



Gowin RiscV_AE350_SOC DSP Software Programming Manual

MUG1032-1.1E, 01/26/2024

Copyright © 2024 Guangdong Gowin Semiconductor Corporation. All Rights Reserved.

GOWIN is a trademark of Guangdong Gowin Semiconductor Corporation and is registered in China, the U.S. Patent and Trademark Office, and other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders. No part of this document may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written consent of GOWINSEMI.

Disclaimer

GOWINSEMI assumes no liability and provides no warranty (either expressed or implied) and is not responsible for any damage incurred to your hardware, software, data, or property resulting from usage of the materials or intellectual property except as outlined in the GOWINSEMI Terms and Conditions of Sale. GOWINSEMI may make changes to this document at any time without prior notice. Anyone relying on this documentation should contact GOWINSEMI for the current documentation and errata.

Revision History

Date	Version	Description
10/31/2023	1.0E	Initial version published.
01/26/2024	1.1E	Minor description changes.

Contents

Contents	i
List of Figures	v
List of Tables	vi
1 About This Manual	1
1.1 Purpose	1
1.2 Terminology and Abbreviations	1
1.3 Support and Feedback	1
2 Overview	2
2.1 Linking Options for the Newlib and MCULib Toolchains	4
2.2 Linking Options for the Glibc Toolchains	5
2.3 Data Alignment for DSP Software Programming Function Library	6
3 Function Interface Descriptions	8
3.1 Basic Functions	8
3.1.1 Absolute Value Function	8
3.1.2 Addition Functions	10
3.1.3 Division Functions	12
3.1.4 Dot Product Functions	14
3.1.5 Multiplication Functions	16
3.1.6 Negation Functions	19
3.1.7 Offset Functions	20
3.1.8 Scaling Functions	22
3.1.9 Shift Functions	26
3.1.10 Subtraction Functions	28
3.2 Complex Functions	30
3.2.1 Complex Conjugate Functions	31

3.2.2 Complex Dot Product Functions	32
3.2.3 Complex Magnitude Functions	36
3.2.4 Complex Magnitude-Squared Functions	37
3.2.5 Complex-Multiply-Complex Functions	40
3.2.6 Complex-Multiply-Real Functions	41
3.3 Controller Functions	44
3.3.1 Clarke Transform Functions	44
3.3.2 Inverse Clarke Transform Functions	45
3.3.3 Park Transform Functions	46
3.3.4 Inverse Park Transform Functions	48
3.3.5 PID Functions	49
3.4 Filtering Functions	56
3.4.1 Biquadratic Cascade Filter (Direct Form I) Functions	56
3.4.2 Biquadratic Cascade Filter (Direct Form II Transposed) Functions	62
3.4.3 Convolution Functions	66
3.4.4 Partial Convolution Functions	68
3.4.5 Correlation Functions	70
3.4.6 Finite Impulse Response (FIR) Filter Functions	73
3.4.7 Decimation FIR Filter Functions	78
3.4.8 Lattice FIR Filter Functions	84
3.4.9 Lattice Infinite Impulse Response (IIR) Filter Functions	87
3.4.10 Least Mean Square (LMS) Filter Functions	92
3.4.11 Normalized Least Mean Square (NLMS) Filter Functions	96
3.4.12 Sparse FIR Filter Functions	100
3.4.13 Upsampling FIR Filter Functions	105
3.5 Matrix Functions	109
3.5.1 Matrix Addition Functions	109
3.5.2 Matrix Inverse Functions	111
3.5.3 Matrix Multiplication Functions	113
3.5.4 Matrix Scale Functions	118
3.5.5 Matrix Subtraction Functions	120
3.5.6 Matrix Transpose Functions	122

3.5.7 Matrix Power 2 Functions	124
3.5.8 Matrix Outer Product Functions	125
3.5.9 Complex Matrix Multiplication Functions	126
3.6 Statistics Functions	129
3.6.1 Maximum Functions	129
3.6.2 Mean Functions	131
3.6.3 Minimum Functions	133
3.6.4 Root Mean Square (RMS) Functions	135
3.6.5 Power Functions	136
3.6.6 Standard Deviation Functions	138
3.6.7 Variance Functions	139
3.6.8 Entropy Function	140
3.6.9 Relative Entropy Function	141
3.6.10 Log-Sum-Exp (LSE) Function	142
3.6.11 Log-Sum-Exp (LSE) Dot Product Function	142
3.6.12 Naive Gaussian Bayesian Estimation Function	143
3.7 Transform Functions	144
3.7.1 Radix-2 CFFT Functions	144
3.7.2 Radix-4 CFFT Functions	149
3.7.3 Complex FFT Functions	154
3.7.4 DCT Type II Functions	159
3.7.5 DCT Type IV Functions	163
3.7.6 Real FFT Functions	167
3.8 Utils Functions	172
3.8.1 arctan Functions	172
3.8.2 arctan2 Functions	173
3.8.3 cos and sin Functions	174
3.8.4 Conversion Functions	177
3.8.5 Duplicate Functions	182
3.8.6 Set Functions	184
3.8.7 Square Root Functions	186
3.8.8 Barycenter Function	187

3.8.9 Weighted Sum Function	187
3.9 Support Vector Machine (SVM) Estimation Functions	188
3.9.1 SVM Linear Estimation	188
3.9.2 SVM Radial Basis Function (RBF) Estimation	189
3.9.3 SVM Polynomial Estimation	191
3.9.4 SVM Sigmoid Estimation	192
3.10 Distance Functions	194
3.10.1 Float Distance	194
3.10.2 Boolean Distance.....	198
3.11 Sort Functions	202
3.11.1 Generic Sort Functions	202
3.11.2 Merge Sort Functions.....	205
4 Application Program.....	208
4.1 Program Description	208
4.2 Application Program.....	208
4.3 Program Running.....	208

List of Figures

Figure 3-1 Clarke Transform Diagram	44
Figure 3-2 Park Transform Diagram	46
Figure 3-3 State of PID Controller.....	49

List of Tables

Table 1-1 Terminology and Abbreviations	1
Table 2-1 Linking Options	5
Table 3-1 The Input and Output Formats	146
Table 3-2 The Input and Output Formats	147
Table 3-3 The Input and Output Formats	148
Table 3-4 The Input and Output Formats	149
Table 3-5 Input and Output Formats	151
Table 3-6 Input and Output Formats	152
Table 3-7 Input and Output Formats	153
Table 3-8 Input and Output Formats	154
Table 3-9 Input and Output Formats	156
Table 3-10 Input and Output Formats	157
Table 3-11 Input and Output Formats	157
Table 3-12 Input and Output Formats	158
Table 3-13 Input and Output Formats	161
Table 3-14 Input and Output Formats	161
Table 3-15 Input and Output Formats	162
Table 3-16 Input and Output Formats	163
Table 3-17 Input and Output Formats	165
Table 3-18 Input and Output Formats	165
Table 3-19 Input and Output Formats	166
Table 3-20 Input and Output Formats	167
Table 3-21 Input and Output Formats	169
Table 3-22 Input and Output Formats	170
Table 3-23 Input and Output Formats	171

Table 3-24 Input and Output Formats	171
---	-----

1 About This Manual

1.1 Purpose

This manual describes the function specification of Gowin RiscV_AE350_SOC DSP software programming function library, and how to use the DSP function interface software programming, the application program demo, etc.

1.2 Terminology and Abbreviations

Table 1-1 shows the abbreviations and terminology used in this manual.

Table 1-1 Terminology and Abbreviations

Terminology and Abbreviations	Meaning
CSR	Control and Status Register
DSP	Digital Signal Processor
ISA	Instruction Set Architecture
RISC-V	Reduced Instruction Set Computer V
SOC	System on Chip

1.3 Support and Feedback

Gowin Semiconductor provides customers with comprehensive technical support. If you have any questions, comments, or suggestions, please feel free to contact us directly using the information provided below.

Website: www.gowinsemi.com

E-mail: support@gowinsemi.com

2 Overview

Gowin RiscV_AE350_SOC DSP software programming library provides comprehensive function interfaces with efficient and streamlined code for easy and fast development of digital signal processing system. These function interfaces include basic vector, matrix and complex vector operations, as well as complex algorithmic operations such as filtering and function transformations. The DSP software programming library encapsulates complex signal processing into user-friendly application function interfaces. Designed and developed using the RDS software toolchain, it benefits users by saving development resources, allowing them to focus more on system design, and reducing development time.

The function interfaces of the DSP software programming library includes the following categories: basic function, complex function, controller function, filter function, matrix function, statistical function, transform function, tool function, SVM function, distance function and sorting function, and they will be introduced in detail in chapter [3 Function Interface Descriptions](#).

When the DSP function interface is called, only the header files of the DSP software programming function library are included. These header files define the prototypes of function interfaces and instance structures, named by functional category, as shown below:

- riscv_dsp_basic_math.h
- riscv_dsp_complex_math.h
- riscv_dsp_controller_math.h
- riscv_dsp_filtering_math.h
- riscv_dsp_matrix_math.h
- riscv_dsp_statistics_math.h
- riscv_dsp_transform_math.h
- riscv_dsp_utils_math.h

- riscv_dsp_svm_math.h
- riscv_dsp_distance_math.h

- `riscv_dsp_sort_math.h`

For example, if you need to use basic functions to perform certain operations on vectors, you can include the header file `riscv_dsp_basic_math.h` in the C file.

Note!

For better performance, the controller functions are directly implemented in the header file `riscv_dsp_controller_math.h`. If these function interfaces need to be optimized using the DSP ISA, the toolchain related functions must be enabled using the compiler flag `-mext-dsp`.

The DSP software programming library defines several data types to hold signal data for subsequent processing. These data types are defined in `riscv_dsp_math_types.h` as follows: `q7_t` for 8-bit integer, `q15_t` for 16-bit integer, `q31_t` for 32-bit integer, `q63_t` for 64-bit integer, `float32_t` for 32-bit single-precision floating-point, and `float64_t` for 64-bit double-precision floating-point, as shown below:

- `typedef int8_t q7_t; /* q7 type */`
- `typedef int16_t q15_t; /* q15 type */`
- `typedef int32_t q31_t; /* q31 type */`
- `typedef int64_t q63_t; /* q63 type */`
- `typedef float float32_t; /* single-precision floating-point type */`
- `typedef double float64_t; /* double-precision floating-point type */`

2.1 Linking Options for the Newlib and MCULib Toolchains

In addition to the header files described above, different toolchains are associated with different linking options, and this section details the options used for Newlib and MCULib, and the options include followings:

- `-ldsp` links to the DSP software programming function library (`libdsp.a`)
- `-mext-dsp` enables DSP ISA extension
- `-lm` links to libm math library

If the target system does not support the DSP ISA extension to accelerate the DSP software programming function library, use `-ldsp` to build an application that links to the DSP software programming function library (`libdsp.a`). If the target system supports DSP ISA extensions, use the `-ldsp -mext-dsp` option to enable the extension for better performance.

Note!

Applications built with the `-ldsp -mext-dsp` option should not be run on systems that do

not support DSP ISA extension, otherwise problems will occur when loading.

If the application needs to use the DSP software programming function interfaces that contain the libm math function library function interfaces (i.e., `riscv_dsp_std_f32`, `riscv_dsp_rms_f32`, `riscv_dsp_cmag_f32`, `riscv_dsp_sqrt_f32` and `riscv_dsp_atan_f32`), then use the `-lm` option.

2.2 Linking Options for the Glibc Toolchains

The linking options available for the glibc toolchain include:

- `-static` uses static linking
- `-ldsp` links to DSP software programming libraries without DSP ISA extension. `-ldsp_p` links to DSP software programming libraries with DSP ISA extension (for backward compatibility, the previous `-ldsp_hw` option, as a link to `-ldsp_p`).
- `-lm` links to libm math library

Static and dynamic links require different options, Table 2-1 summarizes the linking options for different link types and whether DSP ISA extension support is required.

Table 2-1 Linking Options

Type	DSP ISA Extension	Option
Static Linking	Without	<code>-static -ldsp</code>
	With	<code>-static -ldsp_p</code>
Dynamic Linking (default)	Without	<code>-ldsp</code>
	With	<code>-ldsp_p</code>

Among the required options, there are options with DSP ISA extension (i.e., `-static -ldsp_p` and `-ldsp_p`) used to build applications for target systems that support the extension, while the other options (e.g., `-static -ldsp` and `-ldsp`), are used only for target systems that do not support the extension. Users need to ensure that the options specified correspond to the target system used.

If the application is statical linking and requires the use of DSP software programming function interfaces that include the libm math function library function interfaces (i.e., `riscv_dsp_std_f32`, `riscv_dsp_rms_f32`, `riscv_dsp_cmag_f32`, `riscv_dsp_sqrt_f32` and `riscv_dsp_atan_f32`), then make sure to use the `-lm` option.

2.3 Data Alignment for DSP Software Programming Function Library

The DSP software programming function library is designed for the 32-bit RiscV_AE350_SOC MCU platform and uses 4-byte data alignment, i.e., it is assumed that the first addresses of the input arrays (including the input source array and the output destination array) are aligned with 4 bytes. Regardless of the type of input pointer, these function interfaces implement loading/saving a word, which is then processed using SIMD DSP instructions to improve performance. Accessing a word from a non-aligned memory address is called a non-aligned memory access; and an address non-alignment exception will occur in MCUs that do not support non-aligned accesses. For MCUs that support unaligned accesses, unaligned memory accesses exception will not occur, but it may result in a performance decrease of the system. For a detailed description of how the RiscV_AE350_SOC MCU handles non-aligned accesses, refer to the RiscV_AE350_SOC architecture and CSR definitions.

Of the data types supported by the DSP Software Programming Function Library, Q31 and F32 are both 4-byte data and are therefore declared with 4-byte address alignment. Q15 and Q7 are 2-byte and 1-byte data respectively, and by default they are aligned with 2-byte and 1-byte addresses respectively. To avoid inconsistent memory accesses to the 32-bit RiscV_AE350_SOC MCUs, it is necessary to explicitly declare that the input arrays of Q15 and Q7 are accessed at 4-byte-aligned addresses, as shown below:

```
q7_t inputA[blocksize] __attribute__((aligned(4)));  
q15_t outputvec[blocksize] __attribute__((aligned(4)));
```

The following functions do not require an explicit declaration of alignment access:

- Controller functions
- Transformation functions
- Arc tangent functions
- Arc tangent 2 functions
- Cosine functions
- Sine functions
- Square root functions

The following functions also do not require an explicit declaration of alignment access:

- Functions that use the F32 data to input arrays
- Functions that use the Q31 data to input arrays

3 Function Interface Descriptions

The following sections, categorized by functions, describe the function interfaces of the DSP software programming library in details.

3.1 Basic Functions

3.1.1 Absolute Value Function

Absolute value function is used to obtain elements from the source vector to take the absolute value, and sequentially write the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = (src[n] < 0) ? -src[n] : src[n];
```

The DSP software programming library supports absolute value functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each absolute value function in details.

riscv_dsp_abs_f32

Prototype:

```
void riscv_dsp_abs_f32 (float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_abs_q31**Prototype:**

```
void riscv_dsp_abs_q31 (q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0, 0x7FFFFFFF].
- If the input value is INT32_MIN (0x80000000), the output value is INT32_MAX (0x7FFFFFFF).

riscv_dsp_abs_q15**Prototype:**

```
void riscv_dsp_abs_q15 (q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0, 0x7FFF].
- If the input value is INT16_MIN (0x8000), the output value is INT16_MAX (0x7FFF).

riscv_dsp_abs_q7**Prototype:**

```
void riscv_dsp_abs_q7 (q7_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0, 0x7F].
- If the input value is `INT8_MIN` (0x80), the output value is `INT8_MAX` (0x7F).

3.1.2 Addition Functions

The addition function obtains two elements from two different source vectors, adds them together, and sequentially writes the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    dst[n] = src1[n] + src2[n];
```

The DSP software programming library supports addition functions for the following different data types including floating point, Q31, Q15, and Q7, etc. The following sections describe each addition function in details.

`riscv_dsp_add_f32`

Prototype:

```
void riscv_dsp_add_f32 (float32_t *src1, float32_t *src2, float32_t
*dst, uint32_t size)
```

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[out] *dst` pointer of the destination vector
- `[in] size` size of elements in a vector

Return Value:

None

`riscv_dsp_add_q31`

Prototype:

```
void riscv_dsp_add_q31 (q31_t *src1, q31_t *src2, q31_t *dst,
uint32_t size)
```

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[out] *dst` pointer of the destination vector
- `[in] size` number of elements in a vector

Return Value:

None

Note!

The addition results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_add_q15**Prototype:**

```
void riscv_dsp_add_q15 (q15_t *src1, q15_t *src2, q15_t *dst,  
uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The addition results will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_add_q7**Prototype:**

```
void riscv_dsp_add_q7 (q7_t *src1, q7_t *src2, q7_t *dst, uint32_t  
size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The addition results will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_add_u8_u16**Prototype:**

```
void riscv_dsp_add_u8_u16(uint8_t *src1, uint8_t *src2, uint16_t *dst,
```

uint32_t size)

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The addition result of the two uint8_t data will be saved in uint16_t format.

3.1.3 Division Functions

The division function obtains two elements from two different vectors/scalars, divides them, and sequentially writes the results into the destination vector/scalar, and the process is as follows:

```
for (n = 0; n < size; n++)
    dst[n] = src1[n] / src2[n];
```

The DSP software programming library supports division functions for the following different data types including floating point, Q31, and other types. The following sections describe each division function in details.

riscv_dsp_div_f32

Prototype:

```
void riscv_dsp_div_f32 (float32_t *src1, float32_t *src2, float32_t *dst,
uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector (dividend vector)
- [in] *src2 pointer of the second input vector (divisor vector)
- [out] *dst pointer of the destination vector (quotient vector)
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_div_q31

Prototype:

```
q31_t riscv_dsp_div_q31 (q31_t src1, q31_t src2)
```

Parameter:

- [in] `src1` dividend value
- [in] `src2` divisor value

Return Value:

Quotient value

Note!

- The absolute value of `src1` should be less than the absolute value of `src2`.
- This function deals with scalars, not vectors.

riscv_dsp_div_u64_u32**Prototype:**

`q31_t riscv_dsp_div_u64_u32 (uint64_t src1, uint32_t src2)`

Parameter:

- [in] `src1` dividend value
- [in] `src2` divisor value

Return Value:

Quotient value

Note!

- The value of `src1` shifted 32 bits to the right should be less than the value of `src2` shifted 32 bits to the right, otherwise the result is invalid.
- This function deals with scalars, not vectors.

riscv_dsp_div_s64_u32**Prototype:**

`q31_t riscv_dsp_div_s64_u32 (q63_t src1, uint32_t src2)`

Parameter:

- [in] `src1` dividend value
- [in] `src2` divisor value

Return Value:

Quotient value

Note!

- The absolute value of `src1` shifted 31 bits to the right should be less than the absolute value of `src2` shifted 31 bits to the right.
- This function deals with scalars, not vectors.

3.1.4 Dot Product Functions

The dot product function is used to calculate the dot product of two vectors and returns the result value, and the process is as follows:

```
for (output = 0, n = 0; n < size; n++)  
    output += src1[n] * src2[n];
```

The DSP software programming library supports dot product functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each dot product function in details.

riscv_dsp_dprod_f32

Prototype:

```
float32_t riscv_dsp_dprod_f32 (float32_t *src1, float32_t *src2,  
uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

riscv_dsp_dprod_q31

Prototype:

```
q63_t riscv_dsp_dprod_q31 (q31_t *src1, q31_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

The return value is in the format Q48 and of type q63_t.

riscv_dsp_dprod_q15

Prototype:

```
q63_t riscv_dsp_dprod_q15 (q15_t *src1, q15_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

The return value is in the format Q30 and of type `q63_t`.

riscv_dsp_dprod_q7**Prototype:**

`q31_t riscv_dsp_dprod_q7 (q7_t *src1, q7_t *src2, uint32_t size)`

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

The return value is in the format Q14 and of type `q31_t`.

riscv_dsp_dprod_u8**Prototype:**

`uint32_t riscv_dsp_dprod_u8 (uint8_t *src1, uint8_t *src2, uint32_t size)`

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

This function is used to multiply one `uint8_t` data and another `uint8_t` data, and then accumulate the result to a `uint32_t` accumulator.

riscv_dsp_dprod_u8xq15**Prototype:**

```
q31_t riscv_dsp_dprod_u8xq15 (uint8_t *src1, q15_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

This function is used to multiply one `uint8_t` data and another `q15_t` data, and then accumulate the result to a `q31_t` accumulator.

riscv_dsp_dprod_q7xq15**Prototype:**

```
q31_t riscv_dsp_dprod_q7xq15 (q7_t *src1, q15_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size number of elements in a vector

Return Value:

Dot product of the two input vectors

Note!

This function is used to multiply one `q7_t` data and another `q15_t` data, and then accumulate the result to a `q31_t` accumulator.

3.1.5 Multiplication Functions

The multiplication function obtains two elements from two different source vectors, multiplies them together, and sequentially writes the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = src1[n] * src2[n];
```

The DSP software programming library supports multiplication

functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each multiplication function in details.

riscv_dsp_mul_f32

Prototype:

```
void riscv_dsp_mul_f32 (float32_t *src1, float32_t *src2, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_mul_q31

Prototype:

```
void riscv_dsp_mul_q31 (q31_t *src1, q31_t *src2, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The multiplication results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_mul_q15

Prototype:

```
void riscv_dsp_mul_q15 (q15_t *src1, q15_t *src2, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector

- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The multiplication results will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_mul_q7**Prototype:**

```
void riscv_dsp_mul_q7 (q7_t *src1, q7_t *src2, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The multiplication results will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_mul_u8_u16**Prototype:**

```
void riscv_dsp_mul_u8_u16 (uint8_t *src1, uint8_t *src2, uint16_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the destination vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The multiplication result is saved as uint16_t *dst.

3.1.6 Negation Functions

The negation function is used to negate elements from the source vector and sequentially write the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = -src[n];
```

The DSP software programming library supports negation functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each negation function in details.

riscv_dsp_neg_f32

Prototype:

```
void riscv_dsp_neg_f32 (float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_neg_q31

Prototype:

```
void riscv_dsp_neg_q31 (q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0x80000001, 0x7FFFFFFF].
- If the input value is INT32_MIN (0x80000000), the output value is INT32_MAX (0x7FFFFFFF).

riscv_dsp_neg_q15

Prototype:

```
void riscv_dsp_neg_q15 (q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0x8001, 0x7FFF].
- If the input value is INT16_MIN (0x8000), the output value is INT16_MAX (0x7FFF).

```
riscv_dsp_neg_q7
```

Prototype:

```
void riscv_dsp_neg_q7 (q7_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- The output range is [0x81, 0x7F].
- If the input value is INT8_MIN (0x80), the output value is INT8_MAX (0x7F).

3.1.7 Offset Functions

The offset function is used to shift each element from the source vector by a constant value and sequentially write the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = src[n] + offset;
```

The DSP software programming library supports offset functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each offset function in details.

riscv_dsp_offset_f32**Prototype:**

```
void riscv_dsp_offset_f32 (float32_t *src, float32_t offset, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] offset constant offset value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_offset_q31**Prototype:**

```
void riscv_dsp_offset_q31 (q31_t *src, q31_t offset, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] offset constant offset value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The result value will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_offset_q15**Prototype:**

```
void riscv_dsp_offset_q15 (q15_t *src, q15_t offset, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] offset constant offset value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The result value will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_offset_q7**Prototype:**

```
void riscv_dsp_offset_q7 (q7_t *src, q7_t offset, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] offset constant offset value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The result value will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_offset_u8**Prototype:**

```
void riscv_dsp_offset_u8 (uint8_t *src, q7_t offset, uint8_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] offset constant offset value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The result value will be saturated to the uint8_t range [0x00, 0xFF].

3.1.8 Scaling Functions

The scaling function is used to multiply each element from the source

vector by a constant scaling value and sequentially write the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    dst[n] = src[n] * scaling_value;
```

When expressed in fractional notation, the parameter `shift` is introduced to adjust the range of result values. For detailed information about this parameter, see the fractional data type functions in [riscv_dsp_scale_q31](#), [riscv_dsp_scale_q15](#), [riscv_dsp_scale_q7](#), [riscv_dsp_scale_u8](#).

The DSP software programming library supports scaling functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each scaling function in details.

riscv_dsp_scale_f32

Prototype:

```
void riscv_dsp_scale_f32 (float32_t *src, float32_t scale, float32_t
*dst, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[in] scale` constant scaling value
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Return Value:

None

Example:

```
#define size 2
float32_t src[size] = {0.8167, 0.5};
float32_t dst[size] = {0};
float32_t scale = 0.4;
riscv_dsp_scale_f32(src, scale, dst, size);
```

This example also applies to scaling functions of Q31, Q15 or Q7.

riscv_dsp_scale_q31

Prototype:

```
void riscv_dsp_scale_q31 (q31_t *src, q31_t scalefract, int8_t shift,
q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] scalefract constant fractional scaling value
- [in] shift bits for result adjustment (Please refer to the note below)
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- Due to the fractional representation, the result should be within the Q31 type range, and you can use the input parameter shift to adjust the result value. In this function, the range of shift is from 0 to 31.
- This function is roughly realized as follows:

$$\text{dst}[n] = (\text{src}[n] * \text{scalefract}) \gg (31 - \text{shift})$$

Where $0 \leq n < \text{size}$ and $0 \leq \text{shift} \leq 31$.

riscv_dsp_scale_q15**Prototype:**

```
void riscv_dsp_scale_q15 (q15_t *src, q15_t scalefract, int8_t shift,
q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] scalefract constant fractional scaling value
- [in] shift bits for result adjustment (Please refer to the note below)
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- Due to the fractional representation, the result should be within the Q15 type range, and you can use the input parameter shift to adjust the result value. In this function, the range of shift is from 0 to 15.
- This function is roughly realized as follows:

$$\text{dst}[n] = (\text{src}[n] * \text{scalefract}) \gg (15 - \text{shift})$$

Where $0 \leq n < \text{size}$ and $0 \leq \text{shift} \leq 15$.

riscv_dsp_scale_q7**Prototype:**

```
void riscv_dsp_scale_q7 (q7_t *src, q7_t scalefract, int8_t shift, q7_t *dst, uint32_t size)
```

Parameter:

- [in] `*src` pointer of the input vector
- [in] `scalefract` constant fractional scaling value
- [in] `shift` bits for result adjustment (Please refer to the note below)
- [out] `*dst` pointer of the output vector
- [in] `size` number of elements in a vector

Return Value:

None

Note!

- Due to the fractional representation, the result should be within the Q7 type range, and you can use the input parameter `shift` to adjust the result value. In this function, the range of `shift` is from 0 to 7.
- This function is roughly realized as follows:

```
dst[n] = (src[n] * scalefract) >> (7 - shift)
```

Where $0 \leq n < \text{size}$ and $0 \leq \text{shift} \leq 7$.

riscv_dsp_scale_u8**Prototype:**

```
void riscv_dsp_scale_u8 (uint8_t *src, q7_t scalefract, int8_t shift, uint8_t *dst, uint32_t size)
```

Parameter:

- [in] `*src` pointer of the input vector
- [in] `scalefract` constant fractional scaling value
- [in] `shift` bits for result adjustment (Please refer to the note below.)
- [out] `*dst` pointer of the output vector
- [in] `size` number of elements in a vector

Return Value:

None

Note!

- Due to the data type of the destination vector being `uint8_t`, the scaled result will be saturated to the `uint8_t` range [0x00, 0xFF]. You can use the input parameter `shift` to

adjust the result value. In this function, the range of `shift` is from 0 to 7.

- This function is roughly realized as follows:

```
dst[n] = (src[n] * scalefract) >> (7 - shift)
```

Where $0 \leq n < \text{size}$ and $0 \leq \text{shift} \leq 7$.

3.1.9 Shift Functions

The shift function is used to shift each element from the source vector by a constant value, and sequentially write the results into the destination vector. The direction of the element shift (left or right) is determined by the positive/negative value of the shift, and the process is as follows:

```
for (n = 0; n < size; n++)
{
    if (shift < 0)
        dst[n] = src[n] >> (-shift);
    else
        dst[n] = src[n] << (shift);
}
```

The DSP software programming library supports shift functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each shift function in details.

riscv_dsp_shift_q31

Prototype:

```
void riscv_dsp_shift_q31 (q31_t *src, int8_t shift, q31_t *dst, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[in] shift` signed shift value
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Return Value:

None

Note!

- For performance considerations, the value of the shift should not exceed 32, as values greater than this may result in incorrect results.
- The result value will be saturated to the Q31 range `[0x80000000, 0x7FFFFFFF]`.

Example:

```
#define size 1024
q31_t src[size] = {...};
q31_t dst[size];
q31_t shift = 1;
riscv_dsp_shift_q31(src, shift, dst, size);
```

This example also applies to shift functions of Q15 or Q7.

riscv_dsp_shift_q15**Prototype:**

```
void riscv_dsp_shift_q15 (q15_t *src, int8_t shift, q15_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] shift signed shift value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- For performance considerations, the value of the shift should not exceed 16, as values greater than this may result in incorrect results.
- The result value will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_shift_q7**Prototype:**

```
void riscv_dsp_shift_q7 (q7_t *src, int8_t shift, q7_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] shift signed shift value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- For performance considerations, the value of the shift should not exceed 24, as values greater than this may result in incorrect results.
- The result value will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_shift_u8**Prototype:**

```
void riscv_dsp_shift_u8 (uint8_t *src, int8_t shift, uint8_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] shift signed shift value
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- For performance considerations, the value of the shift should not exceed 24, as values greater than this may result in incorrect results.
- The result value will be saturated to the uint8_t range [0x00, 0xFF].

3.1.10 Subtraction Functions

The subtraction function obtains two elements from two different source vectors, subtracts them, and sequentially writes the results into the destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = src1[n] - src2[n];
```

The DSP software programming library supports subtraction functions for the following different data types including floating point, Q31, Q15, Q7, and other types. The following sections describe each subtraction function in details.

riscv_dsp_sub_f32**Prototype:**

```
void riscv_dsp_sub_f32 (float32_t *src1, float32_t *src2, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_sub_q31**Prototype:**

```
void riscv_dsp_sub_q31 (q31_t *src1, q31_t *src2, q31_t *dst,  
uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_sub_q15**Prototype:**

```
void riscv_dsp_sub_q15 (q15_t *src1, q15_t *src2, q15_t *dst,  
uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The results will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_sub_q7**Prototype:**

```
void riscv_dsp_sub_q7 (q7_t *src1, q7_t *src2, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The results will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_sub_u8_q7**Prototype:**

```
void riscv_dsp_sub_u8_q7 (uint8_t *src1, uint8_t *src2, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The results will be saturated to the Q7 range [0x80, 0x7F].

3.2 Complex Functions

Complex functions provide calculations for complex vectors. In the DSP software programming function library, complex vector elements should be interleaved in real and imaginary parts, meaning the memory layout appears as [real, imaginary, real, imaginary, ...].

3.2.1 Complex Conjugate Functions

Complex conjugate functions calculate conjugation values of complex numbers from a source vector and write the results into a destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[2 * n] = src[2 * n];  
    dst[2 * n + 1] = -src[2 * n + 1];
```

The DSP software programming library supports complex conjugate functions for the following different data types including floating point, Q31, and Q15. The following sections describe each complex conjugate function in details.

riscv_dsp_cconj_f32

Prototype:

```
void riscv_dsp_cconj_f32 (const float32_t *src, float32_t *dst, uint32_t  
size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output complex vector
- [in] size number of complex elements in a vector

Return Value:

None

Example:

With a set of complex vector containing three complex elements like $\{3 + 4i, 4 - 7i, -3 + 5i\}$, the complex conjugation is as follows:

//Complex vector is arranged with real and imaginary parts interleaved.

```
#define size 3  
float32_t src[2 * size] = {3.0, 4.0, 4.0, -7.0, -3.0, 5.0};  
float32_t dst[2 * size];  
riscv_dsp_cconj_f32(src, dst, size);
```

This example also applies to conjugate complex functions of Q31 or Q15.

riscv_dsp_cconj_q31

Prototype:

```
void riscv_dsp_cconj_q31 (const q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output complex vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

- The real and imaginary parts in complex vectors are both in the Q31 range.
- When negating the input imaginary part where the value is INT32_MIN (0x80000000), the output imaginary part will be INT32_MAX (0x7FFFFFFF) after complex conjugation.

riscv_dsp_cconj_q15**Prototype:**

```
void riscv_dsp_cconj_q15 (const q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output complex vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

- The real and imaginary parts in complex vectors are both in the Q15 range.
- When negating the input imaginary part where the value is INT16_MIN (0x8000), the output imaginary part will be INT16_MAX (0x7FFF) after complex conjugation.

3.2.2 Complex Dot Product Functions

Complex dot product functions calculate the dot product of two complex vectors and write the results into a destination vector.

There are two kinds of complex dot product functions in DSP programming function library.

The first kind includes riscv_dsp_cdprod_f32, riscv_dsp_cdprod_q31, and riscv_dsp_cdprod_q15, and the process is as follows:

```
for (n = 0; n < size; n++)
    a = src1[2 * n];
```

```

b = src1[2 * n + 1];
c = src2[2 * n];
d = src2[2 * n + 1];
dst[2 * n] = a * c + b * d;
dst[2 * n + 1] = a * d - b * c;

```

The second of complex dot product functions, named as complex dot product type 2 functions, includes `riscv_dsp_cdprod_typ2_f32`, `riscv_dsp_cdprod_typ2_q31`, and `riscv_dsp_cdprod_typ2_q15`, and the process is as follows:

```

for (n = 0; n < size; n++)
    a = src1[2 * n];
    b = src1[2 * n + 1];
    c = src2[2 * n];
    d = src2[2 * n + 1];
    real_sum += a * c - b * d;
    image_sum += a * d + b * c;

```

The following sections describe each complex dot product function in details.

`riscv_dsp_cdprod_f32`

Prototype:

```
void riscv_dsp_cdprod_f32 (const float32_t *src1, const float32_t
*src2, uint32_t size, float32_t *dst)
```

Parameter:

- `[in] *src1` pointer of the first input complex vector
- `[in] *src2` pointer of the second input complex vector
- `[in] size` number of complex elements in a vector
- `[out] *dst` pointer of the output complex vector

Return Value:

None

Example:

With two sets of complex vector each containing three complex numbers like $\{3 + 4i, 4 - 7i, -3 + 5i\}$ and $\{1 - 2i, 5 - 1i, -4 + 3i\}$, the complex dot production is as follows:

```
#define size 3
```

```
float32_t src1[2*size] = {3, 4, 4, -7, -3, 5};  
float32_t src2[2*size] = {1, -2, 5, -1, -4, 3};  
float32_t dst[2*size];  
riscv_dsp_cdprod_f32(src1, src2, size, dst);
```

This example also applies to complex dot product functions of Q31 or Q15.

riscv_dsp_cdprod_q31

Prototype:

```
void riscv_dsp_cdprod_q31 (const q31_t *src1, const q31_t *src2,  
uint32_t size, q31_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input complex vector
- [in] *src2 pointer of the second input complex vector
- [in] size number of complex elements in a vector
- [out] *dst pointer of the output complex vector

Return Value:

None

Note!

The values written into the destination vector is in Q29 format.

riscv_dsp_cdprod_q15

Prototype:

```
void riscv_dsp_cdprod_q15 (const q15_t *src1, const q15_t *src2,  
uint32_t size, q15_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input complex vector
- [in] *src2 pointer of the second input complex vector
- [in] size number of complex elements in a vector
- [out] *dst pointer of the output complex vector

Return Value:

None

Note!

The values written into the destination vector is in Q13 format.

riscv_dsp_cdprod_typ2_f32**Prototype:**

```
void riscv_dsp_cdprod_typ2_f32 (const float32_t *src1, const  
float32_t *src2, uint32_t size, float32_t *rout, float32_t *iout)
```

Parameter:

- [in] **src1* pointer of the first input complex vector
- [in] **src2* pointer of the second input complex vector
- [in] *size* number of complex elements in a vector
- [out] **rout* pointer of the real output
- [out] **iout* pointer of the image output

Return Value:

None

Example:

With two sets of complex vector each containing three complex numbers like $3 + 4i$, $4 - 7i$, $-3 + 5i$ and $\{1 - 2i, 5 - 1i, -4 + 3i$, the complex dot production of this function is as follows:

```
#define size 3  
float32_t src1[2*size] = {3, 4, 4, -7, -3, 5};  
float32_t src2[2*size] = {1, -2, 5, -1, -4, 3};  
float32_t real_out, image_out;  
riscv_dsp_cdprod_typ2_f32(src1, src2, size, &real_out, &image_out);
```

This example also applies to complex dot product type 2 functions of Q31 or Q15.

riscv_dsp_cdprod_typ2_q31**Prototype:**

```
void riscv_dsp_cprod_typ2_q31 (const q31_t *src1, const q31_t *src2,  
uint32_t size, q63_t *rout, q63_t *iout)
```

Parameter:

- [in] **src1* pointer of the first input complex vector
- [in] **src2* pointer of the second input complex vector
- [in] *size* number of complex elements in a vector
- [out] **rout* pointer of the real output
- [out] **iout* pointer of the image output

Return Value:

None

Note!

The values written into the two output variables are in Q48 format.

riscv_dsp_cdprod_typ2_q15

Prototype:

```
void riscv_dsp_cprod_typ2_q15 (const q15_t *src1, const q15_t *src2,
uint32_t size, q31_t *rout, q31_t *iout)
```

Parameter:

- [in] *src1 pointer of the first input complex vector
- [in] *src2 pointer of the second input complex vector
- [in] size number of complex elements in a vector
- [out] *rout pointer of the real output
- [out] *iout pointer of the image output

Return Value:

None

Note!

The values written into the two output variables are in Q24 format.

3.2.3 Complex Magnitude Functions

Complex magnitude functions compute the magnitude of a complex vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    dst[n] = sqrt(src[2 * n]2 + src[2 * n + 1]2);
```

The DSP software programming library supports complex magnitude functions for the following different data types including floating point, Q31, and Q15. The following sections describe each complex magnitude function in details.

riscv_dsp_cmag_f32

Prototype:

```
void riscv_dsp_cmag_f32 (const float32_t *src, float32_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input complex vector

- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

riscv_dsp_cmag_q31**Prototype:**

```
void riscv_dsp_cmag_q31 (const q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q29 format.

riscv_dsp_cmag_q15**Prototype:**

```
void riscv_dsp_cmag_q15 (const q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q13 format.

3.2.4 Complex Magnitude-Squared Functions

Complex magnitude-squared functions compute the magnitude squared of complex numbers from a complex vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    dst[n] = src[2 * n]² + src[2 * n + 1]²
```

The DSP software programming library supports complex magnitude-squared functions for the following different data types including floating point, Q31, and Q15. The following sections describe each magnitude-squared function in details.

riscv_dsp_cmag_sqr_f32

Prototype:

```
void riscv_dsp_cmag_sqr_f32 (const float32_t *src, float32_t *dst,  
uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

riscv_dsp_cmag_sqr_q31

Prototype:

```
void riscv_dsp_cmag_sqr_q31 (const q31_t *src, q31_t *dst, uint32_t  
size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q29 format.

riscv_dsp_cmag_sqr_q15

Prototype:

```
void riscv_dsp_cmag_sqr_q15 (const q15_t *src, q15_t *dst, uint32_t  
size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [out] *dst pointer of the output vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q13 format.

3.2.5 Complex-Multiply-Complex Functions

Complex-multiply-complex functions compute the multiplication of two complex vectors and write the results into a destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    a = src1[2 * n];
    b = src1[2 * n + 1];
    c = src2[2 * n];
    d = src2[2 * n + 1];
    dst[2 * n] = a * c - b * d;
    dst[2 * n + 1] = a * d + b * c;
```

The DSP software programming library supports complex-multiply-complex functions for the following different data types including floating point, Q31, and Q15. The following sections describe each complex-multiply-complex function in details.

riscv_dsp_cmul_f32

Prototype:

```
void riscv_dsp_cmul_f32 (const float32_t *src1, const float32_t *src2,
float32_t *dst, uint32_t size)
```

Parameter:

- **[in] *src1** pointer of the first input complex vector
- **[in] *src2** pointer of the second input complex vector
- **[out] *dst** pointer of the output complex vector
- **[in] size** number of complex elements in a vector

Return Value:

None

Example:

With two sets of complex vector each containing three complex numbers like $[3 + 4i, 4 - 7i, -3 + 5i]$ and $[1 - 2i, 5 - 1i, -4 + 3i]$, the multiplication of the two complex vectors is as follows:

```
#define size 3
float32_t src1[2*size] = {3, 4, 4, -7, -3, 5};
float32_t src2[2*size] = {1, -2, 5, -1, -4, 3};
```

```
float32_t dst[2*size];  
riscv_dsp_cmul_f32(src1, src2, dst, size);
```

This example also applies to complex-multiply-complex functions of Q31 or Q15.

riscv_dsp_cmul_q31

Prototype:

```
void riscv_dsp_cmul_q31 (const q31_t *src1, const q31_t *src2, q31_t  
*dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input complex vector
- [in] *src2 pointer of the second input complex vector
- [out] *dst pointer of the output complex vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q29 format.

riscv_dsp_cmul_q15

Prototype:

```
void riscv_dsp_cmul_q15 (const q15_t *src1, const q15_t *src2, q15_t  
*dst, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input complex vector
- [in] *src2 pointer of the second input complex vector
- [out] *dst pointer of the output complex vector
- [in] size number of complex elements in a vector

Return Value:

None

Note!

The values written into the destination vector is in Q13 format.

3.2.6 Complex-Multiply-Real Functions

Complex-multiply-real functions compute the multiplication of a

complex vector by a real vector and write the results into a destination complex vector, and the process is as follows:

```
for (n = 0; n < size; n++)
    r = real[n];
    dst[2 * n] = src[2 * n] * r;
    dst[2 * n + 1] = src[2 * n + 1] * r;
```

The DSP software programming library supports complex-multiply-real functions for the following different data types including floating point, Q31, and Q15. The following sections describe each complex-multiply-real function in details.

riscv_dsp_cmul_real_f32

Prototype:

```
void riscv_dsp_cmul_real_f32 (const float32_t *src, const float32_t
*real, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [in] *real pointer of the input real vector
- [out] *dst pointer of the output complex vector
- [in] size number of elements in a vector

Return Value:

None

Example:

With a complex vector containing three elements like $[3 + 4i, 4 - 7i, -3 + 5i]$ and a real vector containing three elements like $[1, -3, 2]$, the multiplication of the two vectors is as follows:

```
#define size 3
float32_t src[2*size] = {3, 4, 4, -7, -3, 5};
float32_t real[size] = {1, -3, 2};
float32_t dst[2*size];
riscv_dsp_cmul_real_f32(src, real, dst, size);
```

This example also applies to complex-multiply-real functions of Q31 or Q15.

riscv_dsp_cmul_real_q31

Prototype:

```
void riscv_dsp_cmul_real_q31 (const q31_t *src, const q31_t *real,  
q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [in] *real pointer of the input real vector
- [out] *dst pointer of the output complex vector
- [in] size number of elements in a vector

Return Value:

None

Note!

The result value will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_cmul_real_q15

Prototype:

```
void riscv_dsp_cmul_real_q15 (const q15_t *src, const q15_t *real,  
q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input complex vector
- [in] *real pointer of the input real vector
- [out] *dst pointer of the output complex vector
- [in] size number of elements in a vector

Return Value:

None

Note!

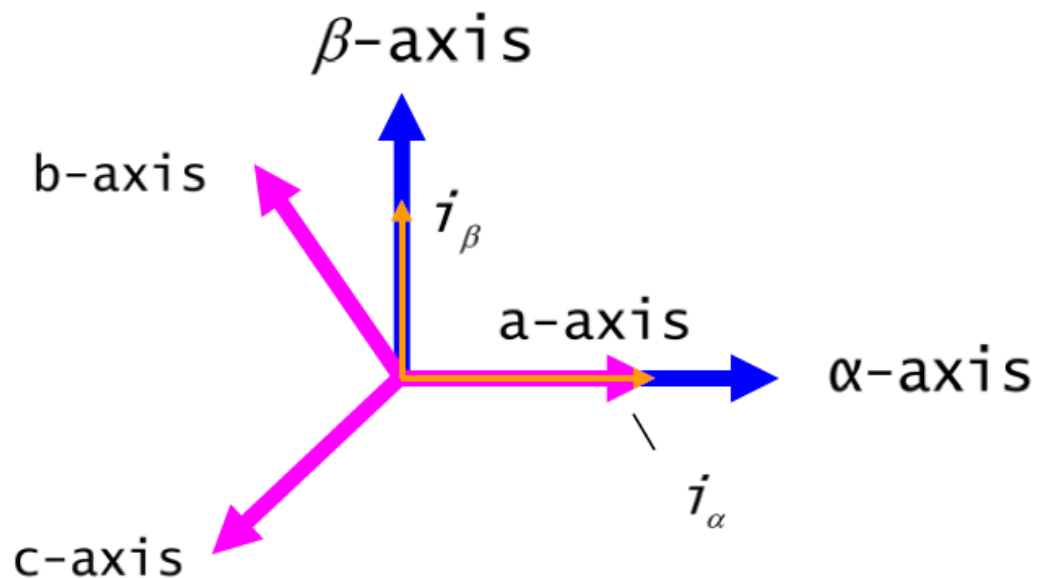
The result value will be saturated to the Q15 range [0x8000, 0x7FFF].

3.3 Controller Functions

3.3.1 Clarke Transform Functions

Clarke transform functions translate quantities from a three-phase system into a two-axis reference frame (alpha- and beta-axis), as shown in Figure 3-1.

Figure 3-1 Clarke Transform Diagram



The Clarke transform can be defined as follows:

let $I_a + I_b + I_c = 0$

such that,

$isqrt3 = 1 / \sqrt{3}$;

$i_\alpha = I_a$;

$i_\beta = isqrt3 * I_a + 2 * isqrt3 * I_b$

The DSP software programming library supports Clarke transform functions for the following different data types including floating point and Q31. The following sections describe each Clarke transform function in details.

riscv_dsp_clarke_f32

Prototype:

```
void riscv_dsp_clarke_f32 (float32_t a, float32_t b, float32_t *alpha,
float32_t *beta)
```

Parameter:

- [in] **a** quantity **a** of a three-phase system
- [in] **b** quantity **b** of a three-phase system
- [out] ***alpha** axis alpha of a two-phase reference frame
- [out] ***beta** axis beta of a two-phase reference frame

Return Value:

None

riscv_dsp_clarke_q31**Prototype:**

```
void riscv_dsp_clarke_q31 (q31_t a, q31_t b, q31_t *alpha, q31_t *beta)
```

Parameter:

- [in] **a** quantity **a** of a three-phase system
- [in] **b** quantity **b** of a three-phase system
- [out] ***alpha** axis alpha of a two-phase reference frame
- [out] ***beta** axis beta of a two-phase reference frame

Return Value:

None

3.3.2 Inverse Clarke Transform Functions

Inverse Clarke transform functions translate a two-axis reference frame (alpha-axis and beta-axis) back to a three-phase system **a**, **b** and **c**, and the process is as follows:

$$I_a = i_\alpha;$$

$$I_b = -0.5 * i_\alpha + 0.5 * \sqrt{3} * i_\beta;$$

The DSP software programming library supports inverse Clarke transform functions for the following different data types including floating point and Q31. The following sections describe each inverse Clarke transform function in details.

riscv_dsp_inv_clarke_f32**Prototype:**

```
void riscv_dsp_inv_clarke_f32 (float32_t alpha, float32_t beta, float32_t *a, float32_t *b)
```

Parameter:

- [in] **alpha** axis alpha of a two-phase reference frame

- [in] **beta** axis beta of a two-phase reference frame
- [out] ***a** quantity **a** of a three-phase system
- [out] ***b** quantity **b** of a three-phase system

Return Value:

None

riscv_dsp_inv_clarke_q31**Prototype:**

```
void riscv_dsp_inv_clarke_q31 (q31_t alpha, q31_t beta, q31_t *a,
q31_t *b)
```

Parameter:

- [in] **alpha** axis alpha of a two-phase reference frame
- [in] **beta** axis beta of a two-phase reference frame
- [out] ***a** quantity **a** of a three-phase system
- [out] ***b** quantity **b** of a three-phase system

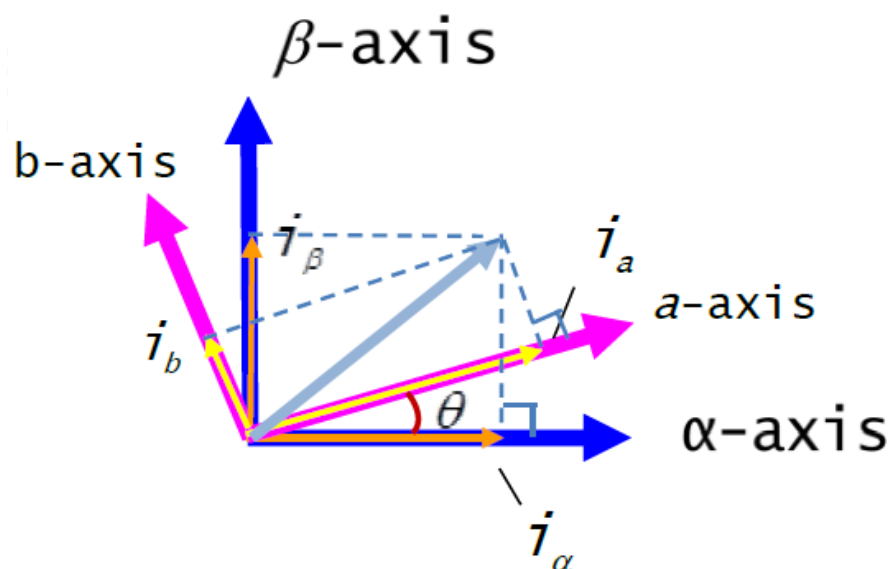
Return Value:

None

3.3.3 Park Transform Functions

Park transform functions translate an input two-phase coordinate (alpha-axis and beta-axis) into a rotor frame (a-axis and b-axis) by rotating an angle θ , as shown in Figure 3-2.

Figure 3-2 Park Transform Diagram



The process of Park transform is as follows:

$$\text{Cosval} = \cos(\theta); \text{Sinval} = \sin(\theta);$$

$$i_a = i_\alpha * \text{Cosval} + i_\beta * \text{Sinval};$$

$$i_b = -i_\alpha * \text{Sinval} + i_\beta * \text{Cosval}$$

The DSP software programming library supports Park transform functions for the following different data types including floating point and Q31. The following sections describe each Park transform function in details.

riscv_dsp_park_f32

Prototype:

```
void riscv_dsp_park_f32 (float32_t alpha, float32_t beta, float32_t *a,
float32_t *b, float32_t sin, float32_t cos)
```

Parameter:

- [in] **alpha** phase alpha of an input two-phase coordinate
- [in] **beta** phase beta of an input two-phase coordinate
- [out] ***a** axis **a** of an output rotor frame
- [out] ***b** axis **b** of an output rotor frame
- [in] **sin** sine value of the rotation angle θ
- [in] **cos** cosine value of the rotation angle θ

Return Value:

None

riscv_dsp_park_q31

Prototype:

```
void riscv_dsp_park_q31 (q31_t alpha, q31_t beta, q31_t *a, q31_t *b,
q31_t cos, q31_t *sin)
```

Parameter:

- [in] **alpha** phase alpha of an input two-phase coordinate
- [in] **beta** phase beta of an input two-phase coordinate
- [out] ***a** axis **a** of an output rotor frame
- [out] ***b** axis **b** of an output rotor frame
- [in] **sin** sine value of the rotation angle θ
- [in] **cos** cosine value of the rotation angle θ

Return Value:

None

3.3.4 Inverse Park Transform Functions

Inverse Park transform functions translate a rotor frame (a-axis and b-axis) back to a two-phase coordinate (alpha-axis and beta-axis) by rotating an angle θ , and the process is as follows:

$$i_{\alpha} = i_a * \text{Cosval} - i_b * \text{Sinval};$$

$$i_{\beta} = i_{\alpha} * \text{Sinval} + i_b * \text{Cosval}$$

The DSP software programming library supports Inverse Park transform functions for the following different data types including floating point and Q31. The following sections describe each Inverse Park transform function in details.

riscv_dsp_inv_park_f32

Prototype:

```
void riscv_dsp_inv_park_f32 (float32_t a, float32_t b, float32_t *alpha,
float32_t *beta, float32_t sin, float32_t cos)
```

Parameter:

- [in] **a** axis **a** of an input rotor frame
- [in] **b** axis **b** of an input rotor frame
- [out] ***alpha** phase alpha of an output two-phase coordinate
- [out] ***beta** phase beta of an output two-phase coordinate
- [in] **sin** sine value of the rotation angle θ
- [in] **cos** cosine value of the rotation angle θ

Return Value:

None

riscv_dsp_inv_park_q31

Prototype:

```
void riscv_dsp_inv_park_f32 (q31_t a, float32_t b, q31_t *alpha, q31_t
*beta, q31_t sin, q31_t cos)
```

Parameter:

- [in] **a** axis **a** of an input rotor frame
- [in] **b** axis **b** of an input rotor frame
- [out] ***alpha** phase alpha of an output two-phase coordinate
- [out] ***beta** phase beta of an output two-phase coordinate

- `[in] sin` sine value of the rotation angle θ
- `[in] cos` cosine value of the rotation angle θ

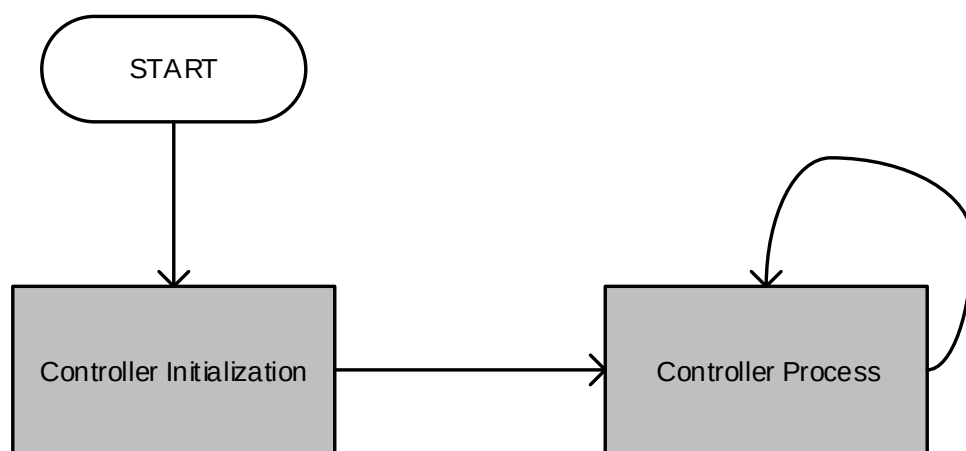
Return Value:

None

3.3.5 PID Functions

PID (Proportional-Integral-Derivative) controller functions provide a continuous-loop feedback mechanism to control output errors according to the inputs periodically. The state of PID controller is shown in Figure 3-3.

Figure 3-3 State of PID Controller



At the controller process when current data is input each time, the behavior of PID controller is as follows:

```

output = Kp * current_data + Ki * state0 + Kd * state1 + state2
state2 = output;
state1 = state0;
state0 = current_data;
  
```

The DSP software programming library supports PID functions for the following different data types including floating point and Q31. The following sections describe each PID function in details.

`riscv_dsp_init_pid_f32`

Prototype:

```

void riscv_dsp_init_pid_f32 (riscv_dsp_pid_f32_t *instance, int32_t
set)
  
```

Parameter:

- `riscv_dsp_pid_f32_t` instance structure of the floating-point PID

function, defined as follows:

```
typedef struct
{
    float32_t gain1;
    float32_t gain2;
    float32_t gain3;
    float32_t state[3];
    float32_t Kp;
    float32_t Ki;
    float32_t Kd;
} riscv_dsp_pid_f32_t;
```

Where

- gain1 is equal to $K_p + K_i + K_d$
- gain2 is equal to $-K_p - 2K_d$
- gain3 is equal to K_d
- state[3] state buffer
- Kp proportional value
- Ki integral value
- Kd derivative value
- [in] *instance pointer of the instance structure
- [in] set do nothing if its value is zero; otherwise, clear the state buffer.

Return Value:

None

Note!

- This function should be called to initialize the controller before the function `riscv_dsp_pid_f32` is executed.
- While parameters `gain1`, `gain2`, `gain3`, and `state[3]` are managed by the controller, you have to input `Kp`, `Ki`, and `Kd`.

Example:

```
riscv_dsp_pid_f32_t instance;
instance.Kp = 0.02F;
instance.Ki = 0.03F;
instance.Kd = 0.014F;
```

```

//initialize controller
riscv_dsp_init_pid_f32(&instance, 1);
//Begin controller
while (1)
{
f32_t output;
f32_t input = ... read data from input port ...
output = riscv_dsp_pid_f32(&instance, input);
...
}

```

This example also applies to PID functions of Q31 or Q15

riscv_dsp_pid_f32

Prototype:

```
float32_t riscv_dsp_pid_f32 (riscv_dsp_pid_f32_t *instance, float32_t src)
```

Parameter:

- `riscv_dsp_pid_f32_t` instance structure of the floating-point PID function, defined as follows:

```

typedef struct
{
    float32_t gain1;
    float32_t gain2;
    float32_t gain3;
    float32_t state[3];
    float32_t Kp;
    float32_t Ki;
    float32_t Kd;
} riscv_dsp_pid_f32_t;

```

Where

- gain1 is equal to $K_p + K_i + K_d$
- gain2 is equal to $-K_p - 2K_d$
- gain3 is equal to K_d
- state[3] state buffer

- K_p proportional value
- K_i integral value
- K_d derivative value
- [in] *instance pointer of the instance structure
- [in] src current data

Return Value:

Output of controller

Note!

The `riscv_dsp_init_pid_f32` function must be called to initialize the controller first, then the `riscv_dsp_pid_f32` function can be executed to do loop process of adjusting the controller's outputs with the current data. See Section `riscv_dsp_init_pid_f32` for an example.

riscv_dsp_init_pid_q31**Prototype:**

```
void riscv_dsp_init_pid_q31 (riscv_dsp_pid_q31_t *instance, int32_t set)
```

Parameter:

- `riscv_dsp_pid_q31_t` instance structure of the Q31 PID function, defined as follows:

```
typedef struct
{
    q31_t gain1;
    q31_t gain2;
    q31_t gain3;
    q31_t state[3];
    q31_t Kp;
    q31_t Ki;
    q31_t Kd;
} riscv_dsp_pid_q31_t;
```

Where

- gain1 is equal to $K_p + K_i + K_d$
- gain2 is equal to $-K_p - 2K_d$
- gain3 is equal to K_d
- state[3] state buffer

- **Kp** proportional value
- **Ki** integral value
- **Kd** derivative value
- **[in] *instance** pointer of the instance structure
- **[in] set** do nothing if its value is zero; otherwise, clear the state buffer.

Return Value:

None

Note!

- This function should be called to initialize the controller before the function `riscv_dsp_pid_q31` is executed.
- While parameters `gain1`, `gain2`, `gain3`, and `state[3]` are managed by the controller, you have to input `Kp`, `Ki`, and `Kd`.

riscv_dsp_pid_q31

Prototype:

`q31_t riscv_dsp_pid_q31 (riscv_dsp_pid_q31_t *instance, q31_t src)`

Parameter:

- `riscv_dsp_pid_q31_t` instance structure of the Q31 PID function, defined as follows:

```
typedef struct
{
    q31_t gain1;
    q31_t gain2;
    q31_t gain3;
    q31_t state[3];
    q31_t Kp;
    q31_t Ki;
    q31_t Kd;
} riscv_dsp_pid_q31_t;
```

Where

- `gain1` is equal to $Kp + Ki + Kd$
- `gain2` is equal to $-Kp - 2Kd$
- `gain3` is equal to `Kd`
- `state[3]` state buffer

- **Kp** proportional value
- **Ki** integral value
- **Kd** derivative value
- **[in] *instance** pointer of the instance structure
- **[in] src** current data

Return Value:

Output of controller

Note!

The `riscv_dsp_init_pid_q31` function must be called to initialize the controller first, then the `riscv_dsp_pid_q31` function can be executed to do loop process of adjusting the controller's outputs with the current data. See `riscv_dsp_init_pid_f32` for an example.

riscv_dsp_init_pid_q15**Prototype:**

```
void riscv_dsp_init_pid_q15 (riscv_dsp_pid_q15_t *instance, int32_t set)
```

Parameter:

- `riscv_dsp_pid_q15_t` instance structure of the Q15 PID function, defined as follows:

```
typedef struct
```

```
{  
    q15_t gain1;  
    q15_t gain2;  
    q15_t gain3;  
    q15_t state[3];  
    q15_t Kp;  
    q15_t Ki;  
    q15_t Kd;  
} riscv_dsp_pid_q15_t;
```

Where

- gain1 is equal to $Kp + Ki + Kd$
- gain2 is equal to $-Kp - 2Kd$

- `gain3` is equal to `Kd`
- `state[3]` state buffer
- `Kp` proportional value
- `Ki` integral value
- `Kd` derivative value
- `[in] *instance` pointer of the instance structure
- `[in] set` do nothing if its value is zero; otherwise, clear the state buffer.

Return Value:

None

Note!

- This function should be called to initialize the controller before the function `riscv_dsp_pid_q15` is executed.
- While parameters `gain1`, `gain2`, `gain3`, and `state[3]` are managed by the controller, you have to input `Kp`, `Ki`, and `Kd`.

riscv_dsp_pid_q15**Prototype:**

```
q15_t riscv_dsp_pid_q15 (riscv_dsp_pid_q15_t *instance, q15_t src)
```

Parameter:

- `riscv_dsp_pid_q15_t` instance structure of the Q15 PID function, defined as follows:

```
typedef struct
{
    q15_t gain1;
    q15_t gain2;
    q15_t gain3;
    q15_t state[3];
    q15_t Kp;
    q15_t Ki;
    q15_t Kd;
} riscv_dsp_pid_q15_t;
```

Where

- `gain1` is equal to $Kp + Ki + Kd$
- `gain2` is equal to $-Kp - 2Kd$

- `gain3` is equal to `Kd`
- `state[3]` state buffer
- `Kp` proportional value
- `Ki` integral value
- `Kd` derivative value
- `[in] *instance` pointer of the instance structure
- `[in] src` current data

Return Value:

Output of controller

Note!

The `riscv_dsp_init_pid_q15` function must be called to initialize the controller first, then the `riscv_dsp_pid_q15` function can be executed to do loop process of adjusting the controller's outputs with the current data. See `riscv_dsp_init_pid_f32` for an example.

3.4 Filtering Functions

3.4.1 Biquadratic Cascade Filter (Direct Form I) Functions

Biquadratic cascade filter functions provide biquadratic filters. DSP programming function library supports direct form I biquadratic cascade filters using the following equation:

$$\text{dst}[n] = b0 * \text{src}[n] + b1 * \text{src}[n-1] + b2 * \text{src}[n-2] + a1 * \text{dst}[n-1] + a2 * \text{dst}[n-2];$$

In the equation, 5 coefficients and 4 state variables are used per stage. The coefficients `b0`, `b1`, and `b2` determine zeros; the coefficients `a1` and `a2` determine the positions of poles.

DSP programming function library supports distinct direct form I biquadratic cascade filter functions and instance structures for floating-point, Q31, Q15 and also 32x64 Q31. Instance structures are used to store information like stages and coefficients, and save the state in each stage. Before you use biquadratic cascade filter functions, make sure you have initialized the corresponding instance structures first.

These functions and their instance structures are introduced in the following sections.

riscv_dsp_bq_df1_f32**Prototype:**

```
void riscv_dsp_bq_df1_f32 (const riscv_dsp_bq_df1_f32_t *instance,
```

```
float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_bq_df1_f32_t` instance structure of the floating-point biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    float32_t *state;
    float32_t *coeff;
} riscv_dsp_bq_df1_f32_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Example:

```
#define nstage 3
#define size 6

float32_t state[4 * nstage] = {0.0};

float32_t coeff[5 * nstage] = {0.40, 0.10, 0.24, -0.40, -0.34, 0.20, 0.06,
0.28, -0.04, -0.20, 0.08, 0.40, 0.60, -1.00, -0.14};

float32_t src[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};

float32_t dst[size];

riscv_dsp_bq_df1_f32_t instance = {nstage, state, coeff};

riscv_dsp_bq_df1_f32(&instance, src, dst, size);
```

`riscv_dsp_bq_df1_q31`

Prototype:

```
void riscv_dsp_bq_df1_q31 (const riscv_dsp_bq_df1_q31_t *instance,
```

q31_t *src, q31_t *dst, uint32_t size)

Parameter:

- `riscv_dsp_bq_df1_q31_t` instance structure of the Q31 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q31_t *state;
    q31_t *coeff;
    int8_t shift;
} riscv_dsp_bq_df1_q31_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 * nstage$
- `*coeff` pointer of the coefficient vector whose size is $5 * nstage$
- `shift` shift bits
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Example:

```
#define shift 0
#define nstage 3
#define size 6
q31_t state_q31[4 * nstage] = {0};
q31_t coeff_q31[5 * nstage], src_q31[size], dst_q31[size];
riscv_dsp_bq_df1_q31_t instance_q31 = {nstage, state_q31,
coeff_q31, shift};

float32_t coeff_f32[5 * nstage] = {0.40, 0.10, 0.24, -0.40, -0.34, 0.20,
0.06, 0.28, -0.04, -0.20, 0.08, 0.40, 0.60, -1.00, -0.14};

float32_t src_f32[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};
```

```

riscv_dsp_convert_f32_q31(coeff_f32, coeff_q31, 5 * nstage);
riscv_dsp_convert_f32_q31(src_f32, src_q31, size);
riscv_dsp_bq_df1_q31(&instance_q31, src_q31, dst_q31, size);

```

This Q31 biquadratic cascade filter function example uses the `riscv_dsp_convert_f32_q31` function to convert the F32 vector to Q31. This example also applies to the biquadratic cascade filter function of Q15.

`riscv_dsp_bq_df1_fast_q31`

Prototype:

```

void riscv_dsp_bq_df1_fast_q31 (const riscv_dsp_bq_df1_q31_t
*instance, q31_t *src, q31_t *dst, uint32_t size)

```

Parameter:

- `riscv_dsp_bq_df1_q31_t` instance structure of the Q31 biquadratic cascade filter, defined as follows:

```

typedef struct
{
    uint32_t nstage;
    q31_t *state;
    q31_t *coeff;
    int8_t shift;
} riscv_dsp_bq_df1_q31_t;

```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is `4*nstage`
- `*coeff` pointer of the coefficient vector whose size is `5*nstage`
- `shift` shift bits
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

`riscv_dsp_bq_df1_q15`

Prototype:

```
void riscv_dsp_bq_df1_q15 (const riscv_dsp_bq_df1_q15_t *instance,
q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_bq_df1_q15_t` instance structure of the Q15 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q15_t *state;
    q15_t *coeff;
    int8_t shift;
} riscv_dsp_bq_df1_q15_t
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$ or $6 \times \text{nstage}$ when using DSP extension. Please see the note below.
- `shift` shift bits
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Note!

The coefficient vector stores coefficients in the order {b0, b1, b2, a1, a2, ...}. The first 5 elements are for the first stage, the next 5 elements are for the next stage and so on. If using DSP extension, the order turns to be {b0, 0, b1, b2, a1, a2, ...} in which one zero (0) element is inserted between b0 and b1 for SIMD load/store instruction. In this case, the first 6 elements are for the first stage, the next 6 elements are for the next stage and so on.

riscv_dsp_bq_df1_fast_q15

Prototype:

```
void riscv_dsp_bq_df1_fast_q15 (const riscv_dsp_bq_df1_q15_t
*instance, q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_bq_df1_q15_t` instance structure of the Q15 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q15_t *state;
    q15_t *coeff;
    int8_t shift;
} riscv_dsp_bq_df1_q15_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$ or $6 \times \text{nstage}$ when using DSP extension. Please see the note below.
- `shift` shift bits
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Note!

The coefficient vector stores coefficients in the order {b0, b1, b2, a1, a2, ...}. The first 5 elements are for the first stage, the next 5 elements are for the next stage and so on. If using DSP extension, the order turns to be {b0, 0, b1, b2, a1, a2, ...} in which one zero (0) element is inserted between b0 and b1 for SIMD load/store instruction. In this case, the first 6 elements are for the first stage, the next 6 elements are for the next stage and so on.

`riscv_dsp_bq_df1_32x64_q31`

Prototype:

```
void riscv_dsp_bq_df1_32x64_q31 (const
riscv_dsp_bq_df1_32x64_q31_t *instance, q31_t *src, q31_t *dst, uint32_t
size)
```

Parameter:

- `riscv_dsp_bq_df1_32x64_q31_t` instance structure of the 32x64 Q31 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q63_t *state;
    q31_t *coeff;
    int8_t shift;
} riscv_dsp_bq_df1_32x64_q31_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$
- `shift` shift bits
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Note!

This function stores and updates intermediate results with a 64-bit state buffer that calculates accumulated values.

3.4.2 Biquadratic Cascade Filter (Direct Form II Transposed) Functions

Biquadratic cascade filter functions provide biquadratic filters. DSP programming function library supports direct form II transposed biquadratic cascade filters using the following equation:

```
for (n = 0; n < size; n++)
    out[n] = b0 * in[n] + d0;
    d0 = b1 * in[n] + a1 * out[n] + d1;
    d1 = b2 * in[n] + a2 * out[n];
```

In the equation, 5 coefficients (`b0`, `b1`, `b2`, `a1`, and `a2`) and 2 state variables (`d0` and `d1`) are used per stage. After the completion of each

stage, the 2 state variables `d0` and `d1` are saved into a state vector.

DSP programming function library also supports a direct form II transposed biquadratic cascade filter for stereo (2 channels) signals. Its equation is as below:

```
for (n = 0; n < size; n+=2)
    out[n] = b0 * in[n] + d00;
    out[n + 1] = b0 * in[n + 1] + d01;
    d00 = b1 * in[n] + a1 * out[n] + d10;
    d01 = b1 * in[n + 1] + a1 * out[n + 1] + d11;
    d10 = b2 * in[n] + a2 * out[n];
    d11 = b2 * in[n + 1] + a2 * out[n + 1];
```

In this equation, 5 coefficients (`b0`, `b1`, `b2`, `a1`, and `a2`) and 4 state variables (`d00`, `d01`, `d10`, and `d11`) are used per stage. After the completion of each stage, the 4 state variables are saved into a state vector.

DSP programming function library supports distinct direct form II transposed biquadratic cascade filter functions and instance structures only for floating-point data types, F32 and F64. Instance structures are used to store information like stages and coefficients, and save the state in each stage. Before you use biquadratic cascade filter functions, make sure you initialize the corresponding instance structures first.

These functions and their instance structures are introduced in the following sections.

`riscv_dsp_bq_df2T_f32`

Prototype:

```
void riscv_dsp_bq_df2T_f32 (const riscv_dsp_bq_df2T_f32_t
*instance, float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_bq_df2T_f32_t` instance structure of the F32 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    float32_t *state;
    float32_t *coeff;
} riscv_dsp_bq_df2T_f32_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $2 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

None

Example:

```
#define nstage 3
#define size 6
float32_t state[2 * nstage] = {0.0};
float32_t coeff[5 * nstage] = {0.40, 0.10, 0.24, -0.40, -0.34, 0.20, 0.06,
0.28, -0.04, -0.20, 0.08, 0.40, 0.60, -1.00, -0.14};
float32_t src[size] = {1, 0.5, 0.4, -0.1, -0.1, 0.3};
float32_t dst[size];
riscv_dsp_bq_df2T_f32_t inst = {nstage, state, coeff};
riscv_dsp_bq_df2T_f32(&inst, src, dst, size);
```

This example also applies to an F64 direct form II transposed biquadratic cascade filter function.

riscv_dsp_bq_df2T_f64

Prototype:

```
void riscv_dsp_bq_df2T_f64 (const riscv_dsp_bq_df2T_f64_t
*instance, float64_t *src, float64_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_bq_df2T_f64_t` instance structure of the F64 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    float64_t *state;
```

```
float64_t *coeff;
} riscv_dsp_bq_df2T_f64_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $2 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples in each stage

Return Value:

`riscv_dsp_bq_stereo_df2T_f32`

Prototype:

```
void riscv_dsp_bq_stereo_df2T_f32(const
riscv_dsp_bq_stereo_df2T_f32_t *instance, float32_t *src, float32_t *dst,
uint32_t size);
```

Parameter:

- `riscv_dsp_bq_stereo_df2T_f32_t` instance structure of the F32 biquadratic cascade filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    float32_t *state;
    float32_t *coeff;
} riscv_dsp_bq_stereo_df2T_f32_t;
```

Where

- `nstage` stages in the filter
- `*state` pointer of the state vector whose size is $4 \times \text{nstage}$
- `*coeff` pointer of the coefficient vector whose size is $5 \times \text{nstage}$
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector

- **[in] size** number of samples in each stage

Return Value:

None

Example:

```
#define nstage 3
#define size 1024

float32_t state[4 * nstage] = {0.0};
float32_t coeff[5 * nstage] = {0.40, 0.10, 0.24, -0.40, -0.34, 0.20, 0.06,
0.28, -0.04, -0.20, 0.08, 0.40, 0.60, -1.00, -0.14};

float32_t src[size * 2] = {...};
float32_t dst[size * 2];

riscv_dsp_bq_stereo_df2T_f32_t inst = {nstage, state, coeff};
riscv_dsp_bq_stereo_df2T_f32(&inst, src, dst, size);
```

3.4.3 Convolution Functions

Convolution functions combine two signals into one signal mathematically. The convolution equation can be presented as:

$$dst[n] = \sum_{k=0}^{len1} (src1[k] \times src2[n - k])$$

Where the size of `src1[n]` is `len1`, the size of `src2[n]` is `len2`, and the size of `dst[n]` is `len1 + len2 - 1`.

The DSP software programming library supports convolution functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each convolution function in details.

riscv_dsp_conv_f32

Prototype:

```
void riscv_dsp_conv_f32 (float32_t *src1, uint32_t len1, float32_t
*src2, uint32_t len2, float32_t *dst)
```

Parameter:

- **[in] *src1** pointer of the first input vector
- **[in] len1** length of the first input vector
- **[in] *src2** pointer of the second input vector
- **[in] len2** length of the second input vector

- [out] *dst pointer of the output vector

Return Value:

None

Example:

```
#define len1 6
#define len2 4
float32_t src1[len1] = {0.3, 0.2, -0.1, 0.2, 0.4, 0.2};
float32_t src2[len2] = {-0.1, 0.3, 0.2, -0.2};
uint32_t dst[len1 + len2 - 1];
riscv_dsp_conv_f32(src1, len1, src2, len2, dst);
```

This example also applies to convolution functions of Q31, Q15 or Q7.

riscv_dsp_conv_q31**Prototype:**

```
void riscv_dsp_conv_q31 (q31_t *src1, uint32_t len1, q31_t *src2,
uint32_t len2, q31_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

riscv_dsp_conv_q15**Prototype:**

```
void riscv_dsp_conv_q15 (q15_t *src1, uint32_t len1, q15_t *src2,
uint32_t len2, q15_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector

- [out] *dst pointer of the output vector

Return Value:

None

riscv_dsp_conv_q7**Prototype:**

```
void riscv_dsp_conv_q7 (q7_t *src1, uint32_t len1, q7_t *src2,  
uint32_t len2, q7_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

3.4.4 Partial Convolution Functions

With a starting index and a size specified, partial convolution functions do the convolution partially for two signals. The partial convolution result will be generated to the destination vector in the range [start_index, ..., start_index + size - 1]. Thus, if the partial convolution result is not within the range [0, ..., len1 + len2 - 2], a return value of -1 will be given for the argument error.

The DSP software programming library supports partial convolution functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each partial convolution function in details.

riscv_dsp_conv_partial_f32**Prototype:**

```
int32_t riscv_dsp_conv_partial_f32 (float32_t *src1, uint32_t len1,  
float32_t *src2, uint32_t len2, float32_t *dst, uint32_t startindex, uint32_t  
size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector

- [in] *scr2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector
- [in] startindex starting index of the partial convolution
- [in] size length of the partial convolution

Return Value:

0 success

-1 failure

Example:

```
#define len1 6
#define len2 4

float32_t src1[len1] = {0.3, 0.2, -0.1, 0.2, 0.4, 0.2};
float32_t src2[len2] = {-0.1, 0.3, 0.2, -0.2};
uint32_t dst[len1 + len2 - 1];

riscv_dsp_conv_partial_f32(src1, len1, src2, len2, dst, 3, 4);
```

This example also applies to partial convolution functions of Q31, Q15 or Q7.

riscv_dsp_conv_partial_q31**Prototype:**

```
int32_t riscv_dsp_conv_partial_q31 (q31_t *src1, uint32_t len1, q31_t
*src2, uint32_t len2, q31_t *dst, uint32_t startindex, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *scr2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector
- [in] startindex starting index of the partial convolution
- [in] size length of the partial convolution

Return Value:

0 success

-1 failure

riscv_dsp_conv_partial_q15**Prototype:**

```
int32_t riscv_dsp_conv_partial_q15(q15_t *src1, uint32_t len1, q15_t *src2, uint32_t len2, q15_t *dst, uint32_t startindex, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector
- [in] startindex starting index of the partial convolution
- [in] size length of the partial convolution

Return Value:

0 success
-1 failure

riscv_dsp_conv_partial_q7**Prototype:**

```
int32_t riscv_dsp_conv_partial_q7(q7_t *src1, uint32_t len1, q7_t *src2, uint32_t len2, q7_t *dst, uint32_t startindex, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector
- [in] startindex starting index of the partial convolution
- [in] size length of the partial convolution

Return Value:

0 success
-1 failure

3.4.5 Correlation Functions

Correlation functions use two signals to form another signal (also

called cross-correlation) mathematically. The mathematical operations of correlation functions are similar to those of convolution functions and can be presented as:

$$\text{dst}[n] = \sum_{k=0}^{\text{len1}} (\text{src}[k] \times \text{src2}[k - n])$$

Where src1 and src2 are input vectors, and their lengths are len1 and len2 separately; dst is the output vector and its size is 2 * max (len1, len2) - 1.

The DSP software programming library supports correlation functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each correlation function in details.

riscv_dsp_corr_f32

Prototype:

```
void riscv_dsp_corr_f32 (float32_t *src1, uint32_t len1, float32_t
*src2, uint32_t len2, float32_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

Example:

```
#define len1 7
#define len2 5
float32_t src1[len1] = {0.3, 0.2, -0.1, 0.2, 0.4, 0.2, -0.1};
float32_t src2[len2] = {-0.1, 0.3, 0.2, -0.2, -0.2};
float32_t dst[2 * len1 - 1];
riscv_dsp_conv_f32(src1, len1, src2, len2, dst);
```

This example also applies to correlation functions of Q31, Q15 or Q7.

riscv_dsp_corr_q31

Prototype:

```
void riscv_dsp_corr_q31 (q31_t *src1, uint32_t len1, q31_t *src2,  
uint32_t len2, q31_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

riscv_dsp_corr_q15

Prototype:

```
void riscv_dsp_corr_q15 (q15_t *src1, uint32_t len1, q15_t *src2,  
uint32_t len2, q15_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

riscv_dsp_corr_q7

Prototype:

```
void riscv_dsp_corr_q7 (q7_t *src1, uint32_t len1, q7_t *src2, uint32_t  
len2, q7_t *dst)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] len1 length of the first input vector
- [in] *src2 pointer of the second input vector
- [in] len2 length of the second input vector
- [out] *dst pointer of the output vector

Return Value:

None

3.4.6 Finite Impulse Response (FIR) Filter Functions

Finite impulse response (FIR) filter functions can be presented by the following equation:

$$dst[n] = \sum_{k=0}^{Size} (b[k] \times src[n - k])$$

Where $b[k]$ is the filter coefficient and $Size$ is the number of filter coefficients.

The DSP software programming library supports FIR filter functions for the following different data types including floating point, Q31, Q15, and Q7. The following sections describe each FIR filter function in details. Before you use FIR filter functions, make sure you initialize the corresponding instance structures first.

The following section will describe the FIR filter function in details.

riscv_dsp_fir_f32

Prototype:

```
void riscv_dsp_fir_f32 (const riscv_dsp_fir_f32_t *instance, float32_t
*src, float32_t *dst, uint32_t size)
```

Parameter:

- **riscv_dsp_fir_f32_t** instance structure of the floating-point FIR filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    float32_t *state;
    float32_t *coeff;
} riscv_dsp_fir_f32_t;
```

Where

- **coeff_size** number of coefficients
- ***state** pointer of the state vector whose size is **coeff_size + size - 1**
- ***coeff** pointer of the time reversed coefficient vector whose size is **coeff_size**
- **[in] *instance** pointer of the instance structure

- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Example:

```
#define coeff_size 4
#define size 6
float32_t state[coeff_size + size - 1];
float32_t coeff[coeff_size] = {0.3, -0.1, 0.4, 0.5};
riscv_dsp_fir_f32_t inst = {coeff_size, state, coeff};
float32_t src[size] = {0.1, -0.2, 0.2, 0.3, 0.4, 0.1};
float32_t dst[size];
riscv_dsp_fir_f32(&inst, src, dst, size);
```

This example also applies to FIR filter functions of Q31, fast_Q31, Q15, fast_Q15, or Q7.

riscv_dsp_fir_q31**Prototype:**

```
void riscv_dsp_fir_q31 (const riscv_dsp_fir_q31_t *instance, q31_t
*src, q31_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_fir_q31_t` instance structure of the Q31 FIR filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q31_t *state;
    q31_t *coeff;
} riscv_dsp_fir_q31_t;
```

Where

- `coeff_size` number of coefficients

- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q31 format and multiplications yield a Q2.62 result. These results are right-shifted by 31 bits, saturated to Q31 format and saved into buffer. Thus, in order to avoid overflow, the input signal must be scaled down by $\log_2(\text{coeff\_size})$ bits before this function is called. When compared with `riscv_dsp_fir_fast_q31`, this function delivers higher precision and smaller code size.

riscv_dsp_fir_fast_q31**Prototype:**

```
void riscv_dsp_fir_fast_q31 (const riscv_dsp_fir_q31_t *instance,
q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_fir_q31_t` instance structure of the Q31 FIR filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q31_t *state;
    q31_t *coeff;
} riscv_dsp_fir_q31_t;
```

Where

- `coeff_size` number of coefficients
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `[in] *instance` pointer of the instance structure

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q31 format and multiplications yield a Q1.31 result. These results are left-shifted by 1 bit and saved into buffer. Thus, in order to avoid overflow, the input signal must be scaled down by $\log_2(\text{coeff\_size})$ bits before this function is called. When compared with `riscv_dsp_fir_q31`, this function delivers higher performance with lower precision and larger code size.

riscv_dsp_fir_q15**Prototype:**

```
void riscv_dsp_fir_q15 (const riscv_dsp_fir_q15_t *instance, q15_t
*src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_fir_q15_t` instance structure of the Q15 FIR filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q15_t *state;
    q15_t *coeff;
} riscv_dsp_fir_q15_t;
```

Where

- `coeff_size` number of coefficients
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- [in] *instance pointer of the instance structure
- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q15 format and multiplications yield a Q30 result. The results are accumulated in a 64-bit accumulator in Q34.30 format and then truncated to Q34.15 format by discarding the low 15 bits. Lastly, the outputs are saturated to yield a result in Q1.15 format. When compared with [riscv_dsp_fir_fast_q15](#), this function delivers higher precision and smaller code size.

riscv_dsp_fir_fast_q15

Prototype:

```
void riscv_dsp_fir_fast_q15 (const riscv_dsp_fir_q15_t *instance,
q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [riscv_dsp_fir_q15_t](#) instance structure of the Q15 FIR filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q15_t *state;
    q15_t *coeff;
} riscv_dsp_fir_q15_t;
```

Where

- [coeff_size](#) number of coefficients
- [*state](#) pointer of the state vector whose size is [coeff_size + size - 1](#)
- [*coeff](#) pointer of the time reversed coefficient vector whose size is [coeff_size](#)
- [\[in\] *instance](#) pointer of the instance structure
- [\[in\] *src](#) pointer of the input vector
- [\[out\] *dst](#) pointer of the output vector
- [\[in\] size](#) number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q15 format and multiplications yield a Q30 result. The results are accumulated in a 32-bit accumulator in Q2.30 format and then saturated to yield a result in Q1.15 format. When compared with

`riscv_dsp_fir_q15`, this function delivers higher performance with lower precision and larger code size.

`riscv_dsp_fir_q7`

Prototype:

```
void riscv_dsp_fir_q7 (const riscv_dsp_fir_q7_t *instance, q7_t *src,
q7_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_fir_q7_t` instance structure of the Q7 FIR filter. defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q7_t *state;
    q7_t *coeff;
} riscv_dsp_fir_q7_t;
```

Where

- `coeff_size` number of coefficients
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`.
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Note!

Both inputs are in Q7 format and multiplications yield a Q14 result. The intermediate results are accumulated in a 32-bit accumulator in Q18.14 format. The result is then converted to Q18.7 format by discarding the low 7 bits and then saturated to Q1.7 format.

3.4.7 Decimation FIR Filter Functions

Decimation FIR filter functions implement FIR filters with decimation by an integer factor `M`. They can be presented by the following equation:

$$dst[n] = \sum_{k=0}^{Size} (b[k] \times src[n - k])$$

Where $b[k]$ is the filter coefficient, src is the input data of the length $Size$, and dst is the output data of the length $Size/M$. $Size$ must be a multiple of the decimation factor M so that the number of output samples can be an integer.

DSP programming function library supports distinct decimation FIR filter functions and instance structures for the following data types including floating-point, Q31 and Q15. Before you use decimation FIR filter functions, make sure you initialize the corresponding instance structure first.

These functions and their instance structures are introduced in the following subsections.

riscv_dsp_dcmfir_f32

Prototype:

```
void riscv_dsp_dcmfir_f32 (const riscv_dsp_dcmfir_f32_t *instance,
float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_dcmfir_f32_t` instance structure of the floating-point decimation FIR filter, defined as follows:

```
typedef struct
{
    uint32_t M;
    uint32_t coeff_size;
    float32_t *coeff;
    float32_t *state;
} riscv_dsp_dcmfir_f32_t;
```

Where

- M decimation factor
- `coeff_size` number of filter coefficients
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector

- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Example:

With the input size as 24, the decimation factor `M` as 4 and the filter coefficient size as 6, set the instance of the decimation FIR filter as:

```
#define size 24
#define M 4
#define coeff_size 6
float32_t coeff[coeff_size] = {...};
float32_t state[coeff_size + size - 1];
riscv_dsp_dcmfir_f32_t inst = {M, coeff_size, coeff, state};
float32_t src[size] = {0.1, -0.2, 0.2, 0.3, 0.4, 0.1, ...};
float32_t dst[size / M];
riscv_dsp_dcmfir_f32(&inst, src, dst, size);
```

This example also applies to decimation FIR filter functions of Q31 or Q15.

riscv_dsp_dcmfir_q31**Prototype:**

```
void riscv_dsp_dcmfir_q31 (const riscv_dsp_dcmfir_q31_t *instance,
q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_dcmfir_q31_t` instance structure of the Q31 decimation FIR filter, defined as follows:

```
typedef struct
{
    uint32_t M;
    uint32_t coeff_size;
    q31_t *coeff;
    q31_t *state;
} riscv_dsp_dcmfir_q31_t;
```

Where

- **M** decimation factor
- **coeff_size** number of filter coefficients
- ***coeff** pointer of the time reversed coefficient vector whose size is **coeff_size**
- ***state** pointer of the state vector whose size is **coeff_size + size - 1**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q31 format. A 64-bit accumulator is used to do multiply-accumulate operations with saturation to yield a Q1.62 result. After all operations are performed, the results are saturated to Q31 format.

riscv_dsp_dcmfir_fast_q31**Prototype:**

```
void riscv_dsp_dcmfir_fast_q31 (const riscv_dsp_dcmfir_q31_t
*instance, q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- **riscv_dsp_dcmfir_q31_t** instance structure of the Q31 decimation FIR filter, defined as follows:

```
typedef struct
{
    uint32_t M;
    uint32_t coeff_size;
    q31_t *coeff;
    q31_t *state;
} riscv_dsp_dcmfir_q31_t;
```

Where

- **M** decimation factor
- **coeff_size** number of filter coefficients
- ***coeff** pointer of the time reversed coefficient vector whose size is

`coeff_size`

- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Note!

- Both coefficients and state variables are represented in Q31 format. A 32-bit accumulator is used to do multiply-accumulate operations and truncated to Q2.30 result. To avoid overflows that distort the result, the input signal must be scaled down by $\log_2(\text{coeff\_size})$ bits. After all operations are performed, the results are saturated to Q1.31 format.
- When compared with `riscv_dsp_dcmfir_q31`, this function delivers higher performance with lower precision.

riscv_dsp_dcmfir_q15

Prototype:

```
void riscv_dsp_dcmfir_q15 (const riscv_dsp_dcmfir_q15_t *instance,
q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_dcmfir_q15_t` instance structure of the Q15 decimation FIR filter, defined as follows:

```
typedef struct
{
    uint32_t M;
    uint32_t coeff_size;
    q15_t *coeff;
    q15_t *state;
} riscv_dsp_dcmfir_q15_t;
```

Where

- `M` decimation factor
- `coeff_size` number of filter coefficients
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`

- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in Q15 format. A 64-bit accumulator is used to do multiply-accumulate operations with saturation to yield a Q33.30 result. After all operations are performed, the results are saturated to Q15 format.

riscv_dsp_dcmfir_fast_q15**Prototype:**

```
void riscv_dsp_dcmfir_fast_q15 (const riscv_dsp_dcmfir_q15_t
*instance, q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_dcmfir_q15_t` instance structure of the Q15 decimation FIR filter, defined as follows:

```
typedef struct
{
    uint32_t M;
    uint32_t coeff_size;
    q15_t *coeff;
    q15_t *state;
} riscv_dsp_dcmfir_q15_t;
```

Where

- `M` decimation factor
- `coeff_size` number of filter coefficients
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector

- `[in]` size number of samples

Return Value:

None

Note!

- Both coefficients and state variables are represented in Q15 format. A 32-bit accumulator is used to do multiply-accumulate operations with saturation to yield a Q2.30 result. If the accumulator result overflows, it wraps around and distorts the result. In order to avoid overflows completely, the input signal must be scaled down by $\log_2(\text{coeff\_size})$ bits. After all operations are performed, the Q2.30 accumulator is then truncated to Q2.15 format and saturated to yield the Q1.15 result.
- When compared with `riscv_dsp_dcmfir_q15`, this function delivers higher performance.

3.4.8 Lattice FIR Filter Functions

Lattice FIR filter functions can be represented by the following equations:

$$y_0[n] = u_0[n] = x[n]$$

$$y_z[n] = y_{z-1}[n] + k_z u_{z-1}[n-1]$$

$$u_z[n] = k_z y_{z-1}[n] + u_{z-1}[n-1]$$

Where x is the input, y is the output, u is the state of previous inputs, $0 \leq z < M$ and M is the stage number, and k_z is the reflection coefficients.

DSP programming function library supports distinct lattice FIR filter functions and instance structures for the following data types including floating-point, Q31 and Q15. Before you use lattice FIR filter functions, make sure you initialize the corresponding instance structure first.

These functions and their instance structures are introduced in the following subsections.

`riscv_dsp_lfir_f32`

Prototype:

```
void riscv_dsp_lfir_f32 (const riscv_dsp_lfir_f32_t *instance, float32_t
*src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_lfir_f32_t` instance structure of the floating-point lattice FIR filter, defined as follows:

```
typedef struct
{
```

```

uint32_t stage;
float32_t *state;
float32_t *coeff;
} riscv_dsp_lfir_f32_t;

```

Where

- **stage** number of filter stages
- ***state** pointer of the state vector whose size is **stage**
- ***coeff** pointer of the coefficient vector whose size is **stage**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples

Return Value:

None

Example:

With the filter length as 6 and the input size as 8, set the instance of the lattice FIR filter as:

```

#define M 6
#define size 8
float32_t state[M];
float32_t coeff[M] = {...};
float32_t src[size] = {...};
float32_t dst[size];
riscv_dsp_lfir_f32_t inst = {M, state, coeff};
riscv_dsp_lfir_f32 (&inst, src, dst, size);

```

This example also applies to lattice FIR filter functions of Q31 or Q15.

riscv_dsp_lfir_q31

Prototype:

```

void riscv_dsp_lfir_q31 (const riscv_dsp_lfir_q31_t *instance, q31_t
*src, q31_t *dst, uint32_t size)

```

Parameter:

- **riscv_dsp_lfir_q31_t** instance structure of the Q31 lattice FIR filter,

defined as follows:

```
typedef struct
{
    uint32_t stage;
    q31_t *state;
    q31_t *coeff;
} riscv_dsp_lfir_q31_t;
```

Where

- **stage** number of filter stages
- ***state** pointer of the state vector whose size is **stage**
- ***coeff** pointer of the coefficient vector whose size is **stage**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples

Return Value:

None

riscv_dsp_lfir_q15

Prototype:

```
void riscv_dsp_lfir_q15 (const riscv_dsp_lfir_q15_t *instance, q15_t
*src, q15_t *dst, uint32_t size)
```

Parameter:

- **riscv_dsp_lfir_q15_t** instance structure of the Q15 lattice FIR filter, defined as follows:

```
typedef struct
{
    uint32_t stage;
    q15_t *state;
    q15_t *coeff;
} riscv_dsp_lfir_q15_t;
```

Where

- **stage** number of filter stages

- `*state` pointer of the state vector whose size is `stage`
- `*coeff` pointer of the coefficient vector whose size is `stage`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

3.4.9 Lattice Infinite Impulse Response (IIR) Filter Functions

Lattice infinite impulse response (IIR) filter functions can be represented as follows:

$$y_0[n] = x[n]$$

$$y_{z-1}[n] = y_z[n] - k_z * u_{z-1}[n-1]$$

$$u_z[n] = k_z * y_{z-1}[n] + u_{z-1}[n-1]$$

$$\text{and } w[n] = v_N * u_N[n] + v_{N-1} * u_{N-1}[n] + \dots + v_0 * u_0[n]$$

Where N is the number of states and $z = N, N-1, \dots, 1$, coefficients $k = \{r_N, r_{N-1}, \dots, r_1\}$, v_N is the ladder coefficients and u is the state.

DSP programming function library supports distinct lattice IIR filter functions and instance structures for the following data types including floating-point, Q31 and Q15. Before you use lattice IIR filter functions, make sure you initialize the corresponding instance structure first.

These functions and their instance structures are introduced in the following subsections.

riscv_dsp_liir_f32**Prototype:**

```
void riscv_dsp_liir_f32 (const riscv_dsp_liir_f32_t *instance, float32_t
*src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_liir_f32_t` instance structure of the floating-point lattice IIR filter, defined as follows:

```
typedef struct
{
```

```

uint32_t nstage;
float32_t *state;
float32_t *rcoeff;
float32_t *lcoeff;
} riscv_dsp_liir_f32_t;

```

Where

- `nstage` number of filter stages
- `*state` pointer of the state vector whose size is `nstage + size`
- `*rcoeff` pointer of the time reversed reflection coefficient vector whose size is `nstage`
- `*lcoeff` pointer of the time reversed ladder coefficient vector whose size is `nstage + 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Example:

```

#define size 8
#define nstage 6
float32_t state[nstage + size];
float32_t rcoeff[nstage] = {...};
float32_t lcoeff[nstage + 1] = {...};
riscv_dsp_liir_f32_t inst = {nstage, state, rcoeff, lcoeff};
float32_t src[size] = {...};
float32_t dst[size];
riscv_dsp_liir_f32(&inst, src, dst, size);

```

This example also applies to lattice IIR filter functions of Q31, fast_Q31, Q15, or fast_Q15.

riscv_dsp_liir_q31

Prototype:

```
void riscv_dsp_liir_q31 (const riscv_dsp_liir_q31_t *instance, q31_t
*src, q31_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_liir_q31_t` instance structure of the Q31 lattice IIR filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q31_t *state;
    q31_t *rcoeff;
    q31_t *lcoeff;
} riscv_dsp_liir_q31_t;
```

Where

- `nstage` number of filter stages
- `*state` pointer of the state vector whose size is `nstage + size`
- `*rcoeff` pointer of the time reversed reflection coefficient vector whose size is `nstage`
- `*lcoeff` pointer of the time reversed ladder coefficient vector whose size is `nstage + 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

riscv_dsp_liir_fast_q31

Prototype:

```
void riscv_dsp_liir_fast_q31 (const riscv_dsp_liir_q31_t *instance,
q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_liir_q31_t` instance structure of the Q31 lattice IIR filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
    q31_t *state;
    q31_t *rcoeff;
    q31_t *lcoeff;
} riscv_dsp_liir_q31_t;
```

Where

- **nstage** number of filter stages
- ***state** pointer of the state vector whose size is **nstage + size**
- ***rcoeff** pointer of the time reversed reflection coefficient vector whose size is **nstage**
- ***lcoeff** pointer of the time reversed ladder coefficient vector whose size is **nstage + 1**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples whose size must be a multiple of 2

Return Value:

None

Note!

In order to avoid overflow, the reflection coefficient must be scaled down by 2.

riscv_dsp_liir_q15

Prototype:

```
void riscv_dsp_liir_q15 (const riscv_dsp_liir_q15_t *instance, q15_t
*src, q15_t *dst, uint32_t size)
```

Parameter:

- **riscv_dsp_liir_q15_t** instance structure of the Q15 lattice IIR filter, defined as follows:

```
typedef struct
{
    uint32_t nstage;
```

```

    q15_t *state;
    q15_t *rcoeff;
    q15_t *lcoeff;
} riscv_dsp_liir_q15_t;

```

Where

- **nstage** number of filter stages
- ***state** pointer of the state vector whose size is **nstage + size**
- ***rcoeff** pointer of the time reversed reflection coefficient vector whose size is **nstage**
- ***lcoeff** pointer of the time reversed ladder coefficient vector whose size is **nstage + 1**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples

Return Value:

None

riscv_dsp_liir_fast_q15

Prototype:

```

void riscv_dsp_liir_fast_q15 (const riscv_dsp_liir_q15_t *instance,
q15_t *src, q15_t *dst, uint32_t size)

```

Parameter:

- **riscv_dsp_liir_q15_t** instance structure of the Q15 lattice IIR filter, defined as follows:

```

typedef struct
{
    uint32_t nstage;
    q15_t *state;
    q15_t *rcoeff;
    q15_t *lcoeff;
} riscv_dsp_liir_q15_t;

```

Where

- `nstage` number of filter stages
- `*state` pointer of the state vector whose size is `nstage + size`
- `*rcoeff` pointer of the time reversed reflection coefficient vector whose size is `nstage`
- `*lcoeff` pointer of the time reversed ladder coefficient vector whose size is `nstage + 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples whose size must be a multiple of 2

Return Value:

None

3.4.10 Least Mean Square (LMS) Filter Functions

Least mean square (LMS) filter functions are adaptive filters which perform the following equations to update coefficients and get the minimum mean square error:

$$y[n] = h[0] * x[n] + h[1] * x[n - 1] + \dots + h[L - 1] * x[n - L + 1];$$

$$e[n] = d[n] - y[n];$$

$$h[k] = h[k - 1] + \mu * e[n] * x[n - k];$$

Where

- `y` is the output vector.
- `h` is the coefficients vector.
- `x` is the input vector.
- `L` is the filter order.
- `e` is the error vector.
- `d` is the desired vector.
- `μ` is the step size which can adjust coefficients.

DSP programming function library supports distinct least mean square filter functions and instance structures for the following data types including floating-point, Q31 and Q15. Before you use LMS filter functions, make sure you initialize the corresponding instance structures first.

The following section will describe the LMS filter function in details.

riscv_dsp_lms_f32**Prototype:**

```
void riscv_dsp_lms_f32 (const riscv_dsp_lms_f32_t *instance,  
float32_t *src, float32_t *ref, float32_t *dst, float32_t *err, uint32_t size)
```

Parameter:

- `riscv_dsp_lms_f32_t` instance structure of the floating-point LMS filter, defined as follows:

```
typedef struct  
{  
    uint32_t coeff_size;  
    float32_t *state;  
    float32_t *coeff;  
    float32_t mu;  
} riscv_dsp_lms_f32_t;
```

Where

- `coeff_size` number of filter coefficients
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `mu` step size which can adjust coefficients
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[in] *ref` pointer of the desired vector
- `[out] *dst` pointer of the output vector
- `[out] *err` pointer of the error vector
- `[in] size` number of samples

Return Value:

None

Example:

```
#define coeff_size 5  
#define size 6
```

```

#define mu 0.05
float32_t state[coeff_size + size - 1] = {0.0};
float32_t coeff[coeff_size] = {0.40, 0.10, 0.24, -0.40, -0.34};
float32_t src[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};
float32_t ref[size] = {0.1, 0.2, -0.1, -0.02, -0.1, 0.2};
float32_t err[size];
float32_t dst[size];
riscv_dsp_lms_f32_t instance = {coeff_size, state, coeff, mu};
riscv_dsp_lms_f32(&instance, src, ref, dst, err, size);

```

This example also applies to LMS filter functions of Q31 and Q15.

riscv_dsp_lms_q31

Prototype:

```

void riscv_dsp_lms_q31 (const riscv_dsp_lms_q31_t *instance, q31_t
*src, q31_t *ref, q31_t *dst, q31_t *err, uint32_t size)

```

Parameter:

- **riscv_dsp_lms_q31_t** instance structure of the Q31 LMS filter, defined as follows:

```

typedef struct
{
    uint32_t coeff_size;
    q31_t *state;
    q31_t *coeff;
    q31_t mu;
    q31_t shift;
} riscv_dsp_lms_q31_t;

```

Where

- **coeff_size** number of filter coefficients
- ***state** pointer of the state vector whose size is **coeff_size + size - 1**
- ***coeff** pointer of the time reversed coefficient vector whose size is **coeff_size**
- **mu** step size which can adjust coefficients
- **shift** number of coefficient bits to be shifted

- [in] *instance pointer of the instance structure
- [in] *src pointer of the input vector
- [in] *ref pointer of the desired vector
- [out] *dst pointer of the output vector
- [out] *err pointer of the error vector
- [in] size number of samples

Return Value:

None

riscv_dsp_lms_q15**Prototype:**

```
void riscv_dsp_lms_q15 (const riscv_dsp_lms_q15_t *instance, q15_t
*src, q15_t *ref, q15_t *dst, q15_t *err, uint32_t size)
```

Parameter:

- riscv_dsp_lms_q15_t instance structure of the Q15 LMS filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q15_t *state;
    q15_t *coeff;
    q15_t mu;
    q15_t shift;
} riscv_dsp_lms_q15_t
```

Where

- coeff_size number of filter coefficients
- *state pointer of the state vector whose size is coeff_size + size - 1
- *coeff pointer of the time reversed coefficient vector whose size is coeff_size
- mu step size which can adjust coefficients
- shift number of coefficient bits to be shifted
- [in] *instance pointer of the instance structure
- [in] *src pointer of the input vector

- [in] *ref pointer of the desired vector
- [out] *dst pointer of the output vector
- [out] *err pointer of the error vector
- [in] size number of samples

Return Value:

None

3.4.11 Normalized Least Mean Square (NLMS) Filter Functions

The Normalized Least Mean Square (NLMS) filter function is an extension of LMS filter functions. It can be represented by the following equations:

$$y[n] = h[0] * x[n] + h[1] * x[n - 1] + \dots + h[L - 1] * x[n - L + 1];$$

$$e[n] = d[n] - y[n];$$

$$\mu[n] = 1 / X[n]^2;$$

$$h[n] = h[n - 1] + \mu[n] * e[n] * x[n];$$

Where

- y is the output vector.
- h is the coefficients vector.
- x is the input vector.
- L is the filter order.
- e is the error vector.
- d is the desired vector.
- μ is the step size which can adjust coefficients and is related to the normalized input signals.

DSP programming function library supports distinct NLMS filter function and instance structure for the following data types including floating point, Q31 and Q15. Before you use NLMS filter functions, make sure you initialize the corresponding instance structures first.

The following section will describe the NLMS filter function in details.

riscv_dsp_nlms_f32**Prototype:**

```
void riscv_dsp_nlms_f32 (riscv_dsp_nlms_f32_t *instance, float32_t
*src, float32_t *ref, float32_t *dst, float32_t *err, uint32_t size)
```

Parameter:

- `riscv_dsp_nlms_f32_t` instance structure of the floating-point NLMS filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    float32_t *state;
    float32_t *coeff;
    float32_t mu;
    float32_t energy
} riscv_dsp_nlms_f32_t;
```

Where

- `coeff_size` number of filter coefficients
- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `mu` step size which can adjust coefficients
- `energy` energy of the previous frame
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[in] *ref` pointer of the desired vector
- `[out] *dst` pointer of the output vector
- `[out] *err` pointer of the error vector
- `[in] size` number of samples

Return Value:

None

Example:

```
#define coeff_size 5
#define size 6
#define mu 0.05
#define energy 0.1
```

```

float32_t state[coeff_size + size - 1] = {0.0};
float32_t coeff[coeff_size] = {0.40, 0.10, 0.24, -0.40, -0.34};
float32_t src[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};
float32_t ref[size] = {0.1, 0.2, -0.1, -0.02, -0.1, 0.2};
float32_t err[size];
float32_t dst[size];

riscv_dsp_nlms_f32_t instance = {coeff_size, state, coeff, mu,
energy};

riscv_dsp_nlms_f32(&instance, src, ref, dst, err, size);

```

This example also applies to NLMS filter functions of Q31 and Q15.

riscv_dsp_nlms_q31

Prototype:

```

void riscv_dsp_nlms_q31(riscv_dsp_nlms_q31_t *instance, q31_t
*src, q31_t *ref, q31_t *dst, q31_t *err, uint32_t size);

```

Parameter:

- **riscv_dsp_nlms_q31_t** instance structure of the Q31 NLMS filter, defined as follows:

```

typedef struct
{
    uint32_t coeff_size;
    q31_t *state;
    q31_t *coeff;
    q31_t mu;
    uint8_t postshift;
    q31_t energy
    q31_t x0;
} riscv_dsp_nlms_q31_t;

```

Where

- **coeff_size** number of filter coefficients
- ***state** pointer of the state vector whose size is **coeff_size + size - 1**
- ***coeff** pointer of the time reversed coefficient vector whose size is **coeff_size**

- `mu` step size which can adjust coefficients
- `postshift` number of coefficient bits to be shifted
- `energy` energy of the previous frame
- `x0` previous input sample
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[in] *ref` pointer of the desired vector
- `[out] *dst` pointer of the output vector
- `[out] *err` pointer of the error vector
- `[in] size` number of samples

Return Value:

None

riscv_dsp_nlms_q15**Prototype:**

```
void riscv_dsp_nlms_q15(riscv_dsp_nlms_q15_t *instance, q15_t
*src, q15_t *ref, q15_t *dst, q15_t *err, uint32_t size);
```

Parameter:

- `riscv_dsp_nlms_q15_t` instance structure of the Q15 NLMS filter, defined as follows:

```
typedef struct
{
    uint32_t coeff_size;
    q15_t *state;
    q15_t *coeff;
    q15_t mu;
    uint8_t postshift;
    q15_t energy;
    q15_t x0;
} riscv_dsp_nlms_q15_t;
```

Where

- `coeff_size` number of filter coefficients

- `*state` pointer of the state vector whose size is `coeff_size + size - 1`
- `*coeff` pointer of the time reversed coefficient vector whose size is `coeff_size`
- `mu` step size which can adjust coefficients
- `postshift` number of coefficient bits to be shifted
- `energy` energy of the previous frame
- `x0` previous input sample
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[in] *ref` pointer of the desired vector
- `[out] *dst` pointer of the output vector
- `[out] *err` pointer of the error vector
- `[in] size` number of samples

Return Value:

None

3.4.12 Sparse FIR Filter Functions

Sparse FIR filter functions are FIR filters with the majority of coefficients being zero. Due to this characteristic, sparse FIR filter only keeps the non-zero coefficients and ignores the multiplication operation of signal and zero coefficients.

DSP programming function library supports distinct sparse FIR filter functions and instance structures for the following data types including floating-point, Q31, Q15 and Q7. Before you use sparse FIR filter functions, make sure you initialize the corresponding instance structures first.

These functions and their instance structures are introduced in the following subsection.

riscv_dsp_spafir_f32**Prototype:**

```
void riscv_dsp_spafir_f32 (riscv_dsp_spafir_f32_t *instance, float32_t
*src, float32_t *dst, float32_t *buf, uint32_t size)
```

Parameter:

- `riscv_dsp_spafir_f32_t` instance structure of the floating-point sparse

FIR filter, defined as follows:

```
typedef struct
{
    uint16_t coeff_size;
    uint16_t index;
    float32_t *state;
    float32_t *coeff;
    uint16_t delay;
    int32_t *nezdelay;
} riscv_dsp_spafir_f32_t;
```

Where

- `coeff_size` number of filter coefficients
- `index` index of the state buffer
- `*state` pointer of the state vector whose size is `delay + size`
- `*coeff` pointer of the coefficient vector whose size is `coeff_size`
- `delay` maximum value in the nezdelay vector
- `*nezdelay` vector which stores non-zero coefficients and the size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] *buf` pointer of the temporary vector
- `[in] size` number of samples

Return Value:

None

Note!

The size of temporary buffer should be equal to the size, size of the input vector.

Example:

The example below can be treated as the following formula:

$$y(n) = \text{coeff}(0) * x(n - \text{nezdelay}(0)) + \text{coeff}(1) * x(n - \text{nezdelay}(1)) + \text{coeff}(2) * x(n - \text{nezdelay}(2)) + \text{coeff}(3) * x(n - \text{nezdelay}(3))$$

```
#define coeff_size 4
```

```

#define index 1
#define size 6
int32_t nezdelay[coeff_size] = {3, 0, 2, 1}
#define delay 3
float32_t state[delay + size] = {0.0};
float32_t coeff[coeff_size] = {0.40, 0.10, 0.24, -0.40};
float32_t src[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};
float32_t buf[size];
float32_t dst[size];
riscv_dsp_spafir_f32_t instance = {coeff_size, index, state, coeff,
delay, nezdelay};
riscv_dsp_spafir_f32(&instance, src, dst, buf, size);

```

This example also applies to sparse FIR filter functions of Q31, Q15 and Q7.

riscv_dsp_spafir_q31

Prototype:

```
void riscv_dsp_spafir_q31 (riscv_dsp_spafir_q31_t *instance, q31_t
*src, q31_t *dst, q31_t *buf, uint32_t size)
```

Parameter:

- **riscv_dsp_spafir_q31_t** instance structure of the Q31 sparse FIR filter, defined as follows:

```

typedef struct
{
    uint16_t coeff_size;
    uint16_t index;
    q31_t *state;
    q31_t *coeff;
    uint16_t delay;
    int32_t *nezdelay;
} riscv_dsp_spafir_q31_t;

```

Where

- **coeff_size** number of filter coefficients

- `index` index of the state buffer
- `*state` pointer of the state vector whose size is `delay + size`
- `*coeff` pointer of the coefficient vector whose size is `coeff_size`
- `delay` maximum value in the `nezdelay` vector
- `*nezdelay` vector which stores non-zero coefficients and the size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] *buf` pointer of the temporary vector
- `[in] size` number of samples

Return Value:

None

Note!

The size of temporary buffer should be equal to the size, size of the input vector.

riscv_dsp_spafir_q15**Prototype:**

```
void riscv_dsp_spafir_q15 (riscv_dsp_spafir_q15_t *instance, q15_t
*src, q15_t *dst, q15_t *buf1, q31_t *buf2, uint32_t size)
```

Parameter:

- `riscv_dsp_spafir_q15_t` instance structure of the Q15 sparse FIR filter, defined as follows:

```
typedef struct
{
    uint16_t coeff_size;
    uint16_t index;
    q15_t *state;
    q15_t *coeff;
    uint16_t delay;
    int32_t *nezdelay;
} riscv_dsp_spafir_q15_t;
```

Where

- `coeff_size` number of filter coefficients
- `index` index of the state buffer
- `*state` pointer of the state vector whose size is `delay + size`
- `*coeff` pointer of the coefficient vector whose size is `coeff_size`
- `delay` maximum value in the `nezdelay` vector
- `*nezdelay` vector which stores non-zero coefficients and the size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] *buf1` pointer of the temporary buffer 1
- `[in] *buf2` pointer of the temporary buffer 2
- `[in] size` number of samples

Return Value:

None

Note!

The size of temporary buffer 1 and 2 should be equal to the size of the input vector.

riscv_dsp_spafir_q7

Prototype:

```
void riscv_dsp_spafir_q7 (riscv_dsp_spafir_q7_t *instance, q7_t *src,
q7_t *dst, q7_t *buf1, q31_t *buf2, uint32_t size)
```

Parameter:

- `riscv_dsp_spafir_q7_t` instance structure of the Q7 sparse FIR filter, defined as follows:

```
typedef struct
{
    uint16_t coeff_size;
    uint16_t index;
    q7_t *state;
    q7_t *coeff;
    uint16_t delay;
    int32_t *nezdelay;
```

```
} riscv_dsp_spafir_q7_t;
```

Where

- `coeff_size` number of filter coefficients
- `index` index of the state buffer
- `*state` pointer of the state vector whose size is `delay + size`
- `*coeff` pointer of the coefficient vector whose size is `coeff_size`
- `delay` maximum value in the `nezdelay` vector
- `*nezdelay` vector which stores non-zero coefficients and the size is `coeff_size`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] *buf1` pointer of the temporary buffer 1
- `[in] *buf2` pointer of the temporary buffer 2
- `[in] size` number of samples

Return Value:

None

Note!

The size of temporary buffer 1 and 2 should be equal to the size of the input vector.

3.4.13 Upsampling FIR Filter Functions

Upsampling FIR filter functions upsample incoming signals to generate multi-rate outputs. They are usually used in audio/sound or images processing like increasing the resolutions of images or adjusting the audio output sample rate from 16kbps to 32kbps.

DSP programming function library supports distinct upsampling FIR filter functions and instance structures for the following data types including floating-point, Q31 and Q15. Before you use upsampling filter functions, make sure you initialize the corresponding instance structure first.

These functions and their instance structures are introduced in the following subsection.

riscv_dsp_upsplfir_f32

Prototype:

```
void riscv_dsp_upsplfir_f32 (const riscv_dsp_upsplfir_f32_t *instance,
float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_upsplfir_f32_t` instance structure of the floating-point upsampling FIR filter, defined as follows:

```
typedef struct
{
    uint32_t L;
    uint32_t plen;
    float32_t *coeff;
    float32_t *state;
} riscv_dsp_upsplfir_f32_t;
```

Where

- `L` factor of the upsampling filter
- `plen` number of coefficients
- `*coeff` pointer of the time reversed coefficient vector whose size is `L * plen`
- `*state` pointer of the state vector whose size is `plen + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Example:

```
#define L 2
#define plen 4
#define size 6
float32_t state[plen + size - 1] = {0.0};
float32_t coeff[L * plen] = {0.40, 0.10, 0.24, -0.40, 1.0, 0.5, 0.4, -0.1};
float32_t src[size] = {1.0, 0.5, 0.4, -0.1, -0.1, 0.3};
float32_t dst[size * L];
```

```
riscv_dsp_upsplfir_f32_t instance = {L, plen, state, coeff};
riscv_dsp_upsplfir_f32(&instance, src, dst, size);
```

This example also applies to upsampling FIR filters of types Q31 and Q15.

riscv_dsp_upsplfir_q31

Prototype:

```
void riscv_dsp_upsplfir_q31 (const riscv_dsp_upsplfir_q31_t
*instance, q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- **riscv_dsp_upsplfir_q31_t** instance structure of the Q31 upsampling FIR filter, defined as follows:

```
typedef struct
{
    uint32_t L;
    uint32_t plen;
    q31_t *coeff;
    q31_t *state;
} riscv_dsp_upsplfir_q31_t;
```

Where

- **L** factor of the upsampling filter
- **plen** number of coefficients
- ***coeff** pointer of the time reversed coefficient vector whose size is **L * plen**
- ***state** pointer of the state vector whose size is **plen + size - 1**
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *dst** pointer of the output vector
- **[in] size** number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in 1.31 format and multiplications

yield a 2.62 result. The 2.62 results are accumulated in a 64-bit accumulator. After all operations are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

riscv_dsp_upsplfir_q15

Prototype:

```
void riscv_dsp_upsplfir_q15 (const riscv_dsp_upsplfir_q15_t
*instance, q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `riscv_dsp_upsplfir_q15_t` instance structure of the Q15 upsampling FIR filter, defined as follows:

```
typedef struct
{
    uint32_t L;
    uint32_t plen;
    q15_t *coeff;
    q15_t *state;
} riscv_dsp_upsplfir_q15_t;
```

Where

- `L` factor of the upsampling filter
- `plen` number of coefficients
- `*coeff` pointer of the time reversed coefficient vector whose size is `L * plen`
- `*state` pointer of the state vector whose size is `plen + size - 1`
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of samples

Return Value:

None

Note!

Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 results are accumulated in a 64-bit accumulator in 34.30 format and the result is truncated to 34.15 format by discarding low 15 bits. Lastly, the

outputs are saturated to yield a result in 1.15 format.

3.5 Matrix Functions

A matrix can be seen as a rectangular array with m rows and n columns and of $m * n$ elements.

In DSP programming function library, for programming convenience, matrices are stored into vectors separately in row-major order. That is, the size of a vector equals to the number of a matrix (size = $m * n$).

3.5.1 Matrix Addition Functions

Matrix addition functions add two elements from two source vectors and write the results one-by-one into a destination vector, and the process is as follows:

```
size = row * col;
for (i = 0; i < size; i++)
    dst[i] = src1[i] + src2[i];
```

DSP programming function library supports distinct matrix addition functions for the following data types including floating-point, Q31, and Q15. These functions are introduced in the following subsection.

riscv_dsp_mat_add_f32

Prototype:

```
void riscv_dsp_mat_add_f32 (const float32_t *src1, const float32_t
*src2, float32_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Example:

The following equation shows the addition of two matrices and the

result.

$$\begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.1 \end{bmatrix} + \begin{bmatrix} -0.2 & -0.1 \\ 0.3 & 0.5 \end{bmatrix} = \begin{bmatrix} -0.1 & 0.3 \\ 0.1 & 0.6 \end{bmatrix}$$

The sample code is as shown below:

```
#define row 2
#define col 2
float32_t src1[row * col] = {0.1, 0.4, -0.2, 0.1};
float32_t src2[row * col] = {-0.2, -0.1, 0.3, 0.5};
float32_t dst[row * col];
riscv_dsp_mat_add_f32(src1, src2, dst, row, col);
```

This example also applies to matrix addition functions of Q32 or Q15.

riscv_dsp_mat_add_q31

Prototype:

```
void riscv_dsp_mat_add_q31 (const q31_t *src1, const q31_t *src2,
q31_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Note!

The results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_mat_add_q15

Prototype:

```
void riscv_dsp_mat_add_q15 (const q15_t *src1, const q15_t *src2,
q15_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix

- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Note!

The results will be saturated to the Q15 range [0x8000, 0x7FFF].

3.5.2 Matrix Inverse Functions

The matrix inverse function returns the inverse of a square matrix. The square matrix has the same dimension in the horizontal and vertical direction. Therefore, the matrix inverse function only accepts one parameter (i.e. size) to specify the number of row and that of column. It not only avoids receiving two different numbers for row and column but also improves performance.

The matrix inverse function implements Gauss-Jordan elimination to find the inverse of a matrix. First, the output matrix is filled with 0 and 1 to form an identity matrix. Next, elementary row operations are performed on the input and output matrices simultaneously to make the input matrix an identity matrix. If the original input matrix is invertible, the output matrix will eventually be its inverse and 0 will be returned. The process is as follows:

$[A \mid I]$ elementary row operations $[I \mid A^{-1}] \rightarrow \rightarrow$

Where A is the input matrix and A^{-1} is its inverse matrix. I is an identity matrix generated in the output matrix before elementary row operations and found in the input matrix after the operations.

If the pivot value is zero during elementary row operations, the function needs to look for a non-zero pivot from following rows. Once it is found, swap the two rows and perform elementary row operations again until the input matrix is transformed into an identity matrix. If a non-zero pivot is not available, return -1, indicating the input matrix is singular.

DSP programming function library supports distinct matrix inverse functions for the following data types including single and double precision floating-point.

riscv_dsp_mat_inv_f32

Prototype:

```
int32_t riscv_dsp_mat_inv_f32 (float32_t *src, float32_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] size number of rows/columns in a matrix (i.e. size = rows = columns)

Return Value:

0 success

-1 failure

Note!

- If the return value is 0, the input matrix is invertible and transformed into an identity matrix; if the return value is -1, the input matrix is a singular matrix.
- No matter what value is returned, the content of both the input and output matrices will be changed after executing the matrix inverse function.

Example:

A square matrix (A) and its inverse are listed below:

$$A = \begin{bmatrix} 0.1 & 0.4 \\ -0.2 & -0.3 \end{bmatrix} \quad \text{inv}(A) = \begin{bmatrix} -6 & -8 \\ 4 & 2 \end{bmatrix}$$

The sample code for the inverse of matrix A is shown below:

```
#define size 2
float32_t src[size * size] = {0.1, 0.4, -0.2, -0.3};
float32_t dst[size * size];
if (riscv_dsp_mat_inv_f32(src, dst, size) == 0)
    Success...
else
    Fail...
```

This example also applies to matrix inverse functions of type F64.

riscv_dsp_mat_inv_f64

Prototype:

```
int32_t riscv_dsp_mat_inv_f64 (float64_t *src, float64_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] size number of rows/columns in a matrix (i.e. size = rows = columns)

Return Value:

0 success

-1 failure

3.5.3 Matrix Multiplication Functions

Matrix multiplication functions compute the multiplication of two source matrices and write the result into a destination matrix. The following shows the multiplication of matrix A by matrix B, in which both matrices have 2 rows and 2 columns.

$$\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \times \begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} = \begin{array}{cc} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{array}$$

According to the definition of matrix multiplication, the number of rows in matrix B must be identical with the number of columns in matrix A. That is,

$$C_{ik} = A_{ij} * B_{jk}$$

where A is a matrix with i rows and j columns, B is a matrix with j rows and k columns, and C is the destination matrix with i rows and k columns.

DSP programming function library supports distinct matrix multiplication functions for the following data types including single and double precision floating-point, Q31, Q15 and Q7. These functions are introduced in the following subsection.

riscv_dsp_mat_mul_f32

Prototype:

```
void riscv_dsp_mat_mul_f32 (const float32_t *src1, const float32_t *src2, float32_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input matrix

- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in the first matrix
- [in] col number of columns in the first matrix
- [in] col2 number of columns in the second matrix

Return Value:

None

Example:

The following equation shows the multiplication of two matrices:

$$A_{2,3} \times B_{3,2} = C_{2,2}$$

Given each element with a value, the result is like

$$\begin{bmatrix} 0.1 & -0.1 & 0.1 \\ 0.2 & -0.2 & 0.3 \end{bmatrix} \times \begin{bmatrix} 0.2 & 0.2 \\ -0.1 & 0.3 \\ -0.7 & -0.2 \end{bmatrix} = \begin{bmatrix} -0.04 & -0.03 \\ -0.15 & -0.08 \end{bmatrix}$$

and the example code is as follows:

```
#define Arow 2
#define Acol 3
#define Bcol 2
float32_t src1[Arow * Acol] = {0.1, -0.1, 0.1, 0.2, -0.2, 0.3};
float32_t src2[Acol * Bcol] = {0.2, 0.2, -0.1, 0.3, -0.7, -0.2};
float32_t dst[Arow * Bcol];
riscv_dsp_mat_mul_f32 (src1, src2, dst, Arow, Acol, Bcol);
```

This example also applies to matrix multiplication functions of type F64, Q31 or Q15.

riscv_dsp_mat_mul_f64**Prototype:**

```
void riscv_dsp_mat_mul_f64 (const float64_t *src1, const float64_t
*src2, float64_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix

- [out] *dst pointer of the output matrix
- [in] row number of rows in the first matrix
- [in] col number of columns in the first matrix
- [in] col2 number of columns in the second matrix

Return Value:

None

riscv_dsp_mat_mul_q31**Prototype:**

```
void riscv_dsp_mat_mul_q31 (const q31_t *src1, const q31_t *src2,  
q31_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in the first matrix
- [in] col number of columns in the first matrix
- [in] col2 number of columns in the second matrix

Return Value:

None

riscv_dsp_mat_mul_fast_q31**Prototype:**

```
void riscv_dsp_mat_mul_fast_q31 (const q31_t *src1, const q31_t  
*src2, q31_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in the first matrix
- [in] col number of columns in the first matrix
- [in] col2 number of columns in the second matrix

Return Value:

None

Note!

- A 32-bit accumulator is used to perform multiply-accumulate operations and truncated to Q2.30 result. To avoid overflows that distort the result, the input matrix must be scaled down by $\log_2(\text{row})$ bits. After all operations are performed, the results are saturated to Q1.31 format.
- When compared with `riscv_dsp_mat_mul_q31`, this function delivers higher performance with lower precision.

riscv_dsp_mat_mul_q15**Prototype:**

```
void riscv_dsp_mat_mul_q15 (const q15_t *src1, const q15_t *src2,  
q15_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- `[in] *src1` pointer of the first input matrix
- `[in] *src2` pointer of the second input matrix
- `[out] *dst` pointer of the output matrix
- `[in] row` number of rows in the first matrix
- `[in] col` number of columns in the first matrix
- `[in] col2` number of columns in the second matrix

Return Value:

None

Note!

For RV64, this function will allocate a temporary buffer of 256 bytes for efficient calculation.

riscv_dsp_mat_mul_fast_q15**Prototype:**

```
void riscv_dsp_mat_mul_fast_q15 (const q15_t *src1, const q15_t  
*src2, q15_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- `[in] *src1` pointer of the first input matrix
- `[in] *src2` pointer of the second input matrix

- [\[out\] *dst](#) pointer of the output matrix
- [\[in\] row](#) number of rows in the first matrix
- [\[in\] col](#) number of columns in the first matrix
- [\[in\] col2](#) number of columns in the second matrix

Return Value:

None

Note!

- For RV64, this function will allocate a temporary buffer of 128 bytes for efficient calculation.
- When compared with [riscv_dsp_mat_mul_q15](#), this function delivers higher performance with lower precision.

riscv_dsp_mat_mul_q7**Prototype:**

```
void riscv_dsp_mat_mul_q7 (const q7_t *src1, const q7_t *src2, q7_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [\[in\] *src1](#) pointer of the first input matrix
- [\[in\] *src2](#) pointer of the second input matrix
- [\[out\] *dst](#) pointer of the output matrix
- [\[in\] row](#) number of rows in the first matrix
- [\[in\] col](#) number of columns in the first matrix
- [\[in\] col2](#) number of columns in the second matrix

Return Value:

None

Note!

For RV64, this function will allocate a temporary buffer of 512 bytes for efficient calculation; for RV32, a temporary buffer of 128 bytes will be allocated.

riscv_dsp_mat_mul_vxm_q7**Prototype:**

```
void riscv_dsp_mat_mul_vxm_q7 (const q7_t *src1, const q7_t *src2, q7_t *dst, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of an input vector
- [in] *src2 pointer of an input matrix
- [out] *dst pointer of the output vector
- [in] col number of columns in the input vector
- [in] col2 number of columns in the input matrix

Return Value:

None

Note!

This function multiplies a vector with col columns, src1[1, col], with a matrix with col rows and col2 columns, src2[col, col2], and stores the result into a vector with col2 columns, dst[1, col2].

3.5.4 Matrix Scale Functions

Matrix scale functions multiply a matrix by a constant scaling value and write the result into a destination matrix, and the process is as follows:

```
size = m * n;
for (n = 0; n < size; n++)
    dst[i] = src[i] * scale;
```

In cases of fractional presentation, a parameter shift is introduced to adjust the range of result values. For details about the parameter, see functions with fractional data types riscv_dsp_mat_scale_q31 and riscv_dsp_mat_scale_q15.

DSP programming function library supports distinct matrix scale functions for the following data types including floating- point, Q31, and Q15. These functions are introduced in the following subsection.

riscv_dsp_mat_scale_f32

Prototype:

```
void riscv_dsp_mat_scale_f32 (const float32_t *src, float32_t scale,
float32_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [in] scale constant scaling value

- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Example:

Given a matrix and a scale value 0.2, the matrix scale equation is as follows:

$$\begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.1 \end{bmatrix} \times \text{scale} = \begin{bmatrix} 0.02 & 0.08 \\ -0.04 & 0.02 \end{bmatrix}$$

and the example code is as follows:

```
#define row 2
#define col 2
float32_t src[row * col] = {0.1, 0.4, -0.2, 0.1};
float32_t scale = 0.2;
float32_t dst[row * col];
riscv_dsp_mat_scale_f32 (src1, scale, dst, row, col);
```

This example also applies to matrix scale functions of type Q31 or Q15.

riscv_dsp_mat_scale_q31**Prototype:**

```
void riscv_dsp_mat_scale_q31 (const q31_t *src, q31_t scale_fract,
int32_t shift, q31_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [in] scale_fract constant fractional scaling value
- [in] shift shift bits
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

riscv_dsp_mat_scale_q15

Prototype:

```
void riscv_dsp_mat_scale_q15 (const q15_t *src, q15_t scale_fract,  
int32_t shift, q15_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [in] scale_fract constant fractional scaling value
- [in] shift shift bits
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

3.5.5 Matrix Subtraction Functions

Matrix subtraction functions subtract two matrices which have same dimensions and write the result into a destination matrix, and process is as follows:

```
size = m * n;  
for (i = 0; i < size; i++)  
    dst[i] = src1[i] - src2[i];
```

DSP programming function library supports distinct matrix subtraction functions for the following data types including floating-point, Q31, and Q15. These functions are introduced in the following subsection.

riscv_dsp_mat_sub_f32

Prototype:

```
void riscv_dsp_mat_sub_f32 (const float32_t *src1, const float32_t  
*src2, float32_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix

- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Example:

Given two matrices and their subtraction like below,

$$\begin{bmatrix} 0.1 & 0.4 \\ -0.2 & 0.1 \end{bmatrix} - \begin{bmatrix} -0.2 & -0.1 \\ 0.3 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.3 & 0.5 \\ -0.5 & -0.4 \end{bmatrix}$$

The example code of matrix subtraction is as follows:

```
#define row 2
#define col 2

float32_t src1[row * col] = {0.1, 0.4, -0.2, 0.1};
float32_t src2[row * col] = {-0.2, -0.1, 0.3, 0.5};
float32_t dst[row * col];

riscv_dsp_mat_sub_f32 (src1, src2, dst, row, col);
```

This example also applies to matrix subtraction functions of type Q31 or Q15.

riscv_dsp_mat_sub_q31**Prototype:**

```
void riscv_dsp_mat_sub_q31 (const q31_t *src1, const q31_t *src2,
q31_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Note!

The results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_mat_sub_q15

Prototype:

```
void riscv_dsp_mat_sub_q15 (const q15_t *src1, const q15_t *src2,  
q15_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src1 pointer of the first input matrix
- [in] *src2 pointer of the second input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

Note!

The results will be saturated to the Q15 range [0x8000, 0x7FFF].

3.5.6 Matrix Transpose Functions

Matrix transpose functions transpose a matrix then write the result into a destination matrix, and the process is as follows:

$dst[n, m] = src[m, n]$, where $0 \leq m < row$, $0 \leq n < col$

DSP programming function library supports distinct matrix transpose functions for floating-point, Q31, Q15 and other data types. These functions are introduced in the following subsection.

riscv_dsp_mat_trans_f32

Prototype:

```
void riscv_dsp_mat_trans_f32 (const float32_t *src, float32_t *dst,  
uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix

- [in] col number of columns in a matrix

Return Value:

None

Example:

Given an equation of matrix transposing like below,

$$\begin{matrix} 0.1 & -0.1 & 0.1 \\ 0.2 & -0.2 & 0.3 \end{matrix}_T = \begin{matrix} 0.1 & 0.2 \\ -0.1 & -0.2 \\ 0.1 & 0.3 \end{matrix}$$

and the example code is as follows:

```
#define row 2
#define col 3
float32_t src[row * col] = {0.1, -0.1, 0.1, 0.2, -0.2, 0.3};
float32_t dst[col * row];
riscv_dsp_mat_trans_f32 (src, dst, row, col);
```

This example also applies to matrix transpose functions of type Q31 or Q15.

riscv_dsp_mat_trans_q31**Prototype:**

```
void riscv_dsp_mat_trans_q31 (const q31_t *src, q31_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

riscv_dsp_mat_trans_q15**Prototype:**

```
void riscv_dsp_mat_trans_q15 (const q15_t *src, q15_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

riscv_dsp_mat_trans_u8

Prototype:

```
void riscv_dsp_mat_trans_u8 (const uint8_t *src, uint8_t *dst, uint32_t row, uint32_t col)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] row number of rows in a matrix
- [in] col number of columns in a matrix

Return Value:

None

3.5.7 Matrix Power 2 Functions

The matrix power 2 function powers a square matrix with 2 and then writes the result to a destination matrix, and the process is as follows:

$dst = src^2$, where src is a square matrix

DSP programming function library only supports the matrix power 2 function for double precision floating-point data, and the function is introduced in the following subsection.

riscv_dsp_mat_pwr2_cache_f64

Prototype:

```
int32_t riscv_dsp_mat_pwr2_cache_f64(const float64_t *src, float64_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input matrix
- [out] *dst pointer of the output matrix
- [in] size number of rows or columns in a matrix

Return Value:

0 success

-1 failure

Note!

- The input matrix must be a square matrix with the number of rows or columns equal to size. In addition, size must be a multiple of four (such as 28, 32, 64, and 1024). Otherwise, a return value of -1 is given for the error.
- This function performs a block matrix multiplication. The block size is the size of one cache line for utilizing each cache line fetched from the memory and reducing cache misses.
- When size is smaller than 40, you are advised to use riscv_dsp_mat_mul_f64 for better performance.

3.5.8 Matrix Outer Product Functions

The matrix outer product function computes the multiplication of two source vectors and writes the result into a destination matrix. The following shows the multiplication of vector A by vector B, in which vector A has 3 rows and vector B has 4 columns.

$$\begin{array}{c}
 A_{11} \\
 A_{21} \times B_{11} \quad B_{12} \quad B_{13} \quad B_{14} = \\
 A_{31}
 \end{array}$$

$$\begin{array}{cccc}
 A_{11} \times B_{11} & A_{11} \times B_{12} & A_{11} \times B_{13} & A_{11} \times B_{14} \\
 A_{21} \times B_{11} & A_{21} \times B_{12} & A_{21} \times B_{13} & A_{21} \times B_{14} \\
 A_{31} \times B_{11} & A_{31} \times B_{12} & A_{31} \times B_{13} & A_{31} \times B_{14}
 \end{array}$$

The outer product can be treated as a matrix multiplication in which source matrices A_{ij} and B_{jk} have a constraint that the value of j must be 1, as shown below.

$$C_{ik} = A_{ij} * B_{jk}, \text{ where } j = 1$$

DSP programming function library only supports the matrix outer product function for Q31, and the function is introduced in the following subsection.

riscv_dsp_mat_oprod_q31

Prototype:

```
void riscv_dsp_mat_oprod_q31 (const q31_t *src1, const q31_t *src2,
q31_t *dst, uint32_t size1, uint32_t size2)
```

Parameter:

- [in] *src1 pointer of the first input matrix with a size of size1*1
- [in] *src2 pointer of the second input matrix with a size of 1*size2
- [out] *dst pointer of the output matrix with a size of size1 * size2
- [in] size1 number of rows in the first input matrix
- [in] size2 number of columns in the second input matrix

Return Value:

None

Note!

This function multiplies a one-column matrix with size1 rows, src1[size1, 1], with a one-row matrix with size2 columns, src2[1, size2], and stores the result into a matrix with size1 rows and size2 columns, dst[size1, size2]. It achieves better efficiency for vector-wise matrix multiplication than for regular matrix multiplication.

Example:

The following equation shows the outer product of two matrices and its result.

$$\begin{array}{c} 0x200000 \\ 0x100000 \\ 0x500000 \end{array} + \begin{array}{c} 0x10000 \\ 0x30000 \end{array} = \begin{array}{cc} 0x40 & 0xc0 \\ 0x20 & 0x60 \\ 0xa & 0x1e \end{array}$$

and the example code is as follows:

```
#define Arow 3
#define Bcol 2
q31_t src1[Arow] = {0x200000, 0x100000, 0x500000};
q31_t src2[Bcol] = {0x10000, 0x30000};
q31_t dst[Arow * Bcol];
riscv_dsp_mat_oprod_q31 (src1, src2, dst, Arow, Bcol);
```

3.5.9 Complex Matrix Multiplication Functions

Complex matrix multiplication functions compute the multiplication of two source complex matrices and write the result into a destination complex matrix. The following shows the multiplication of complex matrix A by matrix complex B, in which both matrices have 2 rows and 2 columns.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$

Each multiplication is treated as a complex multiplication.

$$A_n = a_n(\text{re}) + a_n(\text{im})i, B_n = b_n(\text{re}) + b_n(\text{im})i,$$

$$A_n \times B_n = (a_n(\text{re}) \times b_n(\text{re}) - a_n(\text{im}) \times b_n(\text{im})) + (a_n(\text{re}) \times b_n(\text{im}) + a_n(\text{im}) \times b_n(\text{re}))i$$

According to the definition of matrix multiplication, the number of rows in complex matrix B must be identical with the number of columns in complex matrix A. That is,

$$C_{ik} = A_{ij} * B_{jk}$$

where A is a complex matrix with i rows and j columns, B is a complex matrix with j rows and k columns, and C is the complex matrix with i rows and k columns.

DSP programming function library supports distinct complex matrix multiplication functions for the following data types including floating-point, Q31, and Q15, and these functions are introduced in the following subsection.

riscv_dsp_cmat_mul_f32

Prototype:

```
void riscv_dsp_cmat_mul_f32 (const float32_t *src1, const float32_t *src2, float32_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input complex matrix
- [in] *src2 pointer of the second input complex matrix
- [out] *dst pointer of the output complex matrix
- [in] row number of rows in the first complex matrix
- [in] col number of columns in the first complex matrix
- [in] col2 number of columns in the second complex matrix

Return Value:

None

Example:

The following equation shows the multiplication of two complex

matrices:

$$A_{2,3} \times B_{3,2} = C_{2,2}$$

Given each element with a value, the result is like

$$\begin{array}{cc} 0.1 + 0.2i & -0.1 - 0.3i \\ 0.2 - 0.1i & -0.2 + 0.5i \end{array} \times \begin{array}{cc} 0.1 + 0.4i & 0.2 + 0.5i \\ 0.3 - 0.4i & -0.7 - 0.4i \end{array} = \begin{array}{cc} 0.08 - 0.22i & 0.15 - 0.2i \\ -0.36 + 0.15i & -0.26 + 0.6i \end{array}$$

and the example code is as follows:

```
#define Arow 2
#define Acol 3
#define Bcol 2

float32_t src1[2 * Arow * Acol] = {0.1, 0.2, -0.1, -0.3, 0.1, 0.4, 0.2, -0.1,
-0.2, 0.5, 0.3, -0.4};

float32_t src2[2 * Acol * Bcol] = {0.2, 0.5, 0.2, -0.4, -0.1, 0.2, 0.3, 0.2, -
0.7, -0.4, -0.2, -0.1};

float32_t dst[2 * Arow * Bcol];

riscv_dsp_cmat_mul_f32 (src1, src2, dst, Arow, Acol, Bcol);
```

This example also applies to complex matrix multiplication functions of type Q31 or Q15.

riscv_dsp_cmat_mul_q31

Prototype:

```
void riscv_dsp_cmat_mul_q31 (const q31_t *src1, const q31_t *src2,
q31_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input complex matrix
- [in] *src2 pointer of the second input complex matrix
- [out] *dst pointer of the output complex matrix
- [in] row number of rows in the first complex matrix
- [in] col number of columns in the first complex matrix
- [in] col2 number of columns in the second complex matrix

Return Value:

None

riscv_dsp_cmat_mul_q15

Prototype:

```
void riscv_dsp_cmat_mul_q15 (const q15_t *src1, const q15_t *src2,  
q15_t *dst, uint32_t row, uint32_t col, uint32_t col2)
```

Parameter:

- [in] *src1 pointer of the first input complex matrix
- [in] *src2 pointer of the second input complex matrix
- [out] *dst pointer of the output complex matrix
- [in] row number of rows in the first complex matrix
- [in] col number of columns in the first complex matrix
- [in] col2 number of columns in the second complex matrix

Return Value:

None

3.6 Statistics Functions

3.6.1 Maximum Functions

Maximum functions compare the values in a vector then return the maximum one and its index.

DSP programming function library supports distinct maximum functions for the following data types including floating-point, Q31, Q15, Q7, and other data types. These functions are introduced in the following subsection.

riscv_dsp_max_f32

Prototype:

```
float32_t riscv_dsp_max_f32 (const float32_t *src, uint32_t size,  
uint32_t *index)
```

- Parameter:
- [in] *src pointer of the input vector
- [in] size number of elements in a vector
- [out] *index index of the maximum value

Return Value:

Maximum value

riscv_dsp_max_value_f32

Prototype:

`float32_t riscv_dsp_max_value_f32 (const float32_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Maximum value

riscv_dsp_max_q31

Prototype:

`q31_t riscv_dsp_max_q31 (const q31_t *src, uint32_t size, uint32_t *index)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector
- `[out] *index` index of the maximum value

Return Value:

Maximum value

riscv_dsp_max_q15

Prototype:

`q15_t riscv_dsp_max_q15 (const q15_t *src, uint32_t size, uint32_t *index)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector
- `[out] *index` index of the maximum value

Return Value:

Maximum value

riscv_dsp_max_q7

Prototype:

```
q7_t riscv_dsp_max_q7 (const q7_t *src, uint32_t size, uint32_t *index)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector
- [out] *index index of the maximum value

Return Value:

Maximum value

riscv_dsp_max_u8

Prototype:

```
uint8_t riscv_dsp_max_u8 (const uint8_t *src, uint32_t size, uint32_t *index)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector
- [out] *index index of the maximum value

Return Value:

Maximum value

3.6.2 Mean Functions

Mean functions calculate the arithmetic mean of a vector.

DSP programming function library supports distinct mean functions for the following data types including floating-point, Q31, Q15, Q7, and other data types. These functions are introduced in the following subsection.

riscv_dsp_mean_f32

Prototype:

```
float32_t riscv_dsp_mean_f32 (const float32_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Mean value

riscv_dsp_mean_q31

Prototype:

q31_t riscv_dsp_mean_q31 (const q31_t *src, uint32_t size)

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Mean value

riscv_dsp_mean_q15

Prototype:

q15_t riscv_dsp_mean_q15 (const q15_t *src, uint32_t size)

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Mean value

riscv_dsp_mean_q7

Prototype:

q7_t riscv_dsp_mean_q7 (const q7_t *src, uint32_t size)

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Mean value

riscv_dsp_mean_u8

Prototype:

`uint8_t riscv_dsp_mean_u8 (const uint8_t *src, uint32_t size)`

Parameter:

- **[in]** `*src` pointer of the input vector
- **[in]** `size` number of elements in a vector

Return Value:

Mean value

3.6.3 Minimum Functions

Minimum functions compare the values in a vector then return the minimum one and its index.

DSP programming function library supports distinct minimum functions for the following data types including floating-point, Q31, Q15, Q7, and other data types. These functions are introduced in the following subsection.

riscv_dsp_min_f32

Prototype:

`float32_t riscv_dsp_min_f32 (const float32_t *src, uint32_t size, uint32_t *index)`

Parameter:

- **[in]** `*src` pointer of the input vector
- **[in]** `size` number of elements in a vector
- **[out]** `*index` index of the minimum value

Return Value:

Minimum value

riscv_dsp_min_q31

Prototype:

`q31_t riscv_dsp_min_q31 (const q31_t *src, uint32_t size, uint32_t *index)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector
- `[out] *index` index of the minimum value

Return Value:

Minimum value

`riscv_dsp_min_q15`

Prototype:

`q15_t riscv_dsp_min_q15 (const q15_t *src, uint32_t size, uint32_t *index)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector
- `[out] *index` index of the minimum value

Return Value:

Minimum value

`riscv_dsp_min_q7`

Prototype:

`q7_t riscv_dsp_min_q7 (const q7_t *src, uint32_t size, uint32_t *index)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector
- `[out] *index` index of the minimum value

Return Value:

Minimum value

riscv_dsp_min_u8**Prototype:**

```
uint8_t riscv_dsp_min_u8 (const uint8_t *src, uint32_t size, uint32_t
*index)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector
- [out] *index index of the minimum value

Return Value:

Minimum value

3.6.4 Root Mean Square (RMS) Functions

Root mean square (RMS) functions compute root mean square of a vector. The process is as follows:

$$\text{RMS} = \sqrt{(\text{src}[0]^2 + \text{src}[1]^2 + \text{src}[2]^2 + \dots + \text{src}[\text{size}-1]^2) / \text{size}}$$

For [sqrt\(\)](#) , see [3.8.7 Square Root Functions](#).

DSP programming function library supports distinct RMS functions for the following data types including floating-point, Q31, and Q15. These functions are introduced in the following subsection.

riscv_dsp_rms_f32**Prototype:**

```
float32_t riscv_dsp_rms_f32 (const float32_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

RMS value

riscv_dsp_rms_q31**Prototype:**

```
q31_t riscv_dsp_rms_q31 (const q31_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

RMS value

riscv_dsp_rms_q15

Prototype:

q15_t riscv_dsp_rms_q15 (const q15_t *src, uint32_t size)

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

RMS value

3.6.5 Power Functions

Power functions compute the sum of squares of a vector, and the process is as follows:

$$\text{SOS} = \text{src}[0]^2 + \text{src}[1]^2 + \text{src}[2]^2 + \dots + \text{src}[\text{size}-1]^2$$

DSP programming function library supports distinct power functions for the following data types including floating-point, Q31, Q15, and Q7. These functions are introduced in the following subsection.

riscv_dsp_pwr_f32

Prototype:

float32_t riscv_dsp_pwr_f32 (const float32_t *src, uint32_t size)

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Sum of squares

riscv_dsp_pwr_q31**Prototype:**

`q63_t riscv_dsp_pwr_q31 (const q31_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Sum of squares

Note!

The return value is in Q48 format.

riscv_dsp_pwr_q15**Prototype:**

`q63_t riscv_dsp_pwr_q15 (const q15_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Sum of squares

Note!

The return value is in Q30 format.

riscv_dsp_pwr_q7**Prototype:**

`q31_t riscv_dsp_pwr_q7 (const q7_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Sum of squares

Note!

The return value is in Q14 format.

3.6.6 Standard Deviation Functions

Standard deviation functions compute the standard deviation value of a vector, and the process is as follows:

```

sos = src[0]2 + src[1]2 + ... + src[size-1]2;
sqrsum = (src[0] + src[1] + ... + src[size-1])2;
std = sqrt((sos - sqrsum / size) / (size - 1));

```

For `sqrt()`, see [3.8.7 Square Root Functions](#).

DSP programming function library supports distinct standard deviation functions for the following data types including floating-point, Q31, Q15, and other data types. These functions are introduced in the following subsection.

riscv_dsp_std_f32

Prototype:

```
float32_t riscv_dsp_std_f32 (const float32_t *src, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Standard deviation value

riscv_dsp_std_q31

Prototype:

```
q31_t riscv_dsp_std_q31 (const q31_t *src, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Standard deviation value

riscv_dsp_std_q15**Prototype:**

`q15_t riscv_dsp_std_q15 (const q15_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Standard deviation value

riscv_dsp_std_u8**Prototype:**

`q15_t riscv_dsp_std_u8 (const uint8_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

Standard deviation value

3.6.7 Variance Functions

Variance functions compute the variance value of a vector. The process is as follows:

```
sos = src[0]2 + src[1]2 + ... + src[size-1]2;  
sqrsum = (src[0] + src[1] + ... + src[size-1])2;  
v = (sos - sqrsum / size) / (size - 1);
```

DSP programming function library supports distinct variance functions for the following data types including floating-point, Q31, and Q15. These functions are introduced in the following subsection.

riscv_dsp_var_f32**Prototype:**

`float32_t riscv_dsp_var_f32 (const float32_t *src, uint32_t size)`

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Variance value

riscv_dsp_var_q31**Prototype:**

```
q63_t riscv_dsp_var_q31 (const q31_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Variance value

riscv_dsp_var_q15**Prototype:**

```
q31_t riscv_dsp_var_q15 (const q15_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Variance value

3.6.8 Entropy Function

The entropy function computes the entropy of a vector, and the process is as follows:

$$\text{Entropy} = -\left(\sum_{i=0}^{size-1} src[i] \times \ln(src[i]) \right)$$

DSP programming function library supports entropy function for the floating-point. The function is introduced in the following subsection.

riscv_dsp_entropy_f32**Prototype:**

```
float32_t riscv_dsp_entropy_f32 (const float32_t *src, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [in] size number of elements in a vector

Return Value:

Entropy value

3.6.9 Relative Entropy Function

The relative entropy function computes the entropy of two vectors and is also called the Kullback–Leibler divergence (DKL), and the process is as follows:

$$\text{DKL}(\text{src1} \square \text{src2}) = - \left(\sum_{i=0}^n \text{src1}[i] \times \ln \left(\frac{\text{src2}[i]}{\text{src1}[i]} \right) \right)$$

DSP programming function library supports distinct relative entropy function for the floating-point. The function is introduced in the following subsection.

riscv_dsp_relative_entropy_f32**Prototype:**

```
float32_t riscv_dsp_relative_entropy_f32 (const float32_t *src1, const float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size size of elements in a vector

Return Value:

Relative entropy value

3.6.10 Log-Sum-Exp (LSE) Function

The LSE function computes the logarithm of the sum of the exponentials of a vector, and the process is as follows:

$$\text{LSE}(\text{src}_1, \dots, \text{src}_n) = \text{src}^* + \log(\exp(\text{src}_1 - \text{src}^*) + \dots + \exp(\text{src}_n - \text{src}^*))$$

where $\text{src}^* = \max\{\text{src}_1, \dots, \text{src}_n\}$

DSP programming function library supports LSE function for the floating-point. The function is introduced in the following subsection.

riscv_dsp_lse_f32

Prototype:

`float32_t riscv_dsp_lse_f32 (const float32_t *src, uint32_t size)`

Parameter:

- `[in] *src` pointer of the input vector
- `[in] size` number of elements in a vector

Return Value:

LSE value

3.6.11 Log-Sum-Exp (LSE) Dot Product Function

The LSE dot product function computes the logarithm of the sum of the exponentials of the dot product of two vectors, and the process is as follows:

$$\text{LSE}(X_1, \dots, X_n) = X^* + \log(\exp(X_1 - X^*) + \dots + \exp(X_n - X^*))$$

where $X^* = \max\{X_1, \dots, X_n\}$, $X_n = \text{src1}[n] + \text{src2}[n]$

DSP programming function library supports LSE dot product function for the floating-point. The function is introduced in the following subsection.

riscv_dsp_lse_dprod_f32

Prototype:

`float32_t riscv_dsp_lse_dprod_f32 (const float32_t *src1, const float32_t *src2, uint32_t size)`

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector

- `[in] size` size of elements in a vector

Return Value:

LSE dot product value

3.6.12 Naive Gaussian Bayesian Estimation Function

Naive Gaussian Bayesian Estimation function implements the naive Gaussian Bayes estimator.

DSP programming function library supports Naive Gaussian Bayesian Estimation function for the floating-point. The function is introduced in the following subsection.

`riscv_dsp_gaussian_naive_bayes_est_f32`

Prototype:

```
uint32_t riscv_dsp_gaussian_naive_bayes_est_f32 (const
riscv_dsp_gaussian_naivebayes_f32_t *instance, const float32_t *src,
float32_t *buf)
```

Parameter:

- `riscv_dsp_gaussian_naivebayes_f32_t` instance structure of the floating-point Gaussian naïve bayes estimator, defined as follows:

```
typedef struct
{
    uint32_t dimofvec;
    uint32_t numofclass;
    const float32_t *mean;
    const float32_t *var;
    const float32_t *classprior;
    float32_t additiveofvar;
} riscv_dsp_gaussian_naivebayes_f32_t;
```

Where

- `dimofvec` dimension of vector space
- `numofclass` number of classes
- `*mean` mean value for the Gaussian whose size is `dimofvec*numofclass`
- `*var` variance for the Gaussian whose size is `dimofvec*numofclass`

- `*classprior` class prior probabilities whose size is `numofclass`
- `Additiveofvar` additive value of variance
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[in] *buf` pointer of the buffer vector

Return Value:

Estimated class

3.7 Transform Functions

In DSP programming function library, the resolution is set as 1024 (the logarithm of 1024 to base 2 is 10) for default configuration. This default value affects the resolution of cosine and sine lookup tables (i.e. the number of elements in tables). It also affects transform functions since they use these tables to look up cosine or sine values. Thus, some suggested values for the input vector size are provided for transform functions in this chapter. Please find them from the function parameter `m` or the compilation option `FFT_LOGN` in examples. If values that have not been suggested are used, there may be a risk of poor accuracy, although the program may still run.

Note that overflow may occur in Q31 and Q15 transform functions. To avoid the problem, you may need to perform an arithmetic right shift operation before calling these functions. The shift amount is different according to the input format, which may vary depending on the input size.

For details about input formats, see the note of each Q31 and Q15 transform function in subsections of this chapter.

3.7.1 Radix-2 CFFT Functions

Radix-2 complex fast Fourier transform (CFFT) functions implement the famous Cooley-Tukey algorithm to transform signals from time domain to frequency domain. The complex data in the input vector are arranged as `[real, imaginary, real, imaginary..., real, imaginary]`.

DSP programming function library supports distinct Radix-2 CFFT and CIFFT functions for the following data types including floating-point, Q31 and Q15. For Q31 and Q15 Radix-2 CFFT and CIFFT functions, you may need to do an arithmetic right shift before calling them.

Note!

In this section, the parameter `m` represents the base 2 logarithm value of the input sample number and must be a number in the range `[3, 10]`. That is, if the input vector has

128 samples, then `m` is equal to 7 ($\log_2(128) = 7$).

The following section describes each Radix-2 CFFT and CFFT function in details.

`riscv_dsp_cfft_rd2_f32`

Prototype:

```
int32_t riscv_dsp_cfft_rd2_f32 (float32_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Example:

Given 128 samples (that is, `FFT_LOGN = 7`), the example of floating-point Radix-2 CFFT and CFFT is as follows:

```
#define FFT_LOGN 7
float32_t src[2* (1 << FFT_LOGN)] = {...};
int32_t ret;
ret = riscv_dsp_cfft_rd2_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail
ret = riscv_dsp_cfft_rd2_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail
```

This example also applies to the Radix-2 CFFT and CFFT functions of types Q31 and Q15.

riscv_dsp_cifft_rd2_f32**Prototype:**

```
int32_t riscv_dsp_cifft_rd2_f32 (float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

riscv_dsp_cfft_rd2_q31**Prototype:**

```
int32_t riscv_dsp_cfft_rd2_q31 (q31_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-1. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-1 The Input and Output Formats

m	Size	Input format	Output format
3	8	Q1.31	Q5.27
4	16	Q1.31	Q6.26
5	32	Q1.31	Q7.25

m	Size	Input format	Output format
6	64	Q1.31	Q8.24
7	128	Q1.31	Q9.23
8	256	Q1.31	Q10.22
9	512	Q1.31	Q11.21
10	1024	Q1.31	Q12.20

riscv_dsp_cifft_rd2_q31

Prototype:

```
int32_t riscv_dsp_cifft_rd2_q31 (q31_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-2. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-2 The Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.27	Q1.31
4	16	Q6.26	Q1.31
5	32	Q7.25	Q1.31
6	64	Q8.24	Q1.31
7	128	Q9.23	Q1.31
8	256	Q10.22	Q1.31
9	512	Q11.21	Q1.31
10	1024	Q12.20	Q1.31

riscv_dsp_cfft_rd2_q15**Prototype:**

```
int32_t riscv_dsp_cfft_rd2_q15 (q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-3. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-3 The Input and Output Formats

m	Size	Input format	Output format
3	8	Q1.15	Q5.11
4	16	Q1.15	Q6.10
5	32	Q1.15	Q7.9
6	64	Q1.15	Q8.8
7	128	Q1.15	Q9.7
8	256	Q1.15	Q10.6
9	512	Q1.15	Q11.5
10	1024	Q1.15	Q12.4

riscv_dsp_cifft_rd2_q15**Prototype:**

```
int32_t riscv_dsp_cifft_rd2_q15 (q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.

- **[in] m** base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-4. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-4 The Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.11	Q1.15
4	16	Q6.10	Q1.15
5	32	Q7.9	Q1.15
6	64	Q8.8	Q1.15
7	128	Q9.7	Q1.15
8	256	Q10.6	Q1.15
9	512	Q11.5	Q1.15
10	1024	Q12.4	Q1.15

3.7.2 Radix-4 CFFT Functions

Radix-4 complex fast Fourier transform (CFFT) and functions implement the famous Cooley-Tukey algorithm to transform signals from time domain to frequency domain. The complex data in the input vector are arranged as **[real, imaginary, real, imaginary..., real, imaginary]**.

DSP programming function library supports distinct Radix-4 CFFT and CIFFT functions for the following data types including floating-point, Q31 and Q15. These functions are introduced in the following subsections. For Q31 and Q15 Radix-4 CFFT and CIFFT functions, you may need to do an arithmetic right shift before calling them.

In this section, the parameter **m** represents the base 2 logarithm value of the input sample number and must be a multiple of two in the range **[4, 10]**. That is, if the input vector has 256 samples, then **m** is equal to 7 ($\log_2(256) = 8$).

The following section describes each Radix-4 CFFT and CIFFT function in details.

riscv_dsp_cfft_rd4_f32**Prototype:**

```
int32_t riscv_dsp_cfft_rd4_f32 (float32_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Example:

Given 256 samples (that is, **FFT_LOGN = 8**), the example of floating-point Radix-4 CFFT and CIFFT is as follows:

```
#define FFT_LOGN 8
float32_t src[2* (1 << FFT_LOGN)] = {...};
int32_t ret;
ret = riscv_dsp_cfft_rd4_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail
ret = riscv_dsp_cifft_rd4_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail
```

This example also applies to Radix-4 CFFT and CIFFT functions of type Q31 or Q15.

riscv_dsp_cifft_rd4_f32**Prototype:**

```
int32_t riscv_dsp_cfft_rd4_f32 (float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Example:

See the example in [riscv_dsp_cfft_rd2_f32](#).

riscv_dsp_cfft_rd4_q31

Prototype:

```
int32_t riscv_dsp_cfft_rd4_q31 (q31_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-5. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-5 Input and Output Formats

m	Size	Input format	Output format
4	16	Q1.31	Q6.26
6	64	Q1.31	Q8.24
8	256	Q1.31	Q10.22

m	Size	Input format	Output format
10	1024	Q1.31	Q12.20

riscv_dsp_cifft_rd4_q31

Prototype:

```
int32_t riscv_dsp_cifft_rd4_q31 (q31_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-6. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-6 Input and Output Formats

m	Size	Input format	Output format
4	16	Q6.26	Q1.31
6	64	Q8.24	Q1.31
8	256	Q10.22	Q1.31
10	1024	Q12.20	Q1.31

riscv_dsp_cfft_rd4_q15

Prototype:

```
int32_t riscv_dsp_cfft_rd4_q15 (q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.

- **[in] m** base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-7. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-7 Input and Output Formats

m	Size	Input format	Output format
4	16	Q1.15	Q6.10
6	64	Q1.15	Q8.8
8	256	Q1.15	Q10.6
10	1024	Q1.15	Q12.4

riscv_dsp_cifft_rd4_q15

Prototype:

`int32_t riscv_dsp_cifft_rd4_q15 (q15_t *src, uint32_t m)`

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set as 4, 6, 8 or 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-8. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-8 Input and Output Formats

m	Size	Input format	Output format
4	16	Q6.10	Q1.15
6	64	Q8.8	Q1.15
8	256	Q10.6	Q1.15
10	1024	Q12.4	Q1.15

3.7.3 Complex FFT Functions

Functions in this section internally call radix-2 or radix-4 CFFT/CIFFT functions introduced in [3.7.1 Radix-2 CFFT Functions](#) and [3.7.2 Radix-4 CFFT Functions](#) depending on the parameter `m`, which represents the base 2 logarithm value of the input sample number. If the value of `m` is a multiple of two such as 4 or 6, the radix-4 functions are used to provide a faster performance. Otherwise, radix-2 functions are used.

DSP programming function library supports distinct CFFT and CIFFT functions for the following data types including floating-point, Q31 and Q15. For Q31 and Q15 Radix-4 CFFT and CIFFT functions, you may need to do an arithmetic right shift before calling them.

The following section describes each CFFT and CIFFT function in details.

`riscv_dsp_cfft_f32`

Prototype:

```
int32_t riscv_dsp_cfft_f32 (float32_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Example:

Given 128 samples (that is, `FFT_LOGN = 7`), the example of floating-point CFFT and CIFFT is as follows:

```

#define FFT_LOGN 7
float32_t src[2* (1 << FFT_LOGN)] = {...};
int32_t ret;
ret = riscv_dsp_cfft_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail
ret = riscv_dsp_cifft_f32(src, FFT_LOGN);
if (ret == 0)
    Success
Else
    Fail

```

This example also applies to the CFFT and CFFT functions of types Q31 and Q15.

riscv_dsp_cifft_f32

Prototype:

```
int32_t riscv_dsp_cifft_f32 (float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of the sample number and it can be set from 3 to 10.

Return Value:

0 success

-1 failure

riscv_dsp_cfft_q31

Prototype:

```
int32_t riscv_dsp_cfft_q31 (q31_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed,

the output will be stored in the input vector.

- **[in] m** base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-9. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-9 Input and Output Formats

m	Size	Input format	Output format
3	8	Q1.31	Q5.27
4	16	Q1.31	Q6.26
5	32	Q1.31	Q7.25
6	64	Q1.31	Q8.24
7	128	Q1.31	Q9.23
8	256	Q1.31	Q10.22
9	512	Q1.31	Q11.21
10	1024	Q1.31	Q12.20

riscv_dsp_cifft_q31

Prototype:

`int32_t riscv_dsp_cifft_q31 (q31_t *src, uint32_t m)`

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-10. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-10 Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.27	Q1.31
4	16	Q6.26	Q1.31
5	32	Q7.25	Q1.31
6	64	Q8.24	Q1.31
7	128	Q9.23	Q1.31
8	256	Q10.22	Q1.31
9	512	Q11.21	Q1.31
10	1024	Q12.20	Q1.31

riscv_dsp_cfft_q15**Prototype:**

```
int32_t riscv_dsp_cfft_q15 (q15_t *src, uint32_t m)
```

Parameter:

- [in, out] **src* pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] *m* base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-11. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-11 Input and Output Formats

m	Size	Input format	Output format
3	8	Q1.15	Q5.11

m	Size	Input format	Output format
4	16	Q1.15	Q6.10
5	32	Q1.15	Q7.9
6	64	Q1.15	Q8.8
7	128	Q1.15	Q9.7
8	256	Q1.15	Q10.6
9	512	Q1.15	Q11.5
10	1024	Q1.15	Q12.4

riscv_dsp_cifft_q15

Prototype:

```
int32_t riscv_dsp_cifft_q15 (q15_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of the sample number and it can be set from 3 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-12. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-12 Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.11	Q1.15
4	16	Q6.10	Q1.15
5	32	Q7.9	Q1.15
6	64	Q8.8	Q1.15
7	128	Q9.7	Q1.15
8	256	Q10.6	Q1.15
9	512	Q11.5	Q1.15

m	Size	Input format	Output format
10	1024	Q12.4	Q1.15

3.7.4 DCT Type II Functions

Discrete cosine transform (DCT) type II functions implement DCT with the following equation:

$$y[k] = \sum_{n=0}^{N-1} (x[n] * \cos(\pi * (2 * n + 1) * \frac{k}{(2 * N)}))$$

and IDCT (DCT Type III, also called as the inverse of DCT II), with the following equation:

$$x[k] = \frac{2.0}{N} * \sum_{n=0}^{N-1} (c[n] * y[n] * \cos(\pi * (2 * k + 1) * \frac{n}{(2 * N)}))$$

Where, $c[0] = 1 / 2.0$, $c[n] = 1$ for $n \neq 0$.

DSP programming function library supports distinct DCT type II and IDCT functions for the following data types including floating-point, Q31 and Q15. For Q31 and Q15 DCT type II and IDCT functions, you may need to do an arithmetic right shift before calling them.

These functions are introduced in the following subsection.

riscv_dsp_dct_f32

Prototype:

```
void riscv_dsp_dct_f32(float32_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Example:

Given 256 samples (that is, **FFT_LOGN = 8**), the example of floating-point (DCT) type II and IDCT is as follows:

```
#define FFT_LOGN 8
```

```
float32_t src[(1 << FFT_LOGN)] = {...};  
riscv_dsp_dct_f32(src, FFT_LOGN);  
riscv_dsp_idct_f32(src, FFT_LOGN);
```

This example also applies to DCT Type II and IDCT functions of type Q31 or Q15.

riscv_dsp_idct_f32

Prototype:

```
void riscv_dsp_idct_f32(float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Example:

See the example in [riscv_dsp_dct_f32](#).

riscv_dsp_dct_q31

Prototype:

```
void riscv_dsp_dct_q31(q31_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Note!

The input and output formats are listed in Table 3-13. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-13 Input and Output Formats

m	Size	Input format	Output format
3	8	Q2.30	Q5.27
4	16	Q2.30	Q6.26
5	32	Q2.30	Q7.25
6	64	Q2.30	Q8.24
7	128	Q2.30	Q9.23
8	256	Q2.30	Q10.22

riscv_dsp_idct_q31**Prototype:**

```
void riscv_dsp_idct_q31(q31_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Note!

The input and output formats are listed in Table 3-14. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-14 Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.27	Q2.30
4	16	Q6.26	Q2.30
5	32	Q7.25	Q2.30
6	64	Q8.24	Q2.30
7	128	Q9.23	Q2.30
8	256	Q10.22	Q2.30

riscv_dsp_dct_q15**Prototype:**

```
void riscv_dsp_dct_q15(q15_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Note!

The input and output formats are listed in Table 3-15. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-15 Input and Output Formats

m	Size	Input format	Output format
3	8	Q2.14	Q5.11
4	16	Q2.14	Q6.10
5	32	Q2.14	Q7.9
6	64	Q2.14	Q8.8
7	128	Q2.14	Q9.7
8	256	Q2.14	Q10.6

riscv_dsp_idct_q15**Prototype:**

```
void riscv_dsp_idct_q15(q15_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 3 to 8

Return Value:

None

Note!

The input and output formats are listed in Table 3-16. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-16 Input and Output Formats

m	Size	Input format	Output format
3	8	Q5.11	Q2.14
4	16	Q6.10	Q2.14
5	32	Q7.9	Q2.14
6	64	Q8.8	Q2.14
7	128	Q9.7	Q2.14
8	256	Q10.6	Q2.14

3.7.5 DCT Type IV Functions

Discrete cosine transform (DCT) type IV functions implement DCT transform with the following equation:

$$y[k] = \sum_{n=0}^{N-1} (x[n] * \cos(\pi * (2 * k + 1) * \frac{(2 * n + 1)}{(4 * N)}))$$

and IDCT with the following equation:

$$x[k] = \frac{2}{N} * \sum_{n=0}^{N-1} (y[n] * \cos(\pi * (2 * k + 1) * \frac{(2 * n + 1)}{(4 * N)}))$$

DSP programming function library supports distinct DCT type IV and IDCT functions for the following data types including floating-point, Q31 and Q15. For Q31 and Q15 DCT type IV and IDCT functions, you may need to do an arithmetic right shift before calling them.

These functions are introduced in the following subsection.

riscv_dsp_dct4_f32

Prototype:

```
void riscv_dsp_dct4_f32(float32_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Example:

Given 128 samples (that is, `FFT_LOGN = 7`), the example of floating-point DCT or IDCT type IV transform is as follows:

```
#define FFT_LOGN 8
float32_t src[(1 << FFT_LOGN)] = {...};
riscv_dsp_dct4_f32(src, FFT_LOGN);
riscv_dsp_idct4_f32(src, FFT_LOGN);
```

This example also applies to DCT and IDCT Type IV functions of type Q31 or Q15.

riscv_dsp_idct4_f32**Prototype:**

```
void riscv_dsp_idct4_f32(float32_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Example:

See the example in `riscv_dsp_dct4_f32`.

riscv_dsp_dct4_q31**Prototype:**

```
void riscv_dsp_dct4_q31(q31_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Note!

The input and output formats are listed in Table 3-17 Input and Output Formats. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-17 Input and Output Formats

m	Size	Input format	Output format
3	8	Q2.30	Q6.26
4	16	Q2.30	Q7.25
5	32	Q2.30	Q8.24
6	64	Q2.30	Q9.23
7	128	Q2.30	Q10.22

riscv_dsp_idct4_q31**Prototype:**

```
void riscv_dsp_idct4_q31(q31_t *src, uint32_t m)
```

Parameter:

[in, out] **src* pointer of the input vector. After the function is executed, the output will be stored in the input vector.

[in] *m* base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Note!

The input and output formats are listed in Table 3-18. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-18 Input and Output Formats

m	Size	Input format	Output format
3	8	Q6.26	Q2.30
4	16	Q7.25	Q2.30
5	32	Q8.24	Q2.30
6	64	Q9.23	Q2.30
7	128	Q10.22	Q2.30

riscv_dsp_dct4_q15**Prototype:**

```
void riscv_dsp_dct4_q15(q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Note!

The input and output formats are listed in Table 3-19. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-19 Input and Output Formats

m	Size	Input format	Output format
3	8	Q2.14	Q6.10
4	16	Q2.14	Q7.9
5	32	Q2.14	Q8.8
6	64	Q2.14	Q9.7
7	128	Q2.14	Q10.6

riscv_dsp_idct4_q15**Prototype:**

```
void riscv_dsp_idct4_q15(q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 3 to 7

Return Value:

None

Note!

The input and output formats are listed in Table 3-20. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-20 Input and Output Formats

m	Size	Input format	Output format
3	8	Q6.10	Q2.14
4	16	Q7.9	Q2.14
5	32	Q8.8	Q2.14
6	64	Q9.7	Q2.14
7	128	Q10.6	Q2.14

3.7.6 Real FFT Functions

Real fast Fourier transform (RFFT) and inverse fast Fourier transform (RIFFT) functions transform signals composed of real values from time domain to frequency domain.

The RFFT algorithm treats the N real data in the input as $N/2$ complex and performs a CFFT.

After the CFFT is computed, the real value at $N/2$ of the frequency spectrum is stored at the first imaginary position. The output layout of the RFFT is thus arranged as `[r[0], r[N/2], r[1], i[1], ..., r[N/2 - 1], i[N/2 - 1]]` where `r` is real, `i` is imaginary and `N` is the size of the input data. As no additional array is created during the transformation, the size of the output is identical to that of the input.

As to the RIFFT, its input data has the same format as the output of the RFFT and its output data after computation is converted from the frequency spectrum to the time domain.

DSP programming function library supports distinct RFFT and RIFFT functions for the following data types including floating-point, Q31 and Q15. For Q31 and Q15 RFFT and RIFFT functions, you may need to do an arithmetic right shift before calling them.

The following section describes each RFFT and RIFFT function in details.

riscv_dsp_rfft_f32

Prototype:

```
int32_t riscv_dsp_rfft_f32(float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 4 to 10

Return Value:

0 success

-1 failure

Example:

Given 128 samples (that is, FFT_LOGN = 7), the example of floating-point RFFT and RIFFT is as follows:

```
#define FFT_LOGN 7
float32_t src[(1 << FFT_LOGN)] = {...};
int32_t ret;
ret = riscv_dsp_rfft_f32(src, FFT_LOGN);
if (ret == 0)
    Success
else
    Fail
ret = riscv_dsp_rifft_f32(src, FFT_LOGN);
if (ret == 0)
    Success
else
    Fail
```

This example also applies to the RFFT and RIFFT functions of type Q31 or Q15.

riscv_dsp_rifft_f32**Prototype:**

```
int32_t riscv_dsp_rifft_f32(float32_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 4 to 10

Return Value:

0 success

-1 failure

riscv_dsp_rfft_q31**Prototype:**

```
int32_t riscv_dsp_rfft_q31(q31_t *src, uint32_t m)
```

Parameter:

- [in, out] *src pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- [in] m base 2 logarithm value of samples and it can be set from 4 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-21. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-21 Input and Output Formats

m	Size	Input format	Output format
4	16	Q1.31	Q6.26
5	32	Q1.31	Q7.25
6	64	Q1.31	Q8.24
7	128	Q1.31	Q9.23
8	256	Q1.31	Q10.22
9	512	Q1.31	Q11.21
10	1024	Q1.31	Q12.20

riscv_dsp_rifft_q31**Prototype:**

```
int32_t riscv_dsp_rifft_q31(q31_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of samples and it can be set from 4 to 10.

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-22. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-22 Input and Output Formats

m	Size	Input format	Output format
4	16	Q6.26	Q1.31
5	32	Q7.25	Q1.31
6	64	Q8.24	Q1.31
7	128	Q9.23	Q1.31
8	256	Q10.22	Q1.31
9	512	Q11.21	Q1.31
10	1024	Q12.20	Q1.31

riscv_dsp_rfft_q15**Prototype:**

```
int32_t riscv_dsp_rfft_q15(q15_t *src, uint32_t m)
```

Parameter:

- `[in, out] *src` pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- `[in] m` base 2 logarithm value of samples and it can be set from 4 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-23. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation

before calling this function.

Table 3-23 Input and Output Formats

m	Size	Input format	Output format
4	16	Q1.15	Q6.10
5	32	Q1.15	Q7.9
6	64	Q1.15	Q8.8
7	128	Q1.15	Q9.7
8	256	Q1.15	Q10.6
9	512	Q1.15	Q11.5
10	1024	Q1.15	Q12.4

riscv_dsp_rifft_q15

Prototype:

```
int32_t riscv_dsp_rifft_q15(q15_t *src, uint32_t m)
```

Parameter:

- **[in, out] *src** pointer of the input vector. After the function is executed, the output will be stored in the input vector.
- **[in] m** base 2 logarithm value of samples and it can be set from 4 to 10

Return Value:

0 success

-1 failure

Note!

The input and output formats are listed in Table 3-24. To satisfy the input format corresponding to your input size, you may need to perform an arithmetic shift operation before calling this function.

Table 3-24 Input and Output Formats

m	Size	Input format	Output format
4	16	Q6.10	Q1.15
5	32	Q7.9	Q1.15
6	64	Q8.8	Q1.15
7	128	Q9.7	Q1.15
8	256	Q10.6	Q1.15
9	512	Q11.5	Q1.15

m	Size	Input format	Output format
10	1024	Q12.4	Q1.15

3.8 Utils Functions

3.8.1 arctan Functions

arctan functions are the inverse of trigonometric functions which return the inverse of tan given an input value. The input values and output results of arctan functions are in radians and the output results are always in the range $[-\pi/2, \pi/2]$.

DSP programming function library supports distinct arctan functions for the following data types including floating-point, Q31 and Q15. These functions are introduced in the following subsection.

riscv_dsp_atan_f32

Prototype:

`float32_t riscv_dsp_atan_f32 (float32_t src)`

Parameter:

- `[in] src` input value (radian)

Return Value:

Radian value

Example:

```
float32_t src = 0.5;
float32_t dst;
dst = riscv_dsp_atan_f32 (src);
```

This example also applies to arctan functions of type Q31 or Q15.

riscv_dsp_atan_q31

Prototype:

`q31_t riscv_dsp_atan_q31 (q31_t src)`

Parameter:

- `[in] src` input value (radian)

Return Value:

Radian value

riscv_dsp_atan_q15

Prototype:

`q15_t riscv_dsp_atan_q15 (q15_t src)`

Parameter:

- `[in] src` input value (radian)

Return Value:

Radian value

3.8.2 arctan2 Functions

arctan2 functions are similar to arc tangent functions but accept two inputs y and x to compute the inverse tangent value. Normally, the output results are all in radians and always in the range $[-\pi, \pi]$. For convenient calculation, they are mapped to $[-1, 1]$ in Q number functions, but you can use conversion formulas to convert them to their floating-point counterparts.

DSP programming function library supports distinct arctan2 functions for the following data types including floating-point, Q31 and Q15. These functions are introduced in the following subsection.

riscv_dsp_atan2_f32

Prototype:

`q31_t riscv_dsp_atan2_f32 (float32_t y, float32_t x)`

Parameter:

- `[in] y` input value y
- `[in] x` input value x

Return Value:

Radian value

Note!

The return value is in the floating-point format and always in the range $[-\pi, \pi]$.

riscv_dsp_atan2_q31

Prototype:

`q31_t riscv_dsp_atan2_q31 (q31_t y, q31_t x)`

Parameter:

- `[in] y` input value `y`
- `[in] x` input value `x`

Return Value:

Radian value

Note!

The return value is in the Q31 format and always in the range `[0x80000000, 0x7FFFFFFF]` (namely, `[- 1, 1)`). To convert the value to the floating-point format, apply the following formula: `floating point value = (return value *  $\pi$ ) / (2 ^ 31)`

`riscv_dsp_atan2_q15`

Prototype:

`q15_t riscv_dsp_atan2_q15 (q15_t y, q15_t x)`

Parameter:

- `[in] y` input value `y`
- `[in] x` input value `x`

Return Value:

Radian value

Note!

The return value is in the Q15 format and always in the range `[0x8000, 0x7FFF]` (namely, `[- 1, 1)`). To convert the value to the floating-point format, apply the following formula: `floating point value = (return value *  $\pi$ ) / (2 ^ 15)`

3.8.3 cos and sin Functions

Cosine and sine functions compute the cosine and sine values.

The input values of these functions are in radians and their ranges are different between floating-point and Q number functions. For floating-point functions, there is no limitation for the input range since the inputs, whatever values they are, will be pre-processed by the following snippet into the range `[-2 $\pi$ , 2 $\pi$ ]`:

```
src = abs(src)
while (src >= 2 $\pi$ )
```

```

{
    src -= 2 $\pi$ ;
}

```

where `src` is the input value.

For Q number (Q31 and Q15) functions, they only support the input range `[-1, 1)` to represent the Q format range `[- $\pi$ ,  $\pi$ ]`. Thus, the input values may need to be converted into the Q format range before these functions are called.

DSP programming function library supports distinct cosine and sine functions for the following data types including floating-point, Q31 and Q15. These functions are introduced in the following subsection.

`riscv_dsp_cos_f32`

Prototype:

```
float32_t riscv_dsp_cos_f32 (float32_t src)
```

Parameter:

- `[in] src` input value (radian)

Return Value:

Cosine value of the input

Example:

```

float32_t src = 0.5;
float32_t dst;
dst = riscv_dsp_cos_f32 (src);

```

This example also applies to cosine/sine functions of type Q31 or Q15.

`riscv_dsp_cos_q31`

Prototype:

```
q31_t riscv_dsp_cos_q31 (q31_t src)
```

Parameter:

- `[in] src` input value (radian)

Return Value:

Cosine value of the input

Note!

The input range is [0x80000000, 0x7FFFFFFF] and mapped to the range $[-\pi, \pi]$.

riscv_dsp_cos_q15**Prototype:**

`q15_t riscv_dsp_cos_q15 (q15_t src)`

Parameter:

- **[in] src** input value (radian)

Return Value:

Cosine value of the input

Note!

The input range is [0x8000, 0x7FFF] and mapped to the range $[-\pi, \pi]$.

riscv_dsp_sin_f32**Prototype:**

`float32_t riscv_dsp_sin_f32 (float32_t src)`

Parameter:

- **[in] src** input value (radian)

Return Value:

Sine value of the input

riscv_dsp_sin_q31**Prototype:**

`q31_t riscv_dsp_sin_q31 (q31_t src)`

Parameter:

- **[in] src** input value (radian)

Return Value:

Sine value of the input

Note!

The input range is [0x80000000, 0x7FFFFFFF] and mapped to the range $[-\pi, \pi]$.

riscv_dsp_sin_q15**Prototype:**

```
q15_t riscv_dsp_sin_q15 (q15_t src)
```

Parameter:

- [in] `src` input value (radian)

Return Value:

Sine value of the input

Note!

The input range is [0x8000, 0x7FFF] and mapped to the range $[-\pi, \pi]$.

3.8.4 Conversion Functions

Conversion functions convert element values in a source vector from one data type to another and write the results into a destination vector. There are following data type conversions:

```
float32_t  q31_t
float32_t  q15_t
float32_t  q7_t
q31_t      float32_t
q31_t      q15_t
q31_t      q7_t
q15_t      float32_t
q15_t      q31_t
q15_t      q7_t
q7_t       float32_t
q7_t       q31_t
q7_t       q15_t
```

These conversion functions are introduced in the following subsection.

riscv_dsp_convert_f32_q31**Prototype:**

```
void riscv_dsp_convert_f32_q31 (float32_t *src, q31_t *dst, uint32_t
size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- This function converts values from float32_t to q31_t.
- The results will be saturated to the Q31 range [0x80000000, 0x7FFFFFFF].

riscv_dsp_convert_f32_q15**Prototype:**

```
void riscv_dsp_convert_f32_q15 (float32_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- This function converts values from float32_t to q15_t.
- The results will be saturated to the Q15 range [0x8000, 0x7FFF].

riscv_dsp_convert_f32_q7**Prototype:**

```
void riscv_dsp_convert_f32_q7 (float32_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- This function converts values from float32_t to q7_t.
- The results will be saturated to the Q7 range [0x80, 0x7F].

riscv_dsp_convert_q31_f32**Prototype:**

```
void riscv_dsp_convert_q31_f32 (q31_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

This function converts values from q31_t to float32_t.

riscv_dsp_convert_q31_q15**Prototype:**

```
void riscv_dsp_convert_q31_q15 (q31_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- This function converts values from q31_t to q15_t.
- There is no overflow situation since the values are simply right shifted with 16 bits.

riscv_dsp_convert_q31_q7**Prototype:**

```
void riscv_dsp_convert_q31_q7 (q31_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

- This function converts values from q31_t to q7_t.
- There is no overflow situation since the values are simply right shifted with 24 bits.

riscv_dsp_convert_q15_f32**Prototype:**

```
void riscv_dsp_convert_q15_f32 (q15_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

This function converts values from q15_t to float32_t.

riscv_dsp_convert_q15_q31**Prototype:**

```
void riscv_dsp_convert_q15_q31 (q15_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

This function converts values from `q15_t` to `q31_t` (left-shifted with 16 bits).

riscv_dsp_convert_q15_q7**Prototype:**

```
void riscv_dsp_convert_q15_q7 (q15_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

This function converts values from `q15_t` to `q7_t` (right-shifted with 8 bits).

riscv_dsp_convert_q7_f32**Prototype:**

```
void riscv_dsp_convert_q7_f32 (q7_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

Note!

This function converts values from `q7_t` to `float32_t`.

riscv_dsp_convert_q7_q31

Prototype:

```
void riscv_dsp_convert_q7_q31 (q7_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Return Value:

None

Note!

This function converts values from `q7_t` to `q31_t` (left-shifted with 24 bits).

riscv_dsp_convert_q7_q15

Prototype:

```
void riscv_dsp_convert_q7_q15 (q7_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Return Value:

None

Note!

This function converts values from `q7_t` to `q15_t` (left-shifted with 8 bits).

3.8.5 Duplicate Functions

Duplicate functions copy element values in a source vector and write the values one-by-one into a destination vector, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = src[n];
```

DSP programming function library supports distinct duplicate functions for the following data types including floating-point, Q31, Q15 and Q7. These functions are introduced in the following subsection.

riscv_dsp_dup_f32

Prototype:

```
void riscv_dsp_dup_f32 (float32_t *src, float32_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_dup_q31

Prototype:

```
void riscv_dsp_dup_q31 (q31_t *src, q31_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_dup_q15

Prototype:

```
void riscv_dsp_dup_q15 (q15_t *src, q15_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_dup_q7

Prototype:

```
void riscv_dsp_dup_q7 (q7_t *src, q7_t *dst, uint32_t size)
```

Parameter:

- [in] *src pointer of the input vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

3.8.6 Set Functions

Set functions set all elements in a destination vector to a constant value, and the process is as follows:

```
for (n = 0; n < size; n++)  
    dst[n] = value;
```

DSP programming function library supports distinct set functions for the following data types including floating-point, Q31, Q15 and Q7. These functions are introduced in the following subsection.

riscv_dsp_set_f32

Prototype:

```
void riscv_dsp_set_f32 (float32_t val, float32_t *dst, uint32_t size)
```

Parameter:

- [in] val value to be written to the output vector
- [out] *dst pointer of the output vector
- [in] size number of elements in a vector

Return Value:

None

riscv_dsp_set_q31**Prototype:**

```
void riscv_dsp_set_q31 (q31_t val, q31_t *dst, uint32_t size)
```

Parameter:

- [in] **val** value to be written to the output vector
- [out] ***dst** pointer of the output vector
- [in] **size** number of elements in a vector

Return Value:

None

riscv_dsp_set_q15**Prototype:**

```
void riscv_dsp_set_q15 (q15_t val, q15_t *dst, uint32_t size)
```

Parameter:

- [in] **val** value to be written to the output vector
- [out] ***dst** pointer of the output vector
- [in] **size** number of elements in a vector

Return Value:

None

riscv_dsp_set_q7**Prototype:**

```
void riscv_dsp_set_q7 (q7_t val, q7_t *dst, uint32_t size)
```

Parameter:

- [in] **val** value to be written to the output vector
- [out] ***dst** pointer of the output vector
- [in] **size** number of elements in a vector

Return Value:

None

3.8.7 Square Root Functions

Square root functions return the square root value of the input.

DSP programming function library supports distinct square root functions for the following data types including floating-point, Q31 and Q15. These functions are introduced in the following subsection.

riscv_dsp_sqrt_f32

Prototype:

`float32_t riscv_dsp_sqrt_f32 (float32_t src)`

Parameter:

- **[in] src** input value

Return Value:

Square root of the input

riscv_dsp_sqrt_q31

Prototype:

`q31_t riscv_dsp_sqrt_q31 (q31_t src)`

Parameter:

- **[in] src** input value

Return Value:

Square root of the input

riscv_dsp_sqrt_q15

Prototype:

`q15_t riscv_dsp_sqrt_q15 (q15_t src)`

Parameter:

- **[in] src** input value

Return Value:

Square root of the input

3.8.8 Barycenter Function

The function calculates the barycenter of the input vector.

DSP programming function library only supports the barycenter function for floating-point data.

riscv_dsp_barycenter_f32

Prototype:

```
void riscv_dsp_barycenter_f32(const float32_t *src, const float32_t *weights, float32_t *dst, uint32_t numofvec, uint32_t dimofvec)
```

Parameter:

- [in] **src* pointer of the input vector
- [in] **weights* pointer of the weighted vector
- [out] **dst* pointer of the output vector
- [in] *numofvec* number of input vectors
- [in] *dimofvec* dimension of vector space

Return Value:

None

3.8.9 Weighted Sum Function

This function calculates the weighted sum of the input vector.

DSP programming function library only supports the weighted sum function for floating-point data.

riscv_dsp_weighted_sum_f32

Prototype:

```
float32_t riscv_dsp_weighted_sum_f32(const float32_t *src, const float32_t *weight, uint32_t size)
```

Parameter:

- [in] **src* pointer of the input vector
- [in] **weight* pointer of the weighted vector
- [in] *size* number of elements in a vector

Return Value:

Weighted sum value

3.9 Support Vector Machine (SVM) Estimation Functions

This set of functions implements two-class SVM classification in accordance with the instance structure you provide.

DSP programming function library supports four distinct algorithms of SVM classification for floating-point data, and the functions are introduced in following subsection.

3.9.1 SVM Linear Estimation

riscv_dsp_svm_linear_est_f32

Prototype:

```
void riscv_dsp_svm_linear_est_f32(const riscv_dsp_svm_linear_f32_t
*instance, const float32_t *src, int32_t *result)
```

Parameter:

- **riscv_dsp_svm_linear_f32_t** instance structure of the floating-point SVM linear estimator, defined as follows:

```
typedef struct
{
    uint32_t numofvec;
    uint32_t dimofvec;
    float32_t intercept;
    const float32_t *dualcoe;
    const float32_t *vec;
    const int32_t *class;
} riscv_dsp_svm_linear_f32_t;
```

Where

- **numofvec** number of support vectors
- **dimofvec** dimension of vector space
- **intercept** intercept
- ***dualcoe** dual coefficients
- ***vec** support vectors
- ***class** the 2 classes of SVM

- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *result` decision value

Return Value:

None

Example:

```
#define numofvec 3
#define dimofvec 4
#define intercept 0.1
float32_t src[dimofvec] = {1.0, 0.5, 0.4, -0.1};
float32_t dualcoe[numofvec] = {0.40, 0.10, 0.24};
float32_t vec[dimofvec * numofvec] = {0.40, 0.10, 0.24, -0.40, -0.34,
0.20, 0.06, 0.28, -0.04, -0.20, 0.08, 0.40};
int32_t class[2] = {3, 7};
int32_t *result;
riscv_dsp_svm_linear_f32_t instance = {numofvec, dimofvec,
intercept, dualcoe, vec, class};
riscv_dsp_svm_linear_est_f32(&instance, src, result);
```

3.9.2 SVM Radial Basis Function (RBF) Estimation

`riscv_dsp_svm_rbf_est_f32`

Prototype:

```
void riscv_dsp_svm_rbf_est_f32(const riscv_dsp_svm_rbf_f32_t
*instance, const float32_t *src, int32_t *result)
```

Parameter:

- `riscv_dsp_svm_rbf_f32_t` instance structure of the floating-point SVM RBF estimator, defined as follows:

```
typedef struct
{
    uint32_t numofvec;
    uint32_t dimofvec;
    float32_t intercept;
```

```

    const float32_t *dualcoe;
    const float32_t *vec;
    const int32_t *class;
    float32_t gamma;
} riscv_dsp_svm_rbf_f32_t;

```

Where

- `numofvec` number of support vectors
- `dimofvec` dimension of vector space
- `intercept` intercept
- `*dualcoe` dual coefficients
- `*vec` support vectors
- `*class` the 2 classes of SVM
- `gamma` gamma factor
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *result` decision value

Return Value:

None

Example:

```

#define numofvec 3
#define dimofvec 4
#define intercept 0.1
#define gamma 0.2
float32_t src[dimofvec] = {1.0, 0.5, 0.4, -0.1};
float32_t dualcoe[numofvec] = {0.40, 0.10, 0.24};
float32_t vec[dimofvec * numofvec] = {0.40, 0.10, 0.24, -0.40, -0.34,
0.20, 0.06, 0.28, -0.04, -0.20, 0.08, 0.40};
int32_t class[2] = {2, 5};
int32_t *result;

riscv_dsp_svm_rbf_f32_t instance = {numofvec, dimofvec, intercept,
dualcoe, vec, class, gamma};

riscv_dsp_svm_rbf_est_f32(&instance, src, result);

```

3.9.3 SVM Polynomial Estimation

riscv_dsp_svm_poly_est_f32

Prototype:

```
void riscv_dsp_svm_poly_est_f32(const riscv_dsp_svm_poly_f32_t
*instance, const float32_t *src, int32_t *result)
```

Parameter:

- `riscv_dsp_svm_poly_f32_t` instance structure of the floating-point SVM polynomial estimator, defined as follows:

```
typedef struct
{
    uint32_t numofvec;
    uint32_t dimofvec;
    float32_t intercept;
    const float32_t *dualcoe;
    const float32_t *vec;
    const int32_t *class;
    int32_t exponent;
    float32_t coef0;
    float32_t gamma;
} riscv_dsp_svm_poly_f32_t;
```

Where

- `numofvec` number of support vectors
- `dimofvec` dimension of vector space
- `intercept` intercept
- `*dualcoe` dual coefficients
- `*vec` support vectors
- `*class` the 2 classes of SVM
- `exponent` polynomial exponent
- `coef0` independent constant
- `gamma` gamma factor
- `[in] *instance` pointer of the instance structure

- `[in] *src` pointer of the input vector
- `[out] *result` decision value

Return Value:

None

Example:

```
#define numofvec 3
#define dimofvec 4
#define intercept 0.1
#define gamma 0.2
#define exponent 3
#define coef0 0.3

float32_t src[dimofvec] = {1.0, 0.5, 0.4, -0.1};
float32_t dualcoe[numofvec] = {0.40, 0.10, 0.24};
float32_t vec[dimofvec * numofvec] = {0.40, 0.10, 0.24, -0.40, -0.34,
0.20, 0.06, 0.28, -0.04, -0.20, 0.08, 0.40};
int32_t class[2] = {-1, 4};
int32_t *result;

riscv_dsp_svm_poly_f32_t instance = {numofvec, dimofvec, intercept,
dualcoe, vec, class, exponent, coef0, gamma};
riscv_dsp_svm_poly_est_f32(&instance, src, result);
```

3.9.4 SVM Sigmoid Estimation

`riscv_dsp_svm_sigmoid_est_f32`

Prototype:

```
void riscv_dsp_svm_sigmoid_est_f32(const
riscv_dsp_svm_sigmoid_f32_t *instance, const float32_t *src, int32_t
*result)
```

Parameter:

- `riscv_dsp_svm_sigmoid_f32_t` instance structure of the floating-point SVM sigmoid estimator, defined as follows:

```
typedef struct
{
```

```
uint32_t numofvec;  
uint32_t dimofvec;  
float32_t intercept;  
const float32_t *dualcoe;  
const float32_t *vec;  
const int32_t *class;  
float32_t coef0;  
float32_t gamma;  
} riscv_dsp_svm_sigmoid_f32_t;
```

Where

- **numofvec** number of support vectors
- **dimofvec** dimension of vector space
- **intercept** intercept
- ***dualcoe** dual coefficients
- ***vec** support vectors
- ***class** the 2 classes of SVM
- **coef0** independent constant
- **gamma** gamma factor
- **[in] *instance** pointer of the instance structure
- **[in] *src** pointer of the input vector
- **[out] *result** decision value

Return Value:

None

Example:

```
#define numofvec 3  
#define dimofvec 4  
#define intercept 0.1  
#define gamma 0.2  
#define coef0 0.3  
float32_t src[dimofvec] = {1.0, 0.5, 0.4, -0.1};  
float32_t dualcoe[numofvec] = {0.40, 0.10, 0.24};
```

```

float32_t vec[dimofvec * numofvec] = {0.40, 0.10, 0.24, -0.40, -0.34,
0.20, 0.06, 0.28, -0.04, -0.20, 0.08, 0.40};

int32_t class[2] = {-1, 4};

int32_t *result;

riscv_dsp_svm_sigmoid_f32_t instance = {numofvec, dimofvec,
intercept, dualcoe, vec, class, coef0, gamma};

riscv_dsp_svm_sigmoid_est_f32(&instance, src, result);

```

3.10 Distance Functions

This set of functions computes the distances between two vectors.

DSP programming function library supports distinct distance functions for floating-point and Boolean values, and these functions are introduced in the following subsection.

3.10.1 Float Distance

This set of functions computes distances between two vectors of float values.

The following example applies to all RiscV_AE350_SOC float distance functions except [riscv_dsp_dist_minkowski_f32](#), whose difference is shown in [riscv_dsp_dist_minkowski_f32](#).

```

#define size 4

float32_t src1[size] = {1.0, 0.5, 0.4, -0.1};
float32_t src2[size] = {0.40, -0.2, 0.24, 0.4};
float32_t dist_out;

dist_out = riscv_dsp_dist_bray_curtis_f32(src1, src2, size);
dist_out = riscv_dsp_dist_canberra_f32(src1, src2, size);
dist_out = riscv_dsp_dist_chebyshev_f32(src1, src2, size);
dist_out = riscv_dsp_dist_city_block_f32(src1, src2, size);
dist_out = riscv_dsp_dist_corr_f32(src1, src2, size);
dist_out = riscv_dsp_dist_cos_f32(src1, src2, size);
dist_out = riscv_dsp_dist_euclidean_f32(src1, src2, size);

```

riscv_dsp_dist_bray_curtis_f32

Prototype:

```
float32_t riscv_dsp_dist_bray_curtis_f32 (const float32_t *src1, const
```

`float32_t *src2, uint32_t size)`

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[in] size` size of elements in a vector

Return Value:

Distance value

`riscv_dsp_dist_canberra_f32`

Prototype:

`float32_t riscv_dsp_dist_canberra_f32 (const float32_t *src1, const float32_t *src2, uint32_t size)`

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[in] size` size of elements in a vector

Return Value:

Distance value

`riscv_dsp_dist_chebyshev_f32`

Prototype:

`float32_t riscv_dsp_dist_chebyshev_f32 (const float32_t *src1, const float32_t *src2, uint32_t size)`

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[in] size` size of elements in a vector

Return Value:

Distance value

`riscv_dsp_dist_city_block_f32`

Prototype:

```
float32_t riscv_dsp_dist_city_block_f32 (const float32_t *src1, const
float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size size of elements in a vector

Return Value:

Distance value

riscv_dsp_dist_corr_f32

Prototype:

```
float32_t riscv_dsp_dist_corr_f32 (const float32_t *src1, const
float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size size of elements in a vector

Return Value:

Distance value

riscv_dsp_dist_cos_f32

Prototype:

```
float32_t riscv_dsp_dist_cos_f32 (const float32_t *src1, const
float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] size size of elements in a vector

Return Value:

Distance value

riscv_dsp_dist_euclidean_f32**Prototype:**

```
float32_t riscv_dsp_dist_euclidean_f32 (const float32_t *src1, const
float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1* pointer of the first input vector
- [in] *src2* pointer of the second input vector
- [in] *size* size of elements in a vector

Return Value:

Distance value

riscv_dsp_dist_jensen_shannon_f32**Prototype:**

```
float32_t riscv_dsp_dist_jensen_shannon_f32 (const float32_t *src1,
const float32_t *src2, uint32_t size)
```

Parameter:

- [in] *src1* pointer of the first input vector
- [in] *src2* pointer of the second input vector
- [in] *size* size of elements in a vector

Return Value:

Distance value

riscv_dsp_dist_minkowski_f32**Prototype:**

```
float32_t riscv_dsp_dist_minkowski_f32 (const float32_t *src1, const
float32_t *src2, int32_t order, uint32_t size)
```

Parameter:

- [in] *src1* pointer of the first input vector
- [in] *src2* pointer of the second input vector
- [in] *order* distance order
- [in] *size* size of elements in a vector

Return Value:

Distance value

Example:

```
#define size 4
#define order 3
float32_t src1[size] = {1.0, 0.5, 0.4, -0.1};
float32_t src2[size] = {0.40, -0.2, 0.24, 0.4};
float32_t dist_out;
dist_out = riscv_dsp_dist_minkowski_f32(src1, src2, order, size);
```

3.10.2 Boolean Distance

This set of functions computes distances between two vectors of Boolean values. Each Boolean value is stored as 1 bit and has to be packed in a 32-bit word from most significant bit (MSB).

As shown below, given 120 Boolean numbers, you will need to pack each src input of the desired Boolean distance function with four 32-bit big-endian words. That is, put three 32 bits of Boolean value in three different words and the remaining 24 bits in the MSB of the fourth word in which Bit 0 to 7 are don't-care. This example applies to all RiscV_AE350_SOC Boolean distance functions.

Example:

```
#define numofbool 120
uint32_t src1[4] = {0x12322111, 0x24421111, 0x012ff241,
0x33c31100};
uint32_t src2[4] = {0x22222222, 0x11ea3222, 0x13465648,
0x5b351100};
float32_t dist_out;
dist_out = riscv_dsp_bdist_dice_u32_f32 (src1, src2, numofbool);
dist_out = riscv_dsp_bdist_hamming_u32_f32(src1, src2, numofbool);
dist_out = riscv_dsp_bdist_jaccard_u32_f32(src1, src2, numofbool);
dist_out = riscv_dsp_bdist_kulsinski_u32_f32(src1, src2, numofbool);
dist_out = riscv_dsp_bdist_rogers_tanimoto_u32_f32(src1, src2,
numofbool);
dist_out = riscv_dsp_bdist_russell_rao_u32_f32(src1, src2,
```

```
numofbool);  
    dist_out = riscv_dsp_bdist_sokal_michener_u32_f32(src1, src2,  
numofbool);  
    dist_out = riscv_dsp_bdist_sokal_sneath_u32_f32(src1, src2,  
numofbool);  
    dist_out = riscv_dsp_bdist_yule_u32_f32(src1, src2, numofbool);  
riscv_dsp_bdist_dice_u32_f32
```

Prototype:

```
float32_t riscv_dsp_bdist_dice_u32_f32 (const uint32_t *src1, const  
uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] numofbool number of booleans in a vector

Return Value:

Distance value

```
riscv_dsp_bdist_hamming_u32_f32
```

Prototype:

```
float32_t riscv_dsp_bdist_hamming_u32_f32 (const uint32_t *src1,  
const uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] numofbool number of booleans in a vector

Return Value:

Distance value

```
riscv_dsp_bdist_jaccard_u32_f32
```

Prototype:

```
float32_t riscv_dsp_bdist_jaccard_u32_f32 (const uint32_t *src1,  
const uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] **src1* pointer of the first input vector
- [in] **src2* pointer of the second input vector
- [in] *numofbool* number of booleans in a vector

Return Value:

Distance value

riscv_dsp_bdist_kulsinski_u32_f32

Prototype:

`float32_t riscv_dsp_bdist_kulsinski_u32_f32 (const uint32_t *src1, const uint32_t *src2, uint32_t nomofbool)`

Parameter:

- [in] **src1* pointer of the first input vector
- [in] **src2* pointer of the second input vector
- [in] *numofbool* number of booleans in a vector

Return Value:

Distance value

riscv_dsp_bdist_rogers_tanimoto_u32_f32

Prototype:

`float32_t riscv_dsp_bdist_rogers_tanimoto_u32_f32 (const uint32_t *src1, const uint32_t *src2, uint32_t nomofbool)`

Parameter:

- [in] **src1* pointer of the first input vector
- [in] **src2* pointer of the second input vector
- [in] *numofbool* number of booleans in a vector

Return Value:

Distance value

riscv_dsp_bdist_russell_rao_u32_f32

Prototype:

```
float32_t riscv_dsp_bdist_russell_rao_u32_f32 (const uint32_t *src1,  
const uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] nomofbool number of booleans in a vector

Return Value:

Distance value

riscv_dsp_bdist_sokal_michener_u32_f32

Prototype:

```
float32_t riscv_dsp_bdist_sokal_michener_u32_f32 (const uint32_t  
*src1, const uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] nomofbool number of booleans in a vector

Return Value:

Distance value

riscv_dsp_bdist_sokal_sneath_u32_f32

Prototype:

```
float32_t riscv_dsp_bdist_sokal_sneath_u32_f32 (const uint32_t  
*src1, const uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- [in] *src1 pointer of the first input vector
- [in] *src2 pointer of the second input vector
- [in] nomofbool number of booleans in a vector

Return Value:

Distance value

`riscv_dsp_bdist_yule_u32_f32`

Prototype:

```
float32_t riscv_dsp_bdist_yule_u32_f32 (const uint32_t *src1, const
uint32_t *src2, uint32_t nomofbool)
```

Parameter:

- `[in] *src1` pointer of the first input vector
- `[in] *src2` pointer of the second input vector
- `[in] numofbool` number of booleans in a vector

Return Value:

Distance value

3.11 Sort Functions

3.11.1 Generic Sort Functions

The generic sort function sorts elements of a vector by the algorithm and sorting order specified in its instance structure. The algorithms to be chosen from to perform the generic sorting include bitonic sort, bubble sort, heap sort, insertion sort, quick sort and selection sort.

DSP programming function library only supports the generic sort function for floating-point data. The function and its instance structures are introduced in the following subsection.

`riscv_dsp_sort_init_f32`

Prototype:

```
void riscv_dsp_sort_init_f32 (riscv_dsp_sort_f32_t * instance,
riscv_dsp_sort_alg alg, riscv_dsp_sort_order order)
```

Parameter:

- `riscv_dsp_sort_f32_t` instance structure of the floating-point generic sorting, defined as follows:

```
typedef struct
{
    riscv_dsp_sort_alg alg;
    riscv_dsp_sort_order order;
```

```
} riscv_dsp_sort_f32_t;
```

Where

- `alg` desired sorting algorithm
- `order` desired sorting order
- `[in, out] *instance` pointer of the instance structure
- `[in] alg` desired sorting algorithm
- `[in] order` desired sorting order

Note!

- This function has to be called to initialize the instance structure before the function

`riscv_dsp_sort_f32` is executed. See `riscv_dsp_sort_f32` for code examples.

- The possible sorting algorithms for the generic sorting include:
 - `RISCV_DSP_SORT_BITONIC` bitonic sort
 - `RISCV_DSP_SORT_BUBBLE` bubble sort
 - `RISCV_DSP_SORT_HEAP` heap sort
 - `RISCV_DSP_SORT_INSERTION` insertion sort
 - `RISCV_DSP_SORT_QUICK` quick sort
 - `RISCV_DSP_SORT_SELECTION` selection sort
- The possible sorting orders for the generic sorting include:
 - `RISCV_DSP_SORT_DESCENDING` descending order
 - `RISCV_DSP_SORT_ASCENDING` ascending order

`riscv_dsp_sort_f32`

Prototype:

```
void riscv_dsp_sort_f32 (const riscv_dsp_sort_f32_t * instance,
float32_t * src, float32_t * dst, uint32_t size)
```

Parameter:

- `riscv_dsp_sort_f32_t` instance structure of the floating-point generic sorting, defined as follows:

```
typedef struct
{
    riscv_dsp_sort_alg alg;
    riscv_dsp_sort_order order;
} riscv_dsp_sort_f32_t;
```

Where

- `alg` desired sorting algorithm
- `order` desired sorting order
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Note!

- The possible sorting algorithms for the generic sorting include:
 - `RISCV_DSP_SORT_BITONIC` bitonic sort
 - `RISCV_DSP_SORT_BUBBL` bubble sort
 - `RISCV_DSP_SORT_HEAP` heap sort
 - `RISCV_DSP_SORT_INSERTION` insertion sort
 - `RISCV_DSP_SORT_QUICK` quick sort
 - `RISCV_DSP_SORT_SELECTION` selection sort
- The possible sorting algorithms for the generic sorting include:
 - `RISCV_DSP_SORT_DESCENDING` descending order
 - `RISCV_DSP_SORT_ASCENDING` ascending order
- To ensure correct results, you must initialize the instance structure with the function `riscv_dsp_sort_f32` before using this function `riscv_dsp_sort_init_f32`.

Example:

1. With the input size as 100, sorting order as descending and sorting algorithm as bubble sort, the code example of generic sorting is as follows:

```
#define size 100
riscv_dsp_sort_f32_t *instance;
float32_t src[size] = {...};
float32_t dst[size];
riscv_dsp_sort_init_f32(instance, RISCV_DSP_SORT_BUBBLE,
RISCV_DSP_SORT_DESCENDING);
riscv_dsp_sort_f32(instance, src, dst, size);
```

2. With the input size as 100, sorting order as ascending and sorting algorithm as quick sort, the code example of generic sorting is as follows:

```
#define size 100
riscv_dsp_sort_f32_t *instance;
```

```
float32_t src[size] = {...};
float32_t dst[size];
riscv_dsp_sort_init_f32(instance, RISCV_DSP_SORT_QUICK,
RISCV_DSP_SORT_ASCENDING);
riscv_dsp_sort_f32(instance, src, dst, size);
```

3.11.2 Merge Sort Functions

The merge sort function sorts elements of a vector by the sorting order specified in its instance structure and a divide and conquer algorithm. The divide and conquer algorithm divides the input array in sublists and conquer them to produce longer sorted sublists until there is only one list remaining. A buffer is required by the algorithm to store the conquered values.

DSP programming function library only supports the merge sort function for floating-point data. The function and its instance structures are introduced in the following subsection.

riscv_dsp_sort_merge_init_f32

Prototype:

```
void riscv_dsp_sort_merge_init_f32 (riscv_dsp_sort_merge_f32_t *
instance, riscv_dsp_sort_order order, float32_t * buf)
```

Parameter:

- `riscv_dsp_sort_merge_f32_t` instance structure of the floating-point merge sorting, defined as follows:

```
typedef struct
{
    riscv_dsp_sort_order order;
    float32_t * buf;
} riscv_dsp_sort_f32_t;
```

Where

- `order` desired sorting order
- `*buf` pointer of the working buffer
- `[in, out] *instance` pointer of the instance structure
- `[in] order` desired sorting order
- `[in] *buf` pointer of the working buffer

Note!

- This function has to be called to initialize the instance structure before the function `riscv_dsp_sort_merge_f32` is executed. See `riscv_dsp_sort_merge_f32` for a code example.
- The possible sorting orders for the merge sorting include:
 - `RISCV_DSP_SORT_DESCENDING` descending order
 - `RISCV_DSP_SORT_ASCENDING` ascending order

riscv_dsp_sort_merge_f32**Prototype:**

```
void riscv_dsp_sort_merge_f32 (const riscv_dsp_sort_merge_f32_t *
instance, float32_t * src, float32_t * dst, uint32_t size)
```

Parameter:

- `riscv_dsp_sort_merge_f32_t` instance structure of the floating-point merge sorting, defined as follows:

```
typedef struct
{
    riscv_dsp_sort_order order;
    float32_t * buf;
} riscv_dsp_sort_f32_t;
```

Where

- `order` desired sorting order
- `*buf` pointer of the working buffer
- `[in] *instance` pointer of the instance structure
- `[in] *src` pointer of the input vector
- `[out] *dst` pointer of the output vector
- `[in] size` number of elements in a vector

Note!

- The possible sorting orders for the merge sorting include:
 - `RISCV_DSP_SORT_DESCENDING` descending order
 - `RISCV_DSP_SORT_ASCENDING` ascending order
- To ensure correct results, you must initialize the instance structure with the function `riscv_dsp_sort_merge_init_f32` before using this function `riscv_dsp_sort_merge_f32`.

Example:

With the input size as 100 and sorting order as descending, the code example of merge sorting is as follows:

```
#define size 100  
riscv_dsp_sort_merge_f32_t *instance;  
float32_t src[size] = {...};  
float32_t buf[size];  
float32_t dst[size];  
riscv_dsp_sort_merge_init_f32(instance,  
RISCV_DSP_SORT_DESCENDING, buf);  
riscv_dsp_sort_merge_f32(instance, src, dst, size);
```

4 Application Program

4.1 Program Description

DSP application program describes how to use DSP Software Programming Library and Math Software Programming Library to develop DSP systems quickly and easily.

Calling DSP Software Programming Library and Math Software Programming Library can implement the Radix-2 CFFT and CIFFT. These functions implement the well-known Cooley-Tukey algorithm to transform signals from the time domain to the frequency domain.

4.2 Application Program

RiscV_AE350_SOC provides DSP application programming:

...\ref_design\MCU_RefDesign\ae350_dsp

4.3 Program Running

The results of running the DSP application program are shown below.

```
It's a DSP demo.
randomly generated input array:
0: 0x0000 [0.00000000]
1: 0x0000 [0.00000000]
2: 0x0c7c [0.09753418]
3: 0x0c7c [0.09753418]
4: 0x187e [0.19134521]
5: 0x187e [0.19134521]
6: 0x238e [0.27777100]
7: 0x238e [0.27777100]
8: 0x2d41 [0.35354614]
9: 0x2d41 [0.35354614]
```

```
10: 0x3537 [0.41574097]
11: 0x3537 [0.41574097]
12: 0x3b21 [0.46194458]
13: 0x3b21 [0.46194458]
14: 0x3ec5 [0.49038696]
15: 0x3ec5 [0.49038696]
16: 0x3fff [0.49996948]
17: 0x3fff [0.49996948]
18: 0x3ec5 [0.49038696]
19: 0x3ec5 [0.49038696]
20: 0x3b21 [0.46194458]
21: 0x3b21 [0.46194458]
22: 0x3537 [0.41574097]
23: 0x3537 [0.41574097]
24: 0x2d41 [0.35354614]
25: 0x2d41 [0.35354614]
26: 0x238e [0.27777100]
27: 0x238e [0.27777100]
28: 0x187e [0.19134521]
29: 0x187e [0.19134521]
30: 0x0c7c [0.09753418]
31: 0x0c7c [0.09753418]
32: 0x0000 [0.00000000]
33: 0x0000 [0.00000000]
34: 0xf383 [-0.09756470]
35: 0xf383 [-0.09756470]
36: 0xe782 [-0.19134521]
37: 0xe782 [-0.19134521]
38: 0xdc71 [-0.27780151]
39: 0xdc71 [-0.27780151]
40: 0xd2bf [-0.35354614]
41: 0xd2bf [-0.35354614]
42: 0xcac9 [-0.41574097]
43: 0xcac9 [-0.41574097]
44: 0xc4df [-0.46194458]
45: 0xc4df [-0.46194458]
46: 0xc13b [-0.49038696]
47: 0xc13b [-0.49038696]
48: 0xc000 [-0.50000000]
49: 0xc000 [-0.50000000]
50: 0xc13b [-0.49038696]
51: 0xc13b [-0.49038696]
52: 0xc4df [-0.46194458]
53: 0xc4df [-0.46194458]
```

```
54: 0xcac9 [-0.41574097]
55: 0xcac9 [-0.41574097]
56: 0xd2bf [-0.35354614]
57: 0xd2bf [-0.35354614]
58: 0xdc71 [-0.27780151]
59: 0xdc71 [-0.27780151]
60: 0xe782 [-0.19134521]
61: 0xe782 [-0.19134521]
62: 0xf383 [-0.09756470]
63: 0xf383 [-0.09756470]
```

after cFFT:

```
0: 0xffff [-0.00003052]
1: 0xffff [-0.00003052]
2: 0x0fff [0.12496948]
3: 0xefff [-0.12503052]
4: 0x0000 [0.00000000]
5: 0x0000 [0.00000000]
6: 0xffff [-0.00003052]
7: 0xffff [-0.00003052]
8: 0x0000 [0.00000000]
9: 0x0000 [0.00000000]
10: 0xffff [-0.00003052]
11: 0xffff [-0.00003052]
12: 0x0000 [0.00000000]
13: 0x0000 [0.00000000]
14: 0xffff [-0.00003052]
15: 0xffff [-0.00003052]
16: 0x0000 [0.00000000]
17: 0x0000 [0.00000000]
18: 0x0000 [0.00000000]
19: 0x0000 [0.00000000]
20: 0x0000 [0.00000000]
21: 0x0000 [0.00000000]
22: 0x0000 [0.00000000]
23: 0xffff [-0.00003052]
24: 0x0000 [0.00000000]
25: 0x0000 [0.00000000]
26: 0x0000 [0.00000000]
27: 0xffff [-0.00003052]
28: 0x0000 [0.00000000]
29: 0x0000 [0.00000000]
30: 0xffff [-0.00003052]
31: 0x0000 [0.00000000]
```

```

32: 0x0000 [0.00000000]
33: 0x0000 [0.00000000]
34: 0x0000 [0.00000000]
35: 0x0000 [0.00000000]
36: 0x0000 [0.00000000]
37: 0x0000 [0.00000000]
38: 0x0000 [0.00000000]
39: 0x0000 [0.00000000]
40: 0x0000 [0.00000000]
41: 0x0000 [0.00000000]
42: 0x0000 [0.00000000]
43: 0x0000 [0.00000000]
44: 0x0000 [0.00000000]
45: 0x0000 [0.00000000]
46: 0x0000 [0.00000000]
47: 0x0000 [0.00000000]
48: 0x0000 [0.00000000]
49: 0x0000 [0.00000000]
50: 0x0000 [0.00000000]
51: 0x0000 [0.00000000]
52: 0x0000 [0.00000000]
53: 0x0000 [0.00000000]
54: 0x0000 [0.00000000]
55: 0x0000 [0.00000000]
56: 0x0000 [0.00000000]
57: 0x0000 [0.00000000]
58: 0x0000 [0.00000000]
59: 0x0000 [0.00000000]
60: 0x0000 [0.00000000]
61: 0x0000 [0.00000000]
62: 0xf000 [-0.12500000]
63: 0x1000 [0.12500000]

```

```

***** output [scaleingout] | ** [golden] * | **** abs diff ****

```

```

0: 0xffff [-0.00390625] | [-0.00000045] | 0x007f [0.00390581]
1: 0xffff [-0.00390625] | [-0.00000045] | 0x007f [0.00390581]
2: 0x0fff [15.99609375] | [16.00000000] | 0x0080 [0.00390625]
3: 0xefff [-16.00390625] | [-15.99999809] | 0x0080 [0.00390816]
4: 0x0000 [ 0.00000000] | [-0.00000086] | 0x0000 [0.00000086]
5: 0x0000 [ 0.00000000] | [ 0.00000024] | 0x0000 [0.00000024]
6: 0xffff [-0.00390625] | [-0.00000048] | 0x007f [0.00390577]
7: 0xffff [-0.00390625] | [-0.00000012] | 0x007f [0.00390613]
8: 0x0000 [ 0.00000000] | [ 0.00000025] | 0x0000 [0.00000025]

```

```
9: 0x0000 [ 0.00000000] | [-0.00000040] | 0x0000 [0.00000040]
10: 0xffff [-0.00390625] | [ 0.00000012] | 0x0080 [0.00390637]
11: 0xffff [-0.00390625] | [ 0.00000034] | 0x0080 [0.00390659]
12: 0x0000 [ 0.00000000] | [ 0.00000015] | 0x0000 [0.00000015]
13: 0x0000 [ 0.00000000] | [-0.00000021] | 0x0000 [0.00000021]
14: 0xffff [-0.00390625] | [ 0.00000072] | 0x0080 [0.00390697]
15: 0xffff [-0.00390625] | [ 0.00000048] | 0x0080 [0.00390673]
16: 0x0000 [ 0.00000000] | [-0.00000030] | 0x0000 [0.00000030]
17: 0x0000 [ 0.00000000] | [ 0.00000090] | 0x0000 [0.00000090]
18: 0x0000 [ 0.00000000] | [-0.00000024] | 0x0000 [0.00000024]
19: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
20: 0x0000 [ 0.00000000] | [-0.00000040] | 0x0000 [0.00000040]
21: 0x0000 [ 0.00000000] | [ 0.00000035] | 0x0000 [0.00000035]
22: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
23: 0xffff [-0.00390625] | [-0.00000083] | 0x007f [0.00390542]
24: 0x0000 [ 0.00000000] | [-0.00000078] | 0x0000 [0.00000078]
25: 0x0000 [ 0.00000000] | [-0.00000013] | 0x0000 [0.00000013]
26: 0x0000 [ 0.00000000] | [ 0.00000022] | 0x0000 [0.00000022]
27: 0xffff [-0.00390625] | [-0.00000048] | 0x007f [0.00390577]
28: 0x0000 [ 0.00000000] | [ 0.00000010] | 0x0000 [0.00000010]
29: 0x0000 [ 0.00000000] | [-0.00000008] | 0x0000 [0.00000008]
30: 0xffff [-0.00390625] | [ 0.00000000] | 0x0080 [0.00390625]
31: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
32: 0x0000 [ 0.00000000] | [ 0.00000021] | 0x0000 [0.00000021]
33: 0x0000 [ 0.00000000] | [ 0.00000021] | 0x0000 [0.00000021]
34: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
35: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
36: 0x0000 [ 0.00000000] | [-0.00000008] | 0x0000 [0.00000008]
37: 0x0000 [ 0.00000000] | [ 0.00000010] | 0x0000 [0.00000010]
38: 0x0000 [ 0.00000000] | [-0.00000072] | 0x0000 [0.00000072]
39: 0x0000 [ 0.00000000] | [ 0.00000060] | 0x0000 [0.00000060]
40: 0x0000 [ 0.00000000] | [-0.00000013] | 0x0000 [0.00000013]
41: 0x0000 [ 0.00000000] | [-0.00000078] | 0x0000 [0.00000078]
42: 0x0000 [ 0.00000000] | [ 0.00000012] | 0x0000 [0.00000012]
43: 0x0000 [ 0.00000000] | [-0.00000034] | 0x0000 [0.00000034]
44: 0x0000 [ 0.00000000] | [ 0.00000035] | 0x0000 [0.00000035]
45: 0x0000 [ 0.00000000] | [-0.00000040] | 0x0000 [0.00000040]
46: 0x0000 [ 0.00000000] | [-0.00000072] | 0x0000 [0.00000072]
47: 0x0000 [ 0.00000000] | [-0.00000048] | 0x0000 [0.00000048]
48: 0x0000 [ 0.00000000] | [ 0.00000090] | 0x0000 [0.00000090]
49: 0x0000 [ 0.00000000] | [-0.00000030] | 0x0000 [0.00000030]
50: 0x0000 [ 0.00000000] | [ 0.00000024] | 0x0000 [0.00000024]
51: 0x0000 [ 0.00000000] | [ 0.00000000] | 0x0000 [0.00000000]
52: 0x0000 [ 0.00000000] | [-0.00000021] | 0x0000 [0.00000021]
```

```

53: 0x0000 [ 0.00000000] | [ 0.00000015] | 0x0000 [0.00000015]
54: 0x0000 [ 0.00000000] | [ 0.00000024] | 0x0000 [0.00000024]
55: 0x0000 [ 0.00000000] | [ 0.00000036] | 0x0000 [0.00000036]
56: 0x0000 [ 0.00000000] | [-0.00000040] | 0x0000 [0.00000040]
57: 0x0000 [ 0.00000000] | [ 0.00000025] | 0x0000 [0.00000025]
58: 0x0000 [ 0.00000000] | [-0.00000046] | 0x0000 [0.00000046]
59: 0x0000 [ 0.00000000] | [-0.00000048] | 0x0000 [0.00000048]
60: 0x0000 [ 0.00000000] | [ 0.00000024] | 0x0000 [0.00000024]
61: 0x0000 [ 0.00000000] | [-0.00000086] | 0x0000 [0.00000086]
62: 0xf000 [-16.00000000] | [-15.99999809] | 0x0000 [0.00000191]
63: 0x1000 [16.00000000] | [16.00000000] | 0x0000 [0.00000000]
after CFFT_RD2, maxdiff= 0x0080 [0.00390816]

```

```

-----
MAE is 0.00079375, RMSD is 0.00176055, NRMSD is 0.00005502,
MAXDIFF is 0.00390816, SNR is 67.12823486
CFFT_RD2 out scale up by 64

```

```

-----
randomly generated input array:

```

```

0: 0x0000 [0.00000000]
1: 0x0000 [0.00000000]
2: 0x1000 [0.12500000]
3: 0xf000 [-0.12500000]
4: 0x0000 [0.00000000]
5: 0x0000 [0.00000000]
6: 0x0000 [0.00000000]
7: 0x0000 [0.00000000]
8: 0x0000 [0.00000000]
9: 0x0000 [0.00000000]
10: 0x0000 [0.00000000]
11: 0x0000 [0.00000000]
12: 0x0000 [0.00000000]
13: 0x0000 [0.00000000]
14: 0x0000 [0.00000000]
15: 0x0000 [0.00000000]
16: 0x0000 [0.00000000]
17: 0x0000 [0.00000000]
18: 0x0000 [0.00000000]
19: 0x0000 [0.00000000]
20: 0x0000 [0.00000000]
21: 0x0000 [0.00000000]
22: 0x0000 [0.00000000]
23: 0x0000 [0.00000000]
24: 0x0000 [0.00000000]
25: 0x0000 [0.00000000]

```

```
26: 0x0000 [0.00000000]
27: 0x0000 [0.00000000]
28: 0x0000 [0.00000000]
29: 0x0000 [0.00000000]
30: 0x0000 [0.00000000]
31: 0x0000 [0.00000000]
```

after CFFT:

```
0: 0x0000 [0.00000000]
1: 0x0000 [0.00000000]
2: 0x0c7a [0.09747314]
3: 0x0c7e [0.09759521]
4: 0x187c [0.19128418]
5: 0x1880 [0.19140625]
6: 0x238a [0.27764893]
7: 0x2392 [0.27789307]
8: 0x2d40 [0.35351563]
9: 0x2d42 [0.35357666]
10: 0x3532 [0.41558838]
11: 0x353a [0.41583252]
12: 0x3b1c [0.46179199]
13: 0x3b22 [0.46197510]
14: 0x3ec2 [0.49029541]
15: 0x3eca [0.49053955]
16: 0x3ffe [0.49993896]
17: 0x4000 [0.50000000]
18: 0x3ec4 [0.49035645]
19: 0x3ec6 [0.49041748]
20: 0x3b1e [0.46185303]
21: 0x3b20 [0.46191406]
22: 0x3538 [0.41577148]
23: 0x3536 [0.41571045]
24: 0x2d40 [0.35351563]
25: 0x2d40 [0.35351563]
26: 0x2390 [0.27783203]
27: 0x238e [0.27777100]
28: 0x187c [0.19128418]
29: 0x187c [0.19128418]
30: 0x0c80 [0.09765625]
31: 0x0c7a [0.09747314]
32: 0x0000 [0.00000000]
33: 0x0000 [0.00000000]
34: 0xf386 [-0.09747314]
35: 0xf382 [-0.09759521]
```

```

36: 0xe784 [-0.19128418]
37: 0xe780 [-0.19140625]
38: 0xdc76 [-0.27764893]
39: 0xdc6e [-0.27789307]
40: 0xd2c0 [-0.35351563]
41: 0xd2be [-0.35357666]
42: 0xcace [-0.41558838]
43: 0xcac6 [-0.41583252]
44: 0xc4e4 [-0.46179199]
45: 0xc4de [-0.46197510]
46: 0xc13e [-0.49029541]
47: 0xc136 [-0.49053955]
48: 0xc002 [-0.49993896]
49: 0xc000 [-0.50000000]
50: 0xc13c [-0.49035645]
51: 0xc13a [-0.49041748]
52: 0xc4e2 [-0.46185303]
53: 0xc4e0 [-0.46191406]
54: 0xcac8 [-0.41577148]
55: 0xcaca [-0.41571045]
56: 0xd2c0 [-0.35351563]
57: 0xd2c0 [-0.35351563]
58: 0xdc70 [-0.27783203]
59: 0xdc72 [-0.27777100]
60: 0xe784 [-0.19128418]
61: 0xe784 [-0.19128418]
62: 0xf380 [-0.09765625]
63: 0xf386 [-0.09747314]

```

```

-----
**** output [scaleingout] | ** [golden] * | **** abs diff ****

```

```

0: 0x0000 [ 0.00000000] | [-0.00000001] | 0x0000 [0.00000001]
1: 0x0000 [ 0.00000000] | [-0.00000001] | 0x0000 [0.00000001]
2: 0x0c7a [ 0.19494629] | [ 0.19509025] | 0x0004 [0.00014396]
3: 0x0c7e [ 0.19519043] | [ 0.19509035] | 0x0003 [0.00010008]
4: 0x187c [ 0.38256836] | [ 0.38268337] | 0x0003 [0.00011501]
5: 0x1880 [ 0.38281250] | [ 0.38268340] | 0x0004 [0.00012910]
6: 0x238a [ 0.55529785] | [ 0.55557024] | 0x0008 [0.00027239]
7: 0x2392 [ 0.55578613] | [ 0.55557019] | 0x0007 [0.00021595]
8: 0x2d40 [ 0.70703125] | [ 0.70710671] | 0x0002 [0.00007546]
9: 0x2d42 [ 0.70715332] | [ 0.70710671] | 0x0001 [0.00004661]
10: 0x3532 [ 0.83117676] | [ 0.83146954] | 0x0009 [0.00029278]
11: 0x353a [ 0.83166504] | [ 0.83146954] | 0x0006 [0.00019550]
12: 0x3b1c [ 0.92358398] | [ 0.92387950] | 0x0009 [0.00029552]
13: 0x3b22 [ 0.92395020] | [ 0.92387950] | 0x0002 [0.00007069]

```

```
14: 0x3ec2 [ 0.98059082] | [ 0.98078519] | 0x0006 [0.00019437]
15: 0x3eca [ 0.98107910] | [ 0.98078519] | 0x0009 [0.00029391]
16: 0x3ffe [ 0.99987793] | [ 1.00000000] | 0x0004 [0.00012207]
17: 0x4000 [ 1.00000000] | [ 1.00000000] | 0x0000 [0.00000000]
18: 0x3ec4 [ 0.98071289] | [ 0.98078525] | 0x0002 [0.00007236]
19: 0x3ec6 [ 0.98083496] | [ 0.98078513] | 0x0001 [0.00004983]
20: 0x3b1e [ 0.92370605] | [ 0.92387938] | 0x0005 [0.00017333]
21: 0x3b20 [ 0.92382813] | [ 0.92387938] | 0x0001 [0.00005126]
22: 0x3538 [ 0.83154297] | [ 0.83146954] | 0x0002 [0.00007343]
23: 0x3536 [ 0.83142090] | [ 0.83146954] | 0x0001 [0.00004864]
24: 0x2d40 [ 0.70703125] | [ 0.70710677] | 0x0002 [0.00007552]
25: 0x2d40 [ 0.70703125] | [ 0.70710677] | 0x0002 [0.00007552]
26: 0x2390 [ 0.55566406] | [ 0.55557019] | 0x0003 [0.00009388]
27: 0x238e [ 0.55554199] | [ 0.55557019] | 0x0000 [0.00002819]
28: 0x187c [ 0.38256836] | [ 0.38268328] | 0x0003 [0.00011492]
29: 0x187c [ 0.38256836] | [ 0.38268328] | 0x0003 [0.00011492]
30: 0x0c80 [ 0.19531250] | [ 0.19509026] | 0x0007 [0.00022224]
31: 0x0c7a [ 0.19494629] | [ 0.19509029] | 0x0004 [0.00014400]
32: 0x0000 [ 0.00000000] | [-0.00000007] | 0x0000 [0.00000007]
33: 0x0000 [ 0.00000000] | [-0.00000007] | 0x0000 [0.00000007]
34: 0xf386 [-0.19494629] | [-0.19509040] | 0x0004 [0.00014411]
35: 0xf382 [-0.19519043] | [-0.19509053] | 0x0003 [0.00009990]
36: 0xe784 [-0.38256836] | [-0.38268337] | 0x0003 [0.00011501]
37: 0xe780 [-0.38281250] | [-0.38268340] | 0x0004 [0.00012910]
38: 0xdc76 [-0.55529785] | [-0.55557024] | 0x0008 [0.00027239]
39: 0xdc6e [-0.55578613] | [-0.55557019] | 0x0007 [0.00021595]
40: 0xd2c0 [-0.70703125] | [-0.70710683] | 0x0002 [0.00007558]
41: 0xd2be [-0.70715332] | [-0.70710683] | 0x0001 [0.00004649]
42: 0xcace [-0.83117676] | [-0.83146977] | 0x0009 [0.00029302]
43: 0xcac6 [-0.83166504] | [-0.83146977] | 0x0006 [0.00019526]
44: 0xc4e4 [-0.92358398] | [-0.92387962] | 0x0009 [0.00029564]
45: 0xc4de [-0.92395020] | [-0.92387962] | 0x0002 [0.00007057]
46: 0xc13e [-0.98059082] | [-0.98078507] | 0x0006 [0.00019425]
47: 0xc136 [-0.98107910] | [-0.98078519] | 0x0009 [0.00029391]
48: 0xc002 [-0.99987793] | [-1.00000000] | 0x0004 [0.00012207]
49: 0xc000 [-1.00000000] | [-1.00000000] | 0x0000 [0.00000000]
50: 0xc13c [-0.98071289] | [-0.98078525] | 0x0002 [0.00007236]
51: 0xc13a [-0.98083496] | [-0.98078513] | 0x0001 [0.00004983]
52: 0xc4e2 [-0.92370605] | [-0.92387938] | 0x0005 [0.00017333]
53: 0xc4e0 [-0.92382813] | [-0.92387938] | 0x0001 [0.00005126]
54: 0xcac8 [-0.83154297] | [-0.83146942] | 0x0002 [0.00007355]
55: 0xcaca [-0.83142090] | [-0.83146954] | 0x0001 [0.00004864]
56: 0xd2c0 [-0.70703125] | [-0.70710653] | 0x0002 [0.00007528]
```

```
57: 0xd2c0 [-0.70703125] | [-0.70710653] | 0x0002 [0.00007528]
58: 0xdc70 [-0.55566406] | [-0.55557030] | 0x0003 [0.00009376]
59: 0xdc72 [-0.55554199] | [-0.55557030] | 0x0000 [0.00002831]
60: 0xe784 [-0.38256836] | [-0.38268340] | 0x0003 [0.00011504]
61: 0xe784 [-0.38256836] | [-0.38268340] | 0x0003 [0.00011504]
62: 0xf380 [-0.19531250] | [-0.19509020] | 0x0007 [0.00022230]
63: 0xf386 [-0.19494629] | [-0.19509023] | 0x0004 [0.00014395]
after CIFFT_RD2, maxdiff= 0x0009 [0.00029564]
```

```
-----
MAE is 0.00012192, RMSD is 0.00014900, NRMSD is 0.00007450,
MAXDIFF is 0.00029564, SNR is 73.52618408
CIFFT_RD2 out scale up by 2
```

```
-----
CIFFT_RD2 PASS.
```

```
Demo DSP Completed.
```

