# xCORE-200 DSP Library

This API reference manual describes the XMOS fixed-point digital signal processing software library. The library implements of a suite of common signal processing functions for use on XMOS xCORE-200 multi-core microcontrollers.

## Required tools and libraries

- xTIMEcomposer Tools Version 14.0.1 or later

## Required hardware

Only XMOS xCORE-200 based multicore microcontrollers are supported with this library. The previsous generation XS1 based multicore microntrollers are not supported.

The xCORE-200 has a single cycle 32x32->64 bit multiply/accumulate unit, single cycle double-word load and store, dual issue instruction execution, and other instruction set enhancements. These features make xCORE-200 an efficient platform for executing digital signal processing algorithms.

## Prerequisites

This document assumes familiarity with the XMOS xCORE architecture, the XMOS tool chain, the 'C' programming language, and digital signal processing concepts.

Copyright 2015 XMOS Ltd.

# 1 Overview

## 1.1 Introduction

This API reference manual describes the XMOS xCORE-200 fixed-point digital signal processing firmware library. The library implements of a suite of common signal processing functions for use on XMOS xCORE-200 multicore microcontrollers.

## 1.2 Library Organization

The library is divided into function collections with each collection covering a specific digital signal processing algorithm category. The API and implementation for each category are provided by a single 'C' header file and implementation file.

| Category | Source Files | Functions |
|---|---|---|
| Fixed point | xmos_dsp_qformat | Q16 through Q31 formats, fixed and floating point conversions |
| Filters | xmos_dsp_filters | FIR, biquad, cascaded biquad, and convolution |
| Adaptive | xmos_dsp_adaptive | LMS and NLMS Adaptive filters |
| Scalar math | xmos_dsp_math | Multiply, square root, reciprocal, inverse square root |
| Vector math | xmos_dsp_vector | Scalar/vector add/subtract/multiply, dot product |
| Matrix math | xmos_dsp_matrix | Scalar/matrix add/subtract/multiply, inverse and transpose |
| Statistics | xmos_dsp_statistics | Vector mean, sum-of-squares, root-mean-square, variance |

Copyright 2015 XMOS Ltd.

# 2 Fixed-Point Format

## 2.1 Q Format Introduction

The library functions support 32 bit input and output data, with internal 64 bit accumulator. The output data can be scaled to any of the supported Q Formats (Q16 through Q31). Further details about Q Format numbers is available here : https://en.wikipedia.org/wiki/Q_(number_format).

## 2.2 The 'q_format' Parameter

All XMOS DSP library functions that incorporate a multiply operation accept a parameter called q_format. This parameter can naively be used to specify the fixed point format for all operands and results (if applicable) where the formats are the same for all parameters. For example:

```
result_q28 = xmos_dsp_math_multiply( input1_q28, input2_q28, 28 );
```

The 'q_format' parameter, being used after one or more sequences of multiply and/or multiply-accumulate, is used to right-shift the 64-bit accumulator before truncating the value back to a 32-bit integer (i.e. the 32-bit fixed-point result). Therefore the 'q_format' parameter can be used to perform the proper fixed-point adjustment for any combination of input operand fixed-point format and desired fixed-point result format.

The output fixed-point fraction bit count is equal to the sum of the two input fraction bit counts minus the desired result fraction bit count:

```
q_format = input1 fraction bit count +  input2 fraction bit count - result fraction bit count
```

For example:

```
// q_format_parameter = 31 = 30 + 29 - 28
result_q28 = xmos_dsp_math_multiply( input1_q30, input2_q29, 31 );

// q_format_parameter = 27 = 28 + 29 - 30
result_q30 = xmos_dsp_math_multiply( input1_q28, input2_q29, 27 );
```

# 3 Filter Functions: Finite Impulse Response (FIR) Filter

Implements the Finite Impulse Response (FIR). The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample).

```
int xmos_dsp_filters_fir( int       input_sample,
                          const int filter_coeffs[],
                          int       state_data[],
                          int       tap_count,
                          int       q_format )
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_sample | in | The new sample to be processed. |
| filter_coeffs[] | in | Pointer to FIR coefficients arranged as [b0,b1,b2, ...,bN-1]. |
| state_data[] | in, out | Pointer to filter state data array of length N. |
| | | Must be initialized at startup to all zero's. |
| tap_count | in | Filter tap count where N = tap_count = filter order + 1. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Resulting filter output sample. |

**Algorithm**

The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient h[i] is multiplied by a state variable which equals a previous input sample x[i]:

```
y[n] = x[n] * h[0] + x[n-1] * h[1] + x[n-2] * h[2] ... + x[n-N+1] * h[N-1]
```

'filter_coeffs' points to a coefficient array of size N = 'num_taps'. The filter coefficients are stored in forward order (e.g. h[0], h[1], ..., h[N-1]).

**Behavior**

The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

**Example**

```
// Five-tap (4th order) FIR filter with samples and coefficients represented in Q28 fixed-point format

int filter_coeff[5] = { Q28(0.5), Q(-0.5), Q28(0.0), Q28(-0.5), Q28(0.5) };
int filter_state[4] = { 0, 0, 0, 0 };

int result = xmos_dsp_fir( sample, filter_coeff, filter_state, 5, 28 );
```

# 4 Filter Functions: Interpolating FIR Filter

Implements an interpolating Finite Impulse Response (FIR). The function operates on a single input sample and outputs a set of samples representing the interpolated data, whose sample count is equal to 'interp_factor'. (i.e. and each call to the function processes one sample and result in 'interp_factor' output samples).

```
void xmos_dsp_filters_interpolate
(
  int       output_samples[],
  int       input_sample,
  const int filter_coeffs[],
  int       state_data[],
  int       tap_count,
  int       interp_factor,
  int       q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| output_samples | out | The resulting interpolated samples. |
| input_sample | in | The new sample to be processed. |
| filter_coeffs[] | in | **Pointer to FIR coefficients arranged as:**<br>hM,h(1L+M),h(2L+M),...h((N-1)L+M), ...<br>h1,h(1L+1),h(2L+1),...h((N-1)L+1),<br>h0,h(1L+0),h(2L+0),...h((N-1)L+0), where M = N-1 |
| state_data[] | in, out | Pointer to filter state data array of length N.<br>Must be initialized at startup to all zero's. |
| tap_count | in | Filter tap count where N = tap_count = filter order + 1. |
| interp_factor | in | The interpolation factor/index (i.e. the up-sampling ratio).<br>The interpolation factor/index can range from 2 to 16. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Resulting filter output sample. |

### Algorithm

The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient h[i] is multiplied by a state variable which equals a previous input sample x[i]:

```
y[n] = x[n] * h[0] + x[n-1] * h[1] + x[n-2] * h[2] ... + x[n-N+1] * h[N-1]
```

'filter_coeffs' points to a coefficient array of size N = 'num_taps'. The filter coefficients are stored in forward order (e.g. h[0], h[1], ..., h[N-1]).

### Behavior

The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of

the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

# 5 Filter Functions: Decimating FIR Filter

Implements an decimating Finite Impulse Response (FIR). The function operates on a single set of input samples whose count is equal to the decimation factor. (i.e. and each call to the function processes 'decim_factor' samples and results in one sample).

```
void xmos_dsp_filters_decimate
(
  int       input_samples[],
  const int filter_coeffs[],
  int       state_data[],
  int       tap_count,
  int       decim_factor,
  int       q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_sample | in | The new sample to be processed. |
| filter_coeffs[] | in | **Pointer to FIR coefficients arranged as:**<br>hM,h(1L+M),h(2L+M),...h((N-1)L+M), ...<br>h1,h(1L+1),h(2L+1),...h((N-1)L+1),<br>h0,h(1L+0),h(2L+0),...h((N-1)L+0), where M = N-1 |
| state_data[] | in, out | Pointer to filter state data array of length N.<br>Must be initialized at startup to all zero's. |
| tap_count | in | Filter tap count where N = tap_count = filter order + 1. |
| decim_factor | in | The decimation factor/index (i.e. the down-sampling ratio).<br>The decimation factor/index can range from 2 to 16. |
| output_samples | out | The resulting interpolated samples. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | The resulting decimated sample. |

**Algorithm**

The FIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient h[i] is multiplied by a state variable which equals a previous input sample x[i]:

```
y[n] = x[n] * h[0] + x[n-1] * h[1] + x[n-2] * h[2] ... + x[n-N+1] * h[N-1]
```

'filter_coeffs' points to a coefficient array of size N = 'num_taps'. The filter coefficients are stored in forward order (e.g. h[0], h[1], ..., h[N-1]).

**Behavior**

The FIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

Copyright 2015 XMOS Ltd.

# 6 Filter Functions: BiQuad Infinite Impulse Repsonse (IIR) Filter (direct form I)

Implements a second order Infinite Impulse Response (IIR) direct form I. The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample).

```
int xmos_dsp_filters_biquad
(
  int       input_sample,
  const int filter_coeffs[],
  int       state_data[4],
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_sample | in | The new sample to be processed. |
| filter_coeffs[] | in | Pointer to biquad coefficients arranged as [b0,b1,b2,a1,a2]. |
| state_data[] | in, out | Pointer to filter state data array of length 4. Must be initialized at startup to all zero's. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Resulting filter output sample. |

### Algorithm

The IIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient b[i] is multiplied by a state variable which equals a previous input sample x[i]:

```
y[i] = x[n] * b[0] + x[n-1] * b[1] + x[n-2] * b2 + x[n-1] * a[1] + x[n-2] * a[2]
```

The filter coefficients are stored in forward order (e.g. b0, b1, b2, a1, a2).

### Behavior

The IIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

### Example

```
// Single Biquad filter with samples and coefficients represented in Q28 fixed-point format.

int filter_coeff[5] = { Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1) };
int filter_state[4] = { 0, 0, 0, 0 };

int result = xmos_dsp_biquad( sample, filter_coeff, filter_state, 28 );
```

# 7 Filter Functions: Cascaded BiQuad IIR Filter (direct form I)

Implements multiple second order Infinite Impulse Response (IIR) direct form I filters in series (cascaded Biquads). The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample).

```
int xmos_dsp_filters_biquads
(
  int       input_sample,
  const int filter_coeffs[],
  int       state_data[],
  int       num_sections,
  int       q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_sample | in | The new sample to be processed. |
| filter_coeffs[] | in | Pointer to biquad coefficients for all BiQuad sections. |
| | | Arranged as [section 1: b0,b1,b2,a1,a2, ... section N: b0,b1,b2,a1,a2]. |
| state_data[] | in, out | Pointer to filter state data array of length 4. |
| | | Must be initialized at startup to all zero's. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| num_sections | in | Number of BiQuad sections. |
| Return Value | out | Resulting filter output sample. |

**Algorithm**

The IIR filter algorithm is based upon a sequence of multiply-accumulate (MAC) operations. Each filter coefficient b[i] is multiplied by a state variable which equals a previous input sample x[i]:

```
y[n] = x[n] * b[0] + x[n-1] * b[1] + x[n-2] * b2 + x[n-1] * a[1] + x[n-2] * a[2]
```

The filter coefficients are stored in forward order (e.g. section 1: b0, b1, b2, a1, a2, ..., section N: b0, b1, b2, a1, a2).

**Behavior**

The IIR algorithm involves multiplication between 32-bit filter coefficients and 32-bit state data producing a 64-bit result for each coeffient and state data pair. Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered.

**Example**

```
// 4x Cascaded Biquad filter with samples and coefficients represented in Q28 fixed-point format.

int filter_coeff[20] = { Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),
                         Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),
                         Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1),
                         Q28(+0.5), Q(-0.1), Q28(-0.5), Q28(-0.1), Q28(0.1) };
int filter_state[16] = { 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0 };

int result = xmos_dsp_cascaded_biquad( sample, filter_coeff, filter_state, 4, 28 );
```

# 8 Adaptive Filter Functions: Least-Mean-Squares Adaptive Filter

Implements a least-mean-squares adaptive FIR filter. LMS filters are a class of adaptive filters that adjust filter coefficients in order to create the a transfer function that minimizes the error between the input and reference signals. FIR coefficients are adjusted on a per sample basis by an amount calculated from the given step size and the instantaneous error.

The function operates on a single sample of input and output data (i.e. and each call to the function processes one sample and each call results in changes to the FIR coefficients).

```
int xmos_dsp_adaptive_lms
(
    int source_sample,
    int reference_sample,
    int error_sample[],
    int filter_coeffs[],
    int state_data[],
    int tap_count,
    int step_size,
    int q_format
)
```

**Parameters**

**Algorithm**

The general LMS algorithm, on a per sample basis, is to:

1. Apply the transfer function: output = FIR( input )
2. Compute the instantaneous error value: error = reference - output
3. Compute current coefficient adjustment delta: delta = mu * error
4. Adjust transfer function coefficients: FIR_COEFFS[n] = FIR_COEFFS[n] + FIR_STATE[n] * delta

**Behavior**

The LMS filter algorithm involves multiplication between two 32-bit values and 64-bit accumulation as a result of using an FIR as well as coefficient step size calculations). Multiplication results are accumulated in 64-bit accumulater with the final result shifted to the required fixed-point format. Therefore overflow behavior of the 32-bit multiply operation and truncation behavior from final shifing of the accumulated multiplication results must be considered for both FIR operations as well as for coefficient step size calculation and FIR coefficient adjustment.

**Example**

```
// 100-tap LMS filter with samples and coefficients represented in Q28 fixed-point format.

int filter_coeff[100] = { ... not shown for brevity };
int filter_state[100] = { 0, 0, 0, 0, ... not shown for brevity };

int output_sample = xmos_dsp_adaptive_lms
(
   input_sample, reference_sample, &error_sample,
   filter_coeff_array, filter_state_array, 100, Q28(0.01), 28
);
```

# 9 Scalar Math Functions: Multiply

Multiplies two scalar values and produces a result according to fixed-point format specified by the 'q_format' parameter.

```
int xmos_dsp_math_multiply
(
  int input1_value,
  int input2_value,
  int q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input1_value | in | Multiply operand #1. |
| input2_value | in | Multiply operand #2. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | input1_value * input2_value. |

**Algorithm**

The two operands are multiplied to produce a 64-bit result which is tested for overflow, clamped at the minimum/maximum value given the fixed-point format if overflow occurs, and finally shifted right by 'q_format' bits.

```
1) Y = X1 * X2
2) Y = min( max( Q_FORMAT_MIN, Y ), Q_FORMAT_MAX, Y )
3) Y = Y >> q_format
```

**Behavior**

While saturation is employed after multiplication an overflow condition when preparing the final result must still be considered when specifying a Q-format whose fixed-point numerical range do not accomodate the final result of multiplication and saturation (if applied).

**Example**

```
int result = xmos_dsp_math_multiply( Q28(-0.33), sample, 28 );
```

# 10 Scalar Math Functions: Reciprocal

Computes the reciprocal of the input value using an iterative approximation method.

```
int xmos_dsp_math_reciprocal
(
  int input_value,
  int q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_value | in | Input value for computation. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Reciprocal of the input value. |

**Algorithm**

```
1) result = 1.0
2) result = result + result * (1     input_value * result)
3) Repeat step #2 until desired precision is achieved
```

**Behavior**

**Example**

```
int result = xmos_dsp_math_reciprocal( sample, 28 );
```

# 11 Scalar Math Functions: Inverse Square Root

Computes the reciprocal of the square root of the input value using an iterative approximation method.

```
int xmos_dsp_math_invsqrroot
(
  int input_value,
  int q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_value | in | Input value for computation. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Reciprocal of the square root of the input value. |

**Algorithm**

```
1) result = 1.0
2) result = result + result * (1 - input * result^2) / 2
3) Repeat step #2 until desired precision is achieved
```

**Behavior**

**Example**

```
int result = xmos_dsp_math_invsqrroot( sample, 28 );
```

# 12 Scalar Math Functions: Square Root

Computes the square root of the input value.

```
int xmos_dsp_math_squareroot
(
  int input_value,
  int q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_value | in | Input value for computation. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Square root of the input value. |

**Algorithm**

result = xmos_dsp_math_reciprocal( xmos_dsp_math_invsqrroot( input ))

**Behavior**

See behavior for the functions ‘xmos_dsp_math_invsqrroot’ and ‘xmos_dsp_math_reciprocal’.

**Example**

```
int result = xmos_dsp_math_squareroot( sample, 28 );
```

# 13 Vector Math Functions: Minimum Value

Locates the vector's first occurring minimum value, returning the index of the first occuring minimum value.

```
int xmos_dsp_vector_minimum
(
  const int input_vector_X[],
  int       vector_length,
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| Return Value | out | Index of the first occurring minimum value in the input vector. |

**Algorithm**

```
index = -1, value = maximum 32 bit signed integer
for i = 0 to (vector_length - 1):
  if input_vector_X[i] < result:
    value = input_vector_X[i]
    index = i
return index
```

**Example**

```
int samples[256];
int result = xmos_dsp_vector_minimum( samples, 256 );
```

# 14 Vector Math Functions: Maximum Value

Locates the vector's first occurring maximum value, returning the index of the first occuring maximum value.

```
int xmos_dsp_vector_maximum
(
  const int input_vector_X[],
  int       vector_length,
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| Return Value | out | Index of the first occurring maximum value in the input vector. |

### Algorithm

```
index = -1, value = minimum 32 bit signed integer
for i = 0 to (vector_length - 1):
  if input_vector_X[i] > result:
    value = input_vector_X[i]
    index = i
return index
```

### Example

```
int samples[256];
int result = xmos_dsp_vector_maximum( samples, 256 );
```

# 15 Vector Math Functions: Element Negation

Computes the negative value for each input element and sets the corresponding result element to its negative value: result_vector[i] = -vector_X[i].

```
void xmos_dsp_vector_negate
(
  const int input_vector_X[],
  int       result_vector_R[],
  int       vector_length,
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |

**Algorithm**

```
result = 0
for i = 0 to (vector_length - 1): input_vector_X[i] = -input_vector_X[i]
```

**Behavior**

Each negated element is computed by twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate it's corresponding maximum positive fixed-point value.

For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.

**Example**

```
int samples[256];
int result[256];
xmos_dsp_vector_negate( samples, result, 256 );
```

# 16 Vector Math Functions: Element Absolute Value

Sets each element of the result vector to the absolute value of the corresponding input vector element: result_vector[i] = abs( vector_X[i] ).

```
void xmos_dsp_vector_abs
(
  const int input_vector_X[],
  int       result_vector_R[],
  int       vector_length,
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |

**Algorithm**

```
for i = 0 to (vector_length - 1): result_vector_R[i] = abs( input_vector_X[i] )
```

**Behavior**

If an element is less than zero it is negated to compute it's absolute value. Negation is computed via twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate it's corresponding maximum positive fixed-point value.

For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.

**Example**

```
int samples[256];
int result[256];
xmos_dsp_vector_abs( samples, result, 256 );
```

# 17 Vector Math Functions: Scalar Addition

Adds a scalar value to each vector element: result_vector[i] = vector_X[i] + scalar_A.

```
void xmos_dsp_vector_adds
(
  const int input_vector_X[],
  int       scalar_value_A,
  int       result_vector_R[],
  int       vector_length,
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| scalar_value_A | in | Scalar value to add to each 'input' element. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] * scalar_value_A
```

### Behavior

32-bit addition is used to compute the scaler plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int scalar_value_A = Q28( 0.333 );
int result_vector_R[256];
xmos_dsp_vector_adds( input_vector_X, scalar_value_A, result_vector_R, 256 );
```

# 18 Vector Math Functions: Scalar Multiplication

Multiplies each vector element by a scaler value: result_vector[i] = vector_X[i] * scalar_A.

```
void xmos_dsp_vector_muls
(
  const int input_vector_X[],
  int       scalar_value_A,]
  int       result_vector_R[],
  int       vector_length,]
  int       q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| scalar_value_A | in | Scalar value to multiply each 'input' element by. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

**Algorithm**

**for i = 0 to (vector_length - 1):** result_vector_R[i] = input_vector_X[i] * scalar_value_A

**Behavior**

Each element in the input vectors is multiplied by a scalar using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

**Example**

```
int input_vector_X[256];
int scalar_value_A = Q28( 0.333 );
int result_vector_R[256];
xmos_dsp_vector_muls( input_vector_X, scalar_value_A, result_vector_R, 256, 28 );
```

# 19 Vector Math Functions: Vector Addition

Subtracts two vectors element-by-element: result_vector[i] = vector_X[i] + vector_Y[i].

```
void xmos_dsp_vector_addv
(
  const int input_vector_X[],
  const int input_vector_Y[],
  int       result_vector_R[],
  int       vector_length,
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] + input_vector_Y[i]
```

### Behavior

32-bit addition is used to compute the vector element plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int input_vector_Y[256];
int result_vector_R[256];
xmos_dsp_vector_addv( input_vector_X, input_vector_Y, result_vector_R, 256 );
```

# 20 Vector Math Functions: Vector Subtraction

Subtracts two vectors element-by-element: result_vector[i] = vector_X[i] - vector_Y[i].

```
void xmos_dsp_vector_subv
(
  const int input_vector_X[],]
  const int input_vector_Y[],
  int       result_vector_R[],
  int       vector_length,
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] - input_vector_Y[i]
```

### Behavior

32-bit subtraction is used to compute the vector element minus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int input_vector_Y[256];
int result_vector_R[256];
xmos_dsp_vector_subv( input_vector_X, input_vector_Y, result_vector_R, 256 );
```

# 21 Vector Math Functions: Vector Multiplication

Multiplies two vectors element-by-element: result_vector[i] = vector_X[i] * vector_Y[i].

```
void xmos_dsp_vector_mulv
(
  const int input_vector_X[],
  const int input_vector_Y[],
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] * input_vector_Y[i]
```

### Behavior

Elements in each of the input vectors are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

### Example

```
int input_vector_X[256];
int input_vector_Y[256];
int result_vector_R[256];
xmos_dsp_vector_mulv( input_vector_X, input_vector_Y, result_vector_R, 256, 28 );
```

# 22 Vector Math Functions: Vector multiplication and scalar addition

result_vector[i] = vector_X[i] * vector_Y[i] + scalar_A.

```
void xmos_dsp_vector_mulv_adds
(
  const int input_vector_X[],
  const int input_vector_Y[],
  int       input_scalar_A,
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| scalar_value_A | in | Scalar value to add to each X*Y result. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] * input_vector_Y[i] + input_scalar_A
```

### Behavior

Elements in each of the input vectors are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

32-bit addition is used to compute the vector element plus scalar value result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int input_vector_Y[256];
int scalar_value_A = Q28( 0.333 );
int result_vector_R[256];
xmos_dsp_vector_mulv_adds( input_vector_X, input_vector_Y, scalar_value_A, result_vector_R, 256, 28 );
```

# 23 Vector Math Functions: Scalar multiplication and vector addition

result_vector[i] = vector_X[i] * A + vector_Y[i].

```
void xmos_dsp_vector_muls_addv
(
  const int input_vector_X[],
  int       input_scalar_A,
  const int input_vector_Y[],
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| scalar_value_A | in | Scalar value to multiply each X element by. |
| input_vector_Y | in | Pointer to source data array Y. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_scalar_A * input_vector_X[i] + input_vector_Y[i]
```

### Behavior

Each element in the input vectors is multiplied by a scalar using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

32-bit addition is used to compute the vector element plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int scalar_value_A = Q28( 0.333 );
int input_vector_Y[256];
int result_vector_R[256];
xmos_dsp_vector_muls_addv( input_vector_X, input_scalar_A, input_vector_Y, result_vector_R, 256, 28 );
```

# 24 Vector Math Functions: Scalar multiplication and vector subtraction

result_vector[i] = vector_X[i] * scalar_A - vector_Y[i].

```
void xmos_dsp_vector_muls_subv
(
  const int input_vector_X[],
  int       input_scalar_A,
  const int input_vector_Y[],
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| scalar_value_A | in | Scalar value to multiply each X element by. |
| input_vector_Y | in | Pointer to source data array Y. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_scalar_A * input_vector_X[i] - input_vector_Y[i]
```

### Behavior

Each element in the input vectors is multiplied by a scalar using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

32-bit subtraction is used to compute the vector element minus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int scalar_value_A = Q28( 0.333 );
int input_vector_Y[256];
int result_vector_R[256];
xmos_dsp_vector_muls_subv( input_vector_X, input_scalar_A, input_vector_Y, result_vector_R, 256, 28 );
```

# 25 Vector Math Functions: Vector multiplication and vector addition

result_vector[i] = vector_X[i] * vector_Y[i] + vector_Z[i].

```
void xmos_dsp_vector_mulv_addv
(
  const int input_vector_X[],
  const int input_vector_Y[],
  const int input_vector_Z[],
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| input_vector_Z | in | Pointer to source data array Z. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] * input_vector_Y[i] + input_vector_Z[i]
```

### Behavior

Elements in each of the input vectors are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

32-bit addition is used to compute the vector element plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X[256];
int input_vector_Y[256];
int input_vector_Z[256];
int result_vector_R[256];
xmos_dsp_vector_mulv_addv( input_vector_X, input_vector_Y, input_vector_Z, result_vector_R, 256, 28 );
```

# 26 Vector Math Functions: Vector multiplication and vector subtraction

result_vector[i] = vector_X[i] * vector_Y[i] - vector_Z[i].

```
void xmos_dsp_vector_mulv_subv
(
  const int input_vector_X[],
  const int input_vector_Y[],
  const int input_vector_Z[],
  int       result_vector_R[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| input_vector_Y | in | Pointer to source data array Y. |
| input_vector_Z | in | Pointer to source data array Z. |
| result_vector_R | out | Pointer to the resulting data array. |
| vector_length | in | Length of the input and result vectors. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
for i = 0 to (vector_length - 1):
  result_vector_R[i] = input_vector_X[i] * input_vector_Y[i] - input_vector_Z[i]
```

### Behavior

Elements in each of the input vectors are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

32-bit subtraction is used to compute the vector element plus vector element result. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_vector_X [256];
int input_vector_Y [256];
int input_vector_Z [256];
int result_vector_R[256];
xmos_dsp_vector_mulv_subv( input_vector_X, input_vector_Y, input_vector_Z, result_vector_R, 256, 28 );
```

# 27 Matrix Math Functions: Element Negation

Computes the negative value for each input element and sets the corresponding result element to its negative value:

result_matrix[i][j] = -matrix_X[i][j].

```
void xmos_dsp_matrix_negate
(
  const int input_matrix_X[],
  int       result_matrix_R[],
  int       row_count]
  int       column_count
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| result_matrix_R | out | Pointer to the resulting data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    input_vector_X[k] = -input_vector_X[k]
```

### Behavior

Each negated element is computed by twos-compliment negation therefore the minimum negative fixed-point value can not be negated to generate it's corresponding maximum positive fixed-point value.

For example: -Q28(-8.0) will not result in a fixed-point value representing +8.0.

### Example

```
int samples[8][32];
int result[8][32];
xmos_dsp_matrix_negate( samples, result, 8, 32 );
```

# 28 Matrix Math Functions: Scalar Addition

Adds a scalar value to each element in matrix_X and stores the result in the corresponding result matrix element:

result_matrix[i][j] = matrix_X[i][j] + scalar_A.

```
void xmos_dsp_matrix_adds
(
  const int input_matrix_X[],  // Pointer/reference to 2-dimensional source data.
  int       scalar_value_A,    // Scalar value to add to each 'input' element.
  int       result_matrix_R[], // Pointer to the resulting 2-dimensional data array.
  int       row_count          // Number of rows in input matrix.
  int       column_count       // Number of columns in input matrix.
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| scalar_value_A | in | Scalar value to add to each ‘input’ element. |
| result_matrix_R | out | Pointer to the resulting 2-dimensional data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    result_matrix_R[k] = input_matrix_X[k] * scalar_value_A
```

### Behavior

32-bit addition is used to compute the result for each element. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_matrix_X[8][32];
int scalar_value_A = Q28( 0.333 );
int result_vector_R[8][32];
xmos_dsp_matrix_adds( input_matrix_X, scalar_matrix_A, result_matrix_R, 8, 32 );
```

# 29 Matrix Math Functions: Scalar Multiplication

Multiplies each element in matrix_X by the corresponding element in matrix_Y and stores the result in the corresponding result matrix element:

result_matrix[i][j] = scalar_A * matrix_X[i][j].

```
void xmos_dsp_matrix_muls
(
  const int input_matrix_X[],
  int       scalar_value_A,
  int       result_matrix_R[],
  int       row_count
  int       column_count
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| scalar_value_A | in | Scalar value to multiply to each 'input' element by. |
| result_matrix_R | out | Pointer to the resulting 2-dimensional data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    result_matrix_R[k] = input_matrix_X[k] * scalar_value_A
```

### Behavior

Each element of the input matrix is multiplied by a scalar value using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

### Example

```
int input_matrix_X[8][32];
int scalar_value_A = Q28( 0.333 );
int result_vector_R[8][32];
xmos_dsp_matrix_muls( input_matrix_X, scalar_value_A, result_matrix_R, 256, 8, 32, 28 );
```

Copyright 2015 XMOS Ltd.

# 30 Matrix Math Functions: Matrix Addition

Adds each element in matrix_X by the corresponding element in matrix_Y and stores the result in the corresponding result matrix element:

result_matrix[i][j] = matrix_X[i][j] + matrix_Y[i][j].

```
void xmos_dsp_matrix_addm
(
  const int input_matrix_X[],
  const int input_matrix_Y[],
  int       result_matrix_R[],
  int       row_count
  int       column_count
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| input_matrix_Y | in | Pointer to source 2-dimensional data array Y. |
| result_matrix_R | out | Pointer to the resulting 2-dimensional data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    result_matrix_R[k] = input_matrix_X[k] + input_matrix_Y[k]
```

### Behavior

32-bit addition is used to compute the result for each element. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_matrix_X[8][32];
int input_matrix_Y[8][32];
int result_vector_R[8][32];
xmos_dsp_matrix_addm( input_matrix_X, input_matrix_Y, result_matrix_R, 256, 8, 32, 28 );
```

# 31 Matrix Math Functions: Matrix Subtraction

Subtracts each element in matrix_Y from the corresponding element in matrix_X and stores the result in the corresponding result matrix element:

result_matrix[i][j] = matrix_X[i][j] + matrix_Y[i][j].

```
void xmos_dsp_matrix_addm
(
  const int input_matrix_X[],
  const int input_matrix_Y[],
  int       result_matrix_R[],
  int       row_count
  int       column_count
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| input_matrix_Y | in | Pointer to source 2-dimensional data array Y. |
| result_matrix_R | out | Pointer to the resulting 2-dimensional data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    result_matrix_R[k] = input_matrix_X[k] - input_matrix_Y[k]
```

### Behavior

32-bit addition is used to compute the result for each element. Therefore fixed-point value overflow conditions should be observed. The resulting values are not saturated.

### Example

```
int input_matrix_X[8][32];
int input_matrix_Y[8][32];
int result_vector_R[8][32];
xmos_dsp_matrix_addm( input_matrix_X, input_matrix_Y, result_matrix_R, 256, 8, 32, 28 );
```

Copyright 2015 XMOS Ltd.

# 32 Matrix Math Functions: Matrix Multiplication

Multiplies each element in matrix_X by the corresponding element in matrix_Y and stores the result in the corresponding element in the result matrix:

result_matrix[i][j] = matrix_X[i][j] * matrix_Y[i][j].

```
void xmos_dsp_matrix_mulm
(
  const int input_matrix_X[],
  const int input_matrix_Y[],
  int       result_matrix_R[],
  int       row_count
  int       column_count
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_matrix_X | in | Pointer to 2-dimensional source data X. |
| input_matrix_Y | in | Pointer to source 2-dimensional data array Y. |
| result_matrix_R | out | Pointer to the resulting 2-dimensional data array. |
| row_count | in | Number of rows in input and result matrices. |
| column_count | in | Number of columns in input and result matrices. |
| q_format | in | Fixed point format (number of bits making up fractional part). |

### Algorithm

```
result = 0
for i = 0 to (row_count - 1):
  for j = 0 to (column_count - 1):
    k = i * column_count + j
    result_matrix_R[k] = input_matrix_X[k] * input_matrix_Y[k]
```

### Behavior

Elements in each of the input matrices are multiplied together using a 32bit multiply 64-bit accumulate function therefore fixed-point multiplication and q-format adjustment overflow behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

### Example

```
int input_matrix_X[8][32];
int input_matrix_Y[8][32];
int result_vector_R[8][32];
xmos_dsp_matrix_mulm( input_matrix_X, input_matrix_Y, result_matrix_R, 256, 8, 32, 28 );
```

# 33 Statistics Functions: Vector Mean

Computes the mean of the values contained within the input vector:

result = (vector_X[0] + ... vector_X[N-1]) / vector_length

```
int xmos_dsp_vector_mean
(
  const int input_vector_X[],
  int       vector_length,
  int       q_format
)
```

**Parameters**

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Mean value of vector elements. |

**Algorithm**

```
result = 0
for i = 0 to N-1: result += input_vector_X[i]
return result / vector_length
```

**Behavior**

Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by 'q_format' bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.

**Example**

```
int result = xmos_dsp_vector_mean( input_vector, 256, 28 );
```

Copyright 2015 XMOS Ltd.

# 34 Statistics Functions: Vector Power (Sum-of-Squares)

Computes the power (also know as the sum-of-squares) of the values contained within the input vector:

result = vector_X[0]^2 + ... vector_X[(vector_length)-1]^2

```
int xmos_dsp_vector_power
(
  const int input_vector_X[], // Pointer to source data array X.
  int       vector_length,    // Length of the input vector.
  int       q_format          // Fixed point format (number of bits making up fractional part).
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Sum-of-squares for all vector elements. |

### Algorithm

```
result = 0
for i = 0 to N-1: result += input_vector_X[i] ^ 2
return result
```

### Behavior

Since each element in the vector is squared the behavior for fixed-point multiplication should be considered (see behavior for the function 'xmos_dsp_math_multiply').

Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by 'q_format' bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.

### Example

```
int result = xmos_dsp_vector_power( input_vector, 256, 28 );
```

# 35 Statistics Functions: Root Mean Square (RMS)

Computes the root-mean-square (RMS) of the values contained within the input vector:

result = ((vector_X[0]^2 + ... + vector_X[N-1]^2) / N) ^ 0.5) where N = vector_length

```
int xmos_dsp_vector_rms
(
  const int input_vector_X[], // Pointer to source data array X.
  int       vector_length,    // Length of the input vector.
  int       q_format          // Fixed point format (number of bits making up fractional part).
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | Root-mean-square of all vector elements. |

### Algorithm

```
result = 0
for i = 0 to N-1: result += input_vector_X[i]
return xmos_dsp_math_squareroot( result / vector_length )
```

### Behavior

Since each element in the vector is squared the behavior for fixed-point multiplication should be considered (see behavior for the function 'xmos_dsp_math_multiply').

Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely.

The squareroot of the 'sum-of-squares divided by N value' uses the function 'xmos_dsp_math_squareroot'; see behavior for that function.

The final value, being effectively the result of a left-shift by 'q_format' bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.

### Example

```
int result = xmos_dsp_vector_rms( input_vector, 256, 28 );
```

# 36 Statistics Functions: Dot Product

Computes the dot-product of two equal length vectors:

result = vector_X[0] * vector_Y[0] + ... vector_X[N-1] * vector_Y[N-1] where N = vector_length

```
int xmos_dsp_vector_dotprod
(
  const int input_vector_X[],
  const int input_vector_Y[],
  int       vector_length,
  int       q_format
)
```

### Parameters

| Name | Direction | Description |
|---|---|---|
| input_vector_X | in | Pointer to source data array X. |
| vector_length | in | Length of the input vector. |
| q_format | in | Fixed point format (number of bits making up fractional part). |
| Return Value | out | The dot product of two equal sized vectors. |

### Algorithm

```
result = 0
for i = 0 to N-1: result += input_vector_X[i] * input_vector_Y[i]
return result
```

### Behavior

The elements in the input vectors are multiplied before being summed therefore fixed-point multiplication behavior must be considered (see behavior for the function 'xmos_dsp_math_multiply').

Due to successive 32-bit additions being accumulated using 64-bit arithmetic overflow during the summation process is unlikely. The final value, being effectively the result of a left-shift by 'q_format' bits will potentially overflow the final fixed-point value depending on the resulting summed value and the chosen Q-format.

### Example

```
int result = xmos_dsp_vector_dotprod( input_vector, 256, 28 );
```

XMOS®
Copyright © 2015, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.