bits	8 bits
0x0006	420→444	16 bits	8 bits	8 bits
0x0007	422*→444	16 bits	8 bits	8 bits
0x4000	444→422	8 bits	8 bits	16 bits
0x4001	422→422	8 bits	8 bits	16 bits

Table 6-7. Units for output_width Depending on colorformat (continued)

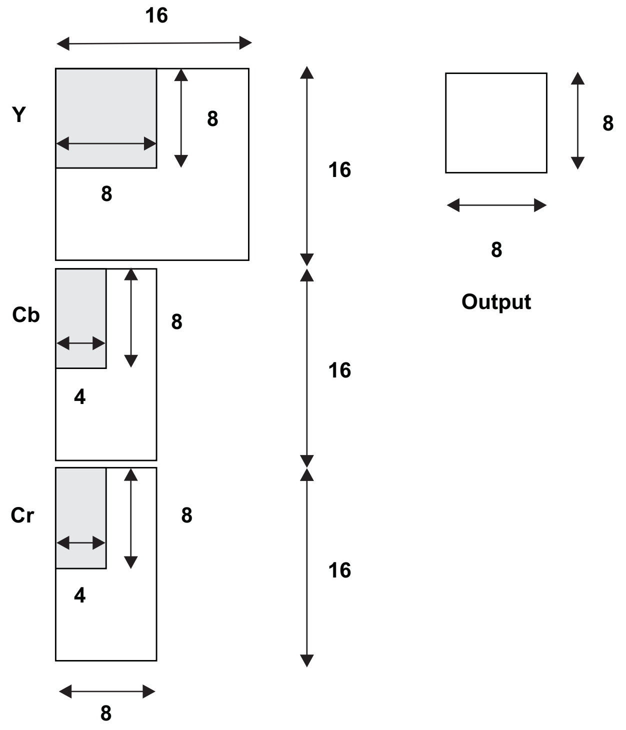
colorformat	In format→out format	Size of Input Element	Size of Each Output Color Component	output_width unit
0x4002	420→422	8 bits	8 bits	16 bits
0x4003	422*→422	8 bits	8 bits	16 bits
0x4004	444→444	8 bits	8 bits	8 bits
0x4005	422→444	8 bits	8 bits	8 bits
0x4006	420→444	8 bits	8 bits	8 bits
0x4007	422*→444	8 bits	8 bits	8 bits
0x8000	444→422	16 bits	16 bits	32 bits
0x8001	422→422	16 bits	16 bits	32 bits
0x8002	420→422	16 bits	16 bits	32 bits
0x8003	422*→422	16 bits	16 bits	32 bits
0x8004	444→444	16 bits	16 bits	16 bits
0x8005	422→444	16 bits	16 bits	16 bits
0x8006	420→444	16 bits	16 bits	16 bits
0x8007	422*→444	16 bits	16 bits	16 bits
0xC000	444→422	8 bits	16 bits	32 bits
0xC001	422→422	8 bits	16 bits	32 bits
0xC002	420→422	8 bits	16 bits	32 bits
0xC003	422*→422	8 bits	16 bits	32 bits
0xC004	444→444	8 bits	16 bits	16 bits
0xC005	422→444	8 bits	16 bits	16 bits
0xC006	420→444	8 bits	16 bits	16 bits
0xC007	422*→444	8 bits	16 bits	16 bits

Each input data point can be scaled by the scaling factor defined in coef_ptr, and being rounded and saturated prior to write out. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

Example 1 Packing from 422 to 422 a 8x8 block of pixels out of an array of size 16x16 pixels and write to an array of size 18x6. The input color format is 422, which means that there will be one Cb and Cr value for each two Y values; so if the Y array is of size 16x16, then each of the Cb and Cr arrays will be 8x16. The output will be 8 rows of 8 words each.

```
Int16 *arrayCol[3]={Y_ptr, U_ptr, V_ptr};

  cmdlen = imxenc_YcbCrPack(
      arrayCol,        /* point to color components */
      coef_ptr,        /* point to scaling coefficient */
      output_ptr,      /* point to output array */
      16,              /* width of input data array */
      16,              /* height of input data array */
      8,               /* width of output data array */
      8,               /* height of output data array */
      8,               /* computation width */
      8,               /* computation height */
      1,               /* 422 -> 422 color format */
      0,               /* number of bits to downshift */
      cmdptr           /* starting point for the cmd sequence in memory */
  );
```

Figure 6-17. imxenc_YCbCrPack



Example 2

Packing from 422 to 444 a 8x8 block out of an array of size 16x16 and write to an array of size 12x8. The input color format is 422, which means the data array for Cb, Cr is each of size 8x16. The output size of each color component is 8 bits.

```
cmdlen = imxenc_YcbCrPack(
    arrayCol,        /* point to color components */
    coef_ptr,        /* point to scaling coefficient */
    output_ptr,      /* point to output array */
    16,              /* width of input data array */
    16,              /* height of input data array */
    3*8,             /* width of output data array */
    8,               /* height of output data array */
    8,               /* computation width */
    8,               /* computation height */
    5,               /* 422 -> 444 color format */
    0,               /* number of bits to downshift */
    cmdptr           /* starting point for the cmd sequence in memory */
);
```

Constraints

- If the input color format is 422 or 420 the size of the input color signal is assumed to be only half (quarter) of the size of the luminance signal Y.
- compute_height and input_height must be < 256
- compute_width must be ≤ 512.
- compute_width must be even.

Performance

The overhead time for this VICP API is ~ 135 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor / speedup\_factor \quad (99)$$

- amount_of_work =

$$compute\_width \times compute\_height \quad (100)$$

- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

- speedup_factor and maximum value for compute_width:

compute_width multiple only of	speedup_factor	Maximum value of compute_width
8	8	1024
4	4	512
2	2	256
1[(1)]	1	128

(1) That is, compute_width is odd.

6.1.59 imxenc_YCbCrUnpack2

imxenc_YCbCrUnpack2 ***Unpack YCbCr 4:2:2 or 4:4:4 composite video data and store as separate component arrays in one of 4:4:4, 4:2:2, or 4:2:0 formats. Complementary function of imxenc_YCbCrPack().***

Syntax

```
cmdlen = imxenc_YCbCrUnpack2(
    input_ptr;          /* Int16*, starting address of input array */
    coeff_ptr,          /* Int16*, starting address of scaling coefficient */
    output_ptr,         /* Int16**, Starting address of the output array;
                           pointer to an array with the addresses of the 3
                           color components */
    input_width,        /* Int16, width of the input array */
    input_height,       /* Int16, height of the input array */
    output_width,       /* Int16, width of the output array */
    output_height,      /* Int16, height of the output array */
    calc_width          /* Int16, computation width */
    calc_height,        /* Int16, computation height */
    colorspace,         /* Int16, Color format of the input & output data:
                        /* 0:422->444, 1:422->422, 2:422->420 */
                        /* 3:444->444, 4:444->422, 5:444->420, 6:422->420 */
                        /* If most significant bit is set */
                        /* then each input color component spans 16 bits */
                        /* instead of 8 bits */
    outputY_type,       /* Int16, type of the unpacked Y components */
                        /* IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    outputCbCr_type,    /* Int16, type of the unpacked chrominance */
                        /* components */
                        /* IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,        /* Int16, number of bits to downshift before output*/
    cmdptr              /* Int16*, starting point of command sequence in
                           memory */
    );

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description

This function unpacks a single YcbCr 4:2:2 or 4:4:4 composite video data array into 3 separate component arrays. The pixels of input array are composed as: YCb YCr YCb YCr for 4:2:2 format or YCb CrY CbrCr YCb for 4:4:4 format. Each input component is assumed to be 8-bit wide or 16 bits, depending on the value of the most significant bit of parameter colorformat. The output data format can be either 4:4:4, 4:2:2, or 4:2:0. In the output, Y block is followed by Cb block and then Cr block. The output Y components are stored as type outputY_type and the output chrominance components are stored as type outputCbCr_type.

The routines takes a sub-array of size compute_width x compute_height of the input data array (size input_width x input_height). The sub-array is reorganized into the corresponding sub-array of an output array of size output_width x output_height for Y component. The Y array is followed by the subsequent output arrays of output_width x output_height for Cb and Cr components for 4:4:4 and 4:2:2 formats, or the subsequent arrays of output_width x output_height/2 in case of 4:2:0 format.

The meaning of input_width can be different depending on the parameter colorformat. Table 6-8 summarizes the different units of input_width:

Table 6-8. Units for input_width Depending on colorformat

colorformat	In format→Out format	Size of Each Input Color Component	input_width Unit	Input Pixel Format
0x0	422→444	8 bits	16 bits	YCbYCr
0x1	422→422	8 bits	16 bits	YCbYCr
0x2	422→420	8 bits	16 bits	YCbYCr
0x3	444→444	8 bits	8 bits	YCb CrY CbrCr YCb
0x4	444→422	8 bits	8 bits	YCb CrY CbrCr YCb

Table 6-8. Units for input_width Depending on colorformat (continued)

colorformat	In format→Out format	Size of Each Input Color Component	input_width Unit	Input Pixel Format
0x5	444→420	8 bits	8 bits	YCb CrY CbrCr YCb
0x6	422*→420	8 bits	16 bits	CbYCrY
0x8000	422→444	16 bits	32 bits	YCbYCr
0x8001	422→422	16 bits	32 bits	YCbYCr
0x8002	422→420	16 bits	32 bits	YCbYCr
0x8003	444→444	16 bits	16 bits	YCb CrY CbrCr YCb
0x8004	444→422	16 bits	16 bits	YCb CrY CbrCr YCb
0x8005	444→420	16 bits	16 bits	YCb CrY CbrCr YCb

Each input data point can be scaled by the scaling factor defined in coef_ptr, and being rounded and saturated prior to write out. Number of down shifts is specified in the command, and saturation bounds can be specified by calling the imxenc_set_saturation function.

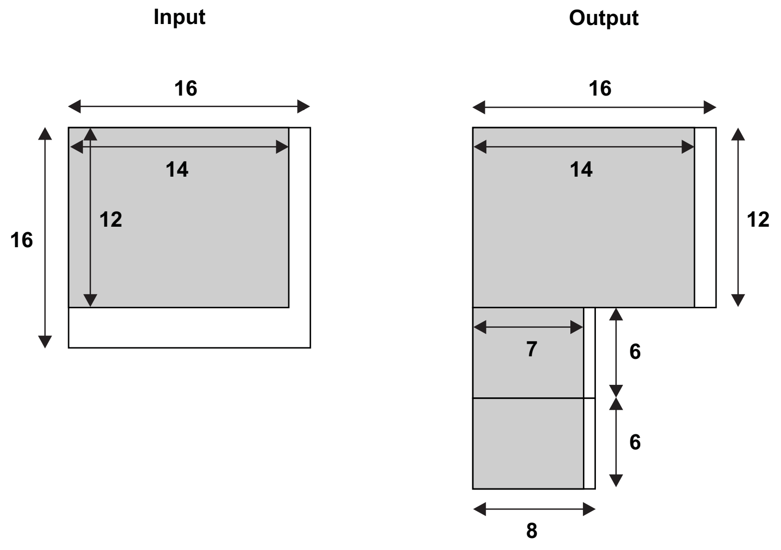
For compatibility with earlier versions of the VICP library, the function imxenc_YCbCrUnpack() is provided as a macro wrapper. This function calls imxenc_YCbCrUnpack2() with the input variables ‘outputY_type” and “outputUV_type” set to IMXOTYPE_SHORT.

Example

Unpacking a 14x12 block out of an array of size 16x16 and write to an array of size 16x12. The input colorformat is 422 and the output color format is 420 which means the output data array for Cb, Cr is of size 10x11.

```
Int16 *arrayCol[3]={Y_ptr, U_ptr, V_ptr};

  cmdlen = imxenc_YcbCrUnpack2(
    data_ptr,        /* point to input data array */
    coef_ptr,        /* point to scaling coefficient */
    arrayCol,        /* point to output array */
    16,              /* width of input data array */
    16,              /* height of input data array */
    16,              /* width of output data array */
    12,              /* height of output data array */
    14,              /* computation width */
    12,              /* computation height */
    2,               /* 420 color format */
    IMXOTYPE_SHORT,
    IMXOTYPE_SHORT,
    0,               /* number of bits to downshift (*coef_ptr = 1) */
    cmdptr           /* starting point for the cmd sequence in memory */
);
```

Figure 6-18. imxenc_YCbCrUnpack2



Constraints

- If the color format is 422 (or 420), the output chrominance data are assumed to be half (or quarter) of the size of the luminance signal Y. The user has to properly set up the starting address of the output chrominance data using *output_ptr[ ].
- Input must be of type SHORT.
- compute_height and input_height must be < 256.
- compute_width must be multiple of 4 and ≤ 512.

Performance

The overhead time for this VICP API is ~ 135 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$\textit{amount\_of\_work} \times \textit{memory\_conflict\_factor} / 4 \qquad (101)$$

- amount_of_work =

$$\textit{compute\_width} \times \textit{compute\_height} + (\textit{compute\_width} + 1) \times \textit{compute\_height} / (vs \times hs) \qquad (102)$$

  The vs and hs parameters have the following values:
  – YUV444: hs=1 and vs=1
  – YUV422: hs=2 and vs=1
  – YUV420: hs=2 and vs=2
- memory_conflict_factor:

Location of data	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	3
IMGBUF	IMGBUF	COEFF	2
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	2
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	2

6.1.60 imxenc_y2blkseq2

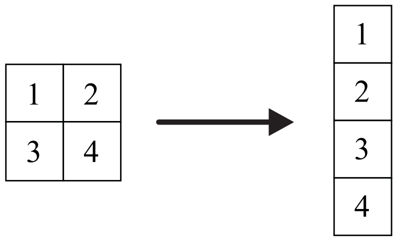
imxenc_y2blkseq2 ***Reorganize a NxM 2D matrix into a (N*M/8)x8 matrix (or storing data in 8x8 block sequentially)***

Syntax

```
cmdlen = imxenc_y2blkseq2(
    input_ptr;     /* Int16*, starting address of input array */
    coeff_ptr,     /* Int16*, starting address of scaling coefficient */
    output_ptr,    /* Int16*, starting address of the output array */
    input_width,   /* Int16, width of the input array */
    input_height,  /* Int16, height of the input array */
    no_blks_x,     /* Int16, width in number of 8x8 blks  */
    no_blks_y,     /* Int16, height in number of 8x8 blks */
    input_type,    /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                   /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    coeff_type,    /* Int16, IMXTYPE_UBYTE, IMXTYPE_BYTE */
                   /* Int16, IMXTYPE_USHORT, IMXTYPE_SHORT */
    output_type,   /* Int16, IMXOTYPE_BYTE, IMXOTYPE_SHORT */
    round_shift,   /* Int16, number of bits to downshift before output */
    cmdptr         /* Int16*, starting point of command sequence in memory */
);

/* Int16, cmdlen is the number of words written to cmd memory starting at cmdptr*/
```

Description This function reorganizes the data into a block sequential fashion.

Figure 6-19. imxenc_y2blkseq2



Note: input's and output's data types are IMXTYPE_SHORT.

Normally, only the data is reorganized and the quantities do not change, so the Int16 word pointed by coeff_ptr contains 1, and round_shift = 0. Otherwise, during the reorganization, each element of the output array is scaled by (*coeff_ptr / 2^round_shift).

For compatibility with earlier versions of the VICP library, the function imxenc_y2blkseq() is provided as a macro wrapper. This function calls imxenc_ y2blkseq2() with the input variables input_type and coeff_type set to IMXTYPE_SHORT and output_type set to IMXOTYPE_SHORT.

Example Mapping a 16x16 block into a 8x64 block.

```
cmdlen = imxenc_y2blkseq2(
    data_ptr,         /* point to input data array */
    coeff_ptr,        /* point to scalar scaling factor */
    output_ptr,       /* point to output array */
    16,               /* width of input data array */
    16,               /* height of input data array */
    2,                /* Number of processed input blocks horizontally */
    2,                /* Number of processed input blocks vertically  */
    IMXTYPE_SHORT,
    IMXTYPE_SHORT,
    IMXOTYPE_SHORT,
    0,                /* number of bits to downshift  */
    cmdptr            /* starting point for command sequence in memory */
);
```

Constraints no_blk_x and no_blks_y ≤ 256

Performance

The overhead time for this VICP API is ~ 30 cycles.

The estimated number of VICP cycles to perform the operation (except overhead time) is:

$$amount\_of\_work \times memory\_conflict\_factor \tag{103}$$

- amount_of_work =

$$8 \times no\_blks\_x \times no\_blks\_y \tag{104}$$

- memory_conflict_factor:

Location of input	Location of coeff	Location of output	memory_conflict_factor
IMGBUF	IMGBUF	IMGBUF	(2+1/ (no_blks_x × no_blks_y × 8))
IMGBUF	IMGBUF	COEFF	1
IMGBUF	COEFF	IMGBUF	2
IMGBUF	COEFF	COEFF	1
COEFF	IMGBUF	IMGBUF	(1+1/ (no_blks_x × no_blks_y × 8))
COEFF	IMGBUF	COEFF	1
COEFF	COEFF	IMGBUF	1
COEFF	COEFF	COEFF	(1+1/ (no_blks_x × no_blks_y × 8))

6.2 Functions Used for Updating Input, Coef and Output Address Pointers

These functions are used to modify specific parameters of existing VICP sequences instead of re-encoding whole imxenc_<computation> functions.

6.2.1 imxUpdate_inputPtr

imxUpdate_inputPtr ***Update the pointer to input data of an VICP opcode sequence previously generated by imxenc_<computation>***

Syntax

```
void  imxUpdate_inputPtr(
    data_init,  /* Int16*, new pointer to input data */
    highbyte,   /* Int16*, Not used, for backward compatibility */
    cmdPtr      /* Int16*, starting point of command sequence in memory */
    );
```

Description

cmdPtr must point into the VICP memory to the first word of the opcode sequence corresponding to the imxenc_<computation> whose pointer to input data needs to be changed.

data_init is in DSP memory map and points to image buffers or VICP memory.

highbyte is not used by the function

This function cannot be used to update the pointer of a sequence generated by the following APIs:

- imxenc_cfa_hq_interpolation
- imxenc_ycbcrpack
- imxenc_ycbcrunpack
- imxenc_alphablend
- imxenc_alphablendYUV422I
- imxenc_blkAverage
- imxenc_blkVariance
- imxenc_color_spc_conv
- imxenc_median3x3
- imxenc_pack422
- imxenc_recursiveFilterVert1stOrder
- imxenc_rgbpack
- imxenc_rgbunpack
- imxenc_sum_abs_diff
- imxenc_sum_array_op
- imxenc_transparentblt

6.2.2 *imxUpdate_coefPtr*

imxUpdate_coefPtr ***Update the pointer to coefficients data or second input data of a VICP opcode sequence previously generated by imxenc_<computation>***

Syntax

```
void imxUpdate_coefPtr(
    ptr_init,  /* Int16*, new pointer to coefficients data */
    highbyte,  /* Int16*, Not used, for backward compatibility */
    cmdptr     /* Int16*, starting point of command sequence in memory */
    );
```

Description

cmdPtr must point into the VICP memory to the first word of the opcode sequence corresponding to the imxenc_<computation> whose pointer to coef or second input data needs to be changed.

ptr_init is in DSP memory map and points to image buffers or VICP memory.

highbyte is not used by the function

This function cannot be used to update the pointer of a sequence generated by the following APIs:

- imxenc_cfa_hq_interpolation
- imxenc_ycbcrpack
- imxenc_ycbcrunpack
- imxenc_alphablend
- imxenc_alphablendYUV422I
- imxenc_blkAverage
- imxenc_blkVariance
- imxenc_color_spc_conv
- imxenc_median3x3
- imxenc_pack422
- imxenc_recursiveFilterVert1stOrder
- imxenc_rgbpack
- imxenc_rgbunpack
- imxenc_sum_abs_diff
- imxenc_sum_array_op
- imxenc_transparentblt

6.2.3 imxUpdate_outputPtr

imxUpdate_outputPtr ***Update the pointer to output data of an VICP opcode sequence previously generated by imxenc_<computation>***

Syntax

```
imxUpdate_outputPtr(
    outPtr,    /* Int16*, new pointer to output data */
    highbyte,  /* Int16*, Not used, for backward compatibility */
    cmdPtr     /* Int16*, starting point of command sequence in memory */
    );
```

Description

cmdPtr must point into the VICP memory to the first word of the opcode sequence corresponding to the imxenc_<computation> whose pointer to output data needs to be changed.

outPtr is in DSP memory map and points to image buffers or VICP memory.

highbyte is not used by the function

This function cannot be used to update the pointer of a sequence generated by the following APIs:

- imxenc_cfa_hq_interpolation
- imxenc_ycbcrpack
- imxenc_ycbcrunpack
- imxenc_alphablend
- imxenc_alphablendYUV422I
- imxenc_blkAverage
- imxenc_blkVariance
- imxenc_color_spc_conv
- imxenc_median3x3
- imxenc_pack422
- imxenc_recursiveFilterVert1stOrder
- imxenc_rgbpack
- imxenc_rgbunpack
- imxenc_sum_abs_diff
- imxenc_sum_array_op
- imxenc_transparentblt

6.2.4 imxUpdate_alphablend

imxUpdate_alphablend ***Update the colorformat (RGB555 or RGB565) value used in an alphablend VICP opcode sequence previously generated by imxenc_alphablend().***

Syntax

```
imxUpdate_alphablend(
    permScratch,  /* Int16* pointer to the 7 words in VICP coef mem for permanent
                     scratch  */
    colorformat,  /* Int16 new value of input colorformat, RGB565=0 or RGB555=1 */
    cmdptr        /* Int16* cmdptr  */
    );
```

Description

This function will update the permanent scratch buffer so that the same alphablend VICP sequence can blend a source bitmap with a background bitmap that has a different colorformat than the one originally used when calling imxenc_alphablend().

permScratch must be the same pointer passed to the imxenc_alphablend() function that encoded the VICP command sequence pointer by cmdptr.

Important Note

Since this function needs to update the VICP command memory and also the permanent scratch memory originally initialized by imxenc_alphablend(), it will automatically switch VICP coefficient buffers and the VICP command buffer to DSP access when it is executed. The buffer switch is also automatically restored to its previous value upon exit of the function. That means this function must be called when VICP is not executing.

6.2.5 imxUpdate_rgbpack

mxUpdate_rgbpack ***Update the colorformat (RGB555 or RGB565) value used in a rgbpack VICP opcode sequence previously generated by imxenc_rgbpack().***

Syntax

```
imxUpdate_rgbpack(
    permScratch, /* Int16* pointer to the 5 words in VICP coef mem for permanent
                    scratch  */
    colorformat  /* Int16 new value of output colorformat, RGB565=0 or RGB555=1 */
    );
```

Description

This function will update the permanent scratch buffer so that the same rgbpack VICP sequence can pack R,G,B planes into a different colorformat than the one originally used when calling imxenc_rgbpack().

permScratch must be the same pointer passed to the imxenc_rgbpack() function that encoded the VICP command sequence pointer by cmdptr.

Important Note

Since this function needs to update the permanent scratch memory originally initialized by imxenc_rgbpack(), it will automatically switch VICP coefficient buffers to DSP access when it is executed. The buffer switch is also automatically restored to its previous value upon exit of the function. That means this function must be called when VICP is not executing.

6.2.6 imxUpdate_rgbunpack

imxUpdate_rgbunpack ***Update the colorformat (RGB555 or RGB565) value used in a rgbunpack VICP opcode sequence previously generated by imxenc_rgbunpack().***

Syntax

```
imxUpdate_rgbunpack(
    permScratch,  /* Int16* pointer to the 4 words in VICP coef mem for permanent
                     scratch  */
    colorformat,  /* Int16 new value of input colorformat, RGB565=0 or RGB555=1 */
    cmdptr        /* Int16* cmdptr  */
    );
```

Description

This function will update the permanent scratch buffer so that the same rgbunpack VICP sequence can unpack 16 bpp RGB data from a different colorformat than the one originally used when calling imxenc_rgbunpack().

permScratch must be the same pointer passed to the imxenc_rgbunpack() function that encoded the VICP command sequence pointer by cmdptr.

Important Note

Since this function needs to update the VICP command memory and also the permanent scratch memory originally initialized by imxenc_rgbunpack(), it will automatically switch VICP coefficient buffers and the VICP command buffer to DSP access when it is executed. The buffer switch is also automatically restored to its previous value upon exit of the function. That means this function must be called when VICP is not executing.

6.2.7 imxUpdate_transparentblt

imxUpdate_transparentblt ***Update the transparent color value used in a transparentblt VICP opcode sequence previously generated by imxenc_transparentblt().***

Syntax

```
imxUpdate_transparentblt(
    permScratch,     /* Int16* pointer to the perm scratch buffer */
    transparentclr,  /* Int16 new value for transparent color */
    cmdptr           /* short* cmdptr  */
    );
```

Description

This function will update the VICP command sequence corresponding to a transparentblt algorithm so that a transparent color value different than the one that was originally used for encoding the sequence is used next time the sequence is run.

cmdPtr must point into the VICP command memory to the first word of the opcode sequence corresponding to the imxenc_transparentblt() whose transparent color value needs to be changed.

permScratch must be the same pointer passed to the imxenc_transparentblt() function that encoded the VICP command sequence pointer by cmdptr.

Important Note

Since this function needs to update the VICP command memory and also the permanent scratch memory originally initialized by imxenc_transparentblt(), it will automatically switch VICP coefficient buffers and the VICP command buffer to DSP access when it is executed. The buffer switch is also automatically restored to its previous value upon exit of the function. That means this function must be called when VICP is not executing.

6.3 Functions Used for Setting Up Coefficients

These functions are used to initialize the VICP coefficient memory with coefficients that need to be arranged in a certain format.

6.3.1 *imx_cfa_hq_setup*

imx_cfa_hq_setup ***Set up coefficient array for high-quality CFA data filtering***

Syntax

```
Int16 length= imx_cfa_hq_setup ( Int16 *coeff_ptr,
                  Int16 filterIdentifier,
                  Int16 phase,
                  Int16 qShift);
```

Parameters

coeff_ptr	Int16*	Pointer to target coefficient array
filterIdentifier	Int16	Identifier to filter used. Currently only VICP_CFA_HQ_5x5_COEFS supported.
phase	Int16	Which color at upper-left corner of input
qShift	Int16	qShift parameter
length	Int16	Number of 16-bit words written

Description

This routine produces the filter coefficients that are required by imxenc_cfa_hq_interpolation(). To select which filter to use, set the argument filterIndentifier. Currently only 5x5 filter supported by selecting VICP_CFA_HQ_5x5_COEFS .

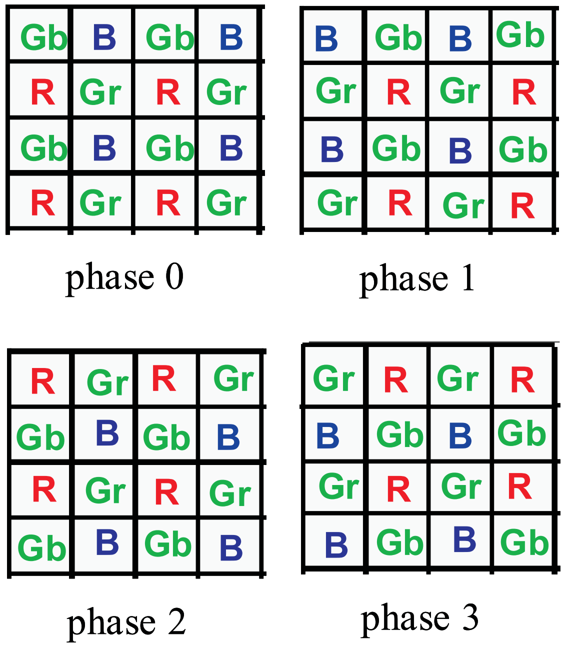
The argument qShift is the same as the one passed to imxenc_cfa_hq_interpolation(). Generally 9 is a good value to pass.

The number of 16-bits words written into memory location pointed by coeff_ptr is:

3 × 4 × coeff_width × coeff_height

The meaning of the phase argument is illustrated in Figure 6-20:

Figure 6-20. imx_cfa_hq_setup



6.3.2 *imx_fir_poly_setup_coeff*

imx_fir_poly_setup_coeff ***Function to setup the coefficients of a 1D filter kernel for polyphase filtering with imxenc_fir_poly_col()***

Syntax

```
length =  imx_fir_poly_setup_coeff (
                Int16 *src_p, Int16  *dst_p,
                Int16 taps
                Int16 smpl_nom, Int16 smpl_denom);
```

Parameters

src_p	Int16*	Starting address of filter coefficients
dst_p	Int16*	Destination address for polyphase filter coefficients
taps	Int16	Number of filter coefficients
smpl_nom	Int16	Nominator of rational sampling factor (upsampling)
smpl_denom	Int16	Denominator of rational sampling factor (downsampling)
length	Int16	Number of coefficients written into memory starting at dst_p

Description

This function arranges the coefficients of a 1D FIR filter in a suitable way for the polyphase filter function imxenc_fir_poly_col(). The ordering and organization depends on the sampling ratios.

The number of coefficients written is:

$$smpl\_nom \times (smpl\_denom + floor(\text{taps} / smpl\_nom)) \quad (106)$$

The filter should be symmetric since imxenc_fir_poly_col() uses a correlation-based approach to perform filtering.

6.3.3 imx_formatTLU

imx_formatTLU ***Format two lookup tables into one destination table to be used for dual table lookup.***

Syntax

```
bytes_generated = imx_formatTLU ( Uint16 numTables,  Uint16 **tablePtrArray,
                                Uint16 *dst, Uint16 numElemTable, Uint16 tableType);
```

Parameters

numTables	Uint16	1, 2, or 4
tablePtrArray	Uint16*	Point to an array of pointers. Each pointer pointing to a lookup table.
dst	Uint16*	Point to location where the formatted table is written.
numElemTable	Uint16	Number of elements in each table
tableType	Uint16	IMXTYPE_SHORT, IMXTYPE_BYTE

Description

This function is used to prepare lookup tables that will be used in the context of imxenc_table_lookup(). This latest function can perform two or four lookups per cycle. In the case of two lookups per cycle, two consecutive data points can initiate lookup operations from two distinct tables. But to enable that, the two tables must be merged/reformatted into one table by imx_formatTLU(). It is the merged table that is passed to imxenc_table_lookup().

When numTables=2 or 4, tablePtrArray[0] is set to the first table, tablePtrArray[1] is set to the second table, and so forth. In the context of speeding up a uniform table lookup, tablePtrArray[0], tablePtrArray[1], etc. can be set to point to the same table.

If numTables is set to 1, it will merely copy the table pointed by tablePtrArra[0] into dst.

The function returns the number of bytes generated.

6.4 Branching Inside a VICP Command Sequence

The VICP computation unit provides some branching commands that could be useful for saving command memory.

6.4.1 imxenc_call

imxenc_call ***Inserts a CALL SINGLE command into the VICP command sequence.***

Syntax

```
length =  imxenc_call(Int16 * address,
                      Int16 * cmd_ptr);
```

Parameters

address	Int16*	Pointer to the command to branch to in the VICP command memory.
cmd_ptr	Int16*	Current command pointer where the CALL SINGLE command should be inserted.

Description

Inserts a CALL SINGLE command into the VICP program sequence. The CALL SINGLE command jumps to the point in the VICP program pointed to by address. The VICP then executes until a computation command is found. After executing this command, control is passed back to the point in the main program following the CALL SINGLE command.

6.4.2 imxenc_call_dataaddr

imxenc_call_dataaddr ***Calls a single command subroutine along with data pointers modification.***

Syntax

```
length = imxenc_call_dataaddr(Int16 * cmdaddr,
                              Int16 * iaddr,
                              Int16 * caddr,
                              Int16 * oaddr,
                              Int16   iaddr_highbyte,
                              Int16   caddr_highbyte,
                              Int16   oaddr_highbyte,
                              Int16 * cmd_ptr);
```

Parameters

cmdaddr	Int16*	A pointer to the start of the subroutine in the VICP command memory.
iaddr	Int16*	Data pointer to substitute into the subroutine command.
caddr	Int16*	Coefficient or second input pointer to substitute into the subroutine command.
oaddr	Int16*	Output pointer to substitute into the subroutine command.
iaddr_highbyte	Int16	If BYTE addressing is used, this is used to address the odd addresses. Set to 0 if SHORT addressing is used in the subroutine command.
caddr_highbyte	Int16	If BYTE addressing is used, this is used to address the odd addresses. Set to 0 if SHORT addressing is used in the subroutine command.
oaddr_highbyte	Int16	If BYTE addressing is used, this is used to address the odd addresses. Set to 0 if SHORT addressing is used in the subroutine command.
cmd_ptr	Int16*	Current command pointer where the CALL command should be inserted.

Description

Calls a single command subroutine along with data pointer modification. A single command subroutine executes a single computation command before returning control to the command following the CALL command. Data pointer modification allows a VICP computation unit to run the same subroutine multiple times using different data pointers.

The computation command in the subroutine can use SHORT addressing or BYTE addressing. If BYTE addressing is used, then the XXXX_highbyte parameters are used to access the odd bytes. Set to zero to access the even bytes and set to one to access the odd bytes. If SHORT addressing is used, the XXXX_highbyte parameters should be set to zero.

6.4.3 imxenc_call_with_ptr_ind

imxenc_call_with_ptr_ind ***Calls a single command subroutine along with data pointer modification. The data pointers are read from the location passed into the function.***

Syntax

```
length = Int16 imxenc_call_with_ptr_ind(
               Int16 *cmdaddr,
               Int16 *iaddr,
               Int16 *caddr,
               Int16 *oaddr,
               Int16 iaddr_highbyte,
               Int16 caddr_highbyte,
               Int16 oaddr_highbyte,
               Int16 *cmd_ptr);
```

Parameters

cmdaddr	Int16	A pointer to the start of the subroutine in the VICP command memory.
iaddr	Int16*	Pointer to a location containing the data pointer to substitute into the subroutine command.
caddr	Int16*	Pointer to a location containing the coefficient or 2nd input pointer to substitute into the subroutine command.
oaddr	Int16*	Pointer to a location containing the output pointer to substitute into the subroutine command.
iaddr_highbyte	Int16	Set to 0.
caddr_highbyte	Int16	Set to 0.
oaddr_highbyte	Int16	Set to 0.
cmd_ptr	Int16*	Current command pointer where the CALL command should be inserted.

Description

Calls a single command subroutine along with data pointer modification. A single command subroutine executes a single computation command before returning control to the command following the CALL command. Data pointer modification allows the VICP computation unit to run the same subroutine multiple times using different data pointers. In this case, the pointers used are read from the location passed into this function. A VICP command sequence can modify the pointers in data memory and call the subroutine multiple times for each set of pointers.

The XXXX_highbyte parameters must be set to zero because the pointers must be aligned on 16-bit boundaries.

6.4.4 imxenc_call_till_return

imxenc_call_till_return *Inserts a CALL UNTIL RETURN command into the VICP program sequence.*

Syntax

```
length = imxenc_call_till_return(Int16 * address,
                                 Int16 * cmd_ptr);
```

Parameters

address	Int16*	Pointer to the command to branch to in the VICP command memory.
cmd_ptr	Int16*	Current command pointer where the CALL command should be inserted.

Description

Inserts a CALL UNTIL RETURN command into the VICP program sequence. The CALL command jumps to the point in the VICP program pointed to by address. The VICP then executes until a RETURN command is found. After finding the RETURN command, control is passed back to the point in the command sequence following the CALL instruction.

6.4.5 imxenc_return_cmd

imxenc_return_cmd *Inserts a RETURN command into the VICP program sequence.*

Syntax

```
length = imxenc_return_cmd(Int16 * cmd_ptr);
```

Parameters

cmd_ptr	Int16*	Current command pointer where the CALL command should be inserted.

Description

Inserts a RETURN command into the VICP command sequence. This is needed at the end of a multiple-command subroutine. When the VICP executes a RETURN command, control is passed back to the point in the command sequence following a previous CALL UNTIL RETURN command. A RETURN command executed without a previous CALL command leads to unpredictable behavior.

6.4.6 imxenc_cmdwrite

imxenc_cmdwrite ***Write a block of data into command memory.***

Syntax

```
length = Int16 imxenc_cmdwrite(
               Int16 *cmdaddr,
               Int16 *data,
               Int16 num,
               Int16 *cmd_ptr);
```

Parameters

cmdaddr	Int16*	A pointer to the start of the block in the VICP command memory to overwrite.
data	Int16*	Pointer to the source block of data to write.
num	Int16	Number of 16-bit words to write into command memory. Writes (num + 1) words.
cmd_ptr	Int16*	Current command pointer where the CALL command should be inserted.

Description

Writes a block of data from data memory (image buffer or coefficient memory) into command memory. This command can be used to modify VICP commands as they are running. An example use of this functionality can be to replace a CALL command with a NOP to implement some level of conditional programming.

If the command block being written is immediately after the CMDWRITE instruction, then cmd_p must be aligned to a 32-bits boundary by adding NOP commmands as follow:

```
if (((cmd_p) - (short*)IMXCMDBUF_BASE)&1){
       cmd_p+= imxenc_nop(cmd_p);
     }
```

6.4.7 imxenc_nop

imxenc_nop — *Inserts a NOP into the VICP command sequence.*

Syntax

```
length = Int16 imxenc_nop(
             Int16 *cmd_ptr);
```

Parameters

cmd_ptr	Int16*	Current command pointer where the command should be inserted.

Description

Inserts a NOP into the VICP command sequence.

6.5 Functions for disabling saturation, rounding, ASAP mode

6.5.1 IMX_setSat

IMX_setSat ***Enable or disable saturation for subsequent computation functions encoded.***

Syntax

```
Int16 IMX_setSat(
    Int16 state, /* Can be IMX_SAT_YES to enable or IMX_SAT_NO to disable */
    );
Returns old value.
```

Description

To enable saturation use:

```
oldSat= IMX_setSat(IMX_SAT_YES);
```

To disable saturation use:

```
oldSat= IMX_setSat(IMX_SAT_NO);
```

Saturation is enabled by default on power up.

6.5.2 IMX_setRound

IMX_setRound ***Enable or disable rounding for subsequent computation functions encoded, which have non zero rightshift.***

Syntax

```
Int16 IMX_setRound(
    Int16 state, /* Can be IMX_ROUMD_YES to enable or IMX_ROUND_NO to disable */
    );
Returns old value.
```

Description

To enable rounding use:

```
oldSat= IMX_setRound(IMX_ROUND_YES);
```

To disable rounding (equivalent of truncating the result or floor()):

```
oldSat= IMX_setRound(IMX_ROUND_NO);
```

Rounding is enabled by default on power up.

6.5.3 IMX_setASAP

IMX_setASAP ***Enable or disable ASAP mode for subsequent computation functions encoded.***

Syntax

```
Int16 IMX_setASAP(
    Int16 ASAPmode, /* Can be IMX_ASAP_ENABLE to enable or
                       IMX_ASAP_DISABLE to disable */
    );
Returns old value.
```

Description

See Section 2.10 for explanations on ASAP mode.

To enable rounding use:

```
oldSat= IMX_setASAP(IMX_ASAP_ENABLE);
```

To disable rounding use:

```
oldSat= IMX_setASAP(IMX_ASAP_DISABLE);
```