NMSIS

Release 1.0.2-RC1

Nuclei

Aug 05, 2021

CONTENTS:

CHAPTER

ONE

NUCLEI MCU SOFTWARE INTERFACE STANDARD(NMSIS)

1.1 About NMSIS

The **NMSIS** is a vendor-independent hardware abstraction layer for micro-controllers that are based on Nuclei Processors[1].

The **NMSIS** defines generic tool interfaces and enables consistent device support. It provides simple software interfaces to the processor and the peripherals, simplifying software re-use, reducing the learning curve for micro-controller developers, and reducing the time to market for new devices.

1.2 NMSIS Components

NMSIS CORE All Nuclei N/NX Class Processors Standardized API for the Nuclei processor core and peripherals.

NMSIS DSP All Nuclei N/NX Class Processors DSP library collection with a lot of functions for various data types: fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit). Implementations optimized for the Nuclei Processors which has RISC-V SIMD instruction set.

NMSIS NN All Nuclei N/NX Class Processors Collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint Nuclei processor cores.

1.3 NMSIS Design

NMSIS is designed to help the Nuclei N/NX Class Processors processors in standardization. It enables consistent software layers and device support across a wide range of development tools and micro-controllers.

NMSIS is a lightweight software interface layer that tried to standardize common Nuclei processor-based SOC, and it didn't define any standard peripherals. The silicon industry can therefore support the wide variations of Nuclei processor-based devices with this common standard.

We can achieve the following benefits of **NMSIS**:

- **NMSIS** reduces the learning curve, development costs, and time-to-market. Developers can write software quicker through a variety of easy-to-use, standardized software interfaces.
- Consistent software interfaces improve the software portability and re-usability. Generic software libraries and interfaces provide consistent software framework.
- It provides interfaces for debug connectivity, debug peripheral views, software delivery, and device support to reduce time-to-market for new micro-controller deployment.

[1] https://doc.nucleisys.com/nuclei_spec

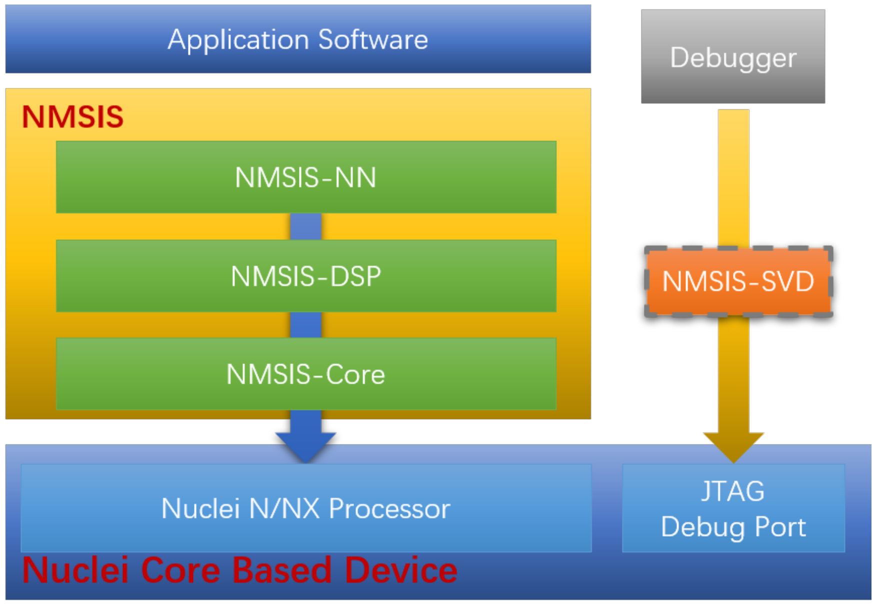


Fig. 1: NMSIS Design Diagram

- Being a compiler independent layer, it allows to use the compiler of your choice. Thus, it is supported by mainstream compilers.
- It enhances program debugging with peripheral information for debuggers.

1.4 How to Access

If you want to access the code of **NMSIS**, you can visit our opensource NMSIS Github Repository[2].

1.5 Coding Rules

The **NMSIS** uses the following essential coding rules and conventions:

- Compliant with ANSI C (C99) and C++ (C++03).
- Uses ANSI C standard data types defined in **stdint.h**.
- Variables and parameters have a complete data type.
- Expressions for *#define* constants are enclosed in parenthesis.

In addition, the **NMSIS** recommends the following conventions for identifiers:

- **CAPITAL** names to identify Core Registers, Peripheral Registers, and CPU Instructions.
- **CamelCase** names to identify function names and interrupt functions.
- **Namespace_** prefixes avoid clashes with user identifiers and provide functional groups (i.e. for peripherals, RTOS, or DSP Library).

The **NMSIS** is documented within the source files with:

[2] https://github.com/Nuclei-Software/NMSIS

- Comments that use the C or C++ style.
- Doxygen compliant comments, which provide:
  - brief function, variable, macro overview.
  - detailed description of the function, variable, macro.
  - detailed parameter explanation.
  - detailed information about return values.

1.6 Validation

Nuclei uses RISC-V GCC Compiler in the various tests of **NMSIS**, and if more compiler is added, it could be easily supported by following the **NMSIS** compiler independent layer. For each component, the section **Validation** describes the scope of the various verifications.

NMSIS components are compatible with a range of C and C++ language standards.

As **NMSIS** defines API interfaces and functions that scale to a wide range of processors and devices, the scope of the run-time test coverage is limited. However, several components are validated using dedicated test suites.

1.7 License

This **NMSIS** is modified based on open-source project **CMSIS** to match Nuclei requirements.

This **NMSIS** is provided free of charge by Nuclei under the Apache 2.0 License[3].

[3] http://www.apache.org/licenses/LICENSE-2.0

CHAPTER

TWO

NMSIS CORE

2.1 Overview

2.1.1 Introduction

NMSIS-Core implements the basic run-time system for a Nuclei N/NX Class Processors based device and gives the user access to the processor core and the device peripherals. In detail it defines:

- **Hardware Abstraction Layer (HAL)** for Nuclei processor registers with standardized definitions for the **CSR Registers**, **TIMER**, **ECLIC**, **PMP Registers**, **DSP Registers**, **FPU registers**, and **Core Access Functions**.
- **Standard core exception/interrupt names** to interface to system exceptions or interrupts without having compatibility issues.
- **Methods to organize header files** that makes it easy to learn new Nuclei micro-controller products and improve software portability. This includes naming conventions for device-specific interrupts.
- **Methods for system initialization** to be used by each Device vendor. For example, the standardized `SystemInit()` function is essential for configuring the clock system of the device.
- **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
- A variable `SystemCoreClock` to determine the **system clock frequency** which simplifies the setup the timer.

The following sections provide details about the **NMSIS-Core**:

- *Using NMSIS in Embedded Applications* (page 6) describes the project setup and shows a simple program example
- *NMSIS-Core Device Templates* (page 11) describes the files of the *NMSIS Core* (page 5) in detail and explains how to adapt template files provided by Nuclei to silicon vendor devices.
- *NMSIS CORE API* (page 56) describe the features and functions of the *Device Header File <device.h>* (page 46) in detail.
- core_api_register_type describe the data structures of the *Device Header File <device.h>* (page 46) in detail.

2.1.2 Processor Support

NMSIS have provided support for all the Nuclei N/NX Class Processors.

Nuclei ISA Spec:

- Nuclei Process Core Instruction Set Architecture Spec[4]

[4] https://doc.nucleisys.com/nuclei_spec

Nuclei N Class Processor Reference Manuals:

- N200 series[5]
- N300 series[6]
- N600 series[7]

Nuclei NX Class Processor Reference Manuals:

- NX600 series[8]

2.1.3 Toolchain Support

The *NMSIS-Core Device Templates* (page 11) provided by Nuclei have been tested and verified using these toolchains:

- GNU Toolchain for RISC-V modified by Nuclei

2.2 Using NMSIS in Embedded Applications

2.2.1 Introduction

To use the **NMSIS-Core**, the following files are added to the embedded application:

- *Startup File startup_<device>.S* (page 13), which provided asm startup code and vector table.
- *Interrupt and Exception Handling File: intexc_<device>.S* (page 20), which provided general exception handling code for non-vector interrupts and exceptions.
- *Device Linker Script: gcc_<device>.ld* (page 30), which provided linker script for the device.
- *System Configuration Files system_<device>.c and system_<device>.h* (page 35), which provided general device configuration (i.e. for clock and BUS setup).
- *Device Header File <device.h>* (page 46) gives access to processor core and all peripherals.

Note: The files *Startup File startup_<device>.S* (page 13), *Interrupt and Exception Handling File: intexc_<device>.S* (page 20), *Device Linker Script: gcc_<device>.ld* (page 30) and *System Configuration Files system_<device>.c and system_<device>.h* (page 35) may require application specific adaptations and therefore should be copied into the application project folder prior configuration.

The *Device Header File <device.h>* (page 46) is included in all source files that need device access and can be stored on a central include folder that is generic for all projects.

The *Startup File startup_<device>.S* (page 13) is executed right after device reset, it will do necessary stack pointer initialization, exception and interrupt entry configuration, then call `SystemInit()`, after system initialization, will return to assemble startup code and do c/c++ runtime initialization which includes data, bss section initialization, c++ runtime initialization, then it will call `main()` function in the application code.

In the *Interrupt and Exception Handling File: intexc_<device>.S* (page 20), it will contain all exception and interrupt vectors and implements a default function for every interrupt. It may also contain stack and heap configurations for the user application.

[5] https://www.nucleisys.com/product.php?site=n200
[6] https://www.nucleisys.com/product.php?site=n300
[7] https://www.nucleisys.com/product.php?site=n600
[8] https://www.nucleisys.com/product.php?site=nx600

The *System Configuration Files system_<device>.c and system_<device>.h* (page 35) performs the setup for the processor clock. The variable `SystemCoreClock` indicates the CPU clock speed. core_api_systick describes the minimum feature set. In addition the file may contain functions for the memory BUS setup and clock re-configuration.

The *Device Header File <device.h>* (page 46) is the central include file that the application programmer is using in the C source code. It provides the following features:

- core_api_periph_access provides a standardized register layout for all peripherals. Optionally functions for device-specific peripherals may be available.
- core_api_interrupt_exception can be accessed with standardized symbols and functions for the **ECLIC** are provided.
- core_api_core_intrinsic allow to access special instructions, for example for activating sleep mode or the NOP instruction.
- *Intrinsic Functions for SIMD Instructions* (page 135) provide access to the DSP-oriented instructions.
- core_api_systick function to configure and start a periodic timer interrupt.
- core_api_csr_access function to access the core csr registers.
- core_api_cache to access the I-CACHE and D-CACHE unit
- core_api_fpu to access the Floating point unit.
- core_api_pmp to access the Physical Memory Protection unit
- core_api_version_control which defines NMSIS release specific macros.
- core_api_compiler_control is compiler agnostic *#define* symbols for generic C/C++ source code

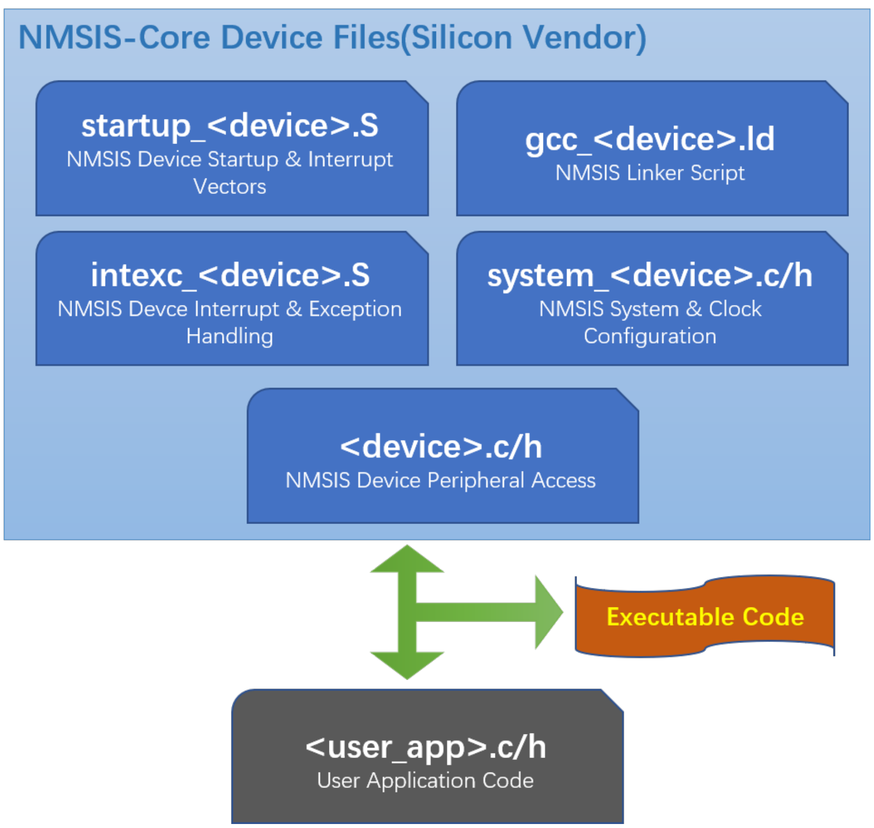


Fig. 1: NMSIS-Core User Files

The NMSIS-Core system files are device specific.

In addition, the *Startup File startup_<device>.S* (page 13) is also compiler vendor specific, currently only GCC version is provided. The versions provided by NMSIS are only generic templates. The adopted versions for a concrete device are typically provided by the device vendor through the according device familiy package.

For example, the following files are provided by the **GD32VF103** device family pack:

Table 1: Files provided by GD32VF103 device family pack

File	Description
./Device/Source/GCC/startup_gd32vf103.S	Startup File startup_<device>.S for the GD32VF103 device variants.
./Device/Source/GCC/intexc_gd32vf103.S	Exception and Interrupt Handling File intexc_<device>.S for the GD32VF103 device variants.
./Device/Source/GCC/gcc_gd32vf103.ld	Linker script File gcc_<device>.ld for the GD32VF103 device variants.
./Device/Source/system_gd32vf103.c	System Configuration File system_<device>.c for the GD32VF103 device families
./Device/Include/system_gd32vf103.h	System Configuration File system_<device>.h for the GD32VF103 device families
./Device/Include/gd32vf103.h	Device Header File <device.h> for the GD32VF103 device families.

Note: The silicon vendors create these device-specific NMSIS-Core files based on *NMSIS-Core Device Templates* (page 11) provided by Nuclei.

Thereafter, the functions described under *NMSIS CORE API* (page 56) can be used in the application.

2.2.2 Basic NMSIS Example

A typical example for using the NMSIS layer is provided below. The example is based on a GD32VF103 Device.

Listing 1: gd32vf103_example.c

```c
#include <gd32vf103.h>                              // File name depends on device used

uint32_t volatile msTicks;                          // Counter for millisecond Interval
#define SysTick_Handler     eclic_mtip_handler
#define CONFIG_TICKS        (SOC_TIMER_FREQ / 1000)

void SysTick_Handler (void) {                       // SysTick Interrupt Handler
  SysTick_Reload(CONFIG_TICKS);
  msTicks++;                                        // Increment Counter
}

void WaitForTick (void)  {
  uint32_t curTicks;

  curTicks = msTicks;                               // Save Current SysTick Value
  while (msTicks == curTicks)  {                    // Wait for next SysTick Interrupt
    __WFI ();                                       // Power-Down until next Event/Interrupt
  }
}

void TIMER0_UP_IRQHandler (void) {                  // Timer Interrupt Handler
  ;                                                 // Add user code here
}

void timer0_init(int frequency) {                   // Set up Timer (device specific)
  ECLIC_SetPriorityIRQ (TIMER0_UP_IRQn, 1);         // Set Timer priority
  ECLIC_EnableIRQ (TIMER0_UP_IRQn);                 // Enable Timer Interrupt
}


void Device_Initialization (void)  {                // Configure & Initialize MCU
  if (SysTick_Config (CONFIG_TICKS)) {
      ; // Handle Error
  }
  timer0_init ();                                   // setup device-specific timer
}

// The processor clock is initialized by NMSIS startup + system file
void main (void) {                                  // user application starts here
  Device_Initialization ();                         // Configure & Initialize MCU
  while (1)  {                                      // Endless Loop (the Super-Loop)
    __disable_irq ();                               // Disable all interrupts
    Get_InputValues ();                             // Read Values
    __enable_irq ();                                // Enable all interrupts
    Calculation_Response ();                        // Calculate Results
    Output_Response ();                             // Output Results
    WaitForTick ();                                 // Synchronize to SysTick Timer
  }
}
```

2.2.3 Using Interrupt and Exception/NMI

Nuclei processors provide **NMI(Non-Maskable Interrupt)**, **Exception**, **Vector Interrupt** and **Non-Vector Interrupt** features.

2.2.4 Using NMSIS with generic Nuclei Processors

Nuclei provides NMSIS-Core files for the supported Nuclei Processors and for various compiler vendors. These files can be used when standard Nuclei processors should be used in a project. The table below lists the folder and device names of the Nuclei processors.

Table 2: Folder and device names of the Nuclei processors

Folder	Processor	RISC-V	Description
./Device/Nuclei/NUCLEI_N	• N200 • N300 • N600	RV32	Contains Include and Source template files configured for the Nuclei N200/N300/N600 processor. The device name is NUCLEI_N and the name of the Device Header File <device.h> is <NUCLEI_N.h>.
./Device/Nuclei/NUCLEI_NX	NX600	RV64	Contains Include and Source template files configured for the Nuclei NX600 processor. The device name is NUCLEI_NX and the name of the Device Header File <device.h> is <NUCLEI_NX.h>.

2.2.5 Create generic Libraries with NMSIS

The NMSIS Processor and Core Peripheral files allow also to create generic libraries. The **NMSIS-DSP** Libraries are an example for such a generic library.

To build a generic library set the define **__NMSIS_GENERIC** and include the *nmsis_core.h* NMSIS CPU & Core Access header file for the processor.

The define **__NMSIS_GENERIC** disables device-dependent features such as the **SysTick timer** and the **Interrupt System**.

Example

The following code section shows the usage of the *nmsis_core.h* header files to build a generic library for N200, N300, N600, NX600.

One of these defines needs to be provided on the compiler command line.

By using this header file, the source code can access the functions for core_api_csr_access, core_api_core_intrinsic and *Intrinsic Functions for SIMD Instructions* (page 135).

Listing 2: core_generic.h

```
#define __NMSIS_GENERIC   // Disable Eclic and Systick functions
#include <nmsis_core.h>
```

2.3 NMSIS-Core Device Templates

2.3.1 Introduction

Nuclei supplies NMSIS-Core device template files for the all supported Nuclei N/NX Class Processors and various compiler vendors. Refer to the list of *supported toolchain* (page 6) for compliance.

These NMSIS-Core device template files include the following:

- Register names of the Core Peripherals and names of the Core Exception/Interrupt Vectors.
- Functions to access core peripherals, special CPU instructions and SIMD instructions
- Generic startup code and system configuration code.

The detailed file structure of the NMSIS-Core device templates is shown in the following picture.

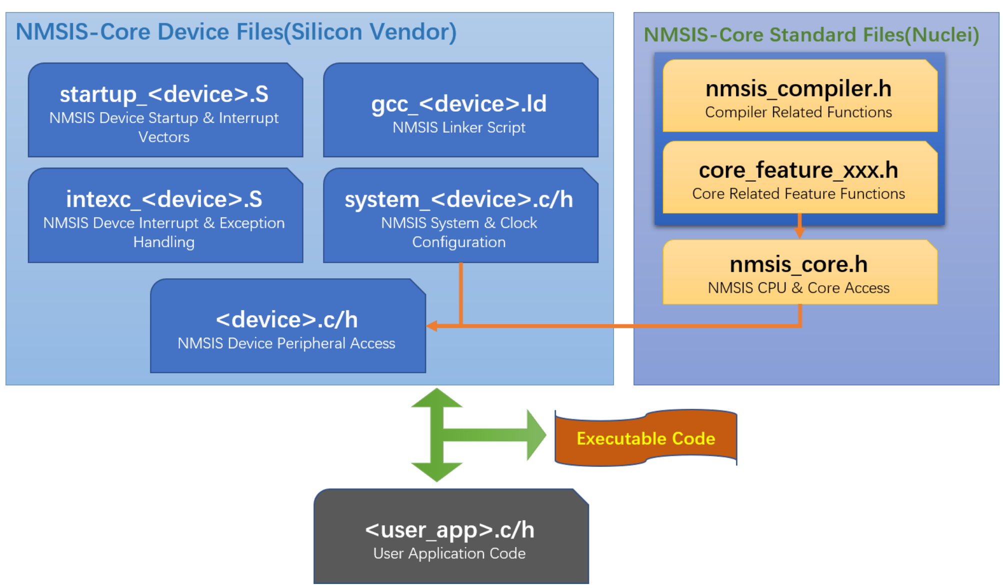


Fig. 2: NMSIS-Core Device Templates

2.3.2 NMSIS-Core Processor Files

The NMSIS-Core processor files provided by Nuclei are in the directory *NMSIS/Core/Include*.

These header files define all processor specific attributes do not need any modifications.

The *nmsis_core.h* defines the core peripherals and provides helper functions that access the core registers.

2.3.3 Device Examples

The NMSIS Software Pack defines several devices that are based on the Nuclei N/NX processors.

The device related NMSIS-Core files are in the directory *Device/Nuclei* and include NMSIS-Core processor file explained before.

The following sample devices are defined as below:

Table 3: Device Examples of Nuclei Processor

Family	Device	Description
Nuclei N	NUCLEI N Class	Nuclei N Class based device
Nuclei NX	NUCLEI NX Class	Nuclei NX Class based device

2.3.4 Template Files

To simplify the creation of NMSIS-Core device files, the following template files are provided that should be extended by the silicon vendor to reflect the actual device and device peripherals.

Silicon vendors add to these template files the following information:

- **Device Peripheral Access Layer** that provides definitions for device-specific peripherals.
- **Access Functions for Peripherals** (optional) that provides additional helper functions to access device-specific peripherals.
- **Interrupt vectors** in the startup file that are device specific.

Table 4: NMSIS-Core Device Template Files

Template File (Under ./Device/_Template_Vendor/Vendor/)	**Description**
Device/Source/GCC/startup_Device.S	Startup file template for GNU GCC RISC-V Embedded Compiler.
Device/Source/GCC/gcc_Device.ld	Link Script file template for GNU GCC RISC-V Embedded Compiler.
Device/Source/GCC/intexc_Device.S	Exception and Interrupt handling file template for GNU GCC RISC-V Embedded Compiler.
Device/Source/system_Device.c	Generic system_Device.c file for system configuration (i.e. processor clock and memory bus system).
Device/Include/Device.h	Generic device header file. Needs to be extended with the device-specific peripheral registers. Optionally functions that access the peripherals can be part of that file.
Device/Include/system_Device.h	Generic system device configuration include file.

Note: The template files for silicon vendors are placed under *./Device/_Template_Vendor/Vendor/*.

Please goto that folder to find the file list in the above table.

2.3.5 Adapt Template Files to a Device

The following steps describe how to adopt the template files to a specific device or device family.

Copy the complete all files in the template directory and replace:

- directory name `Vendor` with the abbreviation for the device vendor e.g.: **GD**.
- directory name `Device` with the specific device name e.g.: **GD32VF103**.
- in the file names `Device` with the specific device name e.g.: **GD32VF103**.

Each template file contains comments that start with **TODO**: that describe a required modification.

The template files contain place holders:

Table 5: Placeholders of Template files

Placeholder	Replaced with
<Device>	the specific device name or device family name; i.e. GD32VF103.
<DeviceInterrupt>	a specific interrupt name of the device; i.e. TIM1 for Timer 1.
<DeviceAbbreviation>	short name or abbreviation of the device family; i.e. GD32VF.
Nuclei-N#	the specific Nuclei Class name; i.e. Nuclei N or Nuclei NX.

2.3.6 Device Templates Explaination

The device configuration of the template files is described in detail on the following pages:

Startup File startup_<device>.S

The Startup File startup_<device>.S contains:

- The reset handler which is executed after CPU reset and typically calls the `SystemInit()` function.
- The setup values for the stack pointer SP.
- Exception vectors of the Nuclei Processor with weak functions that implement default routines.
- Interrupt vectors that are device specific with weak functions that implement default routines.

The processer level start flow is implemented in the *startup_<device>.S*. Detail description as below picture:

Stage1: Interrupt and Exception initialization

- Disable Interrupt
- Initialize GP, stack
- Initialize NMI entry and set default NMI handler
- Initialize Exception entry and set default exception handler
- Initialize vector table entry and set default interrupt handler
- Initialize Interrupt mode as ECLIC mode. (ECLIC mode is proposed. Default mode is CLINT mode)

Stage2: Hardware initialization

- Enable FPU if necessary
- Call user defined `SystemInit()` for system clock initialization.

Stage3: Section initialization

- Copy section, e.g. data section, text section if necessary.
- Clear Block Started by Symbol (BSS) section
- Call `__libc_fini_array` and `__libc_init_array` functions to do C library initialization

- Call `_premain_init` function to do initialization steps before main function
- Jump Main

The file exists for each supported toolchain and is the only toolchain specific NMSIS file.

To adapt the file to a new device only the interrupt vector table needs to be extended with the device-specific interrupt handlers.

The naming convention for the interrupt handler names are `eclic_<interrupt_name>_handler`.

This table needs to be consistent with `IRQn_Type` that defines all the IRQ numbers for each interrupt.

The following example shows the extension of the interrupt vector table for the GD32VF103 device family.

```
    .section .vtable

    .weak  eclic_msip_handler
    .weak  eclic_mtip_handler
    .weak  eclic_pmaf_handler
    /* Adjusted for GD32VF103 interrupt handlers */
    .weak  eclic_wwdgt_handler
    .weak  eclic_lvd_handler
    .weak  eclic_tamper_handler
        :    :
        :    :
    .weak  eclic_can1_ewmc_handler
    .weak  eclic_usbfs_handler

    .globl vector_base
    .type vector_base, @object
vector_base:
    /* Run in FlashXIP download mode */
    j _start                                                  /* 0: Reserved, Jump to _start when reset for vector table not remapped cases.*/
    .align LOG_REGBYTES                                       /*    Need to align 4 byte for RV32, 8 Byte for RV64 */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 1: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 2: Reserved */
    DECLARE_INT_HANDLER     eclic_msip_handler                /* 3: Machine software interrupt */
                    :          :
                    :          :
    /* Adjusted for Vendor Defined External Interrupts */
    DECLARE_INT_HANDLER     eclic_wwdgt_handler               /* 19: Window watchDog timer interrupt */

    DECLARE_INT_HANDLER     eclic_lvd_handler                 /* 20: LVD through EXTI line detect interrupt */
    DECLARE_INT_HANDLER     eclic_tamper_handler              /* 21: tamper through EXTI line detect */
                    :          :
                    :          :
    DECLARE_INT_HANDLER     eclic_can1_ewmc_handler           /* 85: CAN1 EWMC interrupt */
    DECLARE_INT_HANDLER     eclic_usbfs_handler               /* 86: USBFS global interrupt */
```

startup_Device.S Template File

Here provided a riscv-gcc template startup assemble code template file as below. The files for other compilers can slightly differ from this version.

```
/*
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/******************************************************************************
 * \file     startup_<Device>.S
 * \brief    NMSIS Nuclei N/NX Class Core based Core Device Startup File for
 *           Device <Device>
 * \version  V1.10
 * \date     30. July 2021
 *
 ******************************************************************************/

#include "riscv_encoding.h"

.macro DECLARE_INT_HANDLER  INT_HDL_NAME
#if defined(__riscv_xlen) && (__riscv_xlen == 32)
    .word \INT_HDL_NAME
#else
    .dword \INT_HDL_NAME
#endif
.endm

    /*
     * Put the interrupt vectors in this section according to vector remapped or not:
     * .vtable: vector table's LMA and VMA are the same, it is not remapped
     * .vtable_ilm: vector table's LMA and VMA are different, it is remapped, and
     *              VECTOR_TABLE_REMAPPED need to be defined
     */
#if defined(VECTOR_TABLE_REMAPPED)
    .section .vtable_ilm
#else
    .section .vtable
#endif

    .weak  eclic_msip_handler
    .weak  eclic_mtip_handler
    /* TODO: Adjust vendor interrupt handlers */
    .weak  eclic_irq19_handler
```

(continues on next page)

(continued from previous page)

```
    .weak  eclic_irq20_handler
    .weak  eclic_irq21_handler
    .weak  eclic_irq22_handler
    .weak  eclic_irq23_handler
    .weak  eclic_irq24_handler
    .weak  eclic_irq25_handler
    .weak  eclic_irq26_handler
    .weak  eclic_irq27_handler
    .weak  eclic_irq28_handler
    .weak  eclic_irq29_handler
    .weak  eclic_irq30_handler
    .weak  eclic_irq31_handler
    .weak  eclic_irq32_handler
    .weak  eclic_irq33_handler
    .weak  eclic_irq34_handler
    .weak  eclic_irq35_handler
    .weak  eclic_irq36_handler
    .weak  eclic_irq37_handler
    .weak  eclic_irq38_handler
    .weak  eclic_irq39_handler
    .weak  eclic_irq40_handler
    .weak  eclic_irq41_handler
    .weak  eclic_irq42_handler
    .weak  eclic_irq43_handler
    .weak  eclic_irq44_handler
    .weak  eclic_irq45_handler
    .weak  eclic_irq46_handler
    .weak  eclic_irq47_handler
    .weak  eclic_irq48_handler
    .weak  eclic_irq49_handler
    .weak  eclic_irq50_handler

    .globl vector_base
    .type vector_base, @object
vector_base:
    j _start                                                  /* 0: Reserved, Jump to _
↪start when reset for vector table not remapped cases.*/
    .align LOG_REGBYTES                                       /*    Need to align 4␣
↪byte for RV32, 8 Byte for RV64 */

    DECLARE_INT_HANDLER     default_intexc_handler            /* 1: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 2: Reserved */
    DECLARE_INT_HANDLER     eclic_msip_handler                /* 3: Machine software␣
↪interrupt */

    DECLARE_INT_HANDLER     default_intexc_handler            /* 4: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 5: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 6: Reserved */
    DECLARE_INT_HANDLER     eclic_mtip_handler                /* 7: Machine timer␣
↪interrupt */

    DECLARE_INT_HANDLER     default_intexc_handler            /* 8: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 9: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 10: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler            /* 11: Reserved */

    DECLARE_INT_HANDLER     default_intexc_handler            /* 12: Reserved */
```

(continues on next page)

(continued from previous page)

```
    DECLARE_INT_HANDLER     default_intexc_handler          /* 13: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler          /* 14: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler          /* 15: Reserved */

    DECLARE_INT_HANDLER     default_intexc_handler          /* 16: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler          /* 17: Reserved */
    DECLARE_INT_HANDLER     default_intexc_handler          /* 18: Reserved */
    /* TODO: Adjust Vendor Defined External Interrupts */
    DECLARE_INT_HANDLER     eclic_irq19_handler             /* 19: Interrupt 19 */

    DECLARE_INT_HANDLER     eclic_irq20_handler             /* 20: Interrupt 20 */
    DECLARE_INT_HANDLER     eclic_irq21_handler             /* 21: Interrupt 21 */
    DECLARE_INT_HANDLER     eclic_irq22_handler             /* 22: Interrupt 22 */
    DECLARE_INT_HANDLER     eclic_irq23_handler             /* 23: Interrupt 23 */

    DECLARE_INT_HANDLER     eclic_irq24_handler             /* 24: Interrupt 24 */
    DECLARE_INT_HANDLER     eclic_irq25_handler             /* 25: Interrupt 25 */
    DECLARE_INT_HANDLER     eclic_irq26_handler             /* 26: Interrupt 26 */
    DECLARE_INT_HANDLER     eclic_irq27_handler             /* 27: Interrupt 27 */

    DECLARE_INT_HANDLER     eclic_irq28_handler             /* 28: Interrupt 28 */
    DECLARE_INT_HANDLER     eclic_irq29_handler             /* 29: Interrupt 29 */
    DECLARE_INT_HANDLER     eclic_irq30_handler             /* 30: Interrupt 30 */
    DECLARE_INT_HANDLER     eclic_irq31_handler             /* 31: Interrupt 31 */

    DECLARE_INT_HANDLER     eclic_irq32_handler             /* 32: Interrupt 32 */
    DECLARE_INT_HANDLER     eclic_irq33_handler             /* 33: Interrupt 33 */
    DECLARE_INT_HANDLER     eclic_irq34_handler             /* 34: Interrupt 34 */
    DECLARE_INT_HANDLER     eclic_irq35_handler             /* 35: Interrupt 35 */

    DECLARE_INT_HANDLER     eclic_irq36_handler             /* 36: Interrupt 36 */
    DECLARE_INT_HANDLER     eclic_irq37_handler             /* 37: Interrupt 37 */
    DECLARE_INT_HANDLER     eclic_irq38_handler             /* 38: Interrupt 38 */
    DECLARE_INT_HANDLER     eclic_irq39_handler             /* 39: Interrupt 39 */

    DECLARE_INT_HANDLER     eclic_irq40_handler             /* 40: Interrupt 40 */
    DECLARE_INT_HANDLER     eclic_irq41_handler             /* 41: Interrupt 41 */
    DECLARE_INT_HANDLER     eclic_irq42_handler             /* 42: Interrupt 42 */
    DECLARE_INT_HANDLER     eclic_irq43_handler             /* 43: Interrupt 43 */

    DECLARE_INT_HANDLER     eclic_irq44_handler             /* 44: Interrupt 44 */
    DECLARE_INT_HANDLER     eclic_irq45_handler             /* 45: Interrupt 45 */
    DECLARE_INT_HANDLER     eclic_irq46_handler             /* 46: Interrupt 46 */
    DECLARE_INT_HANDLER     eclic_irq47_handler             /* 47: Interrupt 47 */

    DECLARE_INT_HANDLER     eclic_irq48_handler             /* 48: Interrupt 48 */
    DECLARE_INT_HANDLER     eclic_irq49_handler             /* 49: Interrupt 49 */
    DECLARE_INT_HANDLER     eclic_irq50_handler             /* 50: Interrupt 50 */
    /* Please adjust the above part of interrupt definition code
     * according to your device interrupt number and its configuration */


/*** Startup Code Section ***/
    .section .init

    .globl _start
    .type _start,@function
```

(continues on next page)

(continued from previous page)

```

/**
 * Reset Handler called on controller reset
 */
_start:
    /* ===== Startup Stage 1 ===== */
    /* Disable Global Interrupt */
    csrc CSR_MSTATUS, MSTATUS_MIE

    /* Initialize GP and Stack Pointer SP */
    .option push
    .option norelax
    la gp, __global_pointer$

    .option pop
    la sp, _sp

    /*
     * Set the the NMI base mnvec to share
     * with mtvec by setting CSR_MMISC_CTL
     * bit 9 NMI_CAUSE_FFF to 1
     */
    li t0, MMISC_CTL_NMI_CAUSE_FFF
    csrs CSR_MMISC_CTL, t0

    /*
     * Intialize ECLIC vector interrupt
     * base address mtvt to vector_base
     */
    la t0, vector_base
    csrw CSR_MTVT, t0

    /*
     * Set ECLIC non-vector entry to be controlled
     * by mtvt2 CSR register.
     * Intialize ECLIC non-vector interrupt
     * base address mtvt2 to irq_entry.
     */
    la t0, irq_entry
    csrw CSR_MTVT2, t0
    csrs CSR_MTVT2, 0x1

    /*
     * Set Exception Entry MTVEC to exc_entry
     * Due to settings above, Exception and NMI
     * will share common entry.
     */
    la t0, exc_entry
    csrw CSR_MTVEC, t0

    /* Set the interrupt processing mode to ECLIC mode */
    li t0, 0x3f
    csrc CSR_MTVEC, t0
    csrs CSR_MTVEC, 0x3

    /* ===== Startup Stage 2 ===== */

```

(continues on next page)

(continued from previous page)

```
#if defined(__riscv_flen) && __riscv_flen > 0
    /* Enable FPU */
    li t0, MSTATUS_FS
    csrs mstatus, t0
    csrw fcsr, x0
#endif

    /* Enable mcycle and minstret counter */
    csrci CSR_MCOUNTINHIBIT, 0x5

    /*
     * Call vendor defined SystemInit to
     * initialize the micro-controller system.
     * TODO: You need to comment this code when run in Flash download mode.
     * then you need to put this line of code
     * after data/bss section initialization and before main
     */
    call SystemInit

    /* ===== Startup Stage 3 ===== */
    /*
     * Load code section from FLASH to ILM
     * when code LMA is different with VMA
     */
    la a0, _ilm_lma
    la a1, _ilm
    /* If the ILM phy-address same as the logic-address, then quit */
    beq a0, a1, 2f
    la a2, _eilm
    bgeu a1, a2, 2f

1:
    /* Load code section if necessary */
    lw t0, (a0)
    sw t0, (a1)
    addi a0, a0, 4
    addi a1, a1, 4
    bltu a1, a2, 1b
2:
    /* Load data section */
    la a0, _data_lma
    la a1, _data
    la a2, _edata
    bgeu a1, a2, 2f
1:
    lw t0, (a0)
    sw t0, (a1)
    addi a0, a0, 4
    addi a1, a1, 4
    bltu a1, a2, 1b
2:
    /* Clear bss section */
    la a0, __bss_start
    la a1, _end
    bgeu a0, a1, 2f
1:
    sw zero, (a0)
```

(continues on next page)

(continued from previous page)

```
    addi a0, a0, 4
    bltu a0, a1, 1b
2:
    /* TODO: Uncomment this code, if you run in Flash download mode */
    // call SystemInit

    /* Call global constructors */
    la a0, __libc_fini_array
    call atexit
    /* Call C/C++ constructor start up code */
    call __libc_init_array

    /* do pre-init steps before main */
    call _premain_init
    /* ===== Call Main Function  ===== */
    /* argc = argv = 0 */
    li a0, 0
    li a1, 0
    call main
    /* do post-main steps after main */
    call _postmain_fini

1:
    j 1b
```

Interrupt and Exception Handling File: intexc_<device>.S

The intexc File intexc_<device>.S contains:

- Macro to save caller register.
- Macro to restore caller register.
- Default Exception/NMI routine implementation.
- Default Non-Vector Interrupt routine implementation.

Nuclei processors provide **NMI(Non-Maskable Interrupt)**, **Exception**, **Vector Interrupt** and **Non-Vector Interrupt** features.

NMI(Non-Maskable Interrupt)

Click NMI[9] to learn about Nuclei Processor Core NMI in Nuclei ISA Spec.

NMI is used for urgent external HW error. It can't be masked and disabled.

When NMI happened, bit 9 of CSR `MMSIC_CTL` will be checked. If this bit value is 1, then NMI entry address will be the same as exception(CSR_MTVEC), and exception code for NMI will be 0xFFF, otherwise NMI entry will be same as reset_vector.

In NMSIS-Core, the bit 9 of CSR `MMISC_CTL` is set to 1 during core startup, so NMI will be treated as Exception and handled.

[9] https://doc.nucleisys.com/nuclei_spec/isa/nmi.html

Exception

Click Exception[10] to learn about Nuclei Processor Core Exception in Nuclei ISA Spec.

For CPU exception, the entry for exception will be `exc_entry`, in this entry code, it will call default exception handler `core_exception_handler()`.

In the common exception routine(`exc_entry`) to get more information like exception code. Exception handle flow show as below picture:

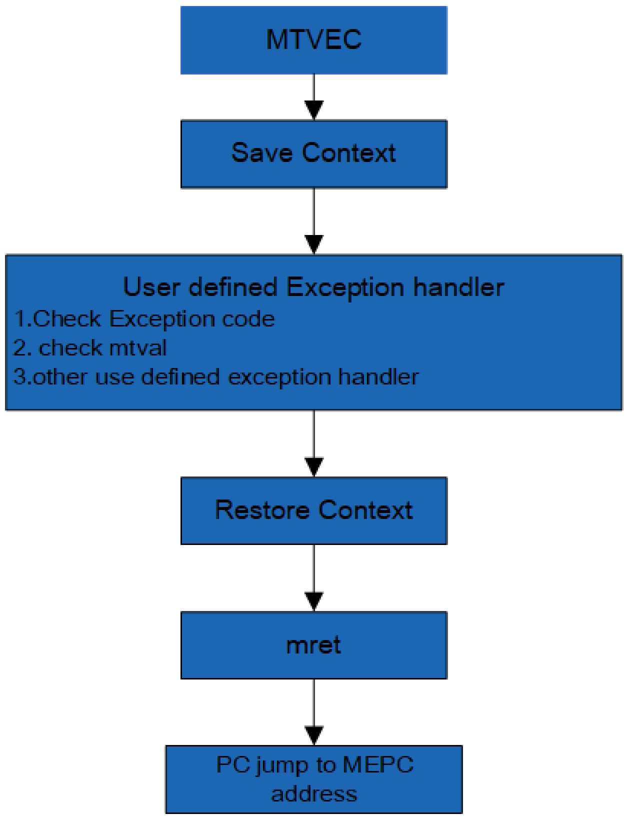


Fig. 3: Exception Handling Flow

NMI and exception could support nesting. Two levels of NMI/Exception state save stacks are supported.

We support three nesting mode as below:

- NMI nesting exception
- Exception nesting exception
- Exception nesting NMI

For software, we have provided the common entry for NMI and exception. Silicon vendor only need adapt the interface defined in core_api_intexc_nmi_handling.

Context save and restore have been handled by `exc_entry` interface.

When exception exception return it will run the intruction which trigger the exception again. It will cause software dead loop. So in the exception handler for each exception code, we propose to set CSR `MEPC` to be `MEPC+4`, then it will start from next instruction of MEPC.

[10] https://doc.nucleisys.com/nuclei_spec/isa/exception.html

Interrupt

Click Interrupt[11] to learn about Nuclei Processor Core Interrupt in Nuclei Spec.

Interrupt could be configured as **CLINT** mode or **ECILC** mode.

In NMSIS-Core, Interrupt has been configured as **ECLIC** mode during startup in *startup_<Devices>.S*, which is also recommended setting using Nuclei Processors.

ECLIC managed interrupt could configured as **vector** and **non-vector** mode.

Detail interrupt handling process as below picture:

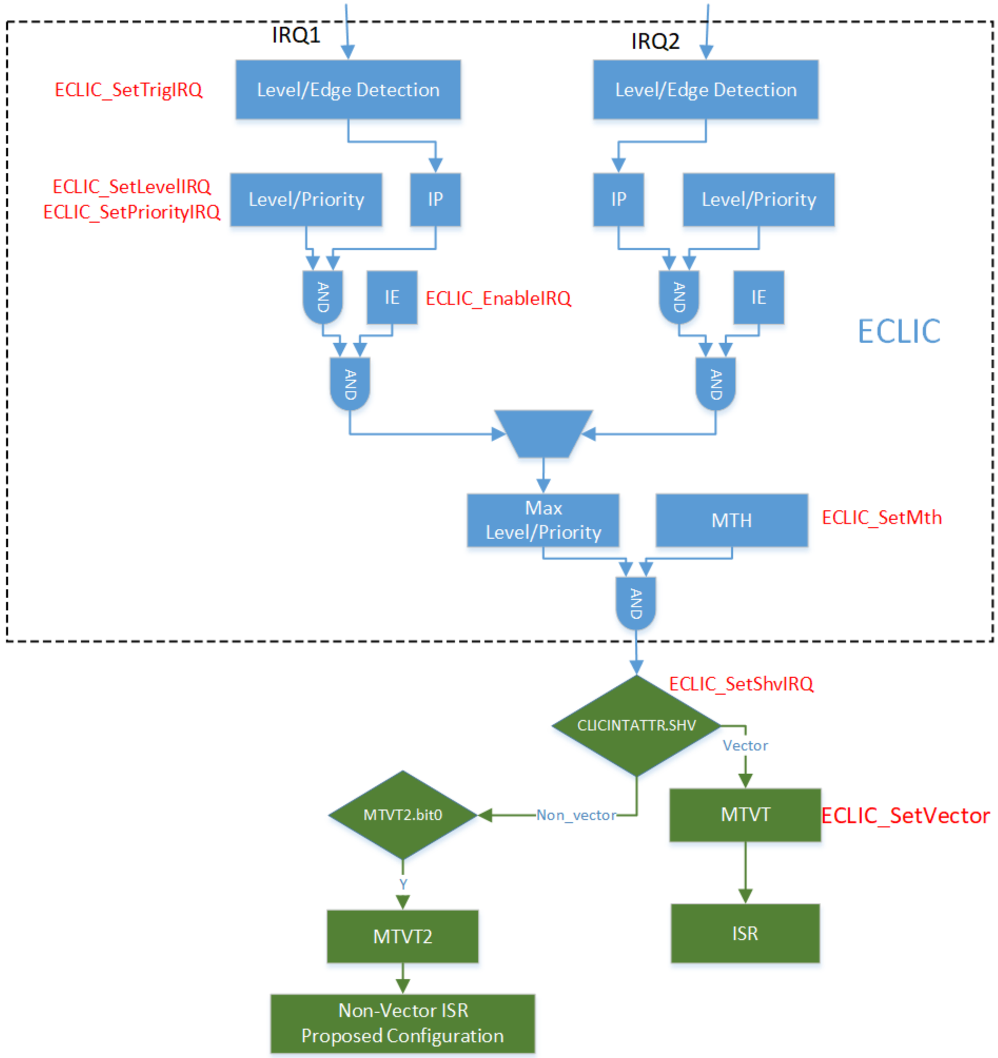


Fig. 4: Interrupt Handling Flow

To get highest priority interrupt we need compare the interrupt level first.If level is the same then compare the priority. High level interrupt could interrupt low level ISR and trigger interrupt nesting. If different priority with same level interrupt pending higher priority will be served first. Interrupt could be configured as vector mode and non-vector mode by vendor. For non-vector mode interrupt handler entry get from MTVT2 and exception/NMI handler entry get

[11] https://doc.nucleisys.com/nuclei_spec/isa/interrupt.html

from MTVEC. If Vendor need set non vector mode interrupt handler entry from `MTVVEC` you need set `MTVT2.BIT0` as 0.

Non-Vector Interrupt SW handler

For **non-vector** mode interrupt it will make the necessary CSR registers and context save and restore. Non-vector mode software handle flow show as below pciture:

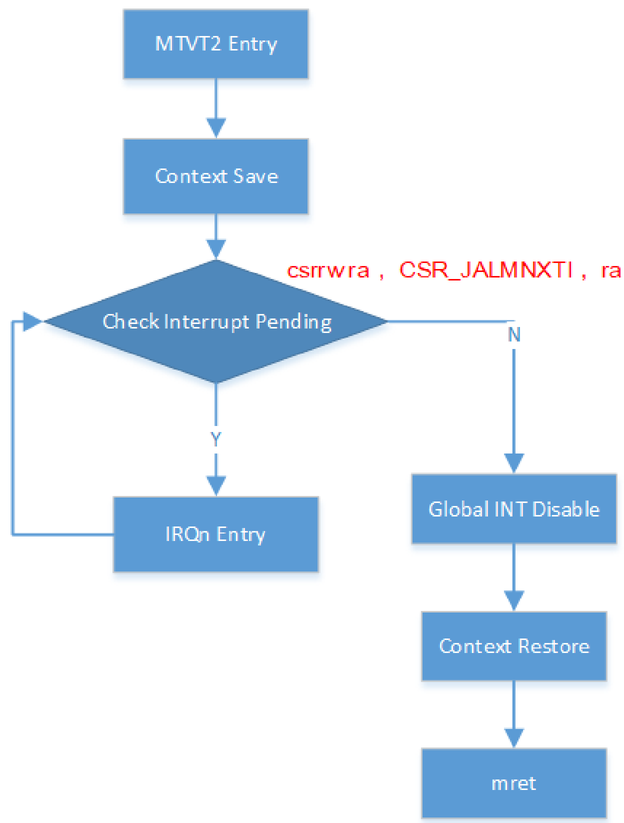


Fig. 5: Non-vector mode interrupt software handle flow

Detail description for non-vector mode interrupt handler as below steps:

1. Get non-vector mode handler entry from `MTVT2` if `MTVT2.BIT0` is 1(proposed configuration).
2. Context save to stack for cpu registers.
3. Save CSR registers `MEPC/MCAUSE/MSUBM` to stack.
4. Run instruction `csrrw ra, CSR_JALMNXTI, ra`. It will enable interrupt, check interrupt pending. If interrupt is pending then get highest priority interrupt and jump to interrupt handler entry in the vector table, otherwise it will go to step 6.
5. Execute the interrupt handler routine, when return from isr routine it will jump to step 4.
6. Global interrupt disable.
7. Restore CSR registers `MEPC/MCAUSE/MSUBM`.
8. Context restore from stack for cpu registers.
9. Execute `mret` to return from handler.

For **non-vector** mode iterrupt it could support **interrupt nesting**.

Interrupt nesting handle flow show as below picture:

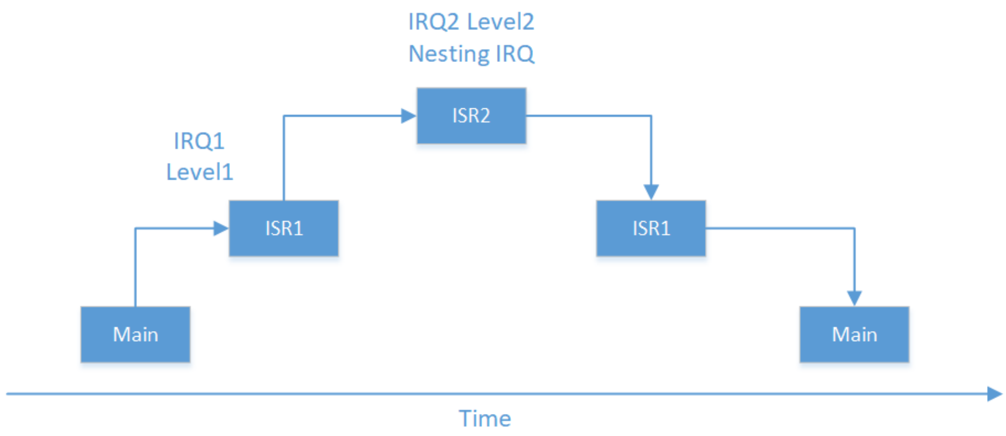


Fig. 6: Nesting interrupt handling flow

Vector Interrupt SW handler

If vector interrupt handler need support nesting or making function call Vector mode software handling flow show as below picture:

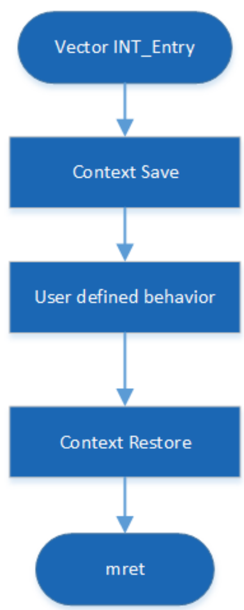


Fig. 7: Vector mode nesting interrupt handling flow

Detail description for nested vector mode interrupt handler as below steps:

1. Get vector mode handler from address of vector table entry `MTVT` added offset.
2. Context save to stack for cpu registers, done in each vector interrupt handler via `__INTERRUPT` (page 58)

3. Save CSR registers `MEPC/MCAUSE/MSUBM` to stack, done in each vector interrupt handler by read and save these CSRs into variables.
4. Execute the interrupt handling.
5. Restore CSR registers `MEPC/MCAUSE/MSUBM` from stack.
6. CSR registers restore from saved variables used in step 3.
7. Execute `mret` to return from handler

Here is sample code for above nested vector interrupt handling process:

```
// Vector interrupt handler for on-board button
__INTERRUPT void SOC_BUTTON_1_HANDLER(void)
{
    // save mepc,mcause,msubm enable interrupts
    SAVE_IRQ_CSR_CONTEXT();

    printf("%s", "----Begin button1 handler----Vector mode\r\n");

    // Green LED toggle
    gpio_toggle(GPIO, SOC_LED_GREEN_GPIO_MASK);

    // Clear the GPIO Pending interrupt by writing 1.
    gpio_clear_interrupt(GPIO, SOC_BUTTON_1_GPIO_OFS, GPIO_INT_RISE);

    wait_seconds(1); // Wait for a while

    printf("%s", "----End button1 handler\r\n");

    // disable interrupts,restore mepc,mcause,msubm
    RESTORE_IRQ_CSR_CONTEXT();
}
```

Detail description for non-nested vector mode interrupt handler as below

To improve the software response latency for vector mode vendor could remove context save/restore and **MEPC/MCAUSE/MSUBM** save/restore.

If so vector mode interrupt will not support nesting and interrupt handler can only be a leaf function which doesn't make any function calls.

Then the vector mode interrupt software flow will be described as below:

1. Get vector mode handler from address of vector table entry `MTVT` added offset.
2. Execute the interrupt handler(leaf function).
3. Execute `mret` to return from handler

Here is sample code for above non-nested vector interrupt handler which is a leaf function handling process:

```
static uint32_t btn_pressed = 0;
// Vector interrupt handler for on-board button
// This function is an leaf function, no function call is allowed
__INTERRUPT void SOC_BUTTON_1_HANDLER(void)
{
    btn_pressed ++;
}
```

intexc_Device.S Template File

The file exists for each supported toolchain and is the only toolchain specific NMSIS file.

Normally this file needn't adapt for different device. If CPU CSR registers have done some changes you may need some adaption.

Here we provided `intexc_Device.S` template file as below:

```
/*
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/******************************************************************************
 * \file     intexc_<Device>.S
 * \brief    NMSIS Interrupt and Exception Handling Template File
 *           for Nuclei N/NX Class Device
 * \version  V1.10
 * \date     30. July 2021
 *
 ******************************************************************************/

#include "riscv_encoding.h"

/**
 * \brief  Global interrupt disabled
 * \details
 *  This function disable global interrupt.
 * \remarks
 *  - All the interrupt requests will be ignored by CPU.
 */
.macro DISABLE_MIE
    csrc CSR_MSTATUS, MSTATUS_MIE
.endm

/**
 * \brief  Macro for context save
 * \details
 * This macro save ABI defined caller saved registers in the stack.
 * \remarks
 * - This Macro could use to save context when you enter to interrupt
 * or exception
*/
/* Save caller registers */
.macro SAVE_CONTEXT
```

(continues on next page)

(continued from previous page)

```
    /* Allocate stack space for context saving */
#ifndef __riscv_32e
    addi sp, sp, -20*REGBYTES
#else
    addi sp, sp, -14*REGBYTES
#endif /* __riscv_32e */

    STORE x1, 0*REGBYTES(sp)
    STORE x4, 1*REGBYTES(sp)
    STORE x5, 2*REGBYTES(sp)
    STORE x6, 3*REGBYTES(sp)
    STORE x7, 4*REGBYTES(sp)
    STORE x10, 5*REGBYTES(sp)
    STORE x11, 6*REGBYTES(sp)
    STORE x12, 7*REGBYTES(sp)
    STORE x13, 8*REGBYTES(sp)
    STORE x14, 9*REGBYTES(sp)
    STORE x15, 10*REGBYTES(sp)
#ifndef __riscv_32e
    STORE x16, 14*REGBYTES(sp)
    STORE x17, 15*REGBYTES(sp)
    STORE x28, 16*REGBYTES(sp)
    STORE x29, 17*REGBYTES(sp)
    STORE x30, 18*REGBYTES(sp)
    STORE x31, 19*REGBYTES(sp)
#endif /* __riscv_32e */
.endm

/**
 * \brief  Macro for restore caller registers
 * \details
 * This macro restore ABI defined caller saved registers from stack.
 * \remarks
 * - You could use this macro to restore context before you want return
 * from interrupt or exeception
 */
/* Restore caller registers */
.macro RESTORE_CONTEXT
    LOAD x1, 0*REGBYTES(sp)
    LOAD x4, 1*REGBYTES(sp)
    LOAD x5, 2*REGBYTES(sp)
    LOAD x6, 3*REGBYTES(sp)
    LOAD x7, 4*REGBYTES(sp)
    LOAD x10, 5*REGBYTES(sp)
    LOAD x11, 6*REGBYTES(sp)
    LOAD x12, 7*REGBYTES(sp)
    LOAD x13, 8*REGBYTES(sp)
    LOAD x14, 9*REGBYTES(sp)
    LOAD x15, 10*REGBYTES(sp)
#ifndef __riscv_32e
    LOAD x16, 14*REGBYTES(sp)
    LOAD x17, 15*REGBYTES(sp)
    LOAD x28, 16*REGBYTES(sp)
    LOAD x29, 17*REGBYTES(sp)
    LOAD x30, 18*REGBYTES(sp)
    LOAD x31, 19*REGBYTES(sp)

```

(continues on next page)

(continued from previous page)

```
    /* De-allocate the stack space */
    addi sp, sp, 20*REGBYTES
#else
    /* De-allocate the stack space */
    addi sp, sp, 14*REGBYTES
#endif /* __riscv_32e */

.endm

/**
 * \brief  Macro for save necessary CSRs to stack
 * \details
 * This macro store MCAUSE, MEPC, MSUBM to stack.
 */
.macro SAVE_CSR_CONTEXT
    /* Store CSR mcause to stack using pushmcause */
    csrrwi  x0, CSR_PUSHMCAUSE, 11
    /* Store CSR mepc to stack using pushmepc */
    csrrwi  x0, CSR_PUSHMEPC, 12
    /* Store CSR msub to stack using pushmsub */
    csrrwi  x0, CSR_PUSHMSUBM, 13
.endm

/**
 * \brief  Macro for restore necessary CSRs from stack
 * \details
 * This macro restore MSUBM, MEPC, MCAUSE from stack.
 */
.macro RESTORE_CSR_CONTEXT
    LOAD x5,  13*REGBYTES(sp)
    csrw CSR_MSUBM, x5
    LOAD x5,  12*REGBYTES(sp)
    csrw CSR_MEPC, x5
    LOAD x5,  11*REGBYTES(sp)
    csrw CSR_MCAUSE, x5
.endm

/**
 * \brief  Exception/NMI Entry
 * \details
 * This function provide common entry functions for exception/nmi.
 * \remarks
 * This function provide a default exception/nmi entry.
 * ABI defined caller save register and some CSR registers
 * to be saved before enter interrupt handler and be restored before return.
 */
.section .text.trap
/* In CLIC mode, the exeception entry must be 64bytes aligned */
.align 6
.global exc_entry
.weak exc_entry
exc_entry:
    /* Save the caller saving registers (context) */
    SAVE_CONTEXT
    /* Save the necessary CSR registers */
    SAVE_CSR_CONTEXT

```

(continues on next page)

(continued from previous page)

```
    /*
     * Set the exception handler function arguments
     * argument 1: mcause value
     * argument 2: current stack point(SP) value
     */
    csrr a0, mcause
    mv a1, sp
    /*
     * TODO: Call the exception handler function
     * By default, the function template is provided in
     * system_Device.c, you can adjust it as you want
     */
    call core_exception_handler

    /* Restore the necessary CSR registers */
    RESTORE_CSR_CONTEXT
    /* Restore the caller saving registers (context) */
    RESTORE_CONTEXT

    /* Return to regular code */
    mret

/**
 * \brief  Non-Vector Interrupt Entry
 * \details
 * This function provide common entry functions for handling
 * non-vector interrupts
 * \remarks
 * This function provide a default non-vector interrupt entry.
 * ABI defined caller save register and some CSR registers need
 * to be saved before enter interrupt handler and be restored before return.
 */
.section      .text.irq
/* In CLIC mode, the interrupt entry must be 4bytes aligned */
.align 2
.global irq_entry
.weak irq_entry
/* This label will be set to MTVT2 register */
irq_entry:
    /* Save the caller saving registers (context) */
    SAVE_CONTEXT
    /* Save the necessary CSR registers */
    SAVE_CSR_CONTEXT

    /* This special CSR read/write operation, which is actually
     * claim the CLIC to find its pending highest ID, if the ID
     * is not 0, then automatically enable the mstatus.MIE, and
     * jump to its vector-entry-label, and update the link register
     */
    csrrw ra, CSR_JALMNXTI, ra

    /* Critical section with interrupts disabled */
    DISABLE_MIE

    /* Restore the necessary CSR registers */
    RESTORE_CSR_CONTEXT
    /* Restore the caller saving registers (context) */
```

(continues on next page)

(continued from previous page)

```
    RESTORE_CONTEXT

    /* Return to regular code */
    mret

/* Default Handler for Exceptions / Interrupts */
.global default_intexc_handler
.weak default_intexc_handler
Undef_Handler:
default_intexc_handler:
1:
    j 1b
```

Device Linker Script: gcc_<device>.ld

The Linker Script File gcc_<device>.ld contains:

- Memory base address and size.
- Code, data section, vector table etc. location.
- Stack & heap location and size.

The file exists for each supported toolchain and is the only toolchain specific NMSIS file.

To adapt the file to a new device only when you need change the memory base address, size, data and code location etc.

gcc_Device.ld Template File

Here we provided `gcc_Device.ld` template file as below:

```
/*
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/******************************************************************************
 * @file     gcc_<Device>.ld
 * @brief    GNU Linker Script for Nuclei N/NX based device
 * @version  V1.1.0
 * @date     30. July 2021
 ******************************************************************************/
```

(continues on next page)

(continued from previous page)

```

/*********** Use Configuration Wizard in Context Menu *************************/

OUTPUT_ARCH( "riscv" )
/******************** Flash Configuration ************************************
 * <h> Flash Configuration
 * <o0> Flash Base Address <0x0-0xFFFFFFFF:8>
 * <o1> Flash Size (in Bytes) <0x0-0xFFFFFFFF:8>
 * </h>
 */
__ROM_BASE = 0x20000000;
__ROM_SIZE = 0x00400000;

/*--------------------- ILM RAM Configuration ---------------------------
 * <h> ILM RAM Configuration
 * <o0> ILM RAM Base Address    <0x0-0xFFFFFFFF:8>
 * <o1> ILM RAM Size (in Bytes) <0x0-0xFFFFFFFF:8>
 * </h>
 */
__ILM_RAM_BASE = 0x80000000;
__ILM_RAM_SIZE = 0x00010000;

/*--------------------- Embedded RAM Configuration ---------------------------
 * <h> RAM Configuration
 * <o0> RAM Base Address    <0x0-0xFFFFFFFF:8>
 * <o1> RAM Size (in Bytes) <0x0-0xFFFFFFFF:8>
 * </h>
*/
__RAM_BASE = 0x90000000;
__RAM_SIZE = 0x00010000;

/******************** Stack / Heap Configuration ****************************
 * <h> Stack / Heap Configuration
 * <o0> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
 * <o1> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
 * </h>
 */
__STACK_SIZE = 0x00000800;
__HEAP_SIZE  = 0x00000800;

/**************************** end of configuration section ********************/

/* Define base address and length of flash and ram */
MEMORY
{
  flash (rxai!w) : ORIGIN = __ROM_BASE, LENGTH = __ROM_SIZE
  ram (wxa!ri) : ORIGIN = __RAM_BASE, LENGTH = __RAM_SIZE
}
/* Linker script to place sections and symbol values. Should be used together
 * with other linker script that defines memory regions FLASH,ILM and RAM.
 * It references following symbols, which must be defined in code:
 *   _start : Entry of reset handler
 *
 * It defines following symbols, which code can use without definition:
 *   _ilm_lma
 *   _ilm
 *   __etext
```

(continues on next page)

(continued from previous page)

```
 *   _etext
 *   etext
 *   _eilm
 *   __preinit_array_start
 *   __preinit_array_end
 *   __init_array_start
 *   __init_array_end
 *   __fini_array_start
 *   __fini_array_end
 *   _data_lma
 *   _edata
 *   edata
 *   __data_end__
 *   __bss_start
 *   __fbss
 *   _end
 *   end
 *   __heap_end
 *   __StackLimit
 *   __StackTop
 *   __STACK_SIZE
 */
/* Define entry label of program */
ENTRY(_start)
SECTIONS
{
  __STACK_SIZE = DEFINED(__STACK_SIZE) ? __STACK_SIZE : 2K;

  .init           :
  {
    /* vector table locate at flash */
    *(.vtable)
    KEEP (*(SORT_NONE(.init)))
  } >flash AT>flash

  .ilalign         :
  {
    . = ALIGN(4);
    /* Create a section label as _ilm_lma which located at flash */
    PROVIDE( _ilm_lma = . );
  } >flash AT>flash

  .ialign         :
  {
    /* Create a section label as _ilm which located at flash */
    PROVIDE( _ilm = . );
  } >flash AT>flash

  /* Code section located at flash */
  .text           :
  {
    *(.text.unlikely .text.unlikely.*)
    *(.text.startup .text.startup.*)
    *(.text .text.*)
    *(.gnu.linkonce.t.*)
  } >flash AT>flash

```

(continues on next page)

(continued from previous page)

```
  .rodata : ALIGN(4)
  {
    . = ALIGN(4);
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
    . = ALIGN(8);
    *(.srodata.cst16)
    *(.srodata.cst8)
    *(.srodata.cst4)
    *(.srodata.cst2)
    *(.srodata .srodata.*)
  } >flash AT>flash

  .fini           :
  {
    KEEP (*(SORT_NONE(.fini)))
  } >flash AT>flash

  . = ALIGN(4);

  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  PROVIDE( _eilm = . );


  .preinit_array  :
  {
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
  } >flash AT>flash

  .init_array     :
  {
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*(SORT_BY_INIT_PRIORITY(.init_array.*) SORT_BY_INIT_PRIORITY(.ctors.*)))
    KEEP (*(.init_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o *crtend?.o ) .ctors))
    PROVIDE_HIDDEN (__init_array_end = .);
  } >flash AT>flash

  .fini_array     :
  {
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(SORT_BY_INIT_PRIORITY(.fini_array.*) SORT_BY_INIT_PRIORITY(.dtors.*)))
    KEEP (*(.fini_array EXCLUDE_FILE (*crtbegin.o *crtbegin?.o *crtend.o *crtend?.o ) .dtors))
    PROVIDE_HIDDEN (__fini_array_end = .);
  } >flash AT>flash

  .ctors          :
  {
    /* gcc uses crtbegin.o to find the start of
     * the constructors, so we make sure it is
     * first.  Because this is a wildcard, it
```

(continues on next page)

(continued from previous page)

```
     * doesn't matter if the user does not
     * actually link against crtbegin.o; the
     * linker won't look for a file to match a
     * wildcard.  The wildcard also means that it
     * doesn't matter which directory crtbegin.o
     * is in.
     */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*crtbegin?.o(.ctors))
    /* We don't want to include the .ctor section from
     * the crtend.o file until after the sorted ctors.
     * The .ctor section from the crtend file contains the
     * end of ctors marker and it must be last
     */
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  } >flash AT>flash

  .dtors          :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  } >flash AT>flash

  .lalign         :
  {
    . = ALIGN(4);
    PROVIDE( _data_lma = . );
  } >flash AT>flash

  .dalign         :
  {
    . = ALIGN(4);
    PROVIDE( _data = . );
  } >ram AT>flash

  /* Define data section virtual address is ram and physical address is flash */
  .data          :
  {
    *(.data .data.*)
    *(.gnu.linkonce.d.*)
    . = ALIGN(8);
    PROVIDE( __global_pointer$ = . + 0x800 );
    *(.sdata .sdata.* .sdata*)
    *(.gnu.linkonce.s.*)
  } >ram AT>flash

  . = ALIGN(4);
  PROVIDE( _edata = . );
  PROVIDE( edata = . );

  PROVIDE( _fbss = . );
  PROVIDE( __bss_start = . );
```

(continues on next page)

(continued from previous page)

```
  .bss            :
  {
    *(.sbss*)
    *(.gnu.linkonce.sb.*)
    *(.bss .bss.*)
    *(.gnu.linkonce.b.*)
    *(COMMON)
    . = ALIGN(4);
  } >ram AT>ram

  . = ALIGN(8);
  PROVIDE( _end = . );
  PROVIDE( end = . );
  /* Define stack and head location at ram */
  .stack ORIGIN(ram) + LENGTH(ram) - __STACK_SIZE :
  {
    PROVIDE( _heap_end = . );
    __StackLimit = .;
    . = __STACK_SIZE;
    __StackTop = .;
    PROVIDE( _sp = . );
  } >ram AT>ram
}
```

System Configuration Files system_<device>.c and system_<device>.h

The **System Configuration Files system_<device>.c and system_<device>.h** provides as a minimum the functions described under core_api_system_device.

These functions are device specific and need adaptations. In addition, the file might have configuration settings for the device such as XTAL frequency or PLL prescaler settings, necessary system initilization, vendor customized interrupt, exception and nmi handling code, refer to core_api_system_device for more details.

For devices with external memory BUS the `system_<device>.c` also configures the BUS system.

The silicon vendor might expose other functions (i.e. for power configuration) in the `system_<device>.c` file. In case of additional features the function prototypes need to be added to the `system_<device>.h` header file.

system_Device.c Template File

Here we provided `system_Device.c` template file as below:

```
/*
 * Copyright (c) 2009-2018 Arm Limited. All rights reserved.
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
```

(continues on next page)

(continued from previous page)

```
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/**************************************************************************//**
 * @file     system_<Device>.c
 * @brief    NMSIS Nuclei N/NX Device Peripheral Access Layer Source File for
 *           Device <Device>
 * @version  V1.10
 * @date     30. July 2021
 ******************************************************************************/

#include <stdint.h>
#include "<Device>.h"

/*----------------------------------------------------------------------------
  Define clocks
 *----------------------------------------------------------------------------*/
/* TODO: add here your necessary defines for device initialization
         following is an example for different system frequencies */
#define XTAL            (12000000U)       /* Oscillator frequency             */

#define SYSTEM_CLOCK    (5 * XTAL)

/**
 * \defgroup  NMSIS_Core_SystemConfig       System Device Configuration
 * \brief Functions for system init, clock setup and interrupt/exception/nmi functions available in system_<device>.c.
 * \details
 * Nuclei provides a template file **system_Device.c** that must be adapted by
 * the silicon vendor to match their actual device. As a <b>minimum requirement</b>,
 * this file must provide:
 *  -  A device-specific system configuration function, \ref SystemInit.
 *  -  A global variable that contains the system frequency, \ref SystemCoreClock.
 *  -  A global eclic configuration initialization, \ref ECLIC_Init.
 *  -  Global c library \ref _init and \ref _fini functions called right before calling main function.
 *  -  Vendor customized interrupt, exception and nmi handling code, see \ref NMSIS_Core_IntExcNMI_Handling
 *
 * The file configures the device and, typically, initializes the oscillator (PLL) that is part
 * of the microcontroller device. This file might export other functions or variables that provide
 * a more flexible configuration of the microcontroller system.
 *
 * And this file also provided common interrupt, exception and NMI exception handling framework template,
 * Silicon vendor can customize these template code as they want.
 *
 * \note Please pay special attention to the static variable \c SystemCoreClock. This variable might be
 * used throughout the whole system initialization and runtime to calculate frequency/time related values.
 * Thus one must assure that the variable always reflects the actual system clock speed.
```

(continues on next page)

(continued from previous page)

```
 *
 * \attention
 * Be aware that a value stored to \c SystemCoreClock during low level initialization␣
↪(i.e. \c SystemInit()) might get
 * overwritten by C libray startup code and/or .bss section initialization.
 * Thus its highly recommended to call \ref SystemCoreClockUpdate at the beginning of␣
↪the user \c main() routine.
 *
 * @{
 */

/*----------------------------------------------------------------------------
  System Core Clock Variable
 *----------------------------------------------------------------------------*/
/* TODO: initialize SystemCoreClock with the system core clock frequency value
         achieved after system initialization.
         This means system core clock frequency after call to SystemInit() */
/**
 * \brief      Variable to hold the system core clock value
 * \details
 * Holds the system core clock, which is the system clock frequency supplied to the␣
↪SysTick
 * timer and the processor core clock. This variable can be used by debuggers to␣
↪query the
 * frequency of the debug timer or to configure the trace clock speed.
 *
 * \attention
 * Compilers must be configured to avoid removing this variable in case the␣
↪application
 * program is not using it. Debugging systems require the variable to be physically
 * present in memory so that it can be examined to configure the debugger.
 */
uint32_t SystemCoreClock = SYSTEM_CLOCK;  /* System Clock Frequency (Core Clock) */


/*----------------------------------------------------------------------------
  Clock functions
 *----------------------------------------------------------------------------*/

/**
 * \brief      Function to update the variable \ref SystemCoreClock
 * \details
 * Updates the variable \ref SystemCoreClock and must be called whenever the core␣
↪clock is changed
 * during program execution. The function evaluates the clock register settings and␣
↪calculates
 * the current core clock.
 */
void SystemCoreClockUpdate (void)  /* Get Core Clock Frequency */
{
    /* TODO: add code to calculate the system frequency based upon the current
     *    register settings.
     * Note: This function can be used to retrieve the system core clock frequeny
     *    after user changed register settings.
     */
    SystemCoreClock = SYSTEM_CLOCK;
}
```

(continues on next page)

(continued from previous page)

```

/**
 * \brief       Function to Initialize the system.
 * \details
 * Initializes the microcontroller system. Typically, this function configures the
 * oscillator (PLL) that is part of the microcontroller device. For systems
 * with a variable clock speed, it updates the variable \ref SystemCoreClock.
 * SystemInit is called from the file <b>startup<i>_device</i></b>.
 */
void SystemInit (void)
{
    /* TODO: add code to initialize the system
     * Warn: do not use global variables because this function is called before
     * reaching pre-main. RW section maybe overwritten afterwards.
     */
    SystemCoreClock = SYSTEM_CLOCK;
}

/**
 * \defgroup  NMSIS_Core_IntExcNMI_Handling   Interrupt and Exception and NMI Handling
 * \brief Functions for interrupt, exception and nmi handle available in system_<device>.c.
 * \details
 * Nuclei provide a template for interrupt, exception and NMI handling. Silicon Vendor could adapat according
 * to their requirement. Silicon vendor could implement interface for different exception code and
 * replace current implementation.
 *
 * @{
 */
/** \brief Max exception handler number, don't include the NMI(0xFFF) one */
#define MAX_SYSTEM_EXCEPTION_NUM        12
/**
 * \brief       Store the exception handlers for each exception ID
 * \note
 * - This SystemExceptionHandlers are used to store all the handlers for all
 * the exception codes Nuclei N/NX core provided.
 * - Exception code 0 - 11, totally 12 exceptions are mapped to SystemExceptionHandlers[0:11]
 * - Exception for NMI is also re-routed to exception handling(exception code 0xFFF) in startup code configuration, the handler itself is mapped to SystemExceptionHandlers[MAX_SYSTEM_EXCEPTION_NUM]
 */
static unsigned long SystemExceptionHandlers[MAX_SYSTEM_EXCEPTION_NUM+1];

/**
 * \brief       Exception Handler Function Typedef
 * \note
 * This typedef is only used internal in this system_<Device>.c file.
 * It is used to do type conversion for registered exception handler before calling it.
 */
typedef void (*EXC_HANDLER) (unsigned long mcause, unsigned long sp);

/**
 * \brief       System Default Exception Handler
```

(continues on next page)

(continued from previous page)

```c
 * \details
 * This function provided a default exception and NMI handling code for all exception ids.
 * By default, It will just print some information for debug, Vendor can customize it according to its requirements.
 */
static void system_default_exception_handler(unsigned long mcause, unsigned long sp)
{
    /* TODO: Uncomment this if you have implement printf function.
     * Or you can implement your own version as you like */
    //printf("MCAUSE: 0x%lx\r\n", mcause);
    //printf("MEPC  : 0x%lx\r\n", __RV_CSR_READ(CSR_MEPC));
    //printf("MTVAL : 0x%lx\r\n", __RV_CSR_READ(CSR_MBADADDR));
    Exception_DumpFrame(sp);
    while(1);
}

/**
 * \brief      Initialize all the default core exception handlers
 * \details
 * The core exception handler for each exception id will be initialized to \ref system_default_exception_handler.
 * \note
 * Called in \ref _init function, used to initialize default exception handlers for all exception IDs
 */
static void Exception_Init(void)
{
    for (int i = 0; i < MAX_SYSTEM_EXCEPTION_NUM+1; i++) {
        SystemExceptionHandlers[i] = (unsigned long)system_default_exception_handler;
    }
}

/**
 * \brief      Dump Exception Frame
 * \details
 * This function provided feature to dump exception frame stored in stack.
 */
void Exception_DumpFrame(unsigned long sp)
{
    EXC_Frame_Type *exc_frame = (EXC_Frame_Type *)sp;

#ifndef __riscv_32e
    printf("ra: 0x%x, tp: 0x%x, t0: 0x%x, t1: 0x%x, t2: 0x%x, t3: 0x%x, t4: 0x%x, t5: 0x%x, t6: 0x%x\n" \
           "a0: 0x%x, a1: 0x%x, a2: 0x%x, a3: 0x%x, a4: 0x%x, a5: 0x%x, a6: 0x%x, a7: 0x%x\n" \
           "mcause: 0x%x, mepc: 0x%x, msubm: 0x%x\n", exc_frame->ra, exc_frame->tp, exc_frame->t0, \
           exc_frame->t1, exc_frame->t2, exc_frame->t3, exc_frame->t4, exc_frame->t5, exc_frame->t6, \
           exc_frame->a0, exc_frame->a1, exc_frame->a2, exc_frame->a3, exc_frame->a4, exc_frame->a5, \
           exc_frame->a6, exc_frame->a7, exc_frame->mcause, exc_frame->mepc, exc_frame->msubm);
#else
    printf("ra: 0x%x, tp: 0x%x, t0: 0x%x, t1: 0x%x, t2: 0x%x\n" \
```

(continues on next page)

(continued from previous page)

```c
            "a0: 0x%x, a1: 0x%x, a2: 0x%x, a3: 0x%x, a4: 0x%x, a5: 0x%x\n" \
            "mcause: 0x%x, mepc: 0x%x, msubm: 0x%x\n", exc_frame->ra, exc_frame->tp,␣
↪exc_frame->t0, \
            exc_frame->t1, exc_frame->t2, exc_frame->a0, exc_frame->a1, exc_frame->a2,␣
↪exc_frame->a3, \
            exc_frame->a4, exc_frame->a5, exc_frame->mcause, exc_frame->mepc, exc_
↪frame->msubm);
#endif
}

/**
 * \brief       Register an exception handler for exception code EXCn
 * \details
 * * For EXCn < \ref MAX_SYSTEM_EXCEPTION_NUM, it will be registered into␣
↪SystemExceptionHandlers[EXCn-1].
 * * For EXCn == NMI_EXCn, it will be registered into SystemExceptionHandlers[MAX_
↪SYSTEM_EXCEPTION_NUM].
 * \param   EXCn    See \ref EXCn_Type
 * \param   exc_handler     The exception handler for this exception code EXCn
 */
void Exception_Register_EXC(uint32_t EXCn, unsigned long exc_handler)
{
    if ((EXCn < MAX_SYSTEM_EXCEPTION_NUM) && (EXCn >= 0)) {
        SystemExceptionHandlers[EXCn] = exc_handler;
    } else if (EXCn == NMI_EXCn) {
        SystemExceptionHandlers[MAX_SYSTEM_EXCEPTION_NUM] = exc_handler;
    }
}

/**
 * \brief       Get current exception handler for exception code EXCn
 * \details
 * * For EXCn < \ref MAX_SYSTEM_EXCEPTION_NUM, it will return␣
↪SystemExceptionHandlers[EXCn-1].
 * * For EXCn == NMI_EXCn, it will return SystemExceptionHandlers[MAX_SYSTEM_
↪EXCEPTION_NUM].
 * \param   EXCn    See \ref EXCn_Type
 * \return  Current exception handler for exception code EXCn, if not found, return 0.
 */
unsigned long Exception_Get_EXC(uint32_t EXCn)
{
    if ((EXCn < MAX_SYSTEM_EXCEPTION_NUM) && (EXCn >= 0)) {
        return SystemExceptionHandlers[EXCn];
    } else if (EXCn == NMI_EXCn) {
        return SystemExceptionHandlers[MAX_SYSTEM_EXCEPTION_NUM];
    } else {
        return 0;
    }
}

/**
 * \brief      Common NMI and Exception handler entry
 * \details
 * This function provided a command entry for NMI and exception. Silicon Vendor could␣
↪modify
 * this template implementation according to requirement.
 * \remarks
```

(continues on next page)

(continued from previous page)

```
 * - RISCV provided common entry for all types of exception. This is proposed code template
 *   for exception entry function, Silicon Vendor could modify the implementation.
 * - For the core_exception_handler template, we provided exception register function \ref Exception_Register_EXCn
 *   which can help developer to register your exception handler for specific exception number.
 */
uint32_t core_exception_handler(unsigned long mcause, unsigned long sp)
{
    uint32_t EXCn = (uint32_t)(mcause & 0X00000fff);
    EXC_HANDLER exc_handler;

    if ((EXCn < MAX_SYSTEM_EXCEPTION_NUM) && (EXCn >= 0)) {
        exc_handler = (EXC_HANDLER)SystemExceptionHandlers[EXCn];
    } else if (EXCn == NMI_EXCn) {
        exc_handler = (EXC_HANDLER)SystemExceptionHandlers[MAX_SYSTEM_EXCEPTION_NUM];
    } else {
        exc_handler = (EXC_HANDLER)system_default_exception_handler;
    }
    if (exc_handler != NULL) {
        exc_handler(mcause, sp);
    }
    return 0;
}

/**
 * \brief Initialize Global ECLIC Config
 * \details
 * ECLIC needs be initialized after boot up,
 * Vendor could also change the initialization
 * configuration.
 */
void ECLIC_Init(void)
{
    /* Global Configuration about MTH and NLBits.
     * TODO: Please adapt it according to your system requirement.
     * This function is called in _init function */
    /* Set CSR MTH to zero */
    ECLIC_SetMth(0);
    /* Set Nlbits to the CLICINTCTLBITS, all the bits are level bits */
    ECLIC_SetCfgNlbits(__ECLIC_INTCTLBITS);
}

/**
 * \brief  Initialize a specific IRQ and register the handler
 * \details
 * This function set vector mode, trigger mode and polarity, interrupt level and priority,
 * assign handler for specific IRQn.
 * \param [in]  IRQn        NMI interrupt handler address
 * \param [in]  shv         \ref ECLIC_NON_VECTOR_INTERRUPT means non-vector mode, and \ref ECLIC_VECTOR_INTERRUPT is vector mode
 * \param [in]  trig_mode   see \ref ECLIC_TRIGGER_Type
 * \param [in]  lvl         interupt level
 * \param [in]  priority    interrupt priority
 * \param [in]  handler     interrupt handler, if NULL, handler will not be installed
```

(continues on next page)

(continued from previous page)

```c
 *
 * \return       -1 means invalid input parameter. 0 means successful.
 * \remarks
 * - This function use to configure specific eclic interrupt and register its␣
↪interrupt handler and enable its interrupt.
 * - If the vector table is placed in read-only section(FLASHXIP mode), handler could␣
↪not be installed
 */
int32_t ECLIC_Register_IRQ(IRQn_Type IRQn, uint8_t shv, ECLIC_TRIGGER_Type trig_mode,␣
↪uint8_t lvl, uint8_t priority, void *handler)
{
    if ((IRQn > SOC_INT_MAX) || (shv > ECLIC_VECTOR_INTERRUPT) \
        || (trig_mode > ECLIC_NEGTIVE_EDGE_TRIGGER )) {
        return -1;
    }

    /* set interrupt vector mode */
    ECLIC_SetShvIRQ(IRQn, shv);
    /* set interrupt trigger mode and polarity */
    ECLIC_SetTrigIRQ(IRQn, trig_mode);
    /* set interrupt level */
    ECLIC_SetLevelIRQ(IRQn, lvl);
    /* set interrupt priority */
    ECLIC_SetPriorityIRQ(IRQn, priority);
    if (handler != NULL) {
        /* set interrupt handler entry to vector table */
        ECLIC_SetVector(IRQn, (rv_csr_t)handler);
    }
    /* enable interrupt */
    ECLIC_EnableIRQ(IRQn);
    return 0;
}
/** @} */ /* End of Doxygen Group NMSIS_Core_ExceptionAndNMI */

/**
 * \brief early init function before main
 * \details
 * This function is executed right before main function.
 * For RISC-V gnu toolchain, _init function might not be called
 * by __libc_init_array function, so we defined a new function
 * to do initialization
 */
void _premain_init(void)
{
    /* TODO: Add your own initialization code here, called before main  */
    /* __ICACHE_PRESENT and __DCACHE_PRESENT are defined in <Device>.h */
#if defined(__ICACHE_PRESENT) && __ICACHE_PRESENT == 1
    EnableICache();
#endif
#if defined(__DCACHE_PRESENT) && __DCACHE_PRESENT == 1
    EnableDCache();
#endif
    // TODO: Add code to set the system clock frequency value SystemCoreClock

    // TODO: Add code to initialize necessary gpio and basic uart for debug print

    /* Initialize exception default handlers */
```

(continues on next page)

(continued from previous page)

```
    Exception_Init();
    /* ECLIC initialization, mainly MTH and NLBIT settings */
    ECLIC_Init();
}

/**
 * \brief finish function after main
 * \param [in]  status     status code return from main
 * \details
 * This function is executed right after main function.
 * For RISC-V gnu toolchain, _fini function might not be called
 * by __libc_fini_array function, so we defined a new function
 * to do initialization
 */
void _postmain_fini(int status)
{
    /* TODO: Add your own finishing code here, called after main */
}

/**
 * \brief _init function called in __libc_init_array()
 * \details
 * This `__libc_init_array()` function is called during startup code,
 * user need to implement this function, otherwise when link it will
 * error init.c:(.text.__libc_init_array+0x26): undefined reference to `_init'
 * \note
 * Please use \ref _premain_init function now
 */
void _init(void)
{
    /* Don't put any code here, please use _premain_init now */
}

/**
 * \brief _fini function called in __libc_fini_array()
 * \details
 * This `__libc_fini_array()` function is called when exit main.
 * user need to implement this function, otherwise when link it will
 * error fini.c:(.text.__libc_fini_array+0x28): undefined reference to `_fini'
 * \note
 * Please use \ref _postmain_fini function now
 */
void _fini(void)
{
    /* Don't put any code here, please use _postmain_fini now */
}

/** @} */ /* End of Doxygen Group NMSIS_Core_SystemAndClock */
```

system_Device.h Template File

Here we provided `system_Device.h` template file as below:

```
/*
 * Copyright (c) 2009-2018 Arm Limited. All rights reserved.
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/**************************************************************************
 * @file     system_<Device>.h
 * @brief    NMSIS Nuclei N/NX Device Peripheral Access Layer Header File for
 *           Device <Device>
 * @version  V1.10
 * @date     30. July 2021
 **************************************************************************/

#ifndef __SYSTEM_<Device>_H__   /* TODO: replace '<Device>' with your device name */
#define __SYSTEM_<Device>_H__

#ifdef __cplusplus
extern "C" {
#endif

#include <stdint.h>

extern uint32_t SystemCoreClock;     /*!< System Clock Frequency (Core Clock)  */

/** \brief Exception frame structure store in stack */
typedef struct EXC_Frame {
    unsigned long ra;                /* ra: x1, return address for jump */
    unsigned long tp;                /* tp: x4, thread pointer */
    unsigned long t0;                /* t0: x5, temporary register 0 */
    unsigned long t1;                /* t1: x6, temporary register 1 */
    unsigned long t2;                /* t2: x7, temporary register 2 */
    unsigned long a0;                /* a0: x10, return value or function argument 0 */
    unsigned long a1;                /* a1: x11, return value or function argument 1 */
    unsigned long a2;                /* a2: x12, function argument 2 */
    unsigned long a3;                /* a3: x13, function argument 3 */
    unsigned long a4;                /* a4: x14, function argument 4 */
    unsigned long a5;                /* a5: x15, function argument 5 */
    unsigned long mcause;            /* mcause: machine cause csr register */
    unsigned long mepc;              /* mepc: machine exception program counter csr register */
    unsigned long msubm;             /* msubm: machine sub-mode csr register, nuclei customized */
#ifndef __riscv_32e
    unsigned long a6;                /* a6: x16, function argument 6 */
```

(continues on next page)

(continued from previous page)

```
    unsigned long a7;                 /* a7: x17, function argument 7 */
    unsigned long t3;                 /* t3: x28, temporary register 3 */
    unsigned long t4;                 /* t4: x29, temporary register 4 */
    unsigned long t5;                 /* t5: x30, temporary register 5 */
    unsigned long t6;                 /* t6: x31, temporary register 6 */
#endif
} EXC_Frame_Type;

/**
 * \brief Setup the microcontroller system.
 * \details
 *  Initialize the System and update the SystemCoreClock variable.
 */
extern void SystemInit(void);

/**
 * \brief  Update SystemCoreClock variable.
 * \details
 *  Updates the SystemCoreClock with current core Clock retrieved from cpu registers.
 */
extern void SystemCoreClockUpdate(void);

/**
 * \brief Dump Exception Frame
 */
void Exception_DumpFrame(unsigned long sp);

/**
 * \brief Register an exception handler for exception code EXCn
 */
extern void Exception_Register_EXC(uint32_t EXCn, unsigned long exc_handler);

/**
 * \brief Get current exception handler for exception code EXCn
 */
extern unsigned long Exception_Get_EXC(uint32_t EXCn);

/**
 * \brief Initialize eclic config
 */
extern void ECLIC_Init(void);

/**
 * \brief  initialize a specific IRQ and register the handler
 * \details
 * This function set vector mode, trigger mode and polarity, interrupt level and␣
↪priority,
 * assign handler for specific IRQn.
 * \param [in]  IRQn        NMI interrupt handler address
 * \param [in]  shv         \ref ECLIC_NON_VECTOR_INTERRUPT means non-vector mode,␣
↪and \ref ECLIC_VECTOR_INTERRUPT is vector mode
 * \param [in]  trig_mode   see \ref ECLIC_TRIGGER_Type
 * \param [in]  lvl         interupt level
 * \param [in]  priority    interrupt priority
 * \param [in]  handler     interrupt handler
 * return       -1 means invalid input parameter. 0 means successful.
 * \remarks
```

(continues on next page)

(continued from previous page)

```
 * - This function use to configure specific eclic interrupt and register its interrupt handler and enable its interrupt.
 */
extern int32_t ECLIC_Register_IRQ(IRQn_Type IRQn, uint8_t shv, ECLIC_TRIGGER_Type trig_mode, uint8_t lvl, uint8_t priority, void *handler);

#ifdef __cplusplus
}
#endif

#endif /* __SYSTEM_<Device>_H__ */
```

Device Header File <device.h>

The *Device Header File <device.h>* (page 46) contains the following sections that are device specific:

- *Interrupt Number Definition* (page 46) provides interrupt numbers (IRQn) for all exceptions and interrupts of the device.
- *Configuration of the Processor and Core Peripherals* (page 47) reflect the features of the device.
- *Device Peripheral Access Layer* (page 49) provides definitions for the core_api_periph_access to all device peripherals. It contains all data structures and the address mapping for device-specific peripherals.
- **Access Functions for Peripherals (optional)** provide additional helper functions for peripherals that are useful for programming of these peripherals. Access Functions may be provided as inline functions or can be extern references to a device-specific library provided by the silicon vendor.

NMSIS CORE API (page 56) describes the standard features and functions of the *Device Header File <device.h>* (page 46) in detail.

Interrupt Number Definition

***Device Header File <device.h>* (page 46) contains the enumeration `IRQn_Type` that defines all exceptions and interrupts of the**

- Negative IRQn values represent processor core exceptions (internal interrupts).
- Positive IRQn values represent device-specific exceptions (external interrupts). The first device-specific interrupt has the IRQn value 0. The IRQn values needs extension to reflect the device-specific interrupt vector table in the *Startup File startup_<device>.S* (page 13).

The following example shows the extension of the interrupt vector table for the GD32VF103 device family.

```
typedef enum IRQn {
    /******  N200 Processor Exceptions Numbers *****************************************/
    Reserved0_IRQn              =   0,      /*!<  Internal reserved                                  */
    Reserved1_IRQn              =   1,      /*!<  Internal reserved                                  */
    Reserved2_IRQn              =   2,      /*!<  Internal reserved                                  */
    SysTimerSW_IRQn             =   3,      /*!<  System Timer SW interrupt                          */
    Reserved3_IRQn              =   4,      /*!<  Internal reserved                                  */
```

(continues on next page)

(continued from previous page)

```
    Reserved4_IRQn              =   5,     /*!<  Internal reserved                 */
    Reserved5_IRQn              =   6,     /*!<  Internal reserved                 */
    SysTimer_IRQn               =   7,     /*!<  System Timer Interrupt            */
    Reserved6_IRQn              =   8,     /*!<  Internal reserved                 */
    Reserved7_IRQn              =   9,     /*!<  Internal reserved                 */
    Reserved8_IRQn              =  10,     /*!<  Internal reserved                 */
    Reserved9_IRQn              =  11,     /*!<  Internal reserved                 */
    Reserved10_IRQn             =  12,     /*!<  Internal reserved                 */
    Reserved11_IRQn             =  13,     /*!<  Internal reserved                 */
    Reserved12_IRQn             =  14,     /*!<  Internal reserved                 */
    Reserved13_IRQn             =  15,     /*!<  Internal reserved                 */
    Reserved14_IRQn             =  16,     /*!<  Internal reserved                 */
    HardFault_IRQn              =  17,     /*!<  Hard Fault, storage access error  */
    Reserved15_IRQn             =  18,     /*!<  Internal reserved                 */

    /******  GD32VF103 Specific Interrupt Numbers *****************************************/
    WWDGT_IRQn                  =  19,     /*!< window watchDog timer interrupt    */
    LVD_IRQn                    =  20,     /*!< LVD through EXTI line detect interrupt    */
    TAMPER_IRQn                 =  21,     /*!< tamper through EXTI line detect    */
                          :       :
                          :       :
    CAN1_EWMC_IRQn              =  85,     /*!< CAN1 EWMC interrupt                */
    USBFS_IRQn                  =  86,     /*!< USBFS global interrupt             */
    SOC_INT_MAX,                           /*!< Number of total Interrupts         */
} IRQn_Type;
```

Configuration of the Processor and Core Peripherals

The *Device Header File <device.h>* (page 46) configures the Nuclei N/NX Class Processors and the core peripherals with `#define` that are set prior to including the file *nmsis_core.h*.

The following tables list the `#define` along with the possible values for N200, N300, N600, NX600. If these `#define` are missing default values are used.

nmsis_core.h

Table 6: Macros used in nmsis_core.h

#define	Value Range	Default	Description
__NUCLEI_N_REV OR __NUCLEI_NX_REV	0x0100 | 0x0104	0x0100	• For Nuclei N class device, define __NUCLEI_N_REV, for NX class device, define __NUCLEI_NX_REV. • Core revision number ([15:8] revision number, [7:0] patch number), 0x0100 -> 1.0, 0x0104 -> 1.4
__SYSTIMER_PRESENT	0 .. 1	1	Define whether Private System Timer is present or not. This SysTimer is a Memory Mapped Unit.
__SYSTIMER_BASEADDR	•	0x02000000	Base address of the System Timer Unit.
__ECLIC_PRESENT	0 .. 1	1	Define whether Enhanced Core Local Interrupt Controller (ECLIC) Unit is present or not
__ECLIC_BASEADDR	•	0x0C000000	Base address of the ECLIC unit.
__ECLIC_INTCTLBITS	1 .. 8	1	Define the number of hardware bits are actually implemented in the clicintctl registers.
__ECLIC_INTNUM	1 .. 1024	1	Define the total interrupt number(including the internal core interrupts) of ECLIC Unit
__PMP_PRESENT	0 .. 1	0	Define whether Physical Memory Protection (PMP) Unit is present or not.
__PMP_ENTRY_NUM	8 or 16	8	Define the numbers of PMP entries.
__FPU_PRESENT	0 .. 2	0	Define whether Floating Point Unit (FPU) is present or not. • 0: Not present • 1: Single precision FPU present • 2: Double precision FPU present
__DSP_PRESENT	0 .. 1	0	Define whether Digital Signal Processing Unit (DSP) is present or not.
__ICACHE_PRESENT	0 .. 1	0	Define whether I-Cache Unit is present or not.
__DCACHE_PRESENT	0 .. 1	0	Define whether D-Cache Unit is present or not.
__Vendor_SysTickConfig	0 .. 1	0	If __SYSTIMER_PRESENT is 1, then the __Vendor_SysTickConfig can be set to 0, otherwise it can only set to 1. If this define is set to 1, then the default SysTick_Config and SysTick_Reload function is excluded. In this case, the file Device.h must contain a vendor specific implementation of this function.

NMSIS Version and Processor Information

The following shows the defines in the *nmsis_core.h* file that may be used in the *NMSIS-Core Device Templates* (page 11) to verify a minimum version or ensure that the right Nuclei N/NX class is used.

Device Peripheral Access Layer

The *Device Header File <device.h>* (page 46) contains for each peripheral:

- Register Layout Typedef
- Base Address
- Access Definitions

The section core_api_periph_access shows examples for peripheral definitions.

Device.h Template File

Here we provided `Device.h` template file as below:

```
/*
 * Copyright (c) 2009-2019 Arm Limited. All rights reserved.
 * Copyright (c) 2019 Nuclei Limited. All rights reserved.
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Licensed under the Apache License, Version 2.0 (the License); you may
 * not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an AS IS BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
/******************************************************************************
 * @file     <Device>.h
 * @brief    NMSIS Nuclei N/NX Core Peripheral Access Layer Header File for
 *           Device <Device>
 * @version  V1.10
 * @date     30. July 2021
 ******************************************************************************/

#ifndef __<Device>_H__     /* TODO: replace '<Device>' with your device name */
#define __<Device>_H__

#ifdef __cplusplus
extern "C" {
#endif

/* TODO: replace '<Vendor>' with vendor name; add your doxygen comment   */
/** @addtogroup <Vendor>
  * @{
  */


/* TODO: replace '<Device>' with device name; add your doxygen comment   */
/** @addtogroup <Device>
```

(continues on next page)

(continued from previous page)

```
  * @{
  */


/** @addtogroup Configuration_of_NMSIS
  * @{
  */

/** \brief SoC Download mode definition */
/* TODO: device vendor can extend more download modes */
typedef enum {
    DOWNLOAD_MODE_FLASHXIP = 0,         /*!< Flashxip download mode */
    DOWNLOAD_MODE_FLASH = 1,            /*!< Flash download mode */
    DOWNLOAD_MODE_ILM = 2,              /*!< ilm download mode */
    DOWNLOAD_MODE_DDR = 3,              /*!< ddr download mode */
    DOWNLOAD_MODE_MAX,
} DownloadMode_Type;

/*␣
↪==================================================================================================
↪*/
/* ================                                Interrupt Number Definition       ␣
↪                    ================ */
/*␣
↪==================================================================================================
↪*/

typedef enum IRQn {
/* ========================================  Nuclei N/NX Specific Interrupt Numbers ␣
↪========================================= */

/* TODO: use this N/NX interrupt numbers if your device is a Nuclei N/NX device */
    Reserved0_IRQn            =   0,              /*!<  Internal reserved */
    Reserved1_IRQn            =   1,              /*!<  Internal reserved */
    Reserved2_IRQn            =   2,              /*!<  Internal reserved */
    SysTimerSW_IRQn           =   3,              /*!<  System Timer SW interrupt */
    Reserved3_IRQn            =   4,              /*!<  Internal reserved */
    Reserved4_IRQn            =   5,              /*!<  Internal reserved */
    Reserved5_IRQn            =   6,              /*!<  Internal reserved */
    SysTimer_IRQn             =   7,              /*!<  System Timer Interrupt */
    Reserved6_IRQn            =   8,              /*!<  Internal reserved */
    Reserved7_IRQn            =   9,              /*!<  Internal reserved */
    Reserved8_IRQn            =  10,              /*!<  Internal reserved */
    Reserved9_IRQn            =  11,              /*!<  Internal reserved */
    Reserved10_IRQn           =  12,              /*!<  Internal reserved */
    Reserved11_IRQn           =  13,              /*!<  Internal reserved */
    Reserved12_IRQn           =  14,              /*!<  Internal reserved */
    Reserved13_IRQn           =  15,              /*!<  Internal reserved */
    Reserved14_IRQn           =  16,              /*!<  Internal reserved */
    Reserved15_IRQn           =  17,              /*!<  Internal reserved */
    Reserved16_IRQn           =  18,              /*!<  Internal reserved */

/* ===========================================  <Device> Specific Interrupt Numbers ␣
↪========================================= */
/* TODO: add here your device specific external interrupt numbers. 19~1023 is␣
↪reserved number for user. Maxmum interrupt supported
         could get from clicinfo.NUM_INTERRUPT. According the interrupt handlers␣
↪defined in startup_Device.s
```

(continues on next page)

(continued from previous page)

```
         eg.: Interrupt for Timer#1       eclic_tim0_handler   ->   TIM0_IRQn */
    <DeviceInterrupt>_IRQn     = 19,                /*!< Device Interrupt */

    SOC_INT_MAX,                                    /* Max SoC interrupt Number */
} IRQn_Type;

/*␣
↪==================================================================================================
↪*/
/* ================                                 Exception Code Definition         ␣
↪                        ================ */
/*␣
↪==================================================================================================
↪*/

typedef enum EXCn {
/* =======================================  Nuclei N/NX Specific Exception Code ␣
↪======================================= */
    InsUnalign_EXCn          =   0,              /*!<  Instruction address misaligned␣
↪*/
    InsAccFault_EXCn         =   1,              /*!<  Instruction access fault */
    IlleIns_EXCn             =   2,              /*!<  Illegal instruction */
    Break_EXCn               =   3,              /*!<  Beakpoint */
    LdAddrUnalign_EXCn       =   4,              /*!<  Load address misaligned */
    LdFault_EXCn             =   5,              /*!<  Load access fault */
    StAddrUnalign_EXCn       =   6,              /*!<  Store or AMO address␣
↪misaligned */
    StAccessFault_EXCn       =   7,              /*!<  Store or AMO access fault */
    UmodeEcall_EXCn          =   8,              /*!<  Environment call from User␣
↪mode */
    MmodeEcall_EXCn          =  11,              /*!<  Environment call from Machine␣
↪mode */
    NMI_EXCn                 = 0xfff,            /*!<  NMI interrupt*/
} EXCn_Type;

/*␣
↪==================================================================================================
↪*/
/* ================                          Processor and Core Peripheral Section  ␣
↪                        ================ */
/*␣
↪==================================================================================================
↪*/

/* ===========================  Configuration of the Nuclei N/NX Processor and Core␣
↪Peripherals  =========================== */
/* TODO: set the defines according your Device */
/* TODO: define the correct core revision
 *       __NUCLEI_N_REV if your device is a Nuclei-N Class device
 *       __NUCLEI_NX_REV if your device is a Nuclei-NX Class device
 */
#define __NUCLEI_N#_REV           0x0100                /*!< Core Revision rXpY,␣
↪version X.Y, change N# to N for Nuclei N class cores, change N# to NX for Nuclei NX␣
↪cores */
/* TODO: define the correct core features for the <Device> */
#define __ECLIC_PRESENT           1                     /*!< Set to 1 if ECLIC is␣
↪present */
```

(continues on next page)

(continued from previous page)

```
#define __ECLIC_BASEADDR          0x0C000000UL          /*!< Set to ECLIC baseaddr of your device */
#define __ECLIC_INTCTLBITS        8                     /*!< Set to 1 - 8, the number of hardware bits are actually implemented in the clicintctl registers. */
#define __ECLIC_INTNUM            51                    /*!< Set to 1 - 1024, total interrupt number of ECLIC Unit */
#define __SYSTIMER_PRESENT        1                     /*!< Set to 1 if System Timer is present */
#define __SYSTIMER_BASEADDR       0x02000000UL          /*!< Set to SysTimer baseaddr of your device */
#define __FPU_PRESENT             1                     /*!< Set to 0, 1, or 2, 0 not present, 1 single floating point unit present, 2 double floating point unit present */
#define __DSP_PRESENT             1                     /*!< Set to 1 if DSP is present */
#define __PMP_PRESENT             1                     /*!< Set to 1 if PMP is present */
#define __PMP_ENTRY_NUM           16                    /*!< Set to 8 or 16, the number of PMP entries */
#define __ICACHE_PRESENT          0                     /*!< Set to 1 if I-Cache is present */
#define __DCACHE_PRESENT          0                     /*!< Set to 1 if D-Cache is present */
#define __Vendor_SysTickConfig    0                     /*!< Set to 1 if different SysTick Config is used */

/** @} */ /* End of group Configuration_of_NMSIS */


#include <nmsis_core.h>
/* TODO: include your system_<Device>.h file
         replace '<Device>' with your device name */
#include "system_<Device>.h"                    /*!< <Device> System */


/* ========================================  Start of section using anonymous unions ========================================= */
#if   defined (__GNUC__)
  /* anonymous unions are enabled by default */
#else
  #warning Not supported compiler type
#endif


/* =========================================================================================================================== */
/* ================                                Device Specific Peripheral Section                         ================ */
/* =========================================================================================================================== */
/* Macros for memory access operations */
#define _REG8P(p, i)                        ((volatile uint8_t *) ((uintptr_t)((p) + (i))))
#define _REG16P(p, i)                       ((volatile uint16_t *) ((uintptr_t)((p) + (i))))
```

(continues on next page)

(continued from previous page)

```
#define _REG32P(p, i)                         ((volatile uint32_t *) ((uintptr_t)((p) + (i))))
#define _REG64P(p, i)                         ((volatile uint64_t *) ((uintptr_t)((p) + (i))))
#define _REG8(p, i)                           (*(_REG8P(p, i)))
#define _REG16(p, i)                          (*(_REG16P(p, i)))
#define _REG32(p, i)                          (*(_REG32P(p, i)))
#define _REG64(p, i)                          (*(_REG64P(p, i)))
#define REG8(addr)                            _REG8((addr), 0)
#define REG16(addr)                           _REG16((addr), 0)
#define REG32(addr)                           _REG32((addr), 0)
#define REG64(addr)                           _REG64((addr), 0)

/* Macros for address type convert and access operations */
#define ADDR16(addr)                          ((uint16_t)(uintptr_t)(addr))
#define ADDR32(addr)                          ((uint32_t)(uintptr_t)(addr))
#define ADDR64(addr)                          ((uint64_t)(uintptr_t)(addr))
#define ADDR8P(addr)                          ((uint8_t *)(uintptr_t)(addr))
#define ADDR16P(addr)                         ((uint16_t *)(uintptr_t)(addr))
#define ADDR32P(addr)                         ((uint32_t *)(uintptr_t)(addr))
#define ADDR64P(addr)                         ((uint64_t *)(uintptr_t)(addr))

/* Macros for Bit Operations */
#if __riscv_xlen == 32
#define BITMASK_MAX                           0xFFFFFFFFUL
#define BITOFS_MAX                            31
#else
#define BITMASK_MAX                           0xFFFFFFFFFFFFFFFFULL
#define BITOFS_MAX                            63
#endif

// BIT/BITS only support bit mask for __riscv_xlen
// For RISC-V 32 bit, it support mask 32 bit wide
// For RISC-V 64 bit, it support mask 64 bit wide
#define BIT(ofs)                              (0x1UL << (ofs))
#define BITS(start, end)                      ((BITMASK_MAX) << (start) & (BITMASK_MAX) >> (BITOFS_MAX - (end)))
#define GET_BIT(regval, bitofs)               (((regval) >> (bitofs)) & 0x1)
#define SET_BIT(regval, bitofs)               ((regval) |= BIT(bitofs))
#define CLR_BIT(regval, bitofs)               ((regval) &= (~BIT(bitofs)))
#define FLIP_BIT(regval, bitofs)              ((regval) ^= BIT(bitofs))
#define WRITE_BIT(regval, bitofs, val)        CLR_BIT(regval, bitofs); ((regval) |= ((val) << bitofs) & BIT(bitofs))
#define CHECK_BIT(regval, bitofs)             (!!((regval) & (0x1UL<<(bitofs))))
#define GET_BITS(regval, start, end)          (((regval) & BITS((start), (end))) >> (start))
#define SET_BITS(regval, start, end)          ((regval) |= BITS((start), (end)))
#define CLR_BITS(regval, start, end)          ((regval) &= (~BITS((start), (end))))
#define FLIP_BITS(regval, start, end)         ((regval) ^= BITS((start), (end)))
#define WRITE_BITS(regval, start, end, val)   CLR_BITS(regval, start, end); ((regval) |= ((val) << start) & BITS((start), (end)))
#define CHECK_BITS_ALL(regval, start, end)    (!((~(regval)) & BITS((start), (end))))
#define CHECK_BITS_ANY(regval, start, end)    ((regval) & BITS((start), (end)))

#define BITMASK_SET(regval, mask)             ((regval) |= (mask))
#define BITMASK_CLR(regval, mask)             ((regval) &= (~(mask)))
#define BITMASK_FLIP(regval, mask)            ((regval) ^= (mask))
```

(continues on next page)

(continued from previous page)

```c
#define BITMASK_CHECK_ALL(regval, mask)     (!((~(regval)) & (mask)))
#define BITMASK_CHECK_ANY(regval, mask)     ((regval) & (mask))

/** @addtogroup Device_Peripheral_peripherals
  * @{
  */

/* TODO: add here your device specific peripheral access structure typedefs
         following is an example for UART */

/* =========================================================================================================================== */
/* ================                                           UART                                           ================ */
/* =========================================================================================================================== */

/**
  * @brief UART (UART)
  */
typedef struct {                                /*!< (@ 0x40000000) UART Structure                                            */
  __IOM uint32_t   TXFIFO;                      /*!< (@ 0x00000000) UART TX FIFO                                              */
  __IM  uint32_t   RXFIFO;                      /*!< (@ 0x00000004) UART RX FIFO                                              */
  __IOM uint32_t   TXCTRL;                      /*!< (@ 0x00000008) UART TX FIFO control                                      */
  __OM  uint32_t   RXCTRL;                      /*!< (@ 0x0000000C) UART RX FIFO control                                      */
  __IM  uint32_t   IE;                          /*!< (@ 0x00000010) UART Interrupt Enable flag                                */
  __IM  uint32_t   IP;                          /*!< (@ 0x00000018) TART Interrupt Pending flag                               */
  __IM  uint32_t   DIV;                         /*!< (@ 0x00000018) UART Baudrate Divider                                     */
} <DeviceAbbreviation>_UART_TypeDef;

/*@}*/ /* end of group <Device>_Peripherals */


/* =========================================  End of section using anonymous unions  ========================================= */
#if defined (__GNUC__)
  /* anonymous unions are enabled by default */
#else
  #warning Not supported compiler type
#endif


/* =========================================================================================================================== */
/* ================                          Device Specific Peripheral Address Map                          ================ */
```

(continues on next page)

(continued from previous page)

```
/* ==========================================================================================
 */


/* TODO: add here your device peripherals base addresses
         following is an example for timer */
/** @addtogroup Device_Peripheral_peripheralAddr
  * @{
  */

/* Peripheral and SRAM base address */
#define <DeviceAbbreviation>_FLASH_BASE         (0x00000000UL)          /*!< (FLASH     ) Base Address */
#define <DeviceAbbreviation>_SRAM_BASE          (0x20000000UL)          /*!< (SRAM      ) Base Address */
#define <DeviceAbbreviation>_PERIPH_BASE        (0x40000000UL)          /*!< (Peripheral) Base Address */

/* Peripheral memory map */
#define <DeviceAbbreviation>UART0_BASE          (<DeviceAbbreviation>_PERIPH_BASE)          /*!< (UART 0  ) Base Address */
#define <DeviceAbbreviation>I2C_BASE            (<DeviceAbbreviation>_PERIPH_BASE + 0x0800) /*!< (I2C     ) Base Address */
#define <DeviceAbbreviation>GPIO_BASE           (<DeviceAbbreviation>_PERIPH_BASE + 0x1000) /*!< (GPIO    ) Base Address */

/** @} */ /* End of group Device_Peripheral_peripheralAddr */


/* ==========================================================================================
 */
/* ================                                  Peripheral declaration                                  ================ */
/* ==========================================================================================
 */


/* TODO: add here your device peripherals pointer definitions
         following is an example for uart0 */
/** @addtogroup Device_Peripheral_declaration
  * @{
  */
#define <DeviceAbbreviation>_UART0              ((<DeviceAbbreviation>_TMR_TypeDef *) <DeviceAbbreviation>UART0_BASE)


/** @} */ /* End of group <Device> */

/** @} */ /* End of group <Vendor> */

#ifdef __cplusplus
}
#endif

```

(continues on next page)

(continued from previous page)

```
#endif  /* __<Device>_H__  */
```

2.4 Register Mapping

The table below associates some common register names used in NMSIS to the register names used in Nuclei ISA Spec[12].

Table 7: Register names used in NMSIS related with the register names in ISA

NMSIS Register Name	N200, N300, N600, NX600	Register Description
Enhanced Core Local Interrupt Controller(ECLIC)		
ECLIC->CFG	cliccfg	ECLIC Global Configuration Register
ECLIC->INFO	clicinfo	ECLIC Global Information Register
ECLIC->MTH	mth	ECLIC Global Machine Mode Threshold Register
ECLIC->CTRL[i].INTIP	clicintip[i]	ECLIC Interrupt Pending Register
ECLIC->CTRL[i].INTIE	clicintie[i]	ECLIC Interrupt Enable Register
ECLIC->CTRL[i].INTATTR	clicintattr[i]	ECLIC Interrupt Attribute Register
ECLIC->CTRL[i].INTCTRL	clicintctl[i]	ECLIC Interrupt Input Control Register
System Timer Unit(SysTimer)		
SysTimer->MTIMER	mtime_hi<<32 + mtime_lo	System Timer current value 64bits Register
SysTimer->MTIMERCMP	mtimecmp_hi<<32 + mtimecmp_lo	System Timer compare value 64bits Register
SysTimer->MSTOP	mstop	System Timer Stop Register
SysTimer->MSIP	msip	System Timer SW interrupt Register

2.5 NMSIS CORE API

If you want to access doxygen generated NMSIS CORE API, please click NMSIS CORE API Doxygen Documentation.

2.5.1 Version Control

__NUCLEI_N_REV (0x0104)

__NUCLEI_NX_REV (0x0100)

__NMSIS_VERSION_MAJOR (1U)

__NMSIS_VERSION_MINOR (0U)

__NMSIS_VERSION_PATCH (1U)

__NMSIS_VERSION ((__NMSIS_VERSION_MAJOR << 16U) | (__NMSIS_VERSION_MINOR << 8) | __NMSIS_VERSION_PAT

[12] https://doc.nucleisys.com/nuclei_spec/

group **NMSIS_Core_VersionControl**

Version #define symbols for NMSIS release specific C/C++ source code.

We followed the semantic versioning 2.0.0[13] to control NMSIS version. The version format is **MAJOR.MINOR.PATCH**, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

The header file `nmsis_version.h` is included by each core header so that these definitions are available.

Example Usage for NMSIS Version Check:

```
#if defined(__NMSIS_VERSION) && (__NMSIS_VERSION >= 0x00010105)
   #warning "Yes, we have NMSIS 1.1.5 or later"
#else
   #error "We need NMSIS 1.1.5 or later!"
#endif
```

Unnamed Group

__NUCLEI_N_REV (0x0104)

Nuclei N class core revision number.

Reversion number format: [15:8] revision number, [7:0] patch number

Attention This define is exclusive with __NUCLEI_NX_REV

__NUCLEI_NX_REV (0x0100)

Nuclei NX class core revision number.

Reversion number format: [15:8] revision number, [7:0] patch number

Attention This define is exclusive with __NUCLEI_N_REV

Defines

__NMSIS_VERSION_MAJOR (1U)

Represent the NMSIS major version.

The NMSIS major version can be used to differentiate between NMSIS major releases.

__NMSIS_VERSION_MINOR (0U)

Represent the NMSIS minor version.

The NMSIS minor version can be used to query a NMSIS release update including new features.

__NMSIS_VERSION_PATCH (1U)

Represent the NMSIS patch version.

The NMSIS patch version can be used to show bug fixes in this package.

__NMSIS_VERSION ((__NMSIS_VERSION_MAJOR << 16U) | (__NMSIS_VERSION_MINOR << 8) | __NMSIS_VERSION

Represent the NMSIS Version.

NMSIS Version format: **MAJOR.MINOR.PATCH**

- MAJOR: __NMSIS_VERSION_MAJOR, stored in `bits [31:16]` of __NMSIS_VERSION

[13] https://semver.org/

- MINOR: __NMSIS_VERSION_MINOR, stored in `bits [15:8]` of __NMSIS_VERSION
- PATCH: __NMSIS_VERSION_PATCH, stored in `bits [7:0]` of __NMSIS_VERSION

2.5.2 Compiler Control

__PACKED_STRUCT **T_UINT16_WRITE**

__PACKED_STRUCT **T_UINT16_READ**

__PACKED_STRUCT **T_UINT32_WRITE**

__PACKED_STRUCT **T_UINT32_READ**

__has_builtin(x) (0)

__ASM __asm

__INLINE inline

__STATIC_INLINE static inline

__STATIC_FORCEINLINE __attribute__((always_inline)) static inline

__NO_RETURN __attribute__((__noreturn__))

__USED __attribute__((used))

__WEAK __attribute__((weak))

__VECTOR_SIZE(x) __attribute__((vector_size(x)))

__PACKED __attribute__((packed, aligned(1)))

__PACKED_STRUCT struct __attribute__((packed, aligned(1)))

__PACKED_UNION union __attribute__((packed, aligned(1)))

__UNALIGNED_UINT16_WRITE(addr, val) (void)((((struct T_UINT16_WRITE *)(void *)(addr))->v) = (val))

__UNALIGNED_UINT16_READ(addr) (((const struct T_UINT16_READ *)(const void *)(addr))->v)

__UNALIGNED_UINT32_WRITE(addr, val) (void)((((struct T_UINT32_WRITE *)(void *)(addr))->v) = (val))

__UNALIGNED_UINT32_READ(addr) (((const struct T_UINT32_READ *)(const void *)(addr))->v)

__ALIGNED(x) __attribute__((aligned(x)))

__RESTRICT __restrict

__COMPILER_BARRIER() __ASM volatile("":::"memory")

__USUALLY(exp) __builtin_expect((exp), 1)

__RARELY(exp) __builtin_expect((exp), 0)

__INTERRUPT __attribute__((interrupt))

group **NMSIS_Core_CompilerControl**

Compiler agnostic #define symbols for generic c/c++ source code.

The NMSIS-Core provides the header file **nmsis_compiler.h** with consistent #define symbols for generate C or C++ source files that should be compiler agnostic. Each NMSIS compliant compiler should support the functionality described in this section.

The header file **nmsis_compiler.h** is also included by each Device Header File <device.h> so that these definitions are available.

Defines

__has_builtin(x) (0)

__ASM __asm
Pass information from the compiler to the assembler.

__INLINE inline
Recommend that function should be inlined by the compiler.

__STATIC_INLINE static inline
Define a static function that may be inlined by the compiler.

__STATIC_FORCEINLINE __attribute__((always_inline)) static inline
Define a static function that should be always inlined by the compiler.

__NO_RETURN __attribute__((__noreturn__))
Inform the compiler that a function does not return.

__USED __attribute__((used))
Inform that a variable shall be retained in executable image.

__WEAK __attribute__((weak))
restrict pointer qualifier to enable additional optimizations.

__VECTOR_SIZE(x) __attribute__((vector_size(x)))
specified the vector size of the variable, measured in bytes

__PACKED __attribute__((packed, aligned(1)))
Request smallest possible alignment.

__PACKED_STRUCT struct __attribute__((packed, aligned(1)))
Request smallest possible alignment for a structure.

__PACKED_UNION union __attribute__((packed, aligned(1)))
Request smallest possible alignment for a union.

__UNALIGNED_UINT16_WRITE(addr, val) (void)((((struct T_UINT16_WRITE *)(void *)(addr))->v) = (val))
Pointer for unaligned write of a uint16_t variable.

__UNALIGNED_UINT16_READ(addr) (((const struct T_UINT16_READ *)(const void *)(addr))->v)
Pointer for unaligned read of a uint16_t variable.

__UNALIGNED_UINT32_WRITE(addr, val) (void)((((struct T_UINT32_WRITE *)(void *)(addr))->v) = (val))
Pointer for unaligned write of a uint32_t variable.

__UNALIGNED_UINT32_READ(addr) (((const struct T_UINT32_READ *)(const void *)(addr))->v)
Pointer for unaligned read of a uint32_t variable.

__ALIGNED(x) __attribute__((aligned(x)))
Minimum x bytes alignment for a variable.

__RESTRICT __restrict
restrict pointer qualifier to enable additional optimizations.

__COMPILER_BARRIER() __ASM volatile("":::"memory")
Barrier to prevent compiler from reordering instructions.

__USUALLY (exp) __builtin_expect((exp), 1)
provide the compiler with branch prediction information, the branch is usually true

__RARELY (exp) __builtin_expect((exp), 0)
provide the compiler with branch prediction information, the branch is rarely true

__INTERRUPT __attribute__((interrupt))
Use this attribute to indicate that the specified function is an interrupt handler.

Variables

__PACKED_STRUCT **T_UINT16_WRITE**
Packed struct for unaligned uint16_t write access.

__PACKED_STRUCT **T_UINT16_READ**
Packed struct for unaligned uint16_t read access.

__PACKED_STRUCT **T_UINT32_WRITE**
Packed struct for unaligned uint32_t write access.

__PACKED_STRUCT **T_UINT32_READ**
Packed struct for unaligned uint32_t read access.

2.5.3 Peripheral Access

__I volatile const

__O volatile

__IO volatile

__IM volatile const

__OM volatile

__IOM volatile

_VAL2FLD (field, value) (((uint32_t)(value) << field ## _Pos) & field ## _Msk)

_FLD2VAL (field, value) (((uint32_t)(value) & field ## _Msk) >> field ## _Pos)

group **NMSIS_Core_PeriphAccess**
Naming conventions and optional features for accessing peripherals.

The section below describes the naming conventions, requirements, and optional features for accessing device specific peripherals. Most of the rules also apply to the core peripherals.

The **Device Header File <device.h>** contains typically these definition and also includes the core specific header files.

Defines

__I volatile const
Defines 'read only' permissions.

__O volatile
Defines 'write only' permissions.

__IO volatile
Defines 'read / write' permissions.

__IM volatile const
Defines 'read only' structure member permissions.

__OM volatile
Defines 'write only' structure member permissions.

__IOM volatile
Defines 'read/write' structure member permissions.

_VAL2FLD (field, value) (((uint32_t)(value) << field ## _Pos) & field ## _Msk)
Mask and shift a bit field value for use in a register bit range.

The macro _VAL2FLD uses the #define's _Pos and _Msk of the related bit field to shift bit-field values for assigning to a register.

Example:

```
ECLIC->CFG = _VAL2FLD(CLIC_CLICCFG_NLBIT, 3);
```

Return Masked and shifted value.

Parameters

- `[in] field`: Name of the register bit field.
- `[in] value`: Value of the bit field. This parameter is interpreted as an uint32_t type.

_FLD2VAL (field, value) (((uint32_t)(value) & field ## _Msk) >> field ## _Pos)
Mask and shift a register value to extract a bit filed value.

The macro _FLD2VAL uses the #define's _Pos and _Msk of the related bit field to extract the value of a bit field from a register.

Example:

```
nlbits = _FLD2VAL(CLIC_CLICCFG_NLBIT, ECLIC->CFG);
```

Return Masked and shifted bit field value.

Parameters

- `[in] field`: Name of the register bit field.
- `[in] value`: Value of register. This parameter is interpreted as an uint32_t type.

2.5.4 Core CSR Encodings

Core CSR Registers

CSR_USTATUS 0x0

CSR_FFLAGS 0x1

CSR_FRM 0x2

CSR_FCSR 0x3

CSR_CYCLE 0xc00

CSR_TIME 0xc01

CSR_INSTRET 0xc02

CSR_HPMCOUNTER3 0xc03

CSR_HPMCOUNTER4 0xc04

CSR_HPMCOUNTER5 0xc05

CSR_HPMCOUNTER6 0xc06

CSR_HPMCOUNTER7 0xc07

CSR_HPMCOUNTER8 0xc08

CSR_HPMCOUNTER9 0xc09

CSR_HPMCOUNTER10 0xc0a

CSR_HPMCOUNTER11 0xc0b

CSR_HPMCOUNTER12 0xc0c

CSR_HPMCOUNTER13 0xc0d

CSR_HPMCOUNTER14 0xc0e

CSR_HPMCOUNTER15 0xc0f

CSR_HPMCOUNTER16 0xc10

CSR_HPMCOUNTER17 0xc11

CSR_HPMCOUNTER18 0xc12

CSR_HPMCOUNTER19 0xc13

CSR_HPMCOUNTER20 0xc14

CSR_HPMCOUNTER21 0xc15

CSR_HPMCOUNTER22 0xc16

CSR_HPMCOUNTER23 0xc17

CSR_HPMCOUNTER24 0xc18

CSR_HPMCOUNTER25 0xc19

CSR_HPMCOUNTER26 0xc1a

CSR_HPMCOUNTER27 0xc1b

CSR_HPMCOUNTER28 0xc1c

CSR_HPMCOUNTER29 0xc1d

CSR_HPMCOUNTER30 0xc1e

CSR_HPMCOUNTER31 0xc1f

CSR_SSTATUS 0x100

CSR_SIE 0x104

CSR_STVEC 0x105

CSR_SSCRATCH 0x140

CSR_SEPC 0x141

CSR_SCAUSE 0x142

CSR_SBADADDR 0x143

CSR_SIP 0x144

CSR_SPTBR 0x180

CSR_MSTATUS 0x300

CSR_MISA 0x301

CSR_MEDELEG 0x302

CSR_MIDELEG 0x303

CSR_MIE 0x304

CSR_MTVEC 0x305

CSR_MCOUNTEREN 0x306

CSR_MSCRATCH 0x340

CSR_MEPC 0x341

CSR_MCAUSE 0x342

CSR_MBADADDR 0x343

CSR_MTVAL 0x343

CSR_MIP 0x344

CSR_PMPCFG0 0x3a0

CSR_PMPCFG1 0x3a1

CSR_PMPCFG2 0x3a2

CSR_PMPCFG3 0x3a3

CSR_PMPADDR0 0x3b0

CSR_PMPADDR1 0x3b1

CSR_PMPADDR2 0x3b2

CSR_PMPADDR3 0x3b3

CSR_PMPADDR4 0x3b4

CSR_PMPADDR5 0x3b5

CSR_PMPADDR6 0x3b6

CSR_PMPADDR7 0x3b7

CSR_PMPADDR8 0x3b8

CSR_PMPADDR9 0x3b9

CSR_PMPADDR10 0x3ba

CSR_PMPADDR11 0x3bb

CSR_PMPADDR12 0x3bc

CSR_PMPADDR13 0x3bd

CSR_PMPADDR14 0x3be

CSR_PMPADDR15 0x3bf

CSR_TSELECT 0x7a0

CSR_TDATA1 0x7a1

CSR_TDATA2 0x7a2

CSR_TDATA3 0x7a3

CSR_DCSR 0x7b0

CSR_DPC 0x7b1

CSR_DSCRATCH 0x7b2

CSR_MCYCLE 0xb00

CSR_MINSTRET 0xb02

CSR_MHPMCOUNTER3 0xb03

CSR_MHPMCOUNTER4 0xb04

CSR_MHPMCOUNTER5 0xb05

CSR_MHPMCOUNTER6 0xb06

CSR_MHPMCOUNTER7 0xb07

CSR_MHPMCOUNTER8 0xb08

CSR_MHPMCOUNTER9 0xb09

CSR_MHPMCOUNTER10 0xb0a

CSR_MHPMCOUNTER11 0xb0b

CSR_MHPMCOUNTER12 0xb0c

CSR_MHPMCOUNTER13 0xb0d

CSR_MHPMCOUNTER14 0xb0e

CSR_MHPMCOUNTER15 0xb0f

CSR_MHPMCOUNTER16 0xb10

CSR_MHPMCOUNTER17 0xb11

CSR_MHPMCOUNTER18 0xb12

CSR_MHPMCOUNTER19 0xb13

CSR_MHPMCOUNTER20 0xb14

CSR_MHPMCOUNTER21 0xb15

CSR_MHPMCOUNTER22 0xb16

CSR_MHPMCOUNTER23 0xb17

CSR_MHPMCOUNTER24 0xb18

CSR_MHPMCOUNTER25 0xb19

CSR_MHPMCOUNTER26 0xb1a

CSR_MHPMCOUNTER27 0xb1b

CSR_MHPMCOUNTER28 0xb1c

CSR_MHPMCOUNTER29 0xb1d

CSR_MHPMCOUNTER30 0xb1e

CSR_MHPMCOUNTER31 0xb1f

CSR_MUCOUNTEREN 0x320

CSR_MSCOUNTEREN 0x321

CSR_MHPMEVENT3 0x323

CSR_MHPMEVENT4 0x324

CSR_MHPMEVENT5 0x325

CSR_MHPMEVENT6 0x326

CSR_MHPMEVENT7 0x327

CSR_MHPMEVENT8 0x328

CSR_MHPMEVENT9 0x329

CSR_MHPMEVENT10 0x32a

CSR_MHPMEVENT11 0x32b

CSR_MHPMEVENT12 0x32c

CSR_MHPMEVENT13 0x32d

CSR_MHPMEVENT14 0x32e

CSR_MHPMEVENT15 0x32f

CSR_MHPMEVENT16 0x330

CSR_MHPMEVENT17 0x331

CSR_MHPMEVENT18 0x332

CSR_MHPMEVENT19 0x333

CSR_MHPMEVENT20 0x334

CSR_MHPMEVENT21 0x335

CSR_MHPMEVENT22 0x336

CSR_MHPMEVENT23 0x337

CSR_MHPMEVENT24 0x338

CSR_MHPMEVENT25 0x339

CSR_MHPMEVENT26 0x33a

CSR_MHPMEVENT27 0x33b

CSR_MHPMEVENT28 0x33c

CSR_MHPMEVENT29 0x33d

CSR_MHPMEVENT30 0x33e

CSR_MHPMEVENT31 0x33f

CSR_MVENDORID 0xf11

CSR_MARCHID 0xf12

CSR_MIMPID 0xf13

CSR_MHARTID 0xf14

CSR_CYCLEH 0xc80

CSR_TIMEH 0xc81

CSR_INSTRETH 0xc82

CSR_HPMCOUNTER3H 0xc83

CSR_HPMCOUNTER4H 0xc84

CSR_HPMCOUNTER5H 0xc85

CSR_HPMCOUNTER6H 0xc86

CSR_HPMCOUNTER7H 0xc87

CSR_HPMCOUNTER8H 0xc88

CSR_HPMCOUNTER9H 0xc89

CSR_HPMCOUNTER10H 0xc8a

CSR_HPMCOUNTER11H 0xc8b

CSR_HPMCOUNTER12H 0xc8c

CSR_HPMCOUNTER13H 0xc8d

CSR_HPMCOUNTER14H 0xc8e

CSR_HPMCOUNTER15H 0xc8f

CSR_HPMCOUNTER16H 0xc90

CSR_HPMCOUNTER17H 0xc91

CSR_HPMCOUNTER18H 0xc92

CSR_HPMCOUNTER19H 0xc93

CSR_HPMCOUNTER20H 0xc94

CSR_HPMCOUNTER21H 0xc95

CSR_HPMCOUNTER22H 0xc96

CSR_HPMCOUNTER23H 0xc97

CSR_HPMCOUNTER24H 0xc98

CSR_HPMCOUNTER25H 0xc99

CSR_HPMCOUNTER26H 0xc9a

CSR_HPMCOUNTER27H 0xc9b

CSR_HPMCOUNTER28H 0xc9c

CSR_HPMCOUNTER29H 0xc9d

CSR_HPMCOUNTER30H 0xc9e

CSR_HPMCOUNTER31H 0xc9f

CSR_MCYCLEH 0xb80

CSR_MINSTRETH 0xb82

CSR_MHPMCOUNTER3H 0xb83

CSR_MHPMCOUNTER4H 0xb84

CSR_MHPMCOUNTER5H 0xb85

CSR_MHPMCOUNTER6H 0xb86

CSR_MHPMCOUNTER7H 0xb87

CSR_MHPMCOUNTER8H 0xb88

CSR_MHPMCOUNTER9H 0xb89

CSR_MHPMCOUNTER10H 0xb8a

CSR_MHPMCOUNTER11H 0xb8b

CSR_MHPMCOUNTER12H 0xb8c

CSR_MHPMCOUNTER13H 0xb8d

CSR_MHPMCOUNTER14H 0xb8e

CSR_MHPMCOUNTER15H 0xb8f

CSR_MHPMCOUNTER16H 0xb90

CSR_MHPMCOUNTER17H 0xb91

CSR_MHPMCOUNTER18H 0xb92

CSR_MHPMCOUNTER19H 0xb93

CSR_MHPMCOUNTER20H 0xb94

CSR_MHPMCOUNTER21H 0xb95

CSR_MHPMCOUNTER22H 0xb96

CSR_MHPMCOUNTER23H 0xb97

CSR_MHPMCOUNTER24H 0xb98

CSR_MHPMCOUNTER25H 0xb99

CSR_MHPMCOUNTER26H 0xb9a

CSR_MHPMCOUNTER27H 0xb9b

CSR_MHPMCOUNTER28H 0xb9c

CSR_MHPMCOUNTER29H 0xb9d

CSR_MHPMCOUNTER30H 0xb9e

CSR_MHPMCOUNTER31H 0xb9f

CSR_MTVT 0x307

CSR_MNXTI 0x345

CSR_MINTSTATUS 0x346

CSR_MSCRATCHCSW 0x348

CSR_MSCRATCHCSWL 0x349

CSR_MCLICBASE 0x350

CSR_UCODE 0x801

CSR_MCOUNTINHIBIT 0x320

CSR_MILM_CTL 0x7C0

CSR_MDLM_CTL 0x7C1

CSR_MECC_CODE 0x7C2

CSR_MNVEC 0x7C3

CSR_MSUBM 0x7C4

CSR_MDCAUSE 0x7C9

CSR_MCACHE_CTL 0x7CA

CSR_MMISC_CTL 0x7D0

CSR_MSAVESTATUS 0x7D6

CSR_MSAVEEPC1 0x7D7

CSR_MSAVECAUSE1 0x7D8

CSR_MSAVEEPC2 0x7D9

CSR_MSAVECAUSE2 0x7DA

CSR_MSAVEDCAUSE1 0x7DB

CSR_MSAVEDCAUSE2 0x7DC

CSR_MTLB_CTL 0x7DD

CSR_MECC_LOCK 0x7DE

CSR_PUSHMSUBM 0x7EB

CSR_MTVT2 0x7EC

CSR_JALMNXTI 0x7ED

CSR_PUSHMCAUSE 0x7EE

CSR_PUSHMEPC 0x7EF

CSR_MPPICFG_INFO 0x7F0

CSR_MFIOCFG_INFO 0x7F1

CSR_SLEEPVALUE 0x811

CSR_TXEVT 0x812

CSR_WFE 0x810

CSR_MICFG_INFO 0xFC0

CSR_MDCFG_INFO 0xFC1

CSR_MCFG_INFO 0xFC2

CSR_MTLBCFG_INFO 0xFC3

CSR_CCM_MBEGINADDR 0x7CB

CSR_CCM_MCOMMAND 0x7CC

CSR_CCM_MDATA 0x7CD

CSR_CCM_SUEN 0x7CE

CSR_CCM_SBEGINADDR 0x5CB

CSR_CCM_SCOMMAND 0x5CC

CSR_CCM_SDATA 0x5CD

CSR_CCM_UBEGINADDR 0x4CB

CSR_CCM_UCOMMAND 0x4CC

CSR_CCM_UDATA 0x4CD

CSR_CCM_FPIPE 0x4CF

group **NMSIS_Core_CSR_Registers**
NMSIS Core CSR Register Definitions.

The following macros are used for CSR Register Defintions.

Defines

CSR_USTATUS 0x0

CSR_FFLAGS 0x1

CSR_FRM 0x2

CSR_FCSR 0x3

CSR_CYCLE 0xc00

CSR_TIME 0xc01

CSR_INSTRET 0xc02

CSR_HPMCOUNTER3 0xc03

CSR_HPMCOUNTER4 0xc04

CSR_HPMCOUNTER5 0xc05

CSR_HPMCOUNTER6 0xc06

CSR_HPMCOUNTER7 0xc07

CSR_HPMCOUNTER8 0xc08

CSR_HPMCOUNTER9 0xc09

CSR_HPMCOUNTER10 0xc0a

CSR_HPMCOUNTER11 0xc0b

CSR_HPMCOUNTER12 0xc0c

CSR_HPMCOUNTER13 0xc0d

CSR_HPMCOUNTER14 0xc0e

CSR_HPMCOUNTER15 0xc0f

CSR_HPMCOUNTER16 0xc10

CSR_HPMCOUNTER17 0xc11

CSR_HPMCOUNTER18 0xc12

CSR_HPMCOUNTER19 0xc13

CSR_HPMCOUNTER20 0xc14

CSR_HPMCOUNTER21 0xc15

CSR_HPMCOUNTER22 0xc16

CSR_HPMCOUNTER23 0xc17

CSR_HPMCOUNTER24 0xc18

CSR_HPMCOUNTER25 0xc19

CSR_HPMCOUNTER26 0xc1a

CSR_HPMCOUNTER27 0xc1b

CSR_HPMCOUNTER28 0xc1c

CSR_HPMCOUNTER29 0xc1d

CSR_HPMCOUNTER30 0xc1e

CSR_HPMCOUNTER31 0xc1f

CSR_SSTATUS 0x100

CSR_SIE 0x104

CSR_STVEC 0x105

CSR_SSCRATCH 0x140

CSR_SEPC 0x141

CSR_SCAUSE 0x142

CSR_SBADADDR 0x143

CSR_SIP 0x144

CSR_SPTBR 0x180

CSR_MSTATUS 0x300

CSR_MISA 0x301

CSR_MEDELEG 0x302

CSR_MIDELEG 0x303

CSR_MIE 0x304

CSR_MTVEC 0x305

CSR_MCOUNTEREN 0x306

CSR_MSCRATCH 0x340

CSR_MEPC 0x341

CSR_MCAUSE 0x342

CSR_MBADADDR 0x343

CSR_MTVAL 0x343

CSR_MIP 0x344

CSR_PMPCFG0 0x3a0

CSR_PMPCFG1 0x3a1

CSR_PMPCFG2 0x3a2

CSR_PMPCFG3 0x3a3

CSR_PMPADDR0 0x3b0

CSR_PMPADDR1 0x3b1

CSR_PMPADDR2 0x3b2

CSR_PMPADDR3 0x3b3

CSR_PMPADDR4 0x3b4

CSR_PMPADDR5 0x3b5

CSR_PMPADDR6 0x3b6

CSR_PMPADDR7 0x3b7

CSR_PMPADDR8 0x3b8

CSR_PMPADDR9 0x3b9

CSR_PMPADDR10 0x3ba

CSR_PMPADDR11 0x3bb

CSR_PMPADDR12 0x3bc

CSR_PMPADDR13 0x3bd

CSR_PMPADDR14 0x3be

CSR_PMPADDR15 0x3bf

CSR_TSELECT 0x7a0

CSR_TDATA1 0x7a1

CSR_TDATA2 0x7a2

CSR_TDATA3 0x7a3

CSR_DCSR 0x7b0

CSR_DPC 0x7b1

CSR_DSCRATCH 0x7b2

CSR_MCYCLE 0xb00

CSR_MINSTRET 0xb02

CSR_MHPMCOUNTER3 0xb03

CSR_MHPMCOUNTER4 0xb04

CSR_MHPMCOUNTER5 0xb05

CSR_MHPMCOUNTER6 0xb06

CSR_MHPMCOUNTER7 0xb07

CSR_MHPMCOUNTER8 0xb08

CSR_MHPMCOUNTER9 0xb09

CSR_MHPMCOUNTER10 0xb0a

CSR_MHPMCOUNTER11 0xb0b

CSR_MHPMCOUNTER12 0xb0c

CSR_MHPMCOUNTER13 0xb0d

CSR_MHPMCOUNTER14 0xb0e

CSR_MHPMCOUNTER15 0xb0f

CSR_MHPMCOUNTER16 0xb10

CSR_MHPMCOUNTER17 0xb11

CSR_MHPMCOUNTER18 0xb12

CSR_MHPMCOUNTER19 0xb13

CSR_MHPMCOUNTER20 0xb14

CSR_MHPMCOUNTER21 0xb15

CSR_MHPMCOUNTER22 0xb16

CSR_MHPMCOUNTER23 0xb17

CSR_MHPMCOUNTER24 0xb18

CSR_MHPMCOUNTER25 0xb19

CSR_MHPMCOUNTER26 0xb1a

CSR_MHPMCOUNTER27 0xb1b

CSR_MHPMCOUNTER28 0xb1c

CSR_MHPMCOUNTER29 0xb1d

CSR_MHPMCOUNTER30 0xb1e

CSR_MHPMCOUNTER31 0xb1f

CSR_MUCOUNTEREN 0x320

CSR_MSCOUNTEREN 0x321

CSR_MHPMEVENT3 0x323

CSR_MHPMEVENT4 0x324

CSR_MHPMEVENT5 0x325

CSR_MHPMEVENT6 0x326

CSR_MHPMEVENT7 0x327

CSR_MHPMEVENT8 0x328

CSR_MHPMEVENT9 0x329

CSR_MHPMEVENT10 0x32a

CSR_MHPMEVENT11 0x32b

CSR_MHPMEVENT12 0x32c

CSR_MHPMEVENT13 0x32d

CSR_MHPMEVENT14 0x32e

CSR_MHPMEVENT15 0x32f

CSR_MHPMEVENT16 0x330

CSR_MHPMEVENT17 0x331

CSR_MHPMEVENT18 0x332

CSR_MHPMEVENT19 0x333

CSR_MHPMEVENT20 0x334

CSR_MHPMEVENT21 0x335

CSR_MHPMEVENT22 0x336

CSR_MHPMEVENT23 0x337

CSR_MHPMEVENT24 0x338

CSR_MHPMEVENT25 0x339

CSR_MHPMEVENT26 0x33a

CSR_MHPMEVENT27 0x33b

CSR_MHPMEVENT28 0x33c

CSR_MHPMEVENT29 0x33d

CSR_MHPMEVENT30 0x33e

CSR_MHPMEVENT31 0x33f

CSR_MVENDORID 0xf11

CSR_MARCHID 0xf12

CSR_MIMPID 0xf13

CSR_MHARTID 0xf14

CSR_CYCLEH 0xc80

CSR_TIMEH 0xc81

CSR_INSTRETH 0xc82

CSR_HPMCOUNTER3H 0xc83

CSR_HPMCOUNTER4H 0xc84

CSR_HPMCOUNTER5H 0xc85

CSR_HPMCOUNTER6H 0xc86

CSR_HPMCOUNTER7H 0xc87

CSR_HPMCOUNTER8H 0xc88

CSR_HPMCOUNTER9H 0xc89

CSR_HPMCOUNTER10H 0xc8a

CSR_HPMCOUNTER11H 0xc8b

CSR_HPMCOUNTER12H 0xc8c

CSR_HPMCOUNTER13H 0xc8d

CSR_HPMCOUNTER14H 0xc8e

CSR_HPMCOUNTER15H 0xc8f

CSR_HPMCOUNTER16H 0xc90

CSR_HPMCOUNTER17H 0xc91

CSR_HPMCOUNTER18H 0xc92

CSR_HPMCOUNTER19H 0xc93

CSR_HPMCOUNTER20H 0xc94

CSR_HPMCOUNTER21H 0xc95

CSR_HPMCOUNTER22H 0xc96

CSR_HPMCOUNTER23H 0xc97

CSR_HPMCOUNTER24H 0xc98

CSR_HPMCOUNTER25H 0xc99

CSR_HPMCOUNTER26H 0xc9a

CSR_HPMCOUNTER27H 0xc9b

CSR_HPMCOUNTER28H 0xc9c

CSR_HPMCOUNTER29H 0xc9d

CSR_HPMCOUNTER30H 0xc9e

CSR_HPMCOUNTER31H 0xc9f

CSR_MCYCLEH 0xb80

CSR_MINSTRETH 0xb82

CSR_MHPMCOUNTER3H 0xb83

CSR_MHPMCOUNTER4H 0xb84

CSR_MHPMCOUNTER5H 0xb85

CSR_MHPMCOUNTER6H 0xb86

CSR_MHPMCOUNTER7H 0xb87

CSR_MHPMCOUNTER8H 0xb88

CSR_MHPMCOUNTER9H 0xb89

CSR_MHPMCOUNTER10H 0xb8a

CSR_MHPMCOUNTER11H 0xb8b

CSR_MHPMCOUNTER12H 0xb8c

CSR_MHPMCOUNTER13H 0xb8d

CSR_MHPMCOUNTER14H 0xb8e

CSR_MHPMCOUNTER15H 0xb8f

CSR_MHPMCOUNTER16H 0xb90

CSR_MHPMCOUNTER17H 0xb91

CSR_MHPMCOUNTER18H 0xb92

CSR_MHPMCOUNTER19H 0xb93

CSR_MHPMCOUNTER20H 0xb94

CSR_MHPMCOUNTER21H 0xb95

CSR_MHPMCOUNTER22H 0xb96

CSR_MHPMCOUNTER23H 0xb97

CSR_MHPMCOUNTER24H 0xb98

CSR_MHPMCOUNTER25H 0xb99

CSR_MHPMCOUNTER26H 0xb9a

CSR_MHPMCOUNTER27H 0xb9b

CSR_MHPMCOUNTER28H 0xb9c

CSR_MHPMCOUNTER29H 0xb9d

CSR_MHPMCOUNTER30H 0xb9e

CSR_MHPMCOUNTER31H 0xb9f

CSR_MTVT 0x307

CSR_MNXTI 0x345

CSR_MINTSTATUS 0x346

CSR_MSCRATCHCSW 0x348

CSR_MSCRATCHCSWL 0x349

CSR_MCLICBASE 0x350

CSR_UCODE 0x801

CSR_MCOUNTINHIBIT 0x320

CSR_MILM_CTL 0x7C0

CSR_MDLM_CTL 0x7C1

CSR_MECC_CODE 0x7C2

CSR_MNVEC 0x7C3

CSR_MSUBM 0x7C4

CSR_MDCAUSE 0x7C9

CSR_MCACHE_CTL 0x7CA

CSR_MMISC_CTL 0x7D0

CSR_MSAVESTATUS 0x7D6

CSR_MSAVEEPC1 0x7D7

CSR_MSAVECAUSE1 0x7D8

CSR_MSAVEEPC2 0x7D9

CSR_MSAVECAUSE2 0x7DA

CSR_MSAVEDCAUSE1 0x7DB

CSR_MSAVEDCAUSE2 0x7DC

CSR_MTLB_CTL 0x7DD

CSR_MECC_LOCK 0x7DE

CSR_PUSHMSUBM 0x7EB

CSR_MTVT2 0x7EC

CSR_JALMNXTI 0x7ED

CSR_PUSHMCAUSE 0x7EE

CSR_PUSHMEPC 0x7EF

CSR_MPPICFG_INFO 0x7F0

CSR_MFIOCFG_INFO 0x7F1

CSR_SLEEPVALUE 0x811

CSR_TXEVT 0x812

CSR_WFE 0x810

CSR_MICFG_INFO 0xFC0

CSR_MDCFG_INFO 0xFC1

CSR_MCFG_INFO 0xFC2

CSR_MTLBCFG_INFO 0xFC3

CSR_CCM_MBEGINADDR 0x7CB

CSR_CCM_MCOMMAND 0x7CC

CSR_CCM_MDATA 0x7CD

CSR_CCM_SUEN 0x7CE

CSR_CCM_SBEGINADDR 0x5CB

CSR_CCM_SCOMMAND 0x5CC

CSR_CCM_SDATA 0x5CD

CSR_CCM_UBEGINADDR 0x4CB

CSR_CCM_UCOMMAND 0x4CC

CSR_CCM_UDATA 0x4CD

CSR_CCM_FPIPE 0x4CF

MSTATUS_UIE 0x00000001

MSTATUS_SIE 0x00000002

MSTATUS_HIE 0x00000004

MSTATUS_MIE 0x00000008

MSTATUS_UPIE 0x00000010

MSTATUS_SPIE 0x00000020

MSTATUS_HPIE 0x00000040

MSTATUS_MPIE 0x00000080

MSTATUS_SPP 0x00000100

MSTATUS_MPP 0x00001800

MSTATUS_FS 0x00006000

MSTATUS_XS 0x00018000

MSTATUS_MPRV 0x00020000

MSTATUS_PUM 0x00040000

MSTATUS_MXR 0x00080000

MSTATUS_VM 0x1F000000

MSTATUS32_SD 0x80000000

MSTATUS64_SD 0x8000000000000000

MSTATUS_FS_INITIAL 0x00002000

MSTATUS_FS_CLEAN 0x00004000

MSTATUS_FS_DIRTY 0x00006000

SSTATUS_UIE 0x00000001

SSTATUS_SIE 0x00000002

SSTATUS_UPIE 0x00000010

SSTATUS_SPIE 0x00000020

SSTATUS_SPP 0x00000100

SSTATUS_FS 0x00006000

SSTATUS_XS 0x00018000

SSTATUS_PUM 0x00040000

SSTATUS32_SD 0x80000000

SSTATUS64_SD 0x8000000000000000

CSR_MCACHE_CTL_IE 0x00000001

CSR_MCACHE_CTL_DE 0x00010000

DCSR_XDEBUGVER (3U<<30)

DCSR_NDRESET (1<<29)

DCSR_FULLRESET (1<<28)

DCSR_EBREAKM (1<<15)

DCSR_EBREAKH (1<<14)

DCSR_EBREAKS (1<<13)

DCSR_EBREAKU (1<<12)

DCSR_STOPCYCLE (1<<10)

DCSR_STOPTIME (1<<9)

DCSR_CAUSE (7<<6)

DCSR_DEBUGINT (1<<5)

DCSR_HALT (1<<3)

DCSR_STEP (1<<2)

DCSR_PRV (3<<0)

DCSR_CAUSE_NONE 0

DCSR_CAUSE_SWBP 1

DCSR_CAUSE_HWBP 2

DCSR_CAUSE_DEBUGINT 3

DCSR_CAUSE_STEP 4

DCSR_CAUSE_HALT 5

MCONTROL_TYPE (xlen) (0xfULL<<((xlen)-4))

MCONTROL_DMODE (xlen) (1ULL<<((xlen)-5))

MCONTROL_MASKMAX (xlen) (0x3fULL<<((xlen)-11))

MCONTROL_SELECT (1<<19)

MCONTROL_TIMING (1<<18)

MCONTROL_ACTION (0x3f<<12)

MCONTROL_CHAIN (1<<11)

MCONTROL_MATCH (0xf<<7)

MCONTROL_M (1<<6)

MCONTROL_H (1<<5)

MCONTROL_S (1<<4)

MCONTROL_U (1<<3)

MCONTROL_EXECUTE (1<<2)

MCONTROL_STORE (1<<1)

MCONTROL_LOAD (1<<0)

MCONTROL_TYPE_NONE 0

MCONTROL_TYPE_MATCH 2

MCONTROL_ACTION_DEBUG_EXCEPTION 0

MCONTROL_ACTION_DEBUG_MODE 1

MCONTROL_ACTION_TRACE_START 2

MCONTROL_ACTION_TRACE_STOP 3

MCONTROL_ACTION_TRACE_EMIT 4

MCONTROL_MATCH_EQUAL 0

MCONTROL_MATCH_NAPOT 1

MCONTROL_MATCH_GE 2

MCONTROL_MATCH_LT 3

MCONTROL_MATCH_MASK_LOW 4

MCONTROL_MATCH_MASK_HIGH 5

MIP_SSIP (1 << IRQ_S_SOFT)

MIP_HSIP (1 << IRQ_H_SOFT)

MIP_MSIP (1 << IRQ_M_SOFT)

MIP_STIP (1 << IRQ_S_TIMER)

MIP_HTIP (1 << IRQ_H_TIMER)

MIP_MTIP (1 << IRQ_M_TIMER)

MIP_SEIP (1 << IRQ_S_EXT)

MIP_HEIP (1 << IRQ_H_EXT)

MIP_MEIP (1 << IRQ_M_EXT)

MIE_SSIE MIP_SSIP

MIE_HSIE MIP_HSIP

MIE_MSIE MIP_MSIP

MIE_STIE MIP_STIP

MIE_HTIE MIP_HTIP

MIE_MTIE MIP_MTIP

MIE_SEIE MIP_SEIP

MIE_HEIE MIP_HEIP

MIE_MEIE MIP_MEIP

UCODE_OV (0x1)

WFE_WFE (0x1)

TXEVT_TXEVT (0x1)

SLEEPVALUE_SLEEPVALUE (0x1)

MCOUNTINHIBIT_IR (1<<2)

MCOUNTINHIBIT_CY (1<<0)

MILM_CTL_ILM_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MILM_CTL_ILM_RWECC (1<<3)

MILM_CTL_ILM_ECC_EXCP_EN (1<<2)

MILM_CTL_ILM_ECC_EN (1<<1)

MILM_CTL_ILM_EN (1<<0)

MDLM_CTL_DLM_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MDLM_CTL_DLM_RWECC (1<<3)

MDLM_CTL_DLM_ECC_EXCP_EN (1<<2)

MDLM_CTL_DLM_ECC_EN (1<<1)

MDLM_CTL_DLM_EN (1<<0)

MSUBM_PTYP (0x3<<8)

MSUBM_TYP (0x3<<6)

MDCAUSE_MDCAUSE (0x3)

MMISC_CTL_NMI_CAUSE_FFF (1<<9)

MMISC_CTL_MISALIGN (1<<6)

MMISC_CTL_BPU (1<<3)

MCACHE_CTL_IC_EN (1<<0)

MCACHE_CTL_IC_SCPD_MOD (1<<1)

MCACHE_CTL_IC_ECC_EN (1<<2)

MCACHE_CTL_IC_ECC_EXCP_EN (1<<3)

MCACHE_CTL_IC_RWTECC (1<<4)

MCACHE_CTL_IC_RWDECC (1<<5)

MCACHE_CTL_DC_EN (1<<16)

MCACHE_CTL_DC_ECC_EN (1<<17)

MCACHE_CTL_DC_ECC_EXCP_EN (1<<18)

MCACHE_CTL_DC_RWTECC (1<<19)

MCACHE_CTL_DC_RWDECC (1<<20)

MTVT2_MTVT2EN (1<<0)

MTVT2_COMMON_CODE_ENTRY (((1ULL<<((__riscv_xlen)-2))-1)<<2)

MCFG_INFO_TEE (1<<0)

MCFG_INFO_ECC (1<<1)

MCFG_INFO_CLIC (1<<2)

MCFG_INFO_PLIC (1<<3)

MCFG_INFO_FIO (1<<4)

MCFG_INFO_PPI (1<<5)

MCFG_INFO_NICE (1<<6)

MCFG_INFO_ILM (1<<7)

MCFG_INFO_DLM (1<<8)

MCFG_INFO_ICACHE (1<<9)

MCFG_INFO_DCACHE (1<<10)

MICFG_IC_SET (0xF<<0)

MICFG_IC_WAY (0x7<<4)

MICFG_IC_LSIZE (0x7<<7)

MICFG_IC_ECC (0x1<<10)

MICFG_ILM_SIZE (0x1F<<16)

MICFG_ILM_XONLY (0x1<<21)

MICFG_ILM_ECC (0x1<<22)

MDCFG_DC_SET (0xF<<0)

MDCFG_DC_WAY (0x7<<4)

MDCFG_DC_LSIZE (0x7<<7)

MDCFG_DC_ECC (0x1<<10)

MDCFG_DLM_SIZE (0x1F<<16)

MDCFG_DLM_ECC (0x1<<21)

MPPICFG_INFO_PPI_SIZE (0x1F<<1)

MPPICFG_INFO_PPI_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MFIOCFG_INFO_FIO_SIZE (0x1F<<1)

MFIOCFG_INFO_FIO_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MECC_LOCK_ECC_LOCK (0x1)

MECC_CODE_CODE (0x1FF)

MECC_CODE_RAMID (0x1F<<16)

MECC_CODE_SRAMID (0x1F<<24)

CCM_SUEN_SUEN (0x1<<0)

CCM_DATA_DATA (0x7<<0)

CCM_COMMAND_COMMAND (0x1F<<0)

SIP_SSIP MIP_SSIP

SIP_STIP MIP_STIP

PRV_U 0

PRV_S 1

PRV_H 2

PRV_M 3

VM_MBARE 0

VM_MBB 1

VM_MBBID 2

VM_SV32 8

VM_SV39 9

VM_SV48 10

IRQ_S_SOFT 1

IRQ_H_SOFT 2

IRQ_M_SOFT 3

IRQ_S_TIMER 5

IRQ_H_TIMER 6

IRQ_M_TIMER 7

IRQ_S_EXT 9

IRQ_H_EXT 10

IRQ_M_EXT 11

IRQ_COP 12

IRQ_HOST 13

FRM_RNDMODE_RNE 0x0

FRM_RNDMODE_RTZ 0x1

FRM_RNDMODE_RDN 0x2

FRM_RNDMODE_RUP 0x3

FRM_RNDMODE_RMM 0x4

FRM_RNDMODE_DYN 0x7

FFLAGS_AE_NX (1<<0)

FFLAGS_AE_UF (1<<1)

FFLAGS_AE_OF (1<<2)

FFLAGS_AE_DZ (1<<3)

FFLAGS_AE_NV (1<<4)

FREG (idx) f##idx

PMP_R 0x01

PMP_W 0x02

PMP_X 0x04

PMP_A 0x18

PMP_A_TOR 0x08

PMP_A_NA4 0x10

PMP_A_NAPOT 0x18

PMP_L 0x80

PMP_SHIFT 2

PMP_COUNT 16

PTE_V 0x001

PTE_R 0x002

PTE_W 0x004

PTE_X 0x008

PTE_U 0x010

PTE_G 0x020

PTE_A 0x040

PTE_D 0x080

PTE_SOFT 0x300

PTE_PPN_SHIFT 10

PTE_TABLE (PTE) (((PTE) & (PTE_V | PTE_R | PTE_W | PTE_X)) == PTE_V)

CAUSE_MISALIGNED_FETCH 0x0

CAUSE_FAULT_FETCH 0x1

CAUSE_ILLEGAL_INSTRUCTION 0x2

CAUSE_BREAKPOINT 0x3

CAUSE_MISALIGNED_LOAD 0x4

CAUSE_FAULT_LOAD 0x5

CAUSE_MISALIGNED_STORE 0x6

CAUSE_FAULT_STORE 0x7

CAUSE_USER_ECALL 0x8

CAUSE_SUPERVISOR_ECALL 0x9

CAUSE_HYPERVISOR_ECALL 0xa

CAUSE_MACHINE_ECALL 0xb

DCAUSE_FAULT_FETCH_PMP 0x1

DCAUSE_FAULT_FETCH_INST 0x2

DCAUSE_FAULT_LOAD_PMP 0x1

DCAUSE_FAULT_LOAD_INST 0x2

DCAUSE_FAULT_LOAD_NICE 0x3

DCAUSE_FAULT_STORE_PMP 0x1

DCAUSE_FAULT_STORE_INST 0x2

group **NMSIS_Core_CSR_Encoding**

NMSIS Core CSR Encodings.

The following macros are used for CSR encodings

Defines

MSTATUS_UIE 0x00000001

MSTATUS_SIE 0x00000002

MSTATUS_HIE 0x00000004

MSTATUS_MIE 0x00000008

MSTATUS_UPIE 0x00000010

MSTATUS_SPIE 0x00000020

MSTATUS_HPIE 0x00000040

MSTATUS_MPIE 0x00000080

MSTATUS_SPP 0x00000100

MSTATUS_MPP 0x00001800

MSTATUS_FS 0x00006000

MSTATUS_XS 0x00018000

MSTATUS_MPRV 0x00020000

MSTATUS_PUM 0x00040000

MSTATUS_MXR 0x00080000

MSTATUS_VM 0x1F000000

MSTATUS32_SD 0x80000000

MSTATUS64_SD 0x8000000000000000

MSTATUS_FS_INITIAL 0x00002000

MSTATUS_FS_CLEAN 0x00004000

MSTATUS_FS_DIRTY 0x00006000

SSTATUS_UIE 0x00000001

SSTATUS_SIE 0x00000002

SSTATUS_UPIE 0x00000010

SSTATUS_SPIE 0x00000020

SSTATUS_SPP 0x00000100

SSTATUS_FS 0x00006000

SSTATUS_XS 0x00018000

SSTATUS_PUM 0x00040000

SSTATUS32_SD 0x80000000

SSTATUS64_SD 0x8000000000000000

CSR_MCACHE_CTL_IE 0x00000001

CSR_MCACHE_CTL_DE 0x00010000

DCSR_XDEBUGVER (3U<<30)

DCSR_NDRESET (1<<29)

DCSR_FULLRESET (1<<28)

DCSR_EBREAKM (1<<15)

DCSR_EBREAKH (1<<14)

DCSR_EBREAKS (1<<13)

DCSR_EBREAKU (1<<12)

DCSR_STOPCYCLE (1<<10)

DCSR_STOPTIME (1<<9)

DCSR_CAUSE (7<<6)

DCSR_DEBUGINT (1<<5)

DCSR_HALT (1<<3)

DCSR_STEP (1<<2)

DCSR_PRV (3<<0)

DCSR_CAUSE_NONE 0

DCSR_CAUSE_SWBP 1

DCSR_CAUSE_HWBP 2

DCSR_CAUSE_DEBUGINT 3

DCSR_CAUSE_STEP 4

DCSR_CAUSE_HALT 5

MCONTROL_TYPE(xlen) (0xfULL<<((xlen)-4))

MCONTROL_DMODE(xlen) (1ULL<<((xlen)-5))

MCONTROL_MASKMAX(xlen) (0x3fULL<<((xlen)-11))

MCONTROL_SELECT (1<<19)

MCONTROL_TIMING (1<<18)

MCONTROL_ACTION (0x3f<<12)

MCONTROL_CHAIN (1<<11)

MCONTROL_MATCH (0xf<<7)

MCONTROL_M (1<<6)

MCONTROL_H (1<<5)

MCONTROL_S (1<<4)

MCONTROL_U (1<<3)

MCONTROL_EXECUTE (1<<2)

MCONTROL_STORE (1<<1)

MCONTROL_LOAD (1<<0)

MCONTROL_TYPE_NONE 0

MCONTROL_TYPE_MATCH 2

MCONTROL_ACTION_DEBUG_EXCEPTION 0

MCONTROL_ACTION_DEBUG_MODE 1

MCONTROL_ACTION_TRACE_START 2

MCONTROL_ACTION_TRACE_STOP 3

MCONTROL_ACTION_TRACE_EMIT 4

MCONTROL_MATCH_EQUAL 0

MCONTROL_MATCH_NAPOT 1

MCONTROL_MATCH_GE 2

MCONTROL_MATCH_LT 3

MCONTROL_MATCH_MASK_LOW 4

MCONTROL_MATCH_MASK_HIGH 5

MIP_SSIP (1 << IRQ_S_SOFT)

MIP_HSIP (1 << IRQ_H_SOFT)

MIP_MSIP (1 << IRQ_M_SOFT)

MIP_STIP (1 << IRQ_S_TIMER)

MIP_HTIP (1 << IRQ_H_TIMER)

MIP_MTIP (1 << IRQ_M_TIMER)

MIP_SEIP (1 << IRQ_S_EXT)

MIP_HEIP (1 << IRQ_H_EXT)

MIP_MEIP (1 << IRQ_M_EXT)

MIE_SSIE MIP_SSIP

MIE_HSIE MIP_HSIP

MIE_MSIE MIP_MSIP

MIE_STIE MIP_STIP

MIE_HTIE MIP_HTIP

MIE_MTIE MIP_MTIP

MIE_SEIE MIP_SEIP

MIE_HEIE MIP_HEIP

MIE_MEIE MIP_MEIP

UCODE_OV (0x1)

WFE_WFE (0x1)

TXEVT_TXEVT (0x1)

SLEEPVALUE_SLEEPVALUE (0x1)

MCOUNTINHIBIT_IR (1<<2)

MCOUNTINHIBIT_CY (1<<0)

MILM_CTL_ILM_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MILM_CTL_ILM_RWECC (1<<3)

MILM_CTL_ILM_ECC_EXCP_EN (1<<2)

MILM_CTL_ILM_ECC_EN (1<<1)

MILM_CTL_ILM_EN (1<<0)

MDLM_CTL_DLM_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MDLM_CTL_DLM_RWECC (1<<3)

MDLM_CTL_DLM_ECC_EXCP_EN (1<<2)

MDLM_CTL_DLM_ECC_EN (1<<1)

MDLM_CTL_DLM_EN (1<<0)

MSUBM_PTYP (0x3<<8)

MSUBM_TYP (0x3<<6)

MDCAUSE_MDCAUSE (0x3)

MMISC_CTL_NMI_CAUSE_FFF (1<<9)

MMISC_CTL_MISALIGN (1<<6)

MMISC_CTL_BPU (1<<3)

MCACHE_CTL_IC_EN (1<<0)

MCACHE_CTL_IC_SCPD_MOD (1<<1)

MCACHE_CTL_IC_ECC_EN (1<<2)

MCACHE_CTL_IC_ECC_EXCP_EN (1<<3)

MCACHE_CTL_IC_RWTECC (1<<4)

MCACHE_CTL_IC_RWDECC (1<<5)

MCACHE_CTL_DC_EN (1<<16)

MCACHE_CTL_DC_ECC_EN (1<<17)

MCACHE_CTL_DC_ECC_EXCP_EN (1<<18)

MCACHE_CTL_DC_RWTECC (1<<19)

MCACHE_CTL_DC_RWDECC (1<<20)

MTVT2_MTVT2EN (1<<0)

MTVT2_COMMON_CODE_ENTRY (((1ULL<<((__riscv_xlen)-2))-1)<<2)

MCFG_INFO_TEE (1<<0)

MCFG_INFO_ECC (1<<1)

MCFG_INFO_CLIC (1<<2)

MCFG_INFO_PLIC (1<<3)

MCFG_INFO_FIO (1<<4)

MCFG_INFO_PPI (1<<5)

MCFG_INFO_NICE (1<<6)

MCFG_INFO_ILM (1<<7)

MCFG_INFO_DLM (1<<8)

MCFG_INFO_ICACHE (1<<9)

MCFG_INFO_DCACHE (1<<10)

MICFG_IC_SET (0xF<<0)

MICFG_IC_WAY (0x7<<4)

MICFG_IC_LSIZE (0x7<<7)

MICFG_IC_ECC (0x1<<10)

MICFG_ILM_SIZE (0x1F<<16)

MICFG_ILM_XONLY (0x1<<21)

MICFG_ILM_ECC (0x1<<22)

MDCFG_DC_SET (0xF<<0)

MDCFG_DC_WAY (0x7<<4)

MDCFG_DC_LSIZE (0x7<<7)

MDCFG_DC_ECC (0x1<<10)

MDCFG_DLM_SIZE (0x1F<<16)

MDCFG_DLM_ECC (0x1<<21)

MPPICFG_INFO_PPI_SIZE (0x1F<<1)

MPPICFG_INFO_PPI_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MFIOCFG_INFO_FIO_SIZE (0x1F<<1)

MFIOCFG_INFO_FIO_BPA (((1ULL<<((__riscv_xlen)-10))-1)<<10)

MECC_LOCK_ECC_LOCK (0x1)

MECC_CODE_CODE (0x1FF)

MECC_CODE_RAMID (0x1F<<16)

MECC_CODE_SRAMID (0x1F<<24)

CCM_SUEN_SUEN (0x1<<0)

CCM_DATA_DATA (0x7<<0)

CCM_COMMAND_COMMAND (0x1F<<0)

SIP_SSIP MIP_SSIP

SIP_STIP MIP_STIP

PRV_U 0

PRV_S 1

PRV_H 2

PRV_M 3

VM_MBARE 0

VM_MBB 1

VM_MBBID 2

VM_SV32 8

VM_SV39 9

VM_SV48 10

IRQ_S_SOFT 1

IRQ_H_SOFT 2

IRQ_M_SOFT 3

IRQ_S_TIMER 5

IRQ_H_TIMER 6

IRQ_M_TIMER 7

IRQ_S_EXT 9

IRQ_H_EXT 10

IRQ_M_EXT 11

IRQ_COP 12

IRQ_HOST 13

FRM_RNDMODE_RNE 0x0
FPU Round to Nearest, ties to Even.

FRM_RNDMODE_RTZ 0x1
FPU Round Towards Zero.

FRM_RNDMODE_RDN 0x2
FPU Round Down (towards -inf)

FRM_RNDMODE_RUP 0x3
FPU Round Up (towards +inf)

FRM_RNDMODE_RMM 0x4
FPU Round to nearest, ties to Max Magnitude.

FRM_RNDMODE_DYN 0x7
In instruction's rm, selects dynamic rounding mode.

In Rounding Mode register, Invalid

FFLAGS_AE_NX (1<<0)
FPU Inexact.

FFLAGS_AE_UF (1<<1)
FPU Underflow.

FFLAGS_AE_OF (1<<2)
FPU Overflow.

FFLAGS_AE_DZ (1<<3)
FPU Divide by Zero.

FFLAGS_AE_NV (1<<4)
FPU Invalid Operation.

FREG (idx) f##idx
Floating Point Register f0-f31, eg.

f0 -> FREG(0)

PMP_R 0x01

PMP_W 0x02

PMP_X 0x04

PMP_A 0x18

PMP_A_TOR 0x08

PMP_A_NA4 0x10

PMP_A_NAPOT 0x18

PMP_L 0x80

PMP_SHIFT 2

PMP_COUNT 16

PTE_V 0x001

PTE_R 0x002

PTE_W 0x004

PTE_X 0x008

PTE_U 0x010

PTE_G 0x020

PTE_A 0x040

PTE_D 0x080

PTE_SOFT 0x300

PTE_PPN_SHIFT 10

PTE_TABLE (PTE) (((PTE) & (PTE_V | PTE_R | PTE_W | PTE_X)) == PTE_V)

CAUSE_MISALIGNED_FETCH 0x0
End of Doxygen Group NMSIS_Core_CSR_Registers.

CAUSE_FAULT_FETCH 0x1

CAUSE_ILLEGAL_INSTRUCTION 0x2

CAUSE_BREAKPOINT 0x3

CAUSE_MISALIGNED_LOAD 0x4

CAUSE_FAULT_LOAD 0x5

CAUSE_MISALIGNED_STORE 0x6

CAUSE_FAULT_STORE 0x7

CAUSE_USER_ECALL 0x8

CAUSE_SUPERVISOR_ECALL 0x9

CAUSE_HYPERVISOR_ECALL 0xa

CAUSE_MACHINE_ECALL 0xb

DCAUSE_FAULT_FETCH_PMP 0x1

DCAUSE_FAULT_FETCH_INST 0x2

DCAUSE_FAULT_LOAD_PMP 0x1

DCAUSE_FAULT_LOAD_INST 0x2

DCAUSE_FAULT_LOAD_NICE 0x3

DCAUSE_FAULT_STORE_PMP 0x1

DCAUSE_FAULT_STORE_INST 0x2

2.5.5 Register Define and Type Definitions

Base Register Define and Type Definitions

union CSR_MISA_Type

Public Members

rv_csr_t (page 104) **a**

rv_csr_t (page 104) **b**

rv_csr_t (page 104) **c**

rv_csr_t (page 104) **d**

rv_csr_t (page 104) **e**

rv_csr_t (page 104) **f**

rv_csr_t (page 104) **g**

rv_csr_t (page 104) **h**

rv_csr_t (page 104) **i**

rv_csr_t (page 104) **j**

rv_csr_t (page 104) **_reserved1**

rv_csr_t (page 104) **l**

rv_csr_t (page 104) **m**

rv_csr_t (page 104) **n**

rv_csr_t (page 104) **_reserved2**

rv_csr_t (page 104) **p**

rv_csr_t (page 104) **q**

rv_csr_t (page 104) **_resreved3**

rv_csr_t (page 104) **s**

rv_csr_t (page 104) **t**

rv_csr_t (page 104) **u**

rv_csr_t (page 104) **v**

rv_csr_t (page 104) **_reserved4**

rv_csr_t (page 104) **x**

rv_csr_t (page 104) **_reserved5**

rv_csr_t (page 104) **mxl**

struct *CSR_MISA_Type* (page 90)::**[anonymous] b**

union CSR_MSTATUS_Type

Public Members

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **sie**

rv_csr_t (page 104) **_reserved1**

rv_csr_t (page 104) **mie**

rv_csr_t (page 104) **_reserved2**

rv_csr_t (page 104) **spie**

rv_csr_t (page 104) **_reserved3**

rv_csr_t (page 104) **mpie**

rv_csr_t (page 104) **_reserved4**

rv_csr_t (page 104) **mpp**

rv_csr_t (page 104) **fs**

rv_csr_t (page 104) **xs**

rv_csr_t (page 104) **mprv**

rv_csr_t (page 104) **sum**

rv_csr_t (page 104) **_reserved6**

rv_csr_t (page 104) **sd**

struct *CSR_MSTATUS_Type* (page 91)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MTVEC_Type

Public Members

rv_csr_t (page 104) **mode**

rv_csr_t (page 104) **addr**

struct *CSR_MTVEC_Type* (page 92)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MCAUSE_Type

Public Members

rv_csr_t (page 104) **exccode**

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **mpil**

rv_csr_t (page 104) **_reserved1**

rv_csr_t (page 104) **mpie**

rv_csr_t (page 104) **mpp**

rv_csr_t (page 104) **minhv**

rv_csr_t (page 104) **interrupt**

struct *CSR_MCAUSE_Type* (page 92)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MCOUNTINHIBIT_Type

Public Members

rv_csr_t (page 104) **cy**

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **ir**

rv_csr_t (page 104) **_reserved1**

struct *CSR_MCOUNTINHIBIT_Type* (page 92)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MSUBM_Type

Public Members

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **typ**

rv_csr_t (page 104) **ptyp**

rv_csr_t (page 104) **_reserved1**

struct *CSR_MSUBM_Type* (page 92)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MMISCCTRL_Type

Public Members

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **bpu**

rv_csr_t (page 104) **_reserved1**

rv_csr_t (page 104) **misalign**

rv_csr_t (page 104) **_reserved2**

rv_csr_t (page 104) **nmi_cause**

rv_csr_t (page 104) **_reserved3**

struct *CSR_MMISCCTRL_Type* (page 93)::**[anonymous] b**

rv_csr_t (page 104) **d**

union CSR_MSAVESTATUS_Type

Public Members

rv_csr_t (page 104) **mpie1**

rv_csr_t (page 104) **mpp1**

rv_csr_t (page 104) **_reserved0**

rv_csr_t (page 104) **ptyp1**

rv_csr_t (page 104) **mpie2**

rv_csr_t (page 104) **mpp2**

rv_csr_t (page 104) **_reserved1**

rv_csr_t (page 104) **ptyp2**

rv_csr_t (page 104) **_reserved2**

struct *CSR_MSAVESTATUS_Type* (page 93)::**[anonymous] b**

rv_csr_t (page 104) **w**

group **NMSIS_Core_Base_Registers**

Type definitions and defines for base core registers.

union CSR_MISA_Type

#include <core_feature_base.h> Union type to access MISA register.

Public Members

rv_csr_t (page 104) **a**
bit: 0 Atomic extension

rv_csr_t (page 104) **b**
bit: 1 Tentatively reserved for Bit-Manipulation extension

rv_csr_t (page 104) **c**
bit: 2 Compressed extension

rv_csr_t (page 104) **d**
bit: 3 Double-precision floating-point extension

Type used for csr data access.

rv_csr_t (page 104) **e**
bit: 4 RV32E base ISA

rv_csr_t (page 104) **f**
bit: 5 Single-precision floating-point extension

rv_csr_t (page 104) **g**
bit: 6 Additional standard extensions present

rv_csr_t (page 104) **h**
bit: 7 Hypervisor extension

rv_csr_t (page 104) **i**
bit: 8 RV32I/64I/128I base ISA

rv_csr_t (page 104) **j**
bit: 9 Tentatively reserved for Dynamically Translated Languages extension

rv_csr_t (page 104) **_reserved1**
bit: 10 Reserved

rv_csr_t (page 104) **l**
bit: 11 Tentatively reserved for Decimal Floating-Point extension

rv_csr_t (page 104) **m**
bit: 12 Integer Multiply/Divide extension

rv_csr_t (page 104) **n**
bit: 13 User-level interrupts supported

rv_csr_t (page 104) **_reserved2**
bit: 14 Reserved

rv_csr_t (page 104) **p**
bit: 15 Tentatively reserved for Packed-SIMD extension

rv_csr_t (page 104) **q**
bit: 16 Quad-precision floating-point extension

rv_csr_t (page 104) **_resreved3**
bit: 17 Reserved

rv_csr_t (page 104) **s**
bit: 18 Supervisor mode implemented

rv_csr_t (page 104) **t**
bit: 19 Tentatively reserved for Transactional Memory extension

rv_csr_t (page 104) **u**
bit: 20 User mode implemented

rv_csr_t (page 104) **v**
bit: 21 Tentatively reserved for Vector extension

rv_csr_t (page 104) **_reserved4**
bit: 22 Reserved

rv_csr_t (page 104) **x**
bit: 23 Non-standard extensions present

rv_csr_t (page 104) **_reserved5**
bit: 24..29 Reserved

rv_csr_t (page 104) **mxl**
bit: 30..31 Machine XLEN

struct *CSR_MISA_Type* (page 90)::**[anonymous] b**
Structure used for bit access.

union CSR_MSTATUS_Type
#include <core_feature_base.h> Union type to access MSTATUS configure register.

Public Members

rv_csr_t (page 104) **_reserved0**
bit: 0 Reserved

rv_csr_t (page 104) **sie**
bit: 1 supervisor interrupt enable flag

rv_csr_t (page 104) **_reserved1**
bit: 2 Reserved

rv_csr_t (page 104) **mie**
bit: 3 Machine mode interrupt enable flag

rv_csr_t (page 104) **_reserved2**
bit: 4 Reserved

rv_csr_t (page 104) **spie**
bit: 3 Supervisor Privilede mode interrupt enable flag

rv_csr_t (page 104) **_reserved3**
bit: Reserved

rv_csr_t (page 104) **mpie**
bit: mirror of MIE flag

rv_csr_t (page 104) **_reserved4**
bit: Reserved

rv_csr_t (page 104) **mpp**
bit: mirror of Privilege Mode

rv_csr_t (page 104) **fs**
bit: FS status flag

rv_csr_t (page 104) **xs**
bit: XS status flag

rv_csr_t (page 104) **mprv**
bit: Machine mode PMP

rv_csr_t (page 104) **sum**
bit: Supervisor Mode load and store protection

rv_csr_t (page 104) **_reserved6**
bit: 19..30 Reserved

rv_csr_t (page 104) **sd**
bit: Dirty status for XS or FS

struct *CSR_MSTATUS_Type* (page 91)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MTVEC_Type
#include <core_feature_base.h> Union type to access MTVEC configure register.

Public Members

rv_csr_t (page 104) **mode**
bit: 0..5 interrupt mode control

rv_csr_t (page 104) **addr**
bit: 6..31 mtvec address

struct *CSR_MTVEC_Type* (page 92)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MCAUSE_Type
#include <core_feature_base.h> Union type to access MCAUSE configure register.

Public Members

rv_csr_t (page 104) **exccode**
bit: 11..0 exception or interrupt code

rv_csr_t (page 104) **_reserved0**
bit: 15..12 Reserved

rv_csr_t (page 104) **mpil**
bit: 23..16 Previous interrupt level

rv_csr_t (page 104) **_reserved1**
bit: 26..24 Reserved

rv_csr_t (page 104) **mpie**
bit: 27 Interrupt enable flag before enter interrupt

rv_csr_t (page 104) **mpp**
bit: 29..28 Privilede mode flag before enter interrupt

rv_csr_t (page 104) **minhv**
bit: 30 Machine interrupt vector table

rv_csr_t (page 104) **interrupt**
bit: 31 trap type.

0 means exception and 1 means interrupt

struct *CSR_MCAUSE_Type* (page 92)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MCOUNTINHIBIT_Type
#include <core_feature_base.h> Union type to access MCOUNTINHIBIT configure register.

Public Members

rv_csr_t (page 104) **cy**
bit: 0 1 means disable mcycle counter

rv_csr_t (page 104) **_reserved0**
bit: 1 Reserved

rv_csr_t (page 104) **ir**
bit: 2 1 means disable minstret counter

rv_csr_t (page 104) **_reserved1**
bit: 3..31 Reserved

struct *CSR_MCOUNTINHIBIT_Type* (page 92)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MSUBM_Type
#include <core_feature_base.h> Union type to access msubm configure register.

Public Members

rv_csr_t (page 104) **_reserved0**
bit: 0..5 Reserved

rv_csr_t (page 104) **typ**
bit: 6..7 current trap type

rv_csr_t (page 104) **ptyp**
bit: 8..9 previous trap type

rv_csr_t (page 104) **_reserved1**
bit: 10..31 Reserved

struct *CSR_MSUBM_Type* (page 92)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MMISCCTRL_Type
#include <core_feature_base.h> Union type to access MMISC_CTRL configure register.

Public Members

rv_csr_t (page 104) **_reserved0**
bit: 0..2 Reserved

rv_csr_t (page 104) **bpu**
bit: 3 dynamic prediction enable flag

rv_csr_t (page 104) **_reserved1**
bit: 4..5 Reserved

rv_csr_t (page 104) **misalign**
bit: 6 misaligned access support flag

rv_csr_t (page 104) **_reserved2**
bit: 7..8 Reserved

rv_csr_t (page 104) **nmi_cause**
bit: 9 mnvec control and nmi mcase exccode

rv_csr_t (page 104) **_reserved3**
bit: 10..31 Reserved

struct *CSR_MMISCCTRL_Type* (page 93)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **d**
Type used for csr data access.

union CSR_MSAVESTATUS_Type
#include <core_feature_base.h> Union type to access MSAVESTATUS configure register.

Public Members

rv_csr_t (page 104) **mpie1**
bit: 0 interrupt enable flag of fisrt level NMI/exception nestting

rv_csr_t (page 104) **mpp1**
bit: 1..2 privilede mode of fisrt level NMI/exception nestting

rv_csr_t (page 104) **_reserved0**
bit: 3..5 Reserved

rv_csr_t (page 104) **ptyp1**
bit: 6..7 NMI/exception type of before first nestting

rv_csr_t (page 104) **mpie2**
bit: 8 interrupt enable flag of second level NMI/exception nestting

rv_csr_t (page 104) **mpp2**
bit: 9..10 privilede mode of second level NMI/exception nestting

rv_csr_t (page 104) **_reserved1**
bit: 11..13 Reserved

rv_csr_t (page 104) **ptyp2**
bit: 14..15 NMI/exception type of before second nestting

rv_csr_t (page 104) **_reserved2**
bit: 16..31 Reserved

struct *CSR_MSAVESTATUS_Type* (page 93)::**[anonymous] b**
Structure used for bit access.

rv_csr_t (page 104) **w**
Type used for csr data access.

Register Define and Type Definitions Of ECLIC

enum NMSIS_Core_ECLIC_Registers::**ECLIC_TRIGGER_Type**
Values:

ECLIC_LEVEL_TRIGGER = 0x0

ECLIC_POSTIVE_EDGE_TRIGGER = 0x1

ECLIC_NEGTIVE_EDGE_TRIGGER = 0x3

ECLIC_MAX_TRIGGER = 0x3

CLIC_CLICCFG_NLBIT_Pos 1U

CLIC_CLICCFG_NLBIT_Msk (0xFUL << CLIC_CLICCFG_NLBIT_Pos)

CLIC_CLICINFO_CTLBIT_Pos 21U

CLIC_CLICINFO_CTLBIT_Msk (0xFUL << CLIC_CLICINFO_CTLBIT_Pos)

CLIC_CLICINFO_VER_Pos 13U

CLIC_CLICINFO_VER_Msk (0xFFUL << CLIC_CLICCFG_NLBIT_Pos)

CLIC_CLICINFO_NUM_Pos 0U

CLIC_CLICINFO_NUM_Msk (0xFFFUL << CLIC_CLICINFO_NUM_Pos)

CLIC_INTIP_IP_Pos 0U

CLIC_INTIP_IP_Msk (0x1UL << CLIC_INTIP_IP_Pos)

CLIC_INTIE_IE_Pos 0U

CLIC_INTIE_IE_Msk (0x1UL << CLIC_INTIE_IE_Pos)

CLIC_INTATTR_TRIG_Pos 1U

CLIC_INTATTR_TRIG_Msk (0x3UL << CLIC_INTATTR_TRIG_Pos)

CLIC_INTATTR_SHV_Pos 0U

CLIC_INTATTR_SHV_Msk (0x1UL << CLIC_INTATTR_SHV_Pos)

ECLIC_MAX_NLBITS 8U

ECLIC_MODE_MTVEC_Msk 3U

ECLIC_NON_VECTOR_INTERRUPT 0x0

ECLIC_VECTOR_INTERRUPT 0x1

ECLIC_BASE __ECLIC_BASEADDR

ECLIC ((CLIC_Type *) ECLIC_BASE)

union CLICCFG_Type

Public Members

uint8_t **_reserved0**

uint8_t **nlbits**

uint8_t **_reserved1**

uint8_t **_reserved2**

struct *CLICCFG_Type* (page 99)::**[anonymous] b**

uint8_t **w**

union CLICINFO_Type

Public Members

uint32_t **numint**

uint32_t **version**

uint32_t **intctlbits**

uint32_t **_reserved0**

struct *CLICINFO_Type* (page 100)::**[anonymous] b**

uint32_t **w**

group **NMSIS_Core_ECLIC_Registers**
Type definitions and defines for eclic registers.

Defines

CLIC_CLICCFG_NLBIT_Pos 1U
CLIC CLICCFG: NLBIT Position.

CLIC_CLICCFG_NLBIT_Msk (0xFUL << CLIC_CLICCFG_NLBIT_Pos)
CLIC CLICCFG: NLBIT Mask.

CLIC_CLICINFO_CTLBIT_Pos 21U
CLIC INTINFO: __ECLIC_GetInfoCtlbits() Position.

CLIC_CLICINFO_CTLBIT_Msk (0xFUL << CLIC_CLICINFO_CTLBIT_Pos)
CLIC INTINFO: __ECLIC_GetInfoCtlbits() Mask.

CLIC_CLICINFO_VER_Pos 13U
CLIC CLICINFO: VERSION Position.

CLIC_CLICINFO_VER_Msk (0xFFUL << CLIC_CLICCFG_NLBIT_Pos)
CLIC CLICINFO: VERSION Mask.

CLIC_CLICINFO_NUM_Pos 0U
CLIC CLICINFO: NUM Position.

CLIC_CLICINFO_NUM_Msk (0xFFFUL << CLIC_CLICINFO_NUM_Pos)
CLIC CLICINFO: NUM Mask.

CLIC_INTIP_IP_Pos 0U
CLIC INTIP: IP Position.

CLIC_INTIP_IP_Msk (0x1UL << CLIC_INTIP_IP_Pos)
CLIC INTIP: IP Mask.

CLIC_INTIE_IE_Pos 0U
CLIC INTIE: IE Position.

CLIC_INTIE_IE_Msk (0x1UL << CLIC_INTIE_IE_Pos)
CLIC INTIE: IE Mask.

CLIC_INTATTR_TRIG_Pos 1U
CLIC INTATTR: TRIG Position.

CLIC_INTATTR_TRIG_Msk (0x3UL << CLIC_INTATTR_TRIG_Pos)
CLIC INTATTR: TRIG Mask.

CLIC_INTATTR_SHV_Pos 0U
CLIC INTATTR: SHV Position.

CLIC_INTATTR_SHV_Msk (0x1UL << CLIC_INTATTR_SHV_Pos)
CLIC INTATTR: SHV Mask.

ECLIC_MAX_NLBITS 8U
Max nlbit of the CLICINTCTLBITS.

ECLIC_MODE_MTVEC_Msk 3U
ECLIC Mode mask for MTVT CSR Register.

ECLIC_NON_VECTOR_INTERRUPT 0x0
Non-Vector Interrupt Mode of ECLIC.

ECLIC_VECTOR_INTERRUPT 0x1
Vector Interrupt Mode of ECLIC.

ECLIC_BASE __ECLIC_BASEADDR
ECLIC Base Address.

ECLIC ((CLIC_Type *) ECLIC_BASE)
CLIC configuration struct.

Enums

enum ECLIC_TRIGGER_Type
ECLIC Trigger Enum for different Trigger Type.

Values:

ECLIC_LEVEL_TRIGGER = 0x0
Level Triggerred, trig[0] = 0.

ECLIC_POSTIVE_EDGE_TRIGGER = 0x1
Postive/Rising Edge Triggered, trig[0] = 1, trig[1] = 0.

ECLIC_NEGTIVE_EDGE_TRIGGER = 0x3
Negtive/Falling Edge Triggered, trig[0] = 1, trig[1] = 1.

ECLIC_MAX_TRIGGER = 0x3
MAX Supported Trigger Mode.

union CLICCFG_Type
#include <core_feature_eclic.h> Union type to access CLICFG configure register.

Public Members

uint8_t **_reserved0**
bit: 0 Overflow condition code flag

uint8_t **nlbits**
bit: 29 Carry condition code flag

uint8_t **_reserved1**
bit: 30 Zero condition code flag

uint8_t **_reserved2**
bit: 31 Negative condition code flag

struct *CLICCFG_Type* (page 99)::**[anonymous] b**
Structure used for bit access.

uint8_t **w**
Type used for byte access.

union CLICINFO_Type
#include <core_feature_eclic.h> Union type to access CLICINFO information register.

Public Members

uint32_t **numint**
bit: 0..12 number of maximum interrupt inputs supported

uint32_t **version**
bit: 13..20 20:17 for architecture version,16:13 for implementation version

uint32_t **intctlbits**
bit: 21..24 specifies how many hardware bits are actually implemented in the clicintctl registers

uint32_t **_reserved0**
bit: 25..31 Reserved

struct *CLICINFO_Type* (page 100)::**[anonymous] b**
Structure used for bit access.

uint32_t **w**
Type used for word access.

struct CLIC_CTRL_Type
#include <core_feature_eclic.h> Access to the structure of a vector interrupt controller.

Register Define and Type Definitions Of System Timer

SysTimer_MTIMECTL_TIMESTOP_Pos 0U

SysTimer_MTIMECTL_TIMESTOP_Msk (1UL << SysTimer_MTIMECTL_TIMESTOP_Pos)

SysTimer_MTIMECTL_CMPCLREN_Pos 1U

SysTimer_MTIMECTL_CMPCLREN_Msk (1UL << SysTimer_MTIMECTL_CMPCLREN_Pos)

SysTimer_MTIMECTL_CLKSRC_Pos 2U

SysTimer_MTIMECTL_CLKSRC_Msk (1UL << SysTimer_MTIMECTL_CLKSRC_Pos)

SysTimer_MSIP_MSIP_Pos 0U

SysTimer_MSIP_MSIP_Msk (1UL << SysTimer_MSIP_MSIP_Pos)

SysTimer_MTIMER_Msk (0xFFFFFFFFFFFFFFFFULL)

SysTimer_MTIMERCMP_Msk (0xFFFFFFFFFFFFFFFFULL)

SysTimer_MTIMECTL_Msk (0xFFFFFFFFUL)

SysTimer_MSIP_Msk (0xFFFFFFFFUL)

SysTimer_MSFTRST_Msk (0xFFFFFFFFUL)

SysTimer_MSFRST_KEY (0x80000A5FUL)

SysTimer_BASE __SYSTIMER_BASEADDR

SysTimer ((*SysTimer_Type* (page 104) *) SysTimer_BASE)

group **NMSIS_Core_SysTimer_Registers**

Type definitions and defines for system timer registers.

Defines

SysTimer_MTIMECTL_TIMESTOP_Pos 0U
SysTick Timer MTIMECTL: TIMESTOP bit Position.

SysTimer_MTIMECTL_TIMESTOP_Msk (1UL << SysTimer_MTIMECTL_TIMESTOP_Pos)
SysTick Timer MTIMECTL: TIMESTOP Mask.

SysTimer_MTIMECTL_CMPCLREN_Pos 1U
SysTick Timer MTIMECTL: CMPCLREN bit Position.

SysTimer_MTIMECTL_CMPCLREN_Msk (1UL << SysTimer_MTIMECTL_CMPCLREN_Pos)
SysTick Timer MTIMECTL: CMPCLREN Mask.

SysTimer_MTIMECTL_CLKSRC_Pos 2U
SysTick Timer MTIMECTL: CLKSRC bit Position.

SysTimer_MTIMECTL_CLKSRC_Msk (1UL << SysTimer_MTIMECTL_CLKSRC_Pos)
SysTick Timer MTIMECTL: CLKSRC Mask.

SysTimer_MSIP_MSIP_Pos 0U
SysTick Timer MSIP: MSIP bit Position.

SysTimer_MSIP_MSIP_Msk (1UL << SysTimer_MSIP_MSIP_Pos)
SysTick Timer MSIP: MSIP Mask.

SysTimer_MTIMER_Msk (0xFFFFFFFFFFFFFFFFULL)
SysTick Timer MTIMER value Mask.

SysTimer_MTIMERCMP_Msk (0xFFFFFFFFFFFFFFFFULL)
SysTick Timer MTIMERCMP value Mask.

SysTimer_MTIMECTL_Msk (0xFFFFFFFFUL)
SysTick Timer MTIMECTL/MSTOP value Mask.

SysTimer_MSIP_Msk (0xFFFFFFFFUL)
SysTick Timer MSIP value Mask.

SysTimer_MSFTRST_Msk (0xFFFFFFFFUL)
SysTick Timer MSFTRST value Mask.

SysTimer_MSFRST_KEY (0x80000A5FUL)
SysTick Timer Software Reset Request Key.

SysTimer_BASE __SYSTIMER_BASEADDR
SysTick Base Address.

SysTimer ((*SysTimer_Type* (page 104) *) SysTimer_BASE)
SysTick configuration struct.

struct SysTimer_Type
#include <core_feature_timer.h> Structure type to access the System Timer (SysTimer).

Structure definition to access the system timer(SysTimer).

Remark

- MSFTRST register is introduced in Nuclei N Core version 1.3(__NUCLEI_N_REV >= 0x0103)
- MSTOP register is renamed to MTIMECTL register in Nuclei N Core version 1.4(__NUCLEI_N_REV >= 0x0104)
- CMPCLREN and CLKSRC bit in MTIMECTL register is introduced in Nuclei N Core version 1.4(__NUCLEI_N_REV >= 0x0104)

typedef uint32_t **rv_csr_t**

__RISCV_XLEN 32

group **NMSIS_Core_Registers**
Type definitions and defines for core registers.

Defines

__RISCV_XLEN 32
Refer to the width of an integer register in bits(either 32 or 64)

Typedefs

typedef uint32_t **rv_csr_t**
Type of Control and Status Register(CSR), depends on the XLEN defined in RISC-V.

2.5.6 Core CSR Register Access

__STATIC_FORCEINLINE void __enable_irq(void)

__STATIC_FORCEINLINE void __disable_irq(void)

__STATIC_FORCEINLINE uint64_t __get_rv_cycle(void)

__STATIC_FORCEINLINE uint64_t __get_rv_instret(void)

__STATIC_FORCEINLINE uint64_t __get_rv_time(void)

__RV_CSR_SWAP (csr, val)

__RV_CSR_READ (csr)

__RV_CSR_WRITE (csr, val)

__RV_CSR_READ_SET (csr, val)

__RV_CSR_SET (csr, val)

__RV_CSR_READ_CLEAR (csr, val)

__RV_CSR_CLEAR (csr, val)

group **NMSIS_Core_CSR_Register_Access**

Functions to access the Core CSR Registers.

The following functions or macros provide access to Core CSR registers.

- *Core CSR Encodings* (page 83)
- *Core CSR Registers* (page 69)

Defines

__RV_CSR_SWAP (csr, val)

CSR operation Macro for csrrw instruction.

Read the content of csr register to __v, then write content of val into csr register, then return __v

Return the CSR register value before written

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: value to store into the CSR register

__RV_CSR_READ (csr)

CSR operation Macro for csrr instruction.

Read the content of csr register to __v and return it

Return the CSR register value

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS

__RV_CSR_WRITE (csr, val)

CSR operation Macro for csrw instruction.

Write the content of val to csr register

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: value to store into the CSR register

__RV_CSR_READ_SET (csr, val)

CSR operation Macro for csrrs instruction.

Read the content of csr register to __v, then set csr register to be __v | val, then return __v

Return the CSR register value before written

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: Mask value to be used wih csrrs instruction

__RV_CSR_SET (csr, val)

CSR operation Macro for csrs instruction.

Set csr register to be csr_content | val

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: Mask value to be used wih csrs instruction

__RV_CSR_READ_CLEAR (csr, val)
CSR operation Macro for csrrc instruction.

Read the content of csr register to __v, then set csr register to be __v & ~val, then return __v

Return the CSR register value before written

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: Mask value to be used wih csrrc instruction

__RV_CSR_CLEAR (csr, val)
CSR operation Macro for csrc instruction.

Set csr register to be csr_content & ~val

Parameters

- `csr`: CSR macro definition defined in *Core CSR Registers* (page 69), eg. CSR_MSTATUS
- `val`: Mask value to be used wih csrc instruction

Functions

`__STATIC_FORCEINLINE void __enable_irq(void)`
Enable IRQ Interrupts.

Enables IRQ interrupts by setting the MIE-bit in the MSTATUS Register.

Remark Can only be executed in Privileged modes.

`__STATIC_FORCEINLINE void __disable_irq(void)`
Disable IRQ Interrupts.

Disables IRQ interrupts by clearing the MIE-bit in the MSTATUS Register.

Remark Can only be executed in Privileged modes.

`__STATIC_FORCEINLINE uint64_t __get_rv_cycle(void)`
Read whole 64 bits value of mcycle counter.

This function will read the whole 64 bits of MCYCLE register

Return The whole 64 bits value of MCYCLE

Remark It will work for both RV32 and RV64 to get full 64bits value of MCYCLE

`__STATIC_FORCEINLINE uint64_t __get_rv_instret(void)`
Read whole 64 bits value of machine instruction-retired counter.

This function will read the whole 64 bits of MINSTRET register

Return The whole 64 bits value of MINSTRET

Remark It will work for both RV32 and RV64 to get full 64bits value of MINSTRET

`__STATIC_FORCEINLINE uint64_t __get_rv_time(void)`
Read whole 64 bits value of real-time clock.

This function will read the whole 64 bits of TIME register

Return The whole 64 bits value of TIME CSR

Remark It will work for both RV32 and RV64 to get full 64bits value of TIME

Attention only available when user mode available

2.5.7 Intrinsic Functions for CPU Intructions

enum NMSIS_Core_CPU_Intrinsic::**WFI_SleepMode_Type**

Values:

WFI_SHALLOW_SLEEP = 0

WFI_DEEP_SLEEP = 1

```
__STATIC_FORCEINLINE void __NOP(void)
__STATIC_FORCEINLINE void __WFI(void)
__STATIC_FORCEINLINE void __WFE(void)
__STATIC_FORCEINLINE void __EBREAK(void)
__STATIC_FORCEINLINE void __ECALL(void)
__STATIC_FORCEINLINE void __set_wfi_sleepmode(WFI_SleepMode_Type mode)
__STATIC_FORCEINLINE void __TXEVT(void)
__STATIC_FORCEINLINE void __enable_mcycle_counter(void)
__STATIC_FORCEINLINE void __disable_mcycle_counter(void)
__STATIC_FORCEINLINE void __enable_minstret_counter(void)
__STATIC_FORCEINLINE void __disable_minstret_counter(void)
__STATIC_FORCEINLINE void __enable_all_counter(void)
__STATIC_FORCEINLINE void __disable_all_counter(void)
__STATIC_FORCEINLINE void __FENCE_I(void)
__STATIC_FORCEINLINE uint8_t __LB(volatile void * addr)
__STATIC_FORCEINLINE uint16_t __LH(volatile void * addr)
__STATIC_FORCEINLINE uint32_t __LW(volatile void * addr)
__STATIC_FORCEINLINE void __SB(volatile void * addr, uint8_t val)
__STATIC_FORCEINLINE void __SH(volatile void * addr, uint16_t val)
__STATIC_FORCEINLINE void __SW(volatile void * addr, uint32_t val)
__STATIC_FORCEINLINE uint32_t __CAS_W(volatile uint32_t * addr, uint32_t oldval, uint32_t n
__STATIC_FORCEINLINE uint32_t __AMOSWAP_W(volatile uint32_t * addr, uint32_t newval)
__STATIC_FORCEINLINE int32_t __AMOADD_W(volatile int32_t * addr, int32_t value)
__STATIC_FORCEINLINE int32_t __AMOAND_W(volatile int32_t * addr, int32_t value)
__STATIC_FORCEINLINE int32_t __AMOOR_W(volatile int32_t * addr, int32_t value)
__STATIC_FORCEINLINE int32_t __AMOXOR_W(volatile int32_t * addr, int32_t value)
__STATIC_FORCEINLINE uint32_t __AMOMAXU_W(volatile uint32_t * addr, uint32_t value)
```

__STATIC_FORCEINLINE int32_t __AMOMAX_W(volatile int32_t * addr, int32_t value)

__STATIC_FORCEINLINE uint32_t __AMOMINU_W(volatile uint32_t * addr, uint32_t value)

__STATIC_FORCEINLINE int32_t __AMOMIN_W(volatile int32_t * addr, int32_t value)

__FENCE (p, s) __ASM volatile ("fence " #p "," #s : : : "memory")

__RWMB () __FENCE(iorw,iorw)

__RMB () __FENCE(ir,ir)

__WMB () __FENCE(ow,ow)

__SMP_RWMB () __FENCE(rw,rw)

__SMP_RMB () __FENCE(r,r)

__SMP_WMB () __FENCE(w,w)

__CPU_RELAX () __ASM volatile ("" : : : "memory")

group **NMSIS_Core_CPU_Intrinsic**

Functions that generate RISC-V CPU instructions.

The following functions generate specified RISC-V instructions that cannot be directly accessed by compiler.

Defines

__FENCE (p, s) __ASM volatile ("fence " #p "," #s : : : "memory")

Execute fence instruction, p -> pred, s -> succ.

the FENCE instruction ensures that all memory accesses from instructions preceding the fence in program order (the `predecessor set`) appear earlier in the global memory order than memory accesses from instructions appearing after the fence in program order (the `successor set`). For details, please refer to The RISC-V Instruction Set Manual

Parameters

- `p`: predecessor set, such as iorw, rw, r, w
- `s`: successor set, such as iorw, rw, r, w

__RWMB () __FENCE(iorw,iorw)

Read & Write Memory barrier.

__RMB () __FENCE(ir,ir)

Read Memory barrier.

__WMB () __FENCE(ow,ow)

Write Memory barrier.

__SMP_RWMB () __FENCE(rw,rw)

SMP Read & Write Memory barrier.

__SMP_RMB () __FENCE(r,r)

SMP Read Memory barrier.

__SMP_WMB () __FENCE(w,w)

SMP Write Memory barrier.

__CPU_RELAX () __ASM volatile ("" : : : "memory")

CPU relax for busy loop.

Enums

enum WFI_SleepMode_Type
WFI Sleep Mode enumeration.

Values:

WFI_SHALLOW_SLEEP = 0
Shallow sleep mode, the core_clk will poweroff.

WFI_DEEP_SLEEP = 1
Deep sleep mode, the core_clk and core_ano_clk will poweroff.

Functions

__STATIC_FORCEINLINE void __NOP(void)
NOP Instruction.

No Operation does nothing. This instruction can be used for code alignment purposes.

__STATIC_FORCEINLINE void __WFI(void)
Wait For Interrupt.

Wait For Interrupt is is executed using CSR_WFE.WFE=0 and WFI instruction. It will suspends execution until interrupt, NMI or Debug happened. When Core is waked up by interrupt, if

1. mstatus.MIE == 1(interrupt enabled), Core will enter ISR code
2. mstatus.MIE == 0(interrupt disabled), Core will resume previous execution

__STATIC_FORCEINLINE void __WFE(void)
Wait For Event.

Wait For Event is executed using CSR_WFE.WFE=1 and WFI instruction. It will suspends execution until event, NMI or Debug happened. When Core is waked up, Core will resume previous execution

__STATIC_FORCEINLINE void __EBREAK(void)
Breakpoint Instruction.

Causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

__STATIC_FORCEINLINE void __ECALL(void)
Environment Call Instruction.

The ECALL instruction is used to make a service request to the execution environment.

__STATIC_FORCEINLINE void __set_wfi_sleepmode(WFI_SleepMode_Type mode)
Set Sleep mode of WFI.

Set the SLEEPVALUE CSR register to control the WFI Sleep mode.

Parameters

- `[in] mode`: The sleep mode to be set

__STATIC_FORCEINLINE void __TXEVT(void)
Send TX Event.

Set the CSR TXEVT to control send a TX Event. The Core will output signal tx_evt as output event signal.

__STATIC_FORCEINLINE void __enable_mcycle_counter(void)
Enable MCYCLE counter.

Clear the CY bit of MCOUNTINHIBIT to 0 to enable MCYCLE Counter

__STATIC_FORCEINLINE void __disable_mcycle_counter(void)
Disable MCYCLE counter.

Set the CY bit of MCOUNTINHIBIT to 1 to disable MCYCLE Counter

__STATIC_FORCEINLINE void __enable_minstret_counter(void)
Enable MINSTRET counter.

Clear the IR bit of MCOUNTINHIBIT to 0 to enable MINSTRET Counter

__STATIC_FORCEINLINE void __disable_minstret_counter(void)
Disable MINSTRET counter.

Set the IR bit of MCOUNTINHIBIT to 1 to disable MINSTRET Counter

__STATIC_FORCEINLINE void __enable_all_counter(void)
Enable MCYCLE & MINSTRET counter.

Clear the IR and CY bit of MCOUNTINHIBIT to 1 to enable MINSTRET & MCYCLE Counter

__STATIC_FORCEINLINE void __disable_all_counter(void)
Disable MCYCLE & MINSTRET counter.

Set the IR and CY bit of MCOUNTINHIBIT to 1 to disable MINSTRET & MCYCLE Counter

__STATIC_FORCEINLINE void __FENCE_I(void)
Fence.i Instruction.

The FENCE.I instruction is used to synchronize the instruction and data streams.

__STATIC_FORCEINLINE uint8_t __LB(volatile void * addr)
Load 8bit value from address (8 bit)

Load 8 bit value.

Return value of type uint8_t at (*addr)

Parameters

- `[in] addr`: Address pointer to data

__STATIC_FORCEINLINE uint16_t __LH(volatile void * addr)
Load 16bit value from address (16 bit)

Load 16 bit value.

Return value of type uint16_t at (*addr)

Parameters

- `[in] addr`: Address pointer to data

__STATIC_FORCEINLINE uint32_t __LW(volatile void * addr)
Load 32bit value from address (32 bit)

Load 32 bit value.

Return value of type uint32_t at (*addr)

Parameters

- `[in] addr`: Address pointer to data

__STATIC_FORCEINLINE void __SB(volatile void * addr, uint8_t val)

Write 8bit value to address (8 bit)

Write 8 bit value.

Parameters

- `[in] addr`: Address pointer to data
- `[in] val`: Value to set

__STATIC_FORCEINLINE void __SH(volatile void * addr, uint16_t val)

Write 16bit value to address (16 bit)

Write 16 bit value.

Parameters

- `[in] addr`: Address pointer to data
- `[in] val`: Value to set

__STATIC_FORCEINLINE void __SW(volatile void * addr, uint32_t val)

Write 32bit value to address (32 bit)

Write 32 bit value.

Parameters

- `[in] addr`: Address pointer to data
- `[in] val`: Value to set

__STATIC_FORCEINLINE uint32_t __CAS_W(volatile uint32_t * addr, uint32_t oldval, uint3

Compare and Swap 32bit value using LR and SC.

Compare old value with memory, if identical, store new value in memory. Return the initial value in memory. Success is indicated by comparing return value with OLD. memory address, return 0 if successful, otherwise return !0

Return return the initial value in memory

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] oldval`: Old value of the data in address
- `[in] newval`: New value to be stored into the address

__STATIC_FORCEINLINE uint32_t __AMOSWAP_W(volatile uint32_t * addr, uint32_t newval)

Atomic Swap 32bit value into memory.

Atomically swap new 32bit value into memory using amoswap.d.

Return return the original value in memory

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] newval`: New value to be stored into the address

__STATIC_FORCEINLINE int32_t __AMOADD_W(volatile int32_t * addr, int32_t value)

Atomic Add with 32bit value.

Atomically ADD 32bit value with value in memory using amoadd.d.

Return return memory value + add value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be ADDed

__STATIC_FORCEINLINE int32_t __AMOAND_W(volatile int32_t * addr, int32_t value)

Atomic And with 32bit value.

Atomically AND 32bit value with value in memory using amoand.d.

Return return memory value & and value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be ANDed

__STATIC_FORCEINLINE int32_t __AMOOR_W(volatile int32_t * addr, int32_t value)

Atomic OR with 32bit value.

Atomically OR 32bit value with value in memory using amoor.d.

Return return memory value | and value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be ORed

__STATIC_FORCEINLINE int32_t __AMOXOR_W(volatile int32_t * addr, int32_t value)

Atomic XOR with 32bit value.

Atomically XOR 32bit value with value in memory using amoxor.d.

Return return memory value ^ and value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be XORed

__STATIC_FORCEINLINE uint32_t __AMOMAXU_W(volatile uint32_t * addr, uint32_t value)

Atomic unsigned MAX with 32bit value.

Atomically unsigned max compare 32bit value with value in memory using amomaxu.d.

Return return the bigger value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be compared

__STATIC_FORCEINLINE int32_t __AMOMAX_W(volatile int32_t * addr, int32_t value)

Atomic signed MAX with 32bit value.

Atomically signed max compare 32bit value with value in memory using amomax.d.

Return the bigger value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be compared

__STATIC_FORCEINLINE uint32_t __AMOMINU_W(volatile uint32_t * addr, uint32_t value)

Atomic unsigned MIN with 32bit value.

Atomically unsigned min compare 32bit value with value in memory using amominu.d.

Return the smaller value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be compared

__STATIC_FORCEINLINE int32_t __AMOMIN_W(volatile int32_t * addr, int32_t value)

Atomic signed MIN with 32bit value.

Atomically signed min compare 32bit value with value in memory using amomin.d.

Return the smaller value

Parameters

- `[in] addr`: Address pointer to data, address need to be 4byte aligned
- `[in] value`: value to be compared

2.5.8 Interrupts and Exceptions

enum NMSIS_Core_IntExc::**IRQn_Type**

Values:

Reserved0_IRQn = 0

Reserved1_IRQn = 1

Reserved2_IRQn = 2

SysTimerSW_IRQn = 3

Reserved3_IRQn = 4

Reserved4_IRQn = 5

Reserved5_IRQn = 6

SysTimer_IRQn = 7

Reserved6_IRQn = 8

Reserved7_IRQn = 9

Reserved8_IRQn = 10

Reserved9_IRQn = 11

Reserved10_IRQn = 12

Reserved11_IRQn = 13

Reserved12_IRQn = 14

Reserved13_IRQn = 15

Reserved14_IRQn = 16

Reserved15_IRQn = 17

Reserved16_IRQn = 18

FirstDeviceSpecificInterrupt_IRQn = 19

SOC_INT_MAX

__STATIC_FORCEINLINE void __ECLIC_SetCfgNlbits(uint32_t nlbits)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetCfgNlbits(void)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoVer(void)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoCtlbits(void)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoNum(void)

__STATIC_FORCEINLINE void __ECLIC_SetMth(uint8_t mth)

__STATIC_FORCEINLINE uint8_t __ECLIC_GetMth(void)

__STATIC_FORCEINLINE void __ECLIC_EnableIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetEnableIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_DisableIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE int32_t __ECLIC_GetPendingIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetPendingIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_ClearPendingIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetTrigIRQ(IRQn_Type IRQn, uint32_t trig)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetTrigIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetShvIRQ(IRQn_Type IRQn, uint32_t shv)

__STATIC_FORCEINLINE uint32_t __ECLIC_GetShvIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetCtrlIRQ(IRQn_Type IRQn, uint8_t intctrl)

__STATIC_FORCEINLINE uint8_t __ECLIC_GetCtrlIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetLevelIRQ(IRQn_Type IRQn, uint8_t lvl_abs)

__STATIC_FORCEINLINE uint8_t __ECLIC_GetLevelIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetPriorityIRQ(IRQn_Type IRQn, uint8_t pri)

__STATIC_FORCEINLINE uint8_t __ECLIC_GetPriorityIRQ(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __ECLIC_SetVector(IRQn_Type IRQn, rv_csr_t vector)

__STATIC_FORCEINLINE rv_csr_t __ECLIC_GetVector(IRQn_Type IRQn)

__STATIC_FORCEINLINE void __set_exc_entry(rv_csr_t addr)

__STATIC_FORCEINLINE rv_csr_t __get_exc_entry(void)

__STATIC_FORCEINLINE void __set_nonvec_entry(rv_csr_t addr)

__STATIC_FORCEINLINE rv_csr_t __get_nonvec_entry(void)

__STATIC_FORCEINLINE rv_csr_t __get_nmi_entry(void)

ECLIC_SetCfgNlbits __ECLIC_SetCfgNlbits

ECLIC_GetCfgNlbits __ECLIC_GetCfgNlbits

ECLIC_GetInfoVer __ECLIC_GetInfoVer

ECLIC_GetInfoCtlbits __ECLIC_GetInfoCtlbits

ECLIC_GetInfoNum __ECLIC_GetInfoNum

ECLIC_SetMth __ECLIC_SetMth

ECLIC_GetMth __ECLIC_GetMth

ECLIC_EnableIRQ __ECLIC_EnableIRQ

ECLIC_GetEnableIRQ __ECLIC_GetEnableIRQ

ECLIC_DisableIRQ __ECLIC_DisableIRQ

ECLIC_SetPendingIRQ __ECLIC_SetPendingIRQ

ECLIC_GetPendingIRQ __ECLIC_GetPendingIRQ

ECLIC_ClearPendingIRQ __ECLIC_ClearPendingIRQ

ECLIC_SetTrigIRQ __ECLIC_SetTrigIRQ

ECLIC_GetTrigIRQ __ECLIC_GetTrigIRQ

ECLIC_SetShvIRQ __ECLIC_SetShvIRQ

ECLIC_GetShvIRQ __ECLIC_GetShvIRQ

ECLIC_SetCtrlIRQ __ECLIC_SetCtrlIRQ

ECLIC_GetCtrlIRQ __ECLIC_GetCtrlIRQ

ECLIC_SetLevelIRQ __ECLIC_SetLevelIRQ

ECLIC_GetLevelIRQ __ECLIC_GetLevelIRQ

ECLIC_SetPriorityIRQ __ECLIC_SetPriorityIRQ

ECLIC_GetPriorityIRQ __ECLIC_GetPriorityIRQ

ECLIC_SetVector __ECLIC_SetVector

ECLIC_GetVector __ECLIC_GetVector

SAVE_IRQ_CSR_CONTEXT()

RESTORE_IRQ_CSR_CONTEXT()

group **NMSIS_Core_IntExc**
Functions that manage interrupts and exceptions via the ECLIC.

Defines

ECLIC_SetCfgNlbits __ECLIC_SetCfgNlbits

ECLIC_GetCfgNlbits __ECLIC_GetCfgNlbits

ECLIC_GetInfoVer __ECLIC_GetInfoVer

ECLIC_GetInfoCtlbits __ECLIC_GetInfoCtlbits

ECLIC_GetInfoNum __ECLIC_GetInfoNum

ECLIC_SetMth __ECLIC_SetMth

ECLIC_GetMth __ECLIC_GetMth

ECLIC_EnableIRQ __ECLIC_EnableIRQ

ECLIC_GetEnableIRQ __ECLIC_GetEnableIRQ

ECLIC_DisableIRQ __ECLIC_DisableIRQ

ECLIC_SetPendingIRQ __ECLIC_SetPendingIRQ

ECLIC_GetPendingIRQ __ECLIC_GetPendingIRQ

ECLIC_ClearPendingIRQ __ECLIC_ClearPendingIRQ

ECLIC_SetTrigIRQ __ECLIC_SetTrigIRQ

ECLIC_GetTrigIRQ __ECLIC_GetTrigIRQ

ECLIC_SetShvIRQ __ECLIC_SetShvIRQ

ECLIC_GetShvIRQ __ECLIC_GetShvIRQ

ECLIC_SetCtrlIRQ __ECLIC_SetCtrlIRQ

ECLIC_GetCtrlIRQ __ECLIC_GetCtrlIRQ

ECLIC_SetLevelIRQ __ECLIC_SetLevelIRQ

ECLIC_GetLevelIRQ __ECLIC_GetLevelIRQ

ECLIC_SetPriorityIRQ __ECLIC_SetPriorityIRQ

ECLIC_GetPriorityIRQ __ECLIC_GetPriorityIRQ

ECLIC_SetVector __ECLIC_SetVector

ECLIC_GetVector __ECLIC_GetVector

SAVE_IRQ_CSR_CONTEXT()

Save necessary CSRs into variables for vector interrupt nesting.

This macro is used to declare variables which are used for saving CSRs(MCAUSE, MEPC, MSUB), and it will read these CSR content into these variables, it need to be used in a vector-interrupt if nesting is required.

Remark

- Interrupt will be enabled after this macro is called
- It need to be used together with RESTORE_IRQ_CSR_CONTEXT
- Don’t use variable names __mcause, __mpec, __msubm in your ISR code
- If you want to enable interrupt nesting feature for vector interrupt, you can do it like this:

```
// __INTERRUPT attribute will generates function entry and exit␣
↪sequences suitable
// for use in an interrupt handler when this attribute is present
__INTERRUPT void eclic_mtip_handler(void)
{
    // Must call this to save CSRs
    SAVE_IRQ_CSR_CONTEXT();
    // !!!Interrupt is enabled here!!!
    // !!!Higher priority interrupt could nest it!!!

    // put you own interrupt handling code here

    // Must call this to restore CSRs
    RESTORE_IRQ_CSR_CONTEXT();
}
```

RESTORE_IRQ_CSR_CONTEXT()

Restore necessary CSRs from variables for vector interrupt nesting.

This macro is used restore CSRs(MCAUSE, MEPC, MSUB) from pre-defined variables in SAVE_IRQ_CSR_CONTEXT macro.

Remark

- Interrupt will be disabled after this macro is called
- It need to be used together with SAVE_IRQ_CSR_CONTEXT

Enums

enum IRQn_Type

Definition of IRQn numbers.

The core interrupt enumeration names for IRQn values are defined in the file **<Device>.h**.

- Interrupt ID(IRQn) from 0 to 18 are reserved for core internal interrupts.
- Interrupt ID(IRQn) start from 19 represent device-specific external interrupts.
- The first device-specific interrupt has the IRQn value 19.

The table below describes the core interrupt names and their availability in various Nuclei Cores.

Values:

Reserved0_IRQn = 0
Internal reserved.

Reserved1_IRQn = 1
Internal reserved.

Reserved2_IRQn = 2
Internal reserved.

SysTimerSW_IRQn = 3
System Timer SW interrupt.

Reserved3_IRQn = 4
Internal reserved.

Reserved4_IRQn = 5
Internal reserved.

Reserved5_IRQn = 6
Internal reserved.

SysTimer_IRQn = 7
System Timer Interrupt.

Reserved6_IRQn = 8
Internal reserved.

Reserved7_IRQn = 9
Internal reserved.

Reserved8_IRQn = 10
Internal reserved.

Reserved9_IRQn = 11
Internal reserved.

Reserved10_IRQn = 12
Internal reserved.

Reserved11_IRQn = 13
Internal reserved.

Reserved12_IRQn = 14
Internal reserved.

Reserved13_IRQn = 15
Internal reserved.

Reserved14_IRQn = 16
Internal reserved.

Reserved15_IRQn = 17
Internal reserved.

Reserved16_IRQn = 18
Internal reserved.

FirstDeviceSpecificInterrupt_IRQn = 19
First Device Specific Interrupt.

SOC_INT_MAX
Number of total interrupts.

Functions

__STATIC_FORCEINLINE void __ECLIC_SetCfgNlbits(uint32_t nlbits)
Set nlbits value.

This function set the nlbits value of CLICCFG register.

Remark

- nlbits is used to set the width of level in the CLICINTCTL[i].

See

- ECLIC_GetCfgNlbits

Parameters

- `[in] nlbits`: nlbits value

__STATIC_FORCEINLINE uint32_t __ECLIC_GetCfgNlbits(void)
Get nlbits value.

This function get the nlbits value of CLICCFG register.

Return nlbits value of CLICCFG register

Remark

- nlbits is used to set the width of level in the CLICINTCTL[i].

See

- ECLIC_SetCfgNlbits

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoVer(void)
Get the ECLIC version number.

This function gets the hardware version information from CLICINFO register.

Return hardware version number in CLICINFO register.

Remark

- This function gets harware version information from CLICINFO register.
- Bit 20:17 for architecture version, bit 16:13 for implementation version.

See

- ECLIC_GetInfoNum

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoCtlbits(void)

Get CLICINTCTLBITS.

This function gets CLICINTCTLBITS from CLICINFO register.

Return CLICINTCTLBITS from CLICINFO register.

Remark

- In the CLICINTCTL[i] registers, with 2 <= CLICINTCTLBITS <= 8.
- The implemented bits are kept left-justified in the most-significant bits of each 8-bit CLICINTCTL[I] register, with the lower unimplemented bits treated as hardwired to 1.

See

- ECLIC_GetInfoNum

__STATIC_FORCEINLINE uint32_t __ECLIC_GetInfoNum(void)

Get number of maximum interrupt inputs supported.

This function gets number of maximum interrupt inputs supported from CLICINFO register.

Return number of maximum interrupt inputs supported from CLICINFO register.

Remark

- This function gets number of maximum interrupt inputs supported from CLICINFO register.
- The num_interrupt field specifies the actual number of maximum interrupt inputs supported in this implementation.

See

- ECLIC_GetInfoCtlbits

__STATIC_FORCEINLINE void __ECLIC_SetMth(uint8_t mth)

Set Machine Mode Interrupt Level Threshold.

This function sets machine mode interrupt level threshold.

See

- ECLIC_GetMth

Parameters

- `[in] mth`: Interrupt Level Threshold.

__STATIC_FORCEINLINE uint8_t __ECLIC_GetMth(void)

Get Machine Mode Interrupt Level Threshold.

This function gets machine mode interrupt level threshold.

Return Interrupt Level Threshold.

See

- ECLIC_SetMth

__STATIC_FORCEINLINE void __ECLIC_EnableIRQ(IRQn_Type IRQn)

Enable a specific interrupt.

This function enables the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_DisableIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE uint32_t __ECLIC_GetEnableIRQ(IRQn_Type IRQn)

Get a specific interrupt enable status.

This function returns the interrupt enable status for the specific interrupt *IRQn*.

Return

- 0 Interrupt is not enabled
- 1 Interrupt is pending

Remark

- IRQn must not be negative.

See

- ECLIC_EnableIRQ
- ECLIC_DisableIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __ECLIC_DisableIRQ(IRQn_Type IRQn)

Disable a specific interrupt.

This function disables the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_EnableIRQ

Parameters

- `[in] IRQn`: Number of the external interrupt to disable

__STATIC_FORCEINLINE int32_t __ECLIC_GetPendingIRQ(IRQn_Type IRQn)

Get the pending specific interrupt.

This function returns the pending status of the specific interrupt *IRQn*.

Return

- 0 Interrupt is not pending

- 1 Interrupt is pending

Remark

- IRQn must not be negative.

See

- ECLIC_SetPendingIRQ
- ECLIC_ClearPendingIRQ

Parameters

- `[in] IRQn`: Interrupt number

`__STATIC_FORCEINLINE void __ECLIC_SetPendingIRQ(IRQn_Type IRQn)`

Set a specific interrupt to pending.

This function sets the pending bit for the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_GetPendingIRQ
- ECLIC_ClearPendingIRQ

Parameters

- `[in] IRQn`: Interrupt number

`__STATIC_FORCEINLINE void __ECLIC_ClearPendingIRQ(IRQn_Type IRQn)`

Clear a specific interrupt from pending.

This function removes the pending state of the specific interrupt *IRQn*. *IRQn* cannot be a negative number.

Remark

- IRQn must not be negative.

See

- ECLIC_SetPendingIRQ
- ECLIC_GetPendingIRQ

Parameters

- `[in] IRQn`: Interrupt number

`__STATIC_FORCEINLINE void __ECLIC_SetTrigIRQ(IRQn_Type IRQn, uint32_t trig)`

Set trigger mode and polarity for a specific interrupt.

This function set trigger mode and polarity of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_GetTrigIRQ

Parameters

- `[in] IRQn`: Interrupt number

- `[in] trig`:
  - 00 level trigger, ECLIC_LEVEL_TRIGGER
  - 01 positive edge trigger, ECLIC_POSTIVE_EDGE_TRIGGER
  - 02 level trigger, ECLIC_LEVEL_TRIGGER
  - 03 negative edge trigger, ECLIC_NEGTIVE_EDGE_TRIGGER

__STATIC_FORCEINLINE uint32_t __ECLIC_GetTrigIRQ(IRQn_Type IRQn)

Get trigger mode and polarity for a specific interrupt.

This function get trigger mode and polarity of the specific interrupt *IRQn*.

Return

- 00 level trigger, ECLIC_LEVEL_TRIGGER
- 01 positive edge trigger, ECLIC_POSTIVE_EDGE_TRIGGER
- 02 level trigger, ECLIC_LEVEL_TRIGGER
- 03 negative edge trigger, ECLIC_NEGTIVE_EDGE_TRIGGER

Remark

- IRQn must not be negative.

See

- ECLIC_SetTrigIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __ECLIC_SetShvIRQ(IRQn_Type IRQn, uint32_t shv)

Set interrupt working mode for a specific interrupt.

This function set selective hardware vector or non-vector working mode of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_GetShvIRQ

Parameters

- `[in] IRQn`: Interrupt number
- `[in] shv`:
  - 0 non-vector mode, ECLIC_NON_VECTOR_INTERRUPT
  - 1 vector mode, ECLIC_VECTOR_INTERRUPT

__STATIC_FORCEINLINE uint32_t __ECLIC_GetShvIRQ(IRQn_Type IRQn)

Get interrupt working mode for a specific interrupt.

This function get selective hardware vector or non-vector working mode of the specific interrupt *IRQn*.

Return shv

- 0 non-vector mode, ECLIC_NON_VECTOR_INTERRUPT
- 1 vector mode, ECLIC_VECTOR_INTERRUPT

Remark

- IRQn must not be negative.

See

- ECLIC_SetShvIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __ECLIC_SetCtrlIRQ(IRQn_Type IRQn, uint8_t intctrl)

Modify ECLIC Interrupt Input Control Register for a specific interrupt.

This function modify ECLIC Interrupt Input Control(CLICINTCTL[i]) register of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.

See

- ECLIC_GetCtrlIRQ

Parameters

- `[in] IRQn`: Interrupt number
- `[in] intctrl`: Set value for CLICINTCTL[i] register

__STATIC_FORCEINLINE uint8_t __ECLIC_GetCtrlIRQ(IRQn_Type IRQn)

Get ECLIC Interrupt Input Control Register value for a specific interrupt.

This function modify ECLIC Interrupt Input Control register of the specific interrupt *IRQn*.

Return value of ECLIC Interrupt Input Control register

Remark

- IRQn must not be negative.

See

- ECLIC_SetCtrlIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __ECLIC_SetLevelIRQ(IRQn_Type IRQn, uint8_t lvl_abs)

Set ECLIC Interrupt level of a specific interrupt.

This function set interrupt level of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.
- If lvl_abs to be set is larger than the max level allowed, it will be force to be max level.
- When you set level value you need use clciinfo.nlbits to get the width of level. Then we could know the maximum of level. CLICINTCTLBITS is how many total bits are present in the CLICINTCTL register.

See

- ECLIC_GetLevelIRQ

Parameters

- `[in] IRQn`: Interrupt number
- `[in] lvl_abs`: Interrupt level

`__STATIC_FORCEINLINE uint8_t __ECLIC_GetLevelIRQ(IRQn_Type IRQn)`

Get ECLIC Interrupt level of a specific interrupt.

This function get interrupt level of the specific interrupt *IRQn*.

Return Interrupt level

Remark

- IRQn must not be negative.

See

- ECLIC_SetLevelIRQ

Parameters

- `[in] IRQn`: Interrupt number

`__STATIC_FORCEINLINE void __ECLIC_SetPriorityIRQ(IRQn_Type IRQn, uint8_t pri)`

Get ECLIC Interrupt priority of a specific interrupt.

This function get interrupt priority of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.
- If pri to be set is larger than the max priority allowed, it will be force to be max priority.
- Priority width is CLICINTCTLBITS minus clciinfo.nlbits if clciinfo.nlbits is less than CLICINTCTLBITS. Otherwise priority width is 0.

See

- ECLIC_GetPriorityIRQ

Parameters

- `[in] IRQn`: Interrupt number
- `[in] pri`: Interrupt priority

`__STATIC_FORCEINLINE uint8_t __ECLIC_GetPriorityIRQ(IRQn_Type IRQn)`

Get ECLIC Interrupt priority of a specific interrupt.

This function get interrupt priority of the specific interrupt *IRQn*.

Return Interrupt priority

Remark

- IRQn must not be negative.

See

- ECLIC_SetPriorityIRQ

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __ECLIC_SetVector(IRQn_Type IRQn, rv_csr_t vector)

Set Interrupt Vector of a specific interrupt.

This function set interrupt handler address of the specific interrupt *IRQn*.

Remark

- IRQn must not be negative.
- You can set the CSR_CSR_MTVT to set interrupt vector table entry address.
- If your vector table is placed in readonly section, the vector for IRQn will not be modified. For this case, you need to use the correct irq handler name defined in your vector table as your irq handler function name.
- This function will only work correctly when the vector table is placed in an read-write enabled section.

See

- ECLIC_GetVector

Parameters

- `[in] IRQn`: Interrupt number
- `[in] vector`: Interrupt handler address

__STATIC_FORCEINLINE rv_csr_t __ECLIC_GetVector(IRQn_Type IRQn)

Get Interrupt Vector of a specific interrupt.

This function get interrupt handler address of the specific interrupt *IRQn*.

Return Interrupt handler address

Remark

- IRQn must not be negative.
- You can read CSR_CSR_MTVT to get interrupt vector table entry address.

See

- ECLIC_SetVector

Parameters

- `[in] IRQn`: Interrupt number

__STATIC_FORCEINLINE void __set_exc_entry(rv_csr_t addr)

Set Exception entry address.

This function set exception handler address to 'CSR_MTVEC'.

Remark

- This function use to set exception handler address to 'CSR_MTVEC'. Address is 4 bytes align.

See

- __get_exc_entry

Parameters

- `[in] addr`: Exception handler address

__STATIC_FORCEINLINE rv_csr_t __get_exc_entry(void)

Get Exception entry address.

This function get exception handler address from 'CSR_MTVEC'.

Return Exception handler address

Remark

- This function use to get exception handler address from 'CSR_MTVEC'. Address is 4 bytes align

See

- __set_exc_entry

__STATIC_FORCEINLINE void __set_nonvec_entry(rv_csr_t addr)

Set Non-vector interrupt entry address.

This function set Non-vector interrupt address.

Remark

- This function use to set non-vector interrupt entry address to 'CSR_MTVT2' if
- CSR_MTVT2 bit0 is 1. If 'CSR_MTVT2' bit0 is 0 then set address to 'CSR_MTVEC'

See

- __get_nonvec_entry

Parameters

- `[in] addr`: Non-vector interrupt entry address

__STATIC_FORCEINLINE rv_csr_t __get_nonvec_entry(void)

Get Non-vector interrupt entry address.

This function get Non-vector interrupt address.

Return Non-vector interrupt handler address

Remark

- This function use to get non-vector interrupt entry address from 'CSR_MTVT2' if
- CSR_MTVT2 bit0 is 1. If 'CSR_MTVT2' bit0 is 0 then get address from 'CSR_MTVEC'.

See

- __set_nonvec_entry

__STATIC_FORCEINLINE rv_csr_t __get_nmi_entry(void)

Get NMI interrupt entry from 'CSR_MNVEC'.

This function get NMI interrupt address from 'CSR_MNVEC'.

Return NMI interrupt handler address

Remark

- This function use to get NMI interrupt handler address from 'CSR_MNVEC'. If CSR_MMISC_CTL[9] = 1 'CSR_MNVEC'
- will be equal as mtvec. If CSR_MMISC_CTL[9] = 0 'CSR_MNVEC' will be equal as reset vector.
- NMI entry is defined via CSR_MMISC_CTL, writing to CSR_MNVEC will be ignored.

2.5.9 SysTimer Functions

__STATIC_FORCEINLINE void SysTimer_SetLoadValue(uint64_t value)

__STATIC_FORCEINLINE uint64_t SysTimer_GetLoadValue(void)

__STATIC_FORCEINLINE void SysTimer_SetCompareValue(uint64_t value)

__STATIC_FORCEINLINE uint64_t SysTimer_GetCompareValue(void)

__STATIC_FORCEINLINE void SysTimer_Start(void)

__STATIC_FORCEINLINE void SysTimer_Stop(void)

__STATIC_FORCEINLINE void SysTimer_SetControlValue(uint32_t mctl)

__STATIC_FORCEINLINE uint32_t SysTimer_GetControlValue(void)

__STATIC_FORCEINLINE void SysTimer_SetSWIRQ(void)

__STATIC_FORCEINLINE void SysTimer_ClearSWIRQ(void)

__STATIC_FORCEINLINE uint32_t SysTimer_GetMsipValue(void)

__STATIC_FORCEINLINE void SysTimer_SetMsipValue(uint32_t msip)

__STATIC_FORCEINLINE void SysTimer_SoftwareReset(void)

__STATIC_INLINE uint32_t SysTick_Config(uint64_t ticks)

__STATIC_FORCEINLINE uint32_t SysTick_Reload(uint64_t ticks)

group **NMSIS_Core_SysTimer**

Functions that configure the Core System Timer.

Functions

__STATIC_FORCEINLINE void SysTimer_SetLoadValue(uint64_t value)

Set system timer load value.

This function set the system timer load value in MTIMER register.

Remark

- Load value is 64bits wide.
- SysTimer_GetLoadValue

Parameters

- `[in] value`: value to set system timer MTIMER register.

__STATIC_FORCEINLINE uint64_t SysTimer_GetLoadValue(void)

Get system timer load value.

This function get the system timer current value in MTIMER register.

Return current value(64bit) of system timer MTIMER register.

Remark

- Load value is 64bits wide.
- SysTimer_SetLoadValue

__STATIC_FORCEINLINE void SysTimer_SetCompareValue(uint64_t value)

Set system timer compare value.

This function set the system Timer compare value in MTIMERCMP register.

Remark

- Compare value is 64bits wide.
- If compare value is larger than current value timer interrupt generate.
- Modify the load value or compare value less to clear the interrupt.
- SysTimer_GetCompareValue

Parameters

- `[in] value`: compare value to set system timer MTIMERCMP register.

__STATIC_FORCEINLINE uint64_t SysTimer_GetCompareValue(void)

Get system timer compare value.

This function get the system timer compare value in MTIMERCMP register.

Return compare value of system timer MTIMERCMP register.

Remark

- Compare value is 64bits wide.
- SysTimer_SetCompareValue

__STATIC_FORCEINLINE void SysTimer_Start(void)

Enable system timer counter running.

Enable system timer counter running by clear TIMESTOP bit in MTIMECTL register.

__STATIC_FORCEINLINE void SysTimer_Stop(void)

Stop system timer counter running.

Stop system timer counter running by set TIMESTOP bit in MTIMECTL register.

__STATIC_FORCEINLINE void SysTimer_SetControlValue(uint32_t mctl)

Set system timer control value.

This function set the system timer MTIMECTL register value.

Remark

- Bit TIMESTOP is used to start and stop timer. Clear TIMESTOP bit to 0 to start timer, otherwise to stop timer.
- Bit CMPCLREN is used to enable auto MTIMER clear to zero when MTIMER >= MTIMERCMP. Clear CMPCLREN bit to 0 to stop auto clear MTIMER feature, otherwise to enable it.
- Bit CLKSRC is used to select timer clock source. Clear CLKSRC bit to 0 to use *mtime_toggle_a*, otherwise use *core_clk_aon*
- SysTimer_GetControlValue

Parameters

- `[in] mctl`: value to set MTIMECTL register

__STATIC_FORCEINLINE uint32_t SysTimer_GetControlValue(void)

Get system timer control value.

This function get the system timer MTIMECTL register value.

Return MTIMECTL register value

Remark

- SysTimer_SetControlValue

__STATIC_FORCEINLINE void SysTimer_SetSWIRQ(void)

Trigger or set software interrupt via system timer.

This function set the system timer MSIP bit in MSIP register.

Remark

- Set system timer MSIP bit and generate a SW interrupt.
- SysTimer_ClearSWIRQ
- SysTimer_GetMsipValue

__STATIC_FORCEINLINE void SysTimer_ClearSWIRQ(void)

Clear system timer software interrupt pending request.

This function clear the system timer MSIP bit in MSIP register.

Remark

- Clear system timer MSIP bit in MSIP register to clear the software interrupt pending.
- SysTimer_SetSWIRQ
- SysTimer_GetMsipValue

__STATIC_FORCEINLINE uint32_t SysTimer_GetMsipValue(void)

Get system timer MSIP register value.

This function get the system timer MSIP register value.

Return Value of Timer MSIP register.

Remark

- Bit0 is SW interrupt flag. Bit0 is 1 then SW interrupt set. Bit0 is 0 then SW interrupt clear.
- SysTimer_SetSWIRQ
- SysTimer_ClearSWIRQ

__STATIC_FORCEINLINE void SysTimer_SetMsipValue(uint32_t msip)

Set system timer MSIP register value.

This function set the system timer MSIP register value.

Parameters

- `[in] msip`: value to set MSIP register

__STATIC_FORCEINLINE void SysTimer_SoftwareReset(void)

Do software reset request.

This function will do software reset request through MTIMER

- Software need to write SysTimer_MSFRST_KEY to generate software reset request
- The software request flag can be cleared by reset operation to clear

  Remark

- The software reset is sent to SoC, SoC need to generate reset signal and send back to Core
- This function will not return, it will do while(1) to wait the Core reset happened

__STATIC_INLINE uint32_t SysTick_Config(uint64_t ticks)

System Tick Configuration.

Initializes the System Timer and its non-vector interrupt, and starts the System Tick Timer.

In our default implementation, the timer counter will be set to zero, and it will start a timer compare non-vector interrupt when it matchs the ticks user set, during the timer interrupt user should reload the system tick using SysTick_Reload function or similar function written by user, so it can produce period timer interrupt.

Return 0 Function succeeded.

Return 1 Function failed.

Remark

- For __NUCLEI_N_REV >= 0x0104, the CMPCLREN bit in MTIMECTL is introduced, but we assume that the CMPCLREN bit is set to 0, so MTIMER register will not be auto cleared to 0 when MTIMER >= MTIMERCMP.
- When the variable __Vendor_SysTickConfig is set to 1, then the function SysTick_Config is not included.
- In this case, the file **<Device>.h** must contain a vendor-specific implementation of this function.
- If user need this function to start a period timer interrupt, then in timer interrupt handler routine code, user should call SysTick_Reload with ticks to reload the timer.
- This function only available when __SYSTIMER_PRESENT == 1 and __ECLIC_PRESENT == 1 and __Vendor_SysTickConfig == 0

See

- SysTimer_SetCompareValue; SysTimer_SetLoadValue

Parameters

- `[in] ticks`: Number of ticks between two interrupts.

__STATIC_FORCEINLINE uint32_t SysTick_Reload(uint64_t ticks)

System Tick Reload.

Reload the System Timer Tick when the MTIMECMP reached TIME value

Return 0 Function succeeded.

Return 1 Function failed.

Remark

- For __NUCLEI_N_REV >= 0x0104, the CMPCLREN bit in MTIMECTL is introduced, but for this SysTick_Config function, we assume this CMPCLREN bit is set to 0, so in interrupt handler function, user still need to set the MTIMERCMP or MTIMER to reload the system tick, if vendor want to use this timer's auto clear feature, they can define __Vendor_SysTickConfig to 1, and implement SysTick_Config and SysTick_Reload functions.
- When the variable __Vendor_SysTickConfig is set to 1, then the function SysTick_Reload is not included.
- In this case, the file **<Device>.h** must contain a vendor-specific implementation of this function.
- This function only available when __SYSTIMER_PRESENT == 1 and __ECLIC_PRESENT == 1 and __Vendor_SysTickConfig == 0

- Since the MTIMERCMP value might overflow, if overflowed, MTIMER will be set to 0, and MTIMERCMP set to ticks

See

- SysTimer_SetCompareValue
- SysTimer_SetLoadValue

Parameters

- `[in] ticks`: Number of ticks between two interrupts.

2.5.10 FPU Functions

typedef uint64_t **rv_fpu_t**

__RISCV_FLEN 64

__get_FCSR() __RV_CSR_READ(CSR_FCSR)

__set_FCSR(val) __RV_CSR_WRITE(CSR_FCSR, (val))

__get_FRM() __RV_CSR_READ(CSR_FRM)

__set_FRM(val) __RV_CSR_WRITE(CSR_FRM, (val))

__get_FFLAGS() __RV_CSR_READ(CSR_FFLAGS)

__set_FFLAGS(val) __RV_CSR_WRITE(CSR_FFLAGS, (val))

__enable_FPU() __RV_CSR_SET(CSR_MSTATUS, MSTATUS_FS)

__disable_FPU() __RV_CSR_CLEAR(CSR_MSTATUS, MSTATUS_FS)

__RV_FLW(freg, addr, ofs)

__RV_FSW(freg, addr, ofs)

__RV_FLD(freg, addr, ofs)

__RV_FSD(freg, addr, ofs)

__RV_FLOAD __RV_FLD

__RV_FSTORE __RV_FSD

SAVE_FPU_CONTEXT()

RESTORE_FPU_CONTEXT()

group **NMSIS_Core_FPU_Functions**

Functions that related to the RISC-V FPU (F and D extension).

Nuclei provided floating point unit by RISC-V F and D extension.

- `F extension` adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard, __RISCV_FLEN = 32. The F extension adds 32 floating-point registers, f0-f31, each 32 bits wide, and a floating-point control and status register fcsr, which contains the operating mode and exception status of the floating-point unit.
- `D extension` adds double-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The D extension widens the 32 floating-point registers, f0-f31, to 64 bits, __RISCV_FLEN = 64

Defines

__RISCV_FLEN 64

__get_FCSR() __RV_CSR_READ(CSR_FCSR)
Get FCSR CSR Register.

__set_FCSR(val) __RV_CSR_WRITE(CSR_FCSR, (val))
Set FCSR CSR Register with val.

__get_FRM() __RV_CSR_READ(CSR_FRM)
Get FRM CSR Register.

__set_FRM(val) __RV_CSR_WRITE(CSR_FRM, (val))
Set FRM CSR Register with val.

__get_FFLAGS() __RV_CSR_READ(CSR_FFLAGS)
Get FFLAGS CSR Register.

__set_FFLAGS(val) __RV_CSR_WRITE(CSR_FFLAGS, (val))
Set FFLAGS CSR Register with val.

__enable_FPU() __RV_CSR_SET(CSR_MSTATUS, MSTATUS_FS)
Enable FPU Unit.

__disable_FPU() __RV_CSR_CLEAR(CSR_MSTATUS, MSTATUS_FS)
Disable FPU Unit.

- We can save power by disable FPU Unit.
- When FPU Unit is disabled, any access to FPU related CSR registers and FPU instructions will cause illegal Instuction Exception.

__RV_FLW(freg, addr, ofs)
Load a single-precision value from memory into float point register freg using flw instruction.

The FLW instruction loads a single-precision floating point value from memory address (addr + ofs) into floating point register freg(f0-f31)

Remark

- FLW and FSW operations need to make sure the address is 4 bytes aligned, otherwise it will cause exception code 4(Load address misaligned) or 6 (Store/AMO address misaligned)
- FLW and FSW do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved

Parameters

- `[in] freg`: The floating point register, eg. FREG(0), f0
- `[in] addr`: The memory base address, 4 byte aligned required
- `[in] ofs`: a 12-bit immediate signed byte offset value, should be an const value

__RV_FSW(freg, addr, ofs)
Store a single-precision value from float point freg into memory using fsw instruction.

The FSW instruction stores a single-precision value from floating point register to memory

Remark

- FLW and FSW operations need to make sure the address is 4 bytes aligned, otherwise it will cause exception code 4(Load address misaligned) or 6 (Store/AMO address misaligned)

- FLW and FSW do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved

Parameters

- `[in] freg`: The floating point register(f0-f31), eg. FREG(0), f0
- `[in] addr`: The memory base address, 4 byte aligned required
- `[in] ofs`: a 12-bit immediate signed byte offset value, should be an const value

__RV_FLD (freg, addr, ofs)

Load a double-precision value from memory into float point register freg using fld instruction.

The FLD instruction loads a double-precision floating point value from memory address (addr + ofs) into floating point register freg(f0-f31)

Attention

- Function only available for double precision floating point unit, FLEN = 64

Remark

- FLD and FSD operations need to make sure the address is 8 bytes aligned, otherwise it will cause exception code 4(Load address misaligned) or 6 (Store/AMO address misaligned)
- FLD and FSD do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

Parameters

- `[in] freg`: The floating point register, eg. FREG(0), f0
- `[in] addr`: The memory base address, 8 byte aligned required
- `[in] ofs`: a 12-bit immediate signed byte offset value, should be an const value

__RV_FSD (freg, addr, ofs)

Store a double-precision value from float point freg into memory using fsd instruction.

The FSD instruction stores double-precision value from floating point register to memory

Attention

- Function only available for double precision floating point unit, FLEN = 64

Remark

- FLD and FSD operations need to make sure the address is 8 bytes aligned, otherwise it will cause exception code 4(Load address misaligned) or 6 (Store/AMO address misaligned)
- FLD and FSD do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

Parameters

- `[in] freg`: The floating point register(f0-f31), eg. FREG(0), f0
- `[in] addr`: The memory base address, 8 byte aligned required
- `[in] ofs`: a 12-bit immediate signed byte offset value, should be an const value

__RV_FLOAD __RV_FLD

Load a float point value from memory into float point register freg using flw/fld instruction.

- For Single-Precison Floating-Point Mode(__FPU_PRESENT == 1, __RISCV_FLEN == 32): It will call __RV_FLW to load a single-precision floating point value from memory to floating point register

- For Double-Precison Floating-Point Mode(__FPU_PRESENT == 2, __RISCV_FLEN == 64): It will call __RV_FLD to load a double-precision floating point value from memory to floating point register

Attention Function behaviour is different for __FPU_PRESENT = 1 or 2, please see the real function this macro represent

__RV_FSTORE __RV_FSD

Store a float value from float point freg into memory using fsw/fsd instruction.

- For Single-Precison Floating-Point Mode(__FPU_PRESENT == 1, __RISCV_FLEN == 32): It will call __RV_FSW to store floating point register into memory
- For Double-Precison Floating-Point Mode(__FPU_PRESENT == 2, __RISCV_FLEN == 64): It will call __RV_FSD to store floating point register into memory

Attention Function behaviour is different for __FPU_PRESENT = 1 or 2, please see the real function this macro represent

SAVE_FPU_CONTEXT()

Save FPU context into variables for interrupt nesting.

This macro is used to declare variables which are used for saving FPU context, and it will store the nessary fpu registers into these variables, it need to be used in a interrupt when in this interrupt fpu registers are used.

Remark

- It need to be used together with RESTORE_FPU_CONTEXT
- Don't use variable names __fpu_context in your ISR code
- If you isr code will use fpu registers, and this interrupt is nested. Then you can do it like this:

```
void eclic_mtip_handler(void)
{
    // !!!Interrupt is enabled here!!!
    // !!!Higher priority interrupt could nest it!!!

    // Necessary only when you need to use fpu registers
    // in this isr handler functions
    SAVE_FPU_CONTEXT();

    // put you own interrupt handling code here

    // pair of SAVE_FPU_CONTEXT()
    RESTORE_FPU_CONTEXT();
}
```

RESTORE_FPU_CONTEXT()

Restore necessary fpu registers from variables for interrupt nesting.

This macro is used restore necessary fpu registers from pre-defined variables in SAVE_FPU_CONTEXT macro.

Remark

- It need to be used together with SAVE_FPU_CONTEXT

Typedefs

typedef uint64_t **rv_fpu_t**

Type of FPU register, depends on the FLEN defined in RISC-V.

2.5.11 Intrinsic Functions for SIMD Instructions

SIMD Data Processing Instructions

SIMD 16-bit Add/Subtract Instructions

```
__STATIC_FORCEINLINE unsigned long __RV_ADD16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_CRAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_CRSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KADD16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KCRAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KCRSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KSTAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KSTSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_KSUB16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RADD16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RCRAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RCRSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RSTAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RSTSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_RSUB16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_STAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_STSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_SUB16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKADD16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKCRAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKCRSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKSTAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKSTSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_UKSUB16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_URADD16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_URCRAS16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_URCRSA16(unsigned long a, unsigned long b)
__STATIC_FORCEINLINE unsigned long __RV_URSTAS16(unsigned long a, unsigned long b)
```

__STATIC_FORCEINLINE unsigned long __RV_URSTSA16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URSUB16(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_16B_ADDSUB**

SIMD 16-bit Add/Subtract Instructions.

Based on the combination of the types of the two 16-bit arithmetic operations, the SIMD 16-bit add/subtract instructions can be classified into 6 main categories: Addition (two 16-bit addition), Subtraction (two 16-bit subtraction), Crossed Add & Sub (one addition and one subtraction), and Crossed Sub & Add (one subtraction and one addition), Straight Add & Sub (one addition and one subtraction), and Straight Sub & Add (one subtraction and one addition). Based on the way of how an overflow condition is handled, the SIMD 16-bit add/subtract instructions can be classified into 5 groups: Wrap-around (dropping overflow), Signed Halving (keeping overflow by dropping 1 LSB bit), Unsigned Halving, Signed Saturation (clipping overflow), and Unsigned Saturation. Together, there are 30 SIMD 16-bit add/subtract instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_ADD16(unsigned long a, unsigned long b)

ADD16 (SIMD 16-bit Addition)

Type: SIMD

Syntax:

```
ADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element additions simultaneously.

Description: This instruction adds the 16-bit integer elements in Rs1 with the 16-bit integer elements in Rs2, and then writes the 16-bit element results to Rd.

Note: This instruction can be used for either signed or unsigned addition.

Operations:

```
Rd.H[x] = Rs1.H[x] + Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_CRAS16(unsigned long a, unsigned long b)

CRAS16 (SIMD 16-bit Cross Addition & Subtraction)

Type: SIMD

Syntax:

```
CRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs2, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it subtracts the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit integer element in [15:0] of 32-bit chunks, and writes the result to [15:0] of 32- bit chunks in Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] + Rs2.W[x][15:0];
Rd.W[x][15:0] = Rs1.W[x][15:0] - Rs2.W[x][31:16];
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_CRSA16(unsigned long a, unsigned long b)`

CRSA16 (SIMD 16-bit Cross Subtraction & Addition)

Type: SIMD

Syntax:

```
CRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit integer element in [31:16] of 32-bit chunks in Rs1, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs1, and writes the result to [15:0] of 32-bit chunks in Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] - Rs2.W[x][15:0];
Rd.W[x][15:0] = Rs1.W[x][15:0] + Rs2.W[x][31:16];
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KADD16(unsigned long a, unsigned long b)`

KADD16 (SIMD 16-bit Signed Saturating Addition)

Type: SIMD

Syntax:

```
KADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > 32767) {
  res[x] = 32767;
  OV = 1;
} else if (res[x] < -32768) {
  res[x] = -32768;
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a
- [in] b: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KCRAS16(unsigned long a, unsigned long b)

KCRAS16 (SIMD 16-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD

Syntax:

```
KCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32- bit chunks.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] - Rs2.W[x][31:16];
for (res in [res1, res2]) {
  if (res > (2^15)-1) {
    res = (2^15)-1;
    OV = 1;
  } else if (res < -2^15) {
    res = -2^15;
```

(continues on next page)

(continued from previous page)

```
    OV = 1;
  }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a
- [in] b: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KCRSA16(unsigned long a, unsigned long b)

KCRSA16 (SIMD 16-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD

Syntax:

```
KCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] + Rs2.W[x][31:16];
for (res in [res1, res2]) {
  if (res > (2^15)-1) {
    res = (2^15)-1;
    OV = 1;
  } else if (res < -2^15) {
    res = -2^15;
    OV = 1;
  }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSTAS16(unsigned long a, unsigned long b)

KSTAS16 (SIMD 16-bit Signed Saturating Straight Addition & Subtraction)

Type: SIMD

Syntax:

```
KSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] - Rs2.W[x][15:0];
for (res in [res1, res2]) {
  if (res > (2^15)-1) {
    res = (2^15)-1;
    OV = 1;
  } else if (res < -2^15) {
    res = -2^15;
    OV = 1;
  }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSTSA16(unsigned long a, unsigned long b)

KSTSA16 (SIMD 16-bit Signed Saturating Straight Subtraction & Addition)

Type: SIMD

Syntax:

```
KSTSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] + Rs2.W[x][15:0];
for (res in [res1, res2]) {
  if (res > (2^15)-1) {
    res = (2^15)-1;
    OV = 1;
  } else if (res < -2^15) {
    res = -2^15;
    OV = 1;
  }
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSUB16(unsigned long a, unsigned long b)

KSUB16 (SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Syntax:

```
KSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] > (2^15)-1) {
  res[x] = (2^15)-1;
  OV = 1;
} else if (res[x] < -2^15) {
  res[x] = -2^15;
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_RADD16(unsigned long a, unsigned long b)`

RADD16 (SIMD 16-bit Signed Halving Addition)

Type: SIMD

Syntax:

```
RADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Rs1 = 0x7FFF, Rs2 = 0x7FFF, Rd = 0x7FFF
* Rs1 = 0x8000, Rs2 = 0x8000, Rd = 0x8000
* Rs1 = 0x4000, Rs2 = 0x8000, Rd = 0xE000
```

Operations:

```
Rd.H[x] = (Rs1.H[x] + Rs2.H[x]) s>> 1; for RV32: x=1...0, for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_RCRAS16(unsigned long a, unsigned long b)`

RCRAS16 (SIMD 16-bit Signed Halving Cross Addition & Subtraction)

Type: SIMD

Syntax:

```
RCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2, and subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `RADD16` and `RSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][15:0]) s>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][31:16]) s>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RCRSA16(unsigned long a, unsigned long b)
RCRSA16 (SIMD 16-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD

Syntax:

```
RCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit signed element integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `RADD16` and `RSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][15:0]) s>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][31:16]) s>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSTAS16(unsigned long a, unsigned long b)
RSTAS16 (SIMD 16-bit Signed Halving Straight Addition & Subtraction)

Type: SIMD

Syntax:

```
RSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2, and subtracts the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `RADD16` and `RSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][31:16]) s>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][15:0]) s>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSTSA16(unsigned long a, unsigned long b)

RSTSA16 (SIMD 16-bit Signed Halving Straight Subtraction & Addition)

Type: SIMD

Syntax:

```
RSTSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit signed integer element in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit signed element integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit signed integer element in [15:0] of 32-bit chunks in Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `RADD16` and `RSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][31:16]) s>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][15:0]) s>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSUB16(unsigned long a, unsigned long b)

RSUB16 (SIMD 16-bit Signed Halving Subtraction)

Type: SIMD

Syntax:

```
RSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFF, Rb = 0x8000, Rt = 0x7FFF
* Ra = 0x8000, Rb = 0x7FFF, Rt = 0x8000
* Ra = 0x8000, Rb = 0x4000, Rt = 0xA000
```

Operations:

```
Rd.H[x] = (Rs1.H[x] - Rs2.H[x]) s>> 1;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_STAS16(unsigned long a, unsigned long b)

STAS16 (SIMD 16-bit Straight Addition & Subtraction)

Type: SIMD

Syntax:

```
STAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit integer element in [31:16] of 32-bit chunks in Rs2, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it subtracts the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit integer element in [15:0] of 32-bit chunks, and writes the result to [15:0] of 32- bit chunks in Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] + Rs2.W[x][31:16];
Rd.W[x][15:0] = Rs1.W[x][15:0] - Rs2.W[x][15:0];
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_STSA16(unsigned long a, unsigned long b)`

STSA16 (SIMD 16-bit Straight Subtraction & Addition)

Type: SIMD

Syntax:

```
STSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit integer element in [31:16] of 32-bit chunks in Rs1, and writes the result to [31:16] of 32-bit chunks in Rd; at the same time, it adds the 16-bit integer element in [15:0] of 32-bit chunks in Rs2 with the 16-bit integer element in [15:0] of 32-bit chunks in Rs1, and writes the result to [15:0] of 32-bit chunks in Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[x][31:16] = Rs1.W[x][31:16] - Rs2.W[x][31:16];
Rd.W[x][15:0] = Rs1.W[x][15:0] + Rs2.W[x][15:0];
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SUB16(unsigned long a, unsigned long b)`

SUB16 (SIMD 16-bit Subtraction)

Type: SIMD

Syntax:

```
SUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 16-bit integer elements in Rs2 from the 16-bit integer elements in Rs1, and then writes the result to Rd.

Note: This instruction can be used for either signed or unsigned subtraction.

Operations:

```
Rd.H[x] = Rs1.H[x] - Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKADD16(unsigned long a, unsigned long b)

UKADD16 (SIMD 16-bit Unsigned Saturating Addition)

Type: SIMD

Syntax:

```
UKADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2. If any of the results are beyond the 16-bit unsigned number range (0 <= RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > (2^16)-1) {
  res[x] = (2^16)-1;
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKCRAS16(unsigned long a, unsigned long b)

UKCRAS16 (SIMD 16-bit Unsigned Saturating Cross Addition & Subtraction)

Type: SIMD

Syntax:

```
UKCRAS16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range (0 <= RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] - Rs2.W[x][31:16];
if (res1 > (2^16)-1) {
  res1 = (2^16)-1;
  OV = 1;
}
if (res2 < 0) {
  res2 = 0;
  OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKCRSA16(unsigned long a, unsigned long b)

UKCRSA16 (SIMD 16-bit Unsigned Saturating Cross Subtraction & Addition)

Type: SIMD

Syntax:

```
UKCRSA16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 with the 16- bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range (0 <= RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][15:0];
res2 = Rs1.W[x][15:0] + Rs2.W[x][31:16];
if (res1 < 0) {
  res1 = 0;
```

(continues on next page)

(continued from previous page)

```
  OV = 1;
} else if (res2 > (2^16)-1) {
  res2 = (2^16)-1;
  OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKSTAS16(unsigned long a, unsigned long b)`

UKSTAS16 (SIMD 16-bit Unsigned Saturating Straight Addition & Subtraction)

Type: SIMD

Syntax:

```
UKSTAS16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction adds the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2; at the same time, it subtracts the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range (0 <= RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for addition and [15:0] of 32-bit chunks in Rd for subtraction.

Operations:

```
res1 = Rs1.W[x][31:16] + Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] - Rs2.W[x][15:0];
if (res1 > (2^16)-1) {
  res1 = (2^16)-1;
  OV = 1;
}
if (res2 < 0) {
  res2 = 0;
  OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKSTSA16(unsigned long a, unsigned long b)
UKSTSA16 (SIMD 16-bit Unsigned Saturating Straight Subtraction & Addition)

Type: SIMD

Syntax:

```
UKSTSA16 Rd, Rs1, Rs2
```

Purpose: Do one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks.

Description: This instruction subtracts the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer element in [31:16] of 32-bit chunks in Rs1; at the same time, it adds the 16-bit unsigned integer element in [15:0] of 32-bit chunks in Rs2 with the 16- bit unsigned integer element in [15:0] of 32-bit chunks in Rs1. If any of the results are beyond the 16-bit unsigned number range (0 <= RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [31:16] of 32-bit chunks in Rd for subtraction and [15:0] of 32-bit chunks in Rd for addition.

Operations:

```
res1 = Rs1.W[x][31:16] - Rs2.W[x][31:16];
res2 = Rs1.W[x][15:0] + Rs2.W[x][15:0];
if (res1 < 0) {
  res1 = 0;
  OV = 1;
} else if (res2 > (2^16)-1) {
  res2 = (2^16)-1;
  OV = 1;
}
Rd.W[x][31:16] = res1;
Rd.W[x][15:0] = res2;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKSUB16(unsigned long a, unsigned long b)
UKSUB16 (SIMD 16-bit Unsigned Saturating Subtraction)

Type: SIMD

Syntax:

```
UKSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 16-bit unsigned integer elements in Rs2 from the 16-bit unsigned integer elements in Rs1. If any of the results are beyond the 16-bit unsigned number range (0 <=

RES <= 2^16-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] < 0) {
  res[x] = 0;
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_URADD16(unsigned long a, unsigned long b)`

URADD16 (SIMD 16-bit Unsigned Halving Addition)

Type: SIMD

Syntax:

```
URADD16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFF, Rb = 0x7FFF Rt = 0x7FFF
* Ra = 0x8000, Rb = 0x8000 Rt = 0x8000
* Ra = 0x4000, Rb = 0x8000 Rt = 0x6000
```

Operations:

```
Rd.H[x] = (Rs1.H[x] + Rs2.H[x]) u>> 1;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_URCRAS16(unsigned long a, unsigned long b)`

URCRAS16 (SIMD 16-bit Unsigned Halving Cross Addition & Subtraction)

Type: SIMD

Syntax:

```
URCRAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2, and subtracts the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32- bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `URADD16` and `URSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][15:0]) u>> 1;
Rd.W[x][15:0]  = (Rs1.W[x][15:0] - Rs2.W[x][31:16]) u>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URCRSA16(unsigned long a, unsigned long b)

URCRSA16 (SIMD 16-bit Unsigned Halving Cross Subtraction & Addition)

Type: SIMD

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk simultaneously. Operands are from crossed positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2. The two results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `URADD16` and `URSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][15:0]) u>> 1;
Rd.W[x][15:0]  = (Rs1.W[x][15:0] + Rs2.W[x][31:16]) u>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSTAS16(unsigned long a, unsigned long b)
URSTAS16 (SIMD 16-bit Unsigned Halving Straight Addition & Subtraction)

Type: SIMD

Syntax:

```
URSTAS16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2, and subtracts the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1. The element results are first logically right-shifted by 1 bit and then written to [31:16] of 32- bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `URADD16` and `URSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] + Rs2.W[x][31:16]) u>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] - Rs2.W[x][15:0]) u>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSTSA16(unsigned long a, unsigned long b)
URSTSA16 (SIMD 16-bit Unsigned Halving Straight Subtraction & Addition)

Type: SIMD

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk simultaneously. Operands are from corresponding positions in 32-bit chunks. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs2 from the 16-bit unsigned integer in [31:16] of 32-bit chunks in Rs1, and adds the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs1 with the 16-bit unsigned integer in [15:0] of 32-bit chunks in Rs2. The two results are first logically right-shifted by 1 bit and then written to [31:16] of 32-bit chunks in Rd and [15:0] of 32-bit chunks in Rd.

Examples:

```
Please see `URADD16` and `URSUB16` instructions.
```

Operations:

```
Rd.W[x][31:16] = (Rs1.W[x][31:16] - Rs2.W[x][31:16]) u>> 1;
Rd.W[x][15:0] = (Rs1.W[x][15:0] + Rs2.W[x][15:0]) u>> 1;
for RV32, x=0
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_URSUB16(unsigned long a, unsigned long b)`

URSUB16 (SIMD 16-bit Unsigned Halving Subtraction)

Type: SIMD

Syntax:

```
URSUB16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 16-bit unsigned integer elements in Rs2 from the 16-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFF, Rb = 0x8000 Rt = 0xFFFF
* Ra = 0x8000, Rb = 0x7FFF Rt = 0x0000
* Ra = 0x8000, Rb = 0x4000 Rt = 0x2000
```

Operations:

```
Rd.H[x] = (Rs1.H[x] - Rs2.H[x]) u>> 1;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

SIMD 8-bit Addition & Subtraction Instructions

__STATIC_FORCEINLINE unsigned long __RV_ADD8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KADD8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KSUB8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RADD8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RSUB8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SUB8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKADD8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKSUB8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URADD8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URSUB8(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_8B_ADDSUB**

SIMD 8-bit Addition & Subtraction Instructions.

Based on the types of the four 8-bit arithmetic operations, the SIMD 8-bit add/subtract instructions can be classified into 2 main categories: Addition (four 8-bit addition), and Subtraction (four 8-bit subtraction). Based on the way of how an overflow condition is handled for singed or unsigned operation, the SIMD 8-bit add/subtract instructions can be classified into 5 groups: Wrap-around (dropping overflow), Signed Halving (keeping overflow by dropping 1 LSB bit), Unsigned Halving, Signed Saturation (clipping overflow), and Unsigned Saturation. Together, there are 10 SIMD 8-bit add/subtract instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_ADD8(unsigned long a, unsigned long b)

ADD8 (SIMD 8-bit Addition)

Type: SIMD

Syntax:

```
ADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit integer element additions simultaneously.

Description: This instruction adds the 8-bit integer elements in Rs1 with the 8-bit integer elements in Rs2, and then writes the 8-bit element results to Rd.

Note: This instruction can be used for either signed or unsigned addition.

Operations:

```
Rd.B[x] = Rs1.B[x] + Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KADD8(unsigned long a, unsigned long b)

KADD8 (SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Syntax:

```
KADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 2^7-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > 127) {
  res[x] = 127;
  OV = 1;
} else if (res[x] < -128) {
  res[x] = -128;
  OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSUB8(unsigned long a, unsigned long b)

KSUB8 (SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Syntax:

```
KSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed elements saturating subtractions simultaneously.

Description: This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 27 -1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] > (2^7)-1) {
  res[x] = (2^7)-1;
  OV = 1;
} else if (res[x] < -2^7) {
  res[x] = -2^7;
  OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RADD8(unsigned long a, unsigned long b)

RADD8 (SIMD 8-bit Signed Halving Addition)

Type: SIMD

Syntax:

```
RADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element additions simultaneously. The element results are halved to avoid overflow or saturation.

Description: This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Rs1 = 0x7F, Rs2 = 0x7F, Rd = 0x7F
* Rs1 = 0x80, Rs2 = 0x80, Rd = 0x80
* Rs1 = 0x40, Rs2 = 0x80, Rd = 0xE0
```

Operations:

```
Rd.B[x] = (Rs1.B[x] + Rs2.B[x]) s>> 1; for RV32: x=3...0, for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSUB8(unsigned long a, unsigned long b)

RSUB8 (SIMD 8-bit Signed Halving Subtraction)

Type: SIMD

Syntax:

```
RSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Rs1 = 0x7F, Rs2 = 0x80, Rd = 0x7F
* Rs1 = 0x80, Rs2 = 0x7F, Rd = 0x80
* Rs1= 0x80, Rs2 = 0x40, Rd = 0xA0
```

Operations:

```
Rd.B[x] = (Rs1.B[x] - Rs2.B[x]) s>> 1;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SUB8(unsigned long a, unsigned long b)

SUB8 (SIMD 8-bit Subtraction)

Type: SIMD

Syntax:

```
SUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 8-bit integer elements in Rs2 from the 8-bit integer elements in Rs1, and then writes the result to Rd.

Note: This instruction can be used for either signed or unsigned subtraction.

Operations:

```
Rd.B[x] = Rs1.B[x] - Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKADD8(unsigned long a, unsigned long b)

UKADD8 (SIMD 8-bit Unsigned Saturating Addition)

Type: SIMD

Syntax:

```
UKADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2. If any of the results are beyond the 8-bit unsigned number range (0 <= RES <= 28-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > (2^8)-1) {
  res[x] = (2^8)-1;
  OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKSUB8(unsigned long a, unsigned long b)`

UKSUB8 (SIMD 8-bit Unsigned Saturating Subtraction)

Type: SIMD

Syntax:

```
UKSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 8-bit unsigned integer elements in Rs2 from the 8-bit unsigned integer elements in Rs1. If any of the results are beyond the 8-bit unsigned number range (0 <= RES <= 28-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] < 0) {
  res[x] = 0;
  OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_URADD8(unsigned long a, unsigned long b)`

URADD8 (SIMD 8-bit Unsigned Halving Addition)

Type: SIMD

Syntax:

```
URADD8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7F, Rb = 0x7F, Rt = 0x7F
* Ra = 0x80, Rb = 0x80, Rt = 0x80
* Ra = 0x40, Rb = 0x80, Rt = 0x60
```

Operations:

```
Rd.B[x] = (Rs1.B[x] + Rs2.B[x]) u>> 1;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSUB8(unsigned long a, unsigned long b)

URSUB8 (SIMD 8-bit Unsigned Halving Subtraction)

Type: SIMD

Syntax:

```
URSUB8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 8-bit unsigned integer elements in Rs2 from the 8-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7F, Rb = 0x80 Rt = 0xFF
* Ra = 0x80, Rb = 0x7F Rt = 0x00
* Ra = 0x80, Rb = 0x40 Rt = 0x20
```

Operations:

```
Rd.B[x] = (Rs1.B[x] - Rs2.B[x]) u>> 1;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

SIMD 16-bit Shift Instructions

__STATIC_FORCEINLINE unsigned long __RV_KSLL16(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_KSLRA16(unsigned long a, int b)

__STATIC_FORCEINLINE unsigned long __RV_KSLRA16_U(unsigned long a, int b)

__STATIC_FORCEINLINE unsigned long __RV_SLL16(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRA16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SRA16_U(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SRL16(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRL16_U(unsigned long a, unsigned int b)

__RV_KSLLI16(a, b)

__RV_SLLI16(a, b)

__RV_SRAI16(a, b)

__RV_SRAI16_U(a, b)

__RV_SRLI16(a, b)

__RV_SRLI16_U(a, b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_16B_SHIFT**

SIMD 16-bit Shift Instructions.

there are 14 SIMD 16-bit shift instructions.

Defines

__RV_KSLLI16(a, b)

KSLLI16 (SIMD 16-bit Saturating Shift Left Logical Immediate)

Type: SIMD

Syntax:

```
KSLLI16 Rd, Rs1, imm4u
```

Purpose: Do 16-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 16-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm4u constant. Any shifted value greater than 2^15-1 is saturated to 2^15-1. Any shifted value smaller than -2^15 is saturated to -2^15. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm4u[3:0];
if (sa != 0) {
  res[(15+sa):0] = Rs1.H[x] << sa;
  if (res > (2^15)-1) {
```

(continues on next page)

(continued from previous page)

```
    res = 0x7fff; OV = 1;
  } else if (res < -2^15) {
    res = 0x8000; OV = 1;
  }
  Rd.H[x] = res[15:0];
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SLLI16 (a, b)

SLLI16 (SIMD 16-bit Shift Left Logical Immediate)

Type: SIMD

Syntax:

```
SLLI16 Rd, Rs1, imm4[3:0]
```

Purpose: Do 16-bit element logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 16-bit elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm4[3:0] constant. And the results are written to Rd.

Operations:

```
sa = imm4[3:0];
Rd.H[x] = Rs1.H[x] << sa;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SRAI16 (a, b)

SRAI16 (SIMD 16-bit Shift Right Arithmetic Immediate)

Type: SIMD

Syntax:

```
SRAI16 Rd, Rs1, imm4u
SRAI16.u Rd, Rs1, imm4u
```

Purpose: Do 16-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 16-bit data elements. The shift amount is specified by the imm4u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm4u[3:0];
if (sa > 0) {
  if (`.u` form) { // SRAI16.u
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else { // SRAI16
    Rd.H[x] = SE16(Rs1.H[x][15:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__RV_SRAI16_U(a, b)

SRAI16.u (SIMD 16-bit Rounding Shift Right Arithmetic Immediate)

Type: SIMD

Syntax:

```
SRAI16 Rd, Rs1, imm4u
SRAI16.u Rd, Rs1, imm4u
```

Purpose: Do 16-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 16-bit data elements. The shift amount is specified by the imm4u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm4u[3:0];
if (sa > 0) {
  if (`.u` form) { // SRAI16.u
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else { // SRAI16
    Rd.H[x] = SE16(Rs1.H[x][15:sa]);
  }
```

(continues on next page)

(continued from previous page)

```
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__RV_SRLI16 (a, b)

SRLI16 (SIMD 16-bit Shift Right Logical Immediate)

Type: SIMD

Syntax:

```
SRLI16 Rt, Ra, imm4u
SRLI16.u Rt, Ra, imm4u
```

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm4u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm4u;
if (sa > 0) {
  if (`.u` form) { // SRLI16.u
    res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[16:1];
  } else { // SRLI16
    Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SRLI16_U (a, b)

SRLI16.u (SIMD 16-bit Rounding Shift Right Logical Immediate)

Type: SIMD

Syntax:

```
SRLI16 Rt, Ra, imm4u
SRLI16.u Rt, Ra, imm4u
```

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm4u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm4u;
if (sa > 0) {
  if (`.u` form) { // SRLI16.u
    res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[16:1];
  } else { // SRLI16
    Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

`__STATIC_FORCEINLINE unsigned long __RV_KSLL16(unsigned long a, unsigned int b)`

KSLL16 (SIMD 16-bit Saturating Shift Left Logical)

Type: SIMD

Syntax:

```
KSLL16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 16-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 4-bits of the value in the Rs2 register. Any shifted value greater than 2^15-1 is saturated to 2^15-1. Any shifted value smaller than -2^15 is saturated to -2^15. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[3:0];
if (sa != 0) {
  res[(15+sa):0] = Rs1.H[x] << sa;
  if (res > (2^15)-1) {
    res = 0x7fff; OV = 1;
  } else if (res < -2^15) {
    res = 0x8000; OV = 1;
  }
  Rd.H[x] = res[15:0];
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA16(unsigned long a, int b)`

KSLRA16 (SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Syntax:

```
KSLRA16 Rd, Rs1, Rs2
KSLRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of [-2^4, 2^4-1]. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of `Rs2[4:0]==-2^4 (0x10)` is defined to be equivalent to the behavior of `Rs2[4:0]==-(2^4-1) (0x11)`. The left-shifted results are saturated to the 16-bit signed integer range of [-2^15, 2^15-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:5] will not affect this instruction.

Operations:

```
if (Rs2[4:0] < 0) {
  sa = -Rs2[4:0];
  sa = (sa == 16)? 15 : sa;
  if (`.u` form) {
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else {
    Rd.H[x] = SE16(Rs1.H[x][15:sa]);
  }
} else {
  sa = Rs2[3:0];
```

(continues on next page)

(continued from previous page)

```
  res[(15+sa):0] = Rs1.H[x] <<(logic) sa;
  if (res > (2^15)-1) {
    res[15:0] = 0x7fff; OV = 1;
  } else if (res < -2^15) {
    res[15:0] = 0x8000; OV = 1;
  }
  d.H[x] = res[15:0];
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA16_U(unsigned long a, int b)`

KSLRA16.u (SIMD 16-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD

Syntax:

```
KSLRA16 Rd, Rs1, Rs2
KSLRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of [-2^4, 2^4-1]. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of `Rs2[4:0]==-2^4 (0x10)` is defined to be equivalent to the behavior of `Rs2[4:0]==-(2^4-1) (0x11)`. The left-shifted results are saturated to the 16-bit signed integer range of [-2^15, 2^15-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:5] will not affect this instruction.

Operations:

```
if (Rs2[4:0] < 0) {
  sa = -Rs2[4:0];
  sa = (sa == 16)? 15 : sa;
  if (`.u` form) {
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else {
    Rd.H[x] = SE16(Rs1.H[x][15:sa]);
  }
} else {
  sa = Rs2[3:0];
  res[(15+sa):0] = Rs1.H[x] <<(logic) sa;
  if (res > (2^15)-1) {
    res[15:0] = 0x7fff; OV = 1;
```

(continues on next page)

(continued from previous page)

```
  } else if (res < -2^15) {
    res[15:0] = 0x8000; OV = 1;
  }
  d.H[x] = res[15:0];
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SLL16(unsigned long a, unsigned int b)`

SLL16 (SIMD 16-bit Shift Left Logical)

Type: SIMD

Syntax:

```
SLL16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 16-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 4-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[3:0];
Rd.H[x] = Rs1.H[x] << sa;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRA16(unsigned long a, unsigned long b)`

SRA16 (SIMD 16-bit Shift Right Arithmetic)

Type: SIMD

Syntax:

```
SRA16 Rd, Rs1, Rs2
SRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[3:0];
if (sa != 0) {
  if (`.u` form) { // SRA16.u
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else { // SRA16
    Rd.H[x] = SE16(Rs1.H[x][15:sa])
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRA16_U(unsigned long a, unsigned long b)`

SRA16.u (SIMD 16-bit Rounding Shift Right Arithmetic)

Type: SIMD

Syntax:

```
SRA16 Rd, Rs1, Rs2
SRA16.u Rd, Rs1, Rs2
```

Purpose: Do 16-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[3:0];
if (sa != 0) {
  if (`.u` form) { // SRA16.u
    res[15:-1] = SE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[15:0];
  } else { // SRA16
    Rd.H[x] = SE16(Rs1.H[x][15:sa])
  }
} else {
```

(continues on next page)

(continued from previous page)

```
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRL16(unsigned long a, unsigned int b)

SRL16 (SIMD 16-bit Shift Right Logical)

Type: SIMD

Syntax:

```
SRL16 Rt, Ra, Rb
 SRL16.u Rt, Ra, Rb
```

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding upoperations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[3:0];
if (sa > 0) {
  if (`.u` form) { // SRL16.u
    res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[16:1];
  } else { // SRL16
    Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRL16_U(unsigned long a, unsigned int b)

SRL16.u (SIMD 16-bit Rounding Shift Right Logical)

Type: SIMD

Syntax:

```
SRL16 Rt, Ra, Rb
 SRL16.u Rt, Ra, Rb
```

Purpose: Do 16-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding upoperations on the shifted results.

Description: The 16-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 4-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 16-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[3:0];
if (sa > 0) {
  if (`.u` form) { // SRL16.u
    res[16:0] = ZE17(Rs1.H[x][15:sa-1]) + 1;
    Rd.H[x] = res[16:1];
  } else { // SRL16
    Rd.H[x] = ZE16(Rs1.H[x][15:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

SIMD 8-bit Shift Instructions

`__STATIC_FORCEINLINE unsigned long __RV_KSLL8(unsigned long a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA8(unsigned long a, int b)`

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA8_U(unsigned long a, int b)`

`__STATIC_FORCEINLINE unsigned long __RV_SLL8(unsigned long a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_SRA8(unsigned long a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_SRA8_U(unsigned long a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_SRL8(unsigned long a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_SRL8_U(unsigned long a, unsigned int b)`

`__RV_KSLLI8` (a, b)

`__RV_SLLI8` (a, b)

`__RV_SRAI8` (a, b)

`__RV_SRAI8_U` (a, b)

`__RV_SRLI8` (a, b)

__RV_SRLI8_U (a, b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_8B_SHIFT**

SIMD 8-bit Shift Instructions.

there are 14 SIMD 8-bit shift instructions.

Defines

__RV_KSLLI8 (a, b)

KSLLI8 (SIMD 8-bit Saturating Shift Left Logical Immediate)

Type: SIMD

Syntax:

```
KSLLI8 Rd, Rs1, imm3u
```

Purpose: Do 8-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 8-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm3u constant. Any shifted value greater than 2^7-1 is saturated to 2^7-1. Any shifted value smaller than -2^7 is saturated to -2^7. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm3u[2:0];
if (sa != 0) {
  res[(7+sa):0] = Rs1.B[x] << sa;
  if (res > (2^7)-1) {
    res = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SLLI8 (a, b)

SLLI8 (SIMD 8-bit Shift Left Logical Immediate)

Type: SIMD

Syntax:

```
SLLI8 Rd, Rs1, imm3u
```

Purpose: Do 8-bit elements logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 8-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the imm3u constant.

Operations:

```
sa = imm3u[2:0];
Rd.B[x] = Rs1.B[x] << sa;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SRAI8 (a, b)

SRAI8 (SIMD 8-bit Shift Right Arithmetic Immediate)

Type: SIMD

Syntax:

```
SRAI8 Rd, Rs1, imm3u
SRAI8.u Rd, Rs1, imm3u
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the imm3u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm3u[2:0];
if (sa > 0) {
  if (`.u` form) { // SRA8.u
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else { // SRA8
    Rd.B[x] = SE8(Rd.B[x][7:sa])
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned int type of value stored in b

`__RV_SRAI8_U` (a, b)

SRAI8.u (SIMD 8-bit Rounding Shift Right Arithmetic Immediate)

Type: SIMD

Syntax:

```
SRAI8 Rd, Rs1, imm3u
SRAI8.u Rd, Rs1, imm3u
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the imm3u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm3u[2:0];
if (sa > 0) {
  if (`.u` form) { // SRA8.u
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else { // SRA8
    Rd.B[x] = SE8(Rd.B[x][7:sa])
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__RV_SRLI8` (a, b)

SRLI8 (SIMD 8-bit Shift Right Logical Immediate)

Type: SIMD

Syntax:

```
SRLI8 Rt, Ra, imm3u
SRLI8.u Rt, Ra, imm3u
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm3u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm3u[2:0];
if (sa > 0) {
  if (`.u` form) { // SRLI8.u
    res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[8:1];
  } else { // SRLI8
    Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_SRLI8_U (a, b)

SRLI8.u (SIMD 8-bit Rounding Shift Right Logical Immediate)

Type: SIMD

Syntax:

```
SRLI8 Rt, Ra, imm3u
SRLI8.u Rt, Ra, imm3u
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm3u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm3u[2:0];
if (sa > 0) {
  if (`.u` form) { // SRLI8.u
    res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[8:1];
  } else { // SRLI8
    Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

`__STATIC_FORCEINLINE unsigned long __RV_KSLL8(unsigned long a, unsigned int b)`

KSLL8 (SIMD 8-bit Saturating Shift Left Logical)

Type: SIMD

Syntax:

```
KSLL8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 8-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 3-bits of the value in the Rs2 register. Any shifted value greater than 2^7-1 is saturated to 2^7-1. Any shifted value smaller than -2^7 is saturated to -2^7. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[2:0];
if (sa != 0) {
  res[(7+sa):0] = Rs1.B[x] << sa;
  if (res > (2^7)-1) {
    res = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA8(unsigned long a, int b)`

KSLRA8 (SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Syntax:

```
KSLRA8 Rd, Rs1, Rs2
KSLRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of [-2^3, 2^3-1]. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of `Rs2[3:0]==-2^3 (0x8)` is defined to be equivalent to the behavior of `Rs2[3:0]==-(2^3-1) (0x9)`. The left-shifted results are saturated to the 8-bit signed integer range of [-2^7, 2^7-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:4] will not affect this instruction.

Operations:

```
if (Rs2[3:0] < 0) {
  sa = -Rs2[3:0];
  sa = (sa == 8)? 7 : sa;
  if (`.u` form) {
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else {
    Rd.B[x] = SE8(Rs1.B[x][7:sa]);
  }
} else {
  sa = Rs2[2:0];
  res[(7+sa):0] = Rs1.B[x] <<(logic) sa;
  if (res > (2^7)-1) {
    res[7:0] = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res[7:0] = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSLRA8_U(unsigned long a, int b)

KSLRA8.u (SIMD 8-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD

Syntax:

```
KSLRA8 Rd, Rs1, Rs2
KSLRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of [-2^3, 2^3-1]. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of `Rs2[3:0]==-2^3 (0x8)` is defined to be equivalent to the behavior of `Rs2[3:0]==-(2^3-1) (0x9)`. The left-shifted results are saturated to the 8-bit signed integer range of [-2^7, 2^7-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:4] will not affect this instruction.

Operations:

```
if (Rs2[3:0] < 0) {
  sa = -Rs2[3:0];
  sa = (sa == 8)? 7 : sa;
  if (`.u` form) {
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else {
    Rd.B[x] = SE8(Rs1.B[x][7:sa]);
  }
} else {
  sa = Rs2[2:0];
  res[(7+sa):0] = Rs1.B[x] <<(logic) sa;
  if (res > (2^7)-1) {
    res[7:0] = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res[7:0] = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SLL8(unsigned long a, unsigned int b)`

SLL8 (SIMD 8-bit Shift Left Logical)

Type: SIMD

Syntax:

```
SLL8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 8-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 3-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[2:0];
Rd.B[x] = Rs1.B[x] << sa;
```

(continues on next page)

(continued from previous page)

```
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRA8(unsigned long a, unsigned int b)

SRA8 (SIMD 8-bit Shift Right Arithmetic)

Type: SIMD

Syntax:

```
SRA8 Rd, Rs1, Rs2
SRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[2:0];
if (sa > 0) {
  if (`.u` form) { // SRA8.u
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else { // SRA8
    Rd.B[x] = SE8(Rd.B[x][7:sa])
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRA8_U(unsigned long a, unsigned int b)

SRA8.u (SIMD 8-bit Rounding Shift Right Arithmetic)

Type: SIMD

Syntax:

```
SRA8 Rd, Rs1, Rs2
SRA8.u Rd, Rs1, Rs2
```

Purpose: Do 8-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[2:0];
if (sa > 0) {
  if (`.u` form) { // SRA8.u
    res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[7:0];
  } else { // SRA8
    Rd.B[x] = SE8(Rd.B[x][7:sa])
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRL8(unsigned long a, unsigned int b)`

SRL8 (SIMD 8-bit Shift Right Logical)

Type: SIMD

Syntax:

```
SRL8 Rt, Ra, Rb
SRL8.u Rt, Ra, Rb
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[2:0];
if (sa > 0) {
  if (`.u` form) { // SRL8.u
```

(continues on next page)

(continued from previous page)

```
    res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[8:1];
  } else { // SRL8
    Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRL8_U(unsigned long a, unsigned int b)`

SRL8.u (SIMD 8-bit Rounding Shift Right Logical)

Type: SIMD

Syntax:

```
SRL8 Rt, Ra, Rb
SRL8.u Rt, Ra, Rb
```

Purpose: Do 8-bit elements logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 8-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 3-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 8-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[2:0];
if (sa > 0) {
  if (`.u` form) { // SRL8.u
    res[8:0] = ZE9(Rs1.B[x][7:sa-1]) + 1;
    Rd.B[x] = res[8:1];
  } else { // SRL8
    Rd.B[x] = ZE8(Rs1.B[x][7:sa]);
  }
} else {
  Rd = Rs1;
}
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

SIMD 16-bit Compare Instructions

__STATIC_FORCEINLINE unsigned long __RV_CMPEQ16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SCMPLE16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SCMPLT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UCMPLE16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UCMPLT16(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_16B_CMP**

SIMD 16-bit Compare Instructions.

there are 5 SIMD 16-bit Compare instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_CMPEQ16(unsigned long a, unsigned long b)

CMPEQ16 (SIMD 16-bit Integer Compare Equal)

Type: SIMD

Syntax:

```
CMPEQ16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit integer elements equal comparisons simultaneously.

Description: This instruction compares the 16-bit integer elements in Rs1 with the 16-bit integer elements in Rs2 to see if they are equal. If they are equal, the result is 0xFFFF; otherwise, the result is 0x0. The 16-bit element comparison results are written to Rt.

Note: This instruction can be used for either signed or unsigned numbers.

Operations:

```
Rd.H[x] = (Rs1.H[x] == Rs2.H[x])? 0xffff : 0x0;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SCMPLE16(unsigned long a, unsigned long b)

SCMPLE16 (SIMD 16-bit Signed Compare Less Than & Equal)

Type: SIMD

Syntax:

```
SCMPLE16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] {le} Rs2.H[x])? 0xffff : 0x0;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SCMPLT16(unsigned long a, unsigned long b)

SCMPLT16 (SIMD 16-bit Signed Compare Less Than)

Type: SIMD

Syntax:

```
SCMPLT16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements less than comparisons simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the two 16- bit signed integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? 0xffff : 0x0;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UCMPLE16(unsigned long a, unsigned long b)

UCMPLE16 (SIMD 16-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Syntax:

```
UCMPLE16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] <=u Rs2.H[x])? 0xffff : 0x0;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UCMPLT16(unsigned long a, unsigned long b)`

UCMPLT16 (SIMD 16-bit Unsigned Compare Less Than)

Type: SIMD

Syntax:

```
UCMPLT16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements less than comparisons simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFFFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] <u Rs2.H[x])? 0xffff : 0x0;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

SIMD 8-bit Compare Instructions

`__STATIC_FORCEINLINE unsigned long __RV_CMPEQ8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_SCMPLE8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_SCMPLT8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_UCMPLE8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_UCMPLT8(unsigned long a, unsigned long b)`

group **`NMSIS_Core_DSP_Intrinsic_SIMD_8B_CMP`**

SIMD 8-bit Compare Instructions.

there are 5 SIMD 8-bit Compare instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_CMPEQ8(unsigned long a, unsigned long b)

CMPEQ8 (SIMD 8-bit Integer Compare Equal)

Type: SIMD

Syntax:

```
CMPEQ8 Rs, Rs1, Rs2
```

Purpose: Do 8-bit integer elements equal comparisons simultaneously.

Description: This instruction compares the 8-bit integer elements in Rs1 with the 8-bit integer elements in Rs2 to see if they are equal. If they are equal, the result is 0xFF; otherwise, the result is 0x0. The 8-bit element comparison results are written to Rd.

Note: This instruction can be used for either signed or unsigned numbers.

Operations:

```
Rd.B[x] = (Rs1.B[x] == Rs2.B[x])? 0xff : 0x0;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SCMPLE8(unsigned long a, unsigned long b)

SCMPLE8 (SIMD 8-bit Signed Compare Less Than & Equal)

Type: SIMD

Syntax:

```
SCMPLE8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd

Operations:

```
Rd.B[x] = (Rs1.B[x] {le} Rs2.B[x])? 0xff : 0x0;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SCMPLT8(unsigned long a, unsigned long b)

SCMPLT8 (SIMD 8-bit Signed Compare Less Than)

Type: SIMD

Syntax:

```
SCMPLT8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements less than comparisons simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? 0xff : 0x0;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UCMPLE8(unsigned long a, unsigned long b)

UCMPLE8 (SIMD 8-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Syntax:

```
UCMPLE8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements less than & equal comparisons simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than or equal to the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The four comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] <=u Rs2.B[x])? 0xff : 0x0;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UCMPLT8(unsigned long a, unsigned long b)

UCMPLT8 (SIMD 8-bit Unsigned Compare Less Than)

Type: SIMD

Syntax:

```
UCMPLT8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements less than comparisons simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 to see if the one in Rs1 is less than the one in Rs2. If it is true, the result is 0xFF; otherwise, the result is 0x0. The element comparison results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] <u Rs2.B[x])? 0xff : 0x0;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

SIMD 16-bit Multiply Instructions

`__STATIC_FORCEINLINE unsigned long __RV_KHM16(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_KHMX16(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long long __RV_SMUL16(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_SMULX16(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_UMUL16(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_UMULX16(unsigned int a, unsigned int b)`

group **`NMSIS_Core_DSP_Intrinsic_SIMD_16B_MULTIPLY`**

SIMD 16-bit Multiply Instructions.

there are 6 SIMD 16-bit Multiply instructions.

Functions

`__STATIC_FORCEINLINE unsigned long __RV_KHM16(unsigned long a, unsigned long b)`

KHM16 (SIMD Signed Saturating Q15 Multiply)

Type: SIMD

Syntax:

```
KHM16 Rd, Rs1, Rs2
KHMX16 Rd, Rs1, Rs2
```

Purpose: Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to Q15 numbers again.

Description: For the `KHM16` instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2. For the `KHMX16` instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit

Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. The Q30 results are then right-shifted 15-bits and saturated into Q15 values. The Q15 results are then written into Rd. When both the two Q15 inputs of a multiplication are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
if (is `KHM16`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top
  op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom
} else if (is `KHMX16`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
  op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x8000 != aop | 0x8000 != bop) {
    res = (aop s* bop) >> 15;
  } else {
    res= 0x7FFF;
    OV = 1;
  }
}
Rd.W[x/2] = concat(rest, resb);
for RV32: x=0
for RV64: x=0,2
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KHMX16(unsigned long a, unsigned long b)

KHMX16 (SIMD Signed Saturating Crossed Q15 Multiply)

Type: SIMD

Syntax:

```
KHM16 Rd, Rs1, Rs2
KHMX16 Rd, Rs1, Rs2
```

Purpose: Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to Q15 numbers again.

Description: For the `KHM16` instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2. For the `KHMX16` instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. The Q30 results are then right-shifted 15-bits and saturated into Q15 values. The Q15 results are then written into Rd. When both the two Q15 inputs of a multiplication are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
if (is `KHM16`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top
  op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom
} else if (is `KHMX16`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
  op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x8000 != aop | 0x8000 != bop) {
    res = (aop s* bop) >> 15;
  } else {
    res= 0x7FFF;
    OV = 1;
  }
}
Rd.W[x/2] = concat(rest, resb);
for RV32: x=0
for RV64: x=0,2
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_SMUL16(unsigned int a, unsigned int b)`

SMUL16 (SIMD Signed 16-bit Multiply)

Type: SIMD

Syntax:

```
SMUL16 Rd, Rs1, Rs2
SMULX16 Rd, Rs1, Rs2
```

Purpose: Do signed 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the `SMUL16` instruction, multiply the top 16-bit Q15 content of Rs1 with the top 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2. For the `SMULX16` instruction, multiply the top 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the top 16- bit Q15 content of Rs2. The two Q30 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even `2d` register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the `SMUL16` instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2. For the `SMULX16` instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2. The two 32-bit Q30 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

```
* RV32:
if (is `SMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `SMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop s* bop;
}
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H] = rest;
R[t_L] = resb;
* RV64:
if (is `SMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `SMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop s* bop;
}
Rd.W[1] = rest;
Rd.W[0] = resb;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_SMULX16(unsigned int a, unsigned int b)

SMULX16 (SIMD Signed Crossed 16-bit Multiply)

Type: SIMD

Syntax:

```
SMUL16 Rd, Rs1, Rs2
SMULX16 Rd, Rs1, Rs2
```

Purpose: Do signed 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the `SMUL16` instruction, multiply the top 16-bit Q15 content of Rs1 with the top 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2. For the `SMULX16` instruction, multiply the top 16-bit Q15 content of Rs1 with the bottom 16-bit Q15 content of Rs2. At the same time, multiply the bottom 16-bit Q15 content of Rs1 with the top 16- bit Q15 content of Rs2. The two Q30 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even `2d` register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the `SMUL16` instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2. For the `SMULX16` instruction, multiply the top 16-bit Q15 content of the lower 32-bit word in Rs1 with the bottom 16-bit Q15 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit Q15 content of the lower 32-bit word in Rs1 with the top 16-bit Q15 content of the lower 32-bit word in Rs2. The two 32-bit Q30 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

```
* RV32:
if (is `SMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `SMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop s* bop;
}
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H] = rest;
R[t_L] = resb;
* RV64:
if (is `SMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `SMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop s* bop;
}
Rd.W[1] = rest;
Rd.W[0] = resb;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_UMUL16(unsigned int a, unsigned int b)`

UMUL16 (SIMD Unsigned 16-bit Multiply)

Type: SIMD

Syntax:

```
UMUL16 Rd, Rs1, Rs2
UMULX16 Rd, Rs1, Rs2
```

Purpose: Do unsigned 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the `UMUL16` instruction, multiply the top 16-bit U16 content of Rs1 with the top 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2. For the `UMULX16` instruction, multiply the top 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the top 16- bit U16 content of Rs2. The two U32 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even `2d` register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the `UMUL16` instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2. For the `UMULX16` instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2. The two 32-bit U32 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

```
* RV32:
if (is `UMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `UMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop u* bop;
}
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H] = rest;
R[t_L] = resb;
* RV64:
if (is `UMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `UMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop u* bop;
}
Rd.W[1] = rest;
Rd.W[0] = resb;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UMULX16(unsigned int a, unsigned int b)

UMULX16 (SIMD Unsigned Crossed 16-bit Multiply)

Type: SIMD

Syntax:

```
UMUL16 Rd, Rs1, Rs2
UMULX16 Rd, Rs1, Rs2
```

Purpose: Do unsigned 16-bit multiplications and generate two 32-bit results simultaneously.

RV32 Description: For the `UMUL16` instruction, multiply the top 16-bit U16 content of Rs1 with the top 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2. For the `UMULX16` instruction, multiply the top 16-bit U16 content of Rs1 with the bottom 16-bit U16 content of Rs2. At the same time, multiply the bottom 16-bit U16 content of Rs1 with the top 16- bit U16 content of Rs2. The two U32 results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the 32-bit result calculated from the top part of Rs1 and the even `2d` register of the pair contains the 32-bit result calculated from the bottom part of Rs1.

RV64 Description: For the `UMUL16` instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2. For the `UMULX16` instruction, multiply the top 16-bit U16 content of the lower 32-bit word in Rs1 with the bottom 16-bit U16 content of the lower 32-bit word in Rs2. At the same time, multiply the bottom 16-bit U16 content of the lower 32-bit word in Rs1 with the top 16-bit U16 content of the lower 32-bit word in Rs2. The two 32-bit U32 results are then written into Rd. The result calculated from the top 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[1]. And the result calculated from the bottom 16-bit of the lower 32-bit word in Rs1 is written to Rd.W[0]

Operations:

```
* RV32:
if (is `UMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `UMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop u* bop;
}
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H] = rest;
R[t_L] = resb;
* RV64:
if (is `UMUL16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[1]; // top
  op1b = Rs1.H[0]; op2b = Rs2.H[0]; // bottom
} else if (is `UMULX16`) {
  op1t = Rs1.H[1]; op2t = Rs2.H[0]; // Rs1 top
  op1b = Rs1.H[0]; op2b = Rs2.H[1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  res = aop u* bop;
```

(continues on next page)

(continued from previous page)

```
}
Rd.W[1] = rest;
Rd.W[0] = resb;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

SIMD 8-bit Multiply Instructions

`__STATIC_FORCEINLINE unsigned long __RV_KHM8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_KHMX8(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long long __RV_SMUL8(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_SMULX8(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_UMUL8(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_UMULX8(unsigned int a, unsigned int b)`

group **`NMSIS_Core_DSP_Intrinsic_SIMD_8B_MULTIPLY`**

SIMD 8-bit Multiply Instructions.

there are 6 SIMD 8-bit Multiply instructions.

Functions

`__STATIC_FORCEINLINE unsigned long __RV_KHM8(unsigned long a, unsigned long b)`

KHM8 (SIMD Signed Saturating Q7 Multiply)

Type: SIMD

Syntax:

```
KHM8 Rd, Rs1, Rs2
KHMX8 Rd, Rs1, Rs2
```

Purpose: Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description: For the `KHM8` instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2. For the `KHMX16` instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```
if (is `KHM8`) {
  op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top
  op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom
} else if (is `KHMX8`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
  op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x80 != aop | 0x80 != bop) {
    res = (aop s* bop) >> 7;
  } else {
    res= 0x7F;
    OV = 1;
  }
}
Rd.H[x/2] = concat(rest, resb);
for RV32, x=0,2
for RV64, x=0,2,4,6
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KHMX8(unsigned long a, unsigned long b)`

KHMX8 (SIMD Signed Saturating Crossed Q7 Multiply)

Type: SIMD

Syntax:

```
KHM8 Rd, Rs1, Rs2
KHMX8 Rd, Rs1, Rs2
```

Purpose: Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description: For the `KHM8` instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2. For the `KHMX16` instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```
if (is `KHM8`) {
  op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top
  op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom
} else if (is `KHMX8`) {
  op1t = Rs1.H[x+1]; op2t = Rs2.H[x]; // Rs1 top
  op1b = Rs1.H[x]; op2b = Rs2.H[x+1]; // Rs1 bottom
}
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
```

(continues on next page)

(continued from previous page)

```
  if (0x80 != aop | 0x80 != bop) {
    res = (aop s* bop) >> 7;
  } else {
    res= 0x7F;
    OV = 1;
  }
}
Rd.H[x/2] = concat(rest, resb);
for RV32, x=0,2
for RV64, x=0,2,4,6
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_SMUL8(unsigned int a, unsigned int b)

SMUL8 (SIMD Signed 8-bit Multiply)

Type: SIMD

Syntax:

```
SMUL8 Rd, Rs1, Rs2
SMULX8 Rd, Rs1, Rs2
```

Purpose: Do signed 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the `SMUL8` instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2. For the `SMULX8` instruction, multiply the first and second 8-bit data elements of Rs1 with the second and first 8-bit data elements of Rs2. At the same time, multiply the third and fourth 8-bit data elements of Rs1 with the fourth and third 8-bit data elements of Rs2. The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even `2d` register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the `SMUL8` instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2. For the `SMULX8` instruction, multiply the first and second 8-bit data elements of Rs1 with the second and first 8-bit data elements of Rs2. At the same time, multiply the third and fourth 8-bit data elements of Rs1 with the fourth and third 8-bit data elements of Rs2. The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

```
* RV32:
if (is `SMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `SMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
```

(continues on next page)

(continued from previous page)

```
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2
* RV64:
if (is `SMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `SMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1];
Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0];
x = 0 and 2
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_SMULX8(unsigned int a, unsigned int b)

SMULX8 (SIMD Signed Crossed 8-bit Multiply)

Type: SIMD

Syntax:

```
SMUL8 Rd, Rs1, Rs2
SMULX8 Rd, Rs1, Rs2
```

Purpose: Do signed 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the `SMUL8` instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2. For the `SMULX8` instruction, multiply the first and second 8-bit data elements of Rs1 with the second and first 8-bit data elements of Rs2. At the same time, multiply the third and fourth 8-bit data elements of Rs1 with the fourth and third 8-bit data elements of Rs2. The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even `2d` register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the `SMUL8` instruction, multiply the 8-bit data elements of Rs1 with the corresponding 8-bit data elements of Rs2. For the `SMULX8` instruction, multiply the first and second 8-bit data elements of Rs1 with the second and first 8-bit data elements of Rs2. At the same time, multiply the third and fourth 8-bit data elements of Rs1 with the fourth and third 8-bit data elements of Rs2. The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

```
* RV32:
if (is `SMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `SMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2
* RV64:
if (is `SMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `SMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] s* op2t[x/2];
resb[x/2] = op1b[x/2] s* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1];
Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0];
x = 0 and 2
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_UMUL8(unsigned int a, unsigned int b)`

UMUL8 (SIMD Unsigned 8-bit Multiply)

Type: SIMD

Syntax:

```
UMUL8 Rd, Rs1, Rs2
UMULX8 Rd, Rs1, Rs2
```

Purpose: Do unsigned 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the `UMUL8` instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2. For the `UMULX8` instruction, multiply the first and second unsigned 8-bit data elements of Rs1 with the second and first unsigned 8-bit data elements of Rs2. At the same time, multiply the third and fourth unsigned 8-bit data elements of Rs1 with the fourth and third unsigned 8-bit data elements of Rs2. The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even `2d` register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the `UMUL8` instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2. For the `UMULX8` instruction, multiply the first and second unsigned 8-bit data elements of Rs1 with the second and first unsigned 8-bit data elements of Rs2. At the same time, multiply the third and fourth unsigned 8-bit data elements of Rs1 with the fourth and third unsigned 8-bit data elements of Rs2. The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

```
* RV32:
if (is `UMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `UMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] u* op2t[x/2];
resb[x/2] = op1b[x/2] u* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2
* RV64:
if (is `UMUL8`) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `UMULX8`) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
    op1b[x/2]  =  Rs1.B[x]; op2b[x/2]  =  Rs2.B[x+1];  //  Rs1  bottom
}
rest[x/2]  =  op1t[x/2]  u*  op2t[x/2];
resb[x/2]  =  op1b[x/2]  u*  op2b[x/2];
t_L  =  CONCAT(Rd(4,1),1'b0); t_H  =  CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1];
Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0]; x = 0 and 2
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_UMULX8(unsigned int a, unsigned int b)`

UMULX8 (SIMD Unsigned Crossed 8-bit Multiply)

Type: SIMD

Syntax:

```
UMUL8 Rd, Rs1, Rs2
UMULX8 Rd, Rs1, Rs2
```

Purpose: Do unsigned 8-bit multiplications and generate four 16-bit results simultaneously.

RV32 Description: For the `UMUL8` instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2. For the `UMULX8` instruction, multiply the first and second unsigned 8-bit data elements of Rs1 with the second and first unsigned 8-bit data elements of Rs2. At the same time, multiply the third and fourth unsigned 8-bit data elements of Rs1 with the fourth and third unsigned 8-bit data elements of Rs2. The four 16-bit results are then written into an even/odd pair of registers specified by Rd(4,1). Rd(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even `2d` register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.

RV64 Description: For the `UMUL8` instruction, multiply the unsigned 8-bit data elements of Rs1 with the corresponding unsigned 8-bit data elements of Rs2. For the `UMULX8` instruction, multiply the first and second unsigned 8-bit data elements of Rs1 with the second and first unsigned 8-bit data elements of Rs2. At the same time, multiply the third and fourth unsigned 8-bit data elements of Rs1 with the fourth and third unsigned 8-bit data elements of Rs2. The four 16-bit results are then written into Rd. The Rd.W[1] contains the two 16-bit results calculated from the top part of Rs1 and the Rd.W[0] contains the two 16-bit results calculated from the bottom part of Rs1.

Operations:

```
* RV32:
if (is `UMUL8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `UMULX8`) {
  op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
  op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x+1]; // Rs1 bottom
}
rest[x/2] = op1t[x/2] u* op2t[x/2];
resb[x/2] = op1b[x/2] u* op2b[x/2];
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].H[1] = rest[1]; R[t_H].H[0] = resb[1];
R[t_L].H[1] = rest[0]; R[t_L].H[0] = resb[0];
x = 0 and 2
* RV64:
if (is `UMUL8`) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x+1]; // top
    op1b[x/2] = Rs1.B[x]; op2b[x/2] = Rs2.B[x]; // bottom
} else if (is `UMULX8`) {
    op1t[x/2] = Rs1.B[x+1]; op2t[x/2] = Rs2.B[x]; // Rs1 top
    op1b[x/2]  =  Rs1.B[x]; op2b[x/2]  =  Rs2.B[x+1];  //  Rs1  bottom
}
rest[x/2]  =  op1t[x/2]  u*  op2t[x/2];
resb[x/2]  =  op1b[x/2]  u*  op2b[x/2];
t_L  =  CONCAT(Rd(4,1),1'b0); t_H  =  CONCAT(Rd(4,1),1'b1);
Rd.W[1].H[1] = rest[1]; Rd.W[1].H[0] = resb[1];
Rd.W[0].H[1] = rest[0]; Rd.W[0].H[0] = resb[0]; x = 0 and 2
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

SIMD 16-bit Miscellaneous Instructions

`__STATIC_FORCEINLINE unsigned long __RV_CLRS16(unsigned long a)`

__STATIC_FORCEINLINE unsigned long __RV_CLO16(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_CLZ16(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_KABS16(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SMAX16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SMIN16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMAX16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMIN16(unsigned long a, unsigned long b)

__RV_SCLIP16(a, b)

__RV_UCLIP16(a, b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_16B_MISC**

SIMD 16-bit Miscellaneous Instructions.

there are 10 SIMD 16-bit Misc instructions.

Defines

__RV_SCLIP16(a, b)

SCLIP16 (SIMD 16-bit Signed Clip Value)

Type: SIMD

Syntax:

```
SCLIP16 Rd, Rs1, imm4u[3:0]
```

Purpose: Limit the 16-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 16-bit signed integer elements stored in Rs1 into a signed integer range between 2imm4u-1 and -2imm4u, and writes the limited results to Rd. For example, if imm4u is 3, the 16-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm4u)-1) {
  src = (2^imm4u)-1;
  OV = 1;
} else if (src < -2^imm4u) {
  src = -2^imm4u;
  OV = 1;
}
Rd.H[x] = src
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a
- [in] b: unsigned int type of value stored in b

__RV_UCLIP16 (a, b)

UCLIP16 (SIMD 16-bit Unsigned Clip Value)

Type: SIMD

Syntax:

```
UCLIP16 Rt, Ra, imm4u
```

Purpose: Limit the 16-bit signed elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 16-bit signed elements stored in Rs1 into an unsigned integer range between 2imm4u-1 and 0, and writes the limited results to Rd. For example, if imm4u is 3, the 16-bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm4u)-1) {
  src = (2^imm4u)-1;
  OV = 1;
} else if (src < 0) {
  src = 0;
  OV = 1;
}
Rd.H[x] = src;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

__STATIC_FORCEINLINE unsigned long __RV_CLRS16(unsigned long a)

CLRS16 (SIMD 16-bit Count Leading Redundant Sign)

Type: SIMD

Syntax:

```
CLRS16 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 16-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 16-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 16- bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 14 to 0) {
  if (snum[x](i) == snum[x](15)) {
    cnt[x] = cnt[x] + 1;
```

(continues on next page)

(continued from previous page)

```
  } else {
    break;
  }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_CLO16(unsigned long a)`

CLO16 (SIMD 16-bit Count Leading One)

Type: SIMD

Syntax:

```
CLO16 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 16-bit elements of a general register.

Description: Starting from the most significant bits of the 16-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 16-bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 15 to 0) {
  if (snum[x](i) == 1) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_CLZ16(unsigned long a)`

CLZ16 (SIMD 16-bit Count Leading Zero)

Type: SIMD

Syntax:

```
CLZ16 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 16-bit elements of a general register.

Description: Starting from the most significant bits of the 16-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 16-bit elements of Rd.

Operations:

```
snum[x] = Rs1.H[x];
cnt[x] = 0;
for (i = 15 to 0) {
  if (snum[x](i) == 0) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.H[x] = cnt[x];
for RV32: x=1...0
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_KABS16(unsigned long a)`

KABS16 (SIMD 16-bit Saturating Absolute)

Type: SIMD

Syntax:

```
KABS16 Rd, Rs1
```

Purpose: Get the absolute value of 16-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 16-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000, this instruction generates 0x7fff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src == 0x8000) {
  src = 0x7fff;
  OV = 1;
} else if (src[15] == 1)
  src = -src;
}
Rd.H[x] = src;
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_SMAX16(unsigned long a, unsigned long b)`

SMAX16 (SIMD 16-bit Signed Maximum)

Type: SIMD

Syntax:

```
SMAX16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] > Rs2.H[x])? Rs1.H[x] : Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SMIN16(unsigned long a, unsigned long b)
SMIN16 (SIMD 16-bit Signed Minimum)

Type: SIMD

Syntax:

```
SMIN16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] < Rs2.H[x])? Rs1.H[x] : Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMAX16(unsigned long a, unsigned long b)
UMAX16 (SIMD 16-bit Unsigned Maximum)

Type: SIMD

Syntax:

```
UMAX16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] >u Rs2.H[x])? Rs1.H[x] : Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMIN16(unsigned long a, unsigned long b)

UMIN16 (SIMD 16-bit Unsigned Minimum)

Type: SIMD

Syntax:

```
UMIN16 Rd, Rs1, Rs2
```

Purpose: Do 16-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the 16-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.H[x] = (Rs1.H[x] <u Rs2.H[x])? Rs1.H[x] : Rs2.H[x];
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

SIMD 8-bit Miscellaneous Instructions

__STATIC_FORCEINLINE unsigned long __RV_CLRS8(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_CLO8(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_CLZ8(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_KABS8(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SMAX8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SMIN8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMAX8(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMIN8(unsigned long a, unsigned long b)

__RV_SCLIP8 (a, b)

__RV_UCLIP8 (a, b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_8B_MISC**

SIMD 8-bit Miscellaneous Instructions.

there are 10 SIMD 8-bit Miscellaneous instructions.

Defines

__RV_SCLIP8 (a, b)

SCLIP8 (SIMD 8-bit Signed Clip Value)

Type: SIMD

Syntax:

```
SCLIP8 Rd, Rs1, imm3u[2:0]
```

Purpose: Limit the 8-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 8-bit signed integer elements stored in Rs1 into a signed integer range between 2^imm3u-1 and -2^imm3u, and writes the limited results to Rd. For example, if imm3u is 3, the 8-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src > (2^imm3u)-1) {
  src = (2^imm3u)-1;
  OV = 1;
} else if (src < -2^imm3u) {
  src = -2^imm3u;
  OV = 1;
}
Rd.B[x] = src
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_UCLIP8 (a, b)

UCLIP8 (SIMD 8-bit Unsigned Clip Value)

Type: SIMD

Syntax:

```
UCLIP8 Rt, Ra, imm3u
```

Purpose: Limit the 8-bit signed elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 8-bit signed elements stored in Rs1 into an unsigned integer range between 2^imm3u-1 and 0, and writes the limited results to Rd. For example, if imm3u is 3, the 8- bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src > (2^imm3u)-1) {
  src = (2^imm3u)-1;
  OV = 1;
} else if (src < 0) {
  src = 0;
  OV = 1;
}
Rd.H[x] = src;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

`__STATIC_FORCEINLINE unsigned long __RV_CLRS8(unsigned long a)`

CLRS8 (SIMD 8-bit Count Leading Redundant Sign)

Type: SIMD

Syntax:

```
CLRS8 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 8-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 8-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
for (i = 6 to 0) {
  if (snum[x](i) == snum[x](7)) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.B[x] = cnt[x];
```

(continues on next page)

(continued from previous page)

```
for RV32: x=3...0
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_CLO8(unsigned long a)

CLO8 (SIMD 8-bit Count Leading One)

Type: SIMD

Syntax:

```
CLO8 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 8-bit elements of a general register.

Description: Starting from the most significant bits of the 8-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
  for (i = 7 to 0) {
  if (snum[x](i) == 1) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.B[x] = cnt[x];
for RV32: x=3...0
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_CLZ8(unsigned long a)

CLZ8 (SIMD 8-bit Count Leading Zero)

Type: SIMD

Syntax:

```
CLZ8 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 8-bit elements of a general register.

Description: Starting from the most significant bits of the 8-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 8-bit elements of Rd.

Operations:

```
snum[x] = Rs1.B[x];
cnt[x] = 0;
for (i = 7 to 0) {
  if (snum[x](i) == 0) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.B[x] = cnt[x];
for RV32: x=3...0
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_KABS8(unsigned long a)

KABS8 (SIMD 8-bit Saturating Absolute)

Type: SIMD

Syntax:

```
KABS8 Rd, Rs1
```

Purpose: Get the absolute value of 8-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 8-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x80, this instruction generates 0x7f as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src == 0x80) {
  src = 0x7f;
  OV = 1;
} else if (src[7] == 1)
  src = -src;
}
Rd.B[x] = src;
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_SMAX8(unsigned long a, unsigned long b)

SMAX8 (SIMD 8-bit Signed Maximum)

Type: SIMD

Syntax:

```
SMAX8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] > Rs2.B[x])? Rs1.B[x] : Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SMIN8(unsigned long a, unsigned long b)

SMIN8 (SIMD 8-bit Signed Minimum)

Type: SIMD

Syntax:

```
SMIN8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] < Rs2.B[x])? Rs1.B[x] : Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMAX8(unsigned long a, unsigned long b)

UMAX8 (SIMD 8-bit Unsigned Maximum)

Type: SIMD

Syntax:

```
UMAX8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the four 8- bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The two selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] >u Rs2.B[x])? Rs1.B[x] : Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMIN8(unsigned long a, unsigned long b)

UMIN8 (SIMD 8-bit Unsigned Minimum)

Type: SIMD

Syntax:

```
UMIN8 Rd, Rs1, Rs2
```

Purpose: Do 8-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the 8-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.B[x] = (Rs1.B[x] <u Rs2.B[x])? Rs1.B[x] : Rs2.B[x];
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

SIMD 8-bit Unpacking Instructions

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD810(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD820(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD830(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD831(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD832(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD810(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD820(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD830(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD831(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD832(unsigned long a)

group **NMSIS_Core_DSP_Intrinsic_SIMD_8B_UNPACK**

SIMD 8-bit Unpacking Instructions.

there are 8 SIMD 8-bit Unpacking instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD810(unsigned long a)

SUNPKD810 (Signed Unpacking Bytes 1 & 0)

Type: DSP

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte *x and byte y* of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description: For the `SUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD820(unsigned long a)

SUNPKD820 (Signed Unpacking Bytes 2 & 0)

Type: DSP

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte *x and byte y* of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description: For the `SUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_SUNPKD830(unsigned long a)`

SUNPKD830 (Signed Unpacking Bytes 3 & 0)

Type: DSP

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte *x and byte y* of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description: For the `SUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD831(unsigned long a)

SUNPKD831 (Signed Unpacking Bytes 3 & 1)

Type: DSP

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte *x and byte y* of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description: For the `SUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_SUNPKD832(unsigned long a)

SUNPKD832 (Signed Unpacking Bytes 3 & 2)

Type: DSP

Syntax:

```
SUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte *x and byte y* of 32-bit chunks in a register into two 16-bit signed halfwords of 32-bit chunks in a register.

Description: For the `SUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit signed halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = SE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = SE16(Rs1.W[m].B[y])
// SUNPKD810, x=1,y=0
// SUNPKD820, x=2,y=0
// SUNPKD830, x=3,y=0
// SUNPKD831, x=3,y=1
// SUNPKD832, x=3,y=2
```

(continues on next page)

(continued from previous page)

```
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD810(unsigned long a)

ZUNPKD810 (Unsigned Unpacking Bytes 1 & 0)

Type: DSP

Syntax:

```
ZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description: For the `ZUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD820(unsigned long a)

ZUNPKD820 (Unsigned Unpacking Bytes 2 & 0)

Type: DSP

Syntax:

```
ZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description: For the `ZUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD830(unsigned long a)

ZUNPKD830 (Unsigned Unpacking Bytes 3 & 0)

Type: DSP

Syntax:

```
ZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description: For the `ZUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD831(unsigned long a)

ZUNPKD831 (Unsigned Unpacking Bytes 3 & 1)

Type: DSP

Syntax:

```
ZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description: For the `ZUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_ZUNPKD832(unsigned long a)

ZUNPKD832 (Unsigned Unpacking Bytes 3 & 2)

Type: DSP

Syntax:

```
ZUNPKD8xy Rd, Rs1
xy = {10, 20, 30, 31, 32}
```

Purpose: Unpack byte x and byte y of 32-bit chunks in a register into two 16-bit unsigned halfwords of 32-bit chunks in a register.

Description: For the `ZUNPKD8(x)(*y*)` instruction, it unpacks byte *x and byte y* of 32-bit chunks in Rs1 into two 16-bit unsigned halfwords and writes the results to the top part and the bottom part of 32-bit chunks in Rd.

Operations:

```
Rd.W[m].H[1] = ZE16(Rs1.W[m].B[x])
Rd.W[m].H[0] = ZE16(Rs1.W[m].B[y])
// ZUNPKD810, x=1,y=0
// ZUNPKD820, x=2,y=0
// ZUNPKD830, x=3,y=0
// ZUNPKD831, x=3,y=1
// ZUNPKD832, x=3,y=2
for RV32: m=0,
for RV64: m=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

group **NMSIS_Core_DSP_Intrinsic_SIMD_DATA_PROCESS**
SIMD Data Processing Instructions.

Non-SIMD Instructions

Non-SIMD Q15 saturation ALU Instructions

__STATIC_FORCEINLINE long __RV_KADDH(int a, int b)

__STATIC_FORCEINLINE long __RV_KHMBB(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KHMBT(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KHMTT(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KSUBH(int a, int b)

__STATIC_FORCEINLINE unsigned long __RV_UKADDH(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_UKSUBH(unsigned int a, unsigned int b)

group **NMSIS_Core_DSP_Intrinsic_NON_SIMD_Q15_SAT_ALU**
Non-SIMD Q15 saturation ALU Instructions.

there are 7 Non-SIMD Q15 saturation ALU Instructions

Functions

__STATIC_FORCEINLINE long __RV_KADDH(int a, int b)
KADDH (Signed Addition with Q15 Saturation)

Type: DSP

Syntax:

```
KADDH Rd, Rs1, Rs2
```

Purpose: Add the signed lower 32-bit content of two registers with Q15 saturation.

Description: The signed lower 32-bit content of Rs1 is added with the signed lower 32-bit content of Rs2. And the result is saturated to the 16-bit signed integer range of [-2^15, 2^15-1] and then sign- extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > 32767) {
  res = 32767;
  OV = 1;
} else if (tmp < -32768) {
  res = -32768;
  OV = 1
} else {
  res = tmp;
}
Rd = SE(tmp[15:0]);
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE long __RV_KHMBB(unsigned int a, unsigned int b)

KHMBB (Signed Saturating Half Multiply B16 x B16)

Type: DSP

Syntax:

```
KHMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 number contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then right-shift 15 bits to turn the Q30 result into a Q15 number again and saturate the Q15 result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then right- shifted 15-bits and saturated into a Q15 value. The Q15 value is then sing-extended and written into Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KHMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KHMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KHMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
}
Rd = SE32(res[15:0]); // Rv32
Rd = SE64(res[15:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KHMBT(unsigned int a, unsigned int b)

KHMBT (Signed Saturating Half Multiply B16 x T16)

Type: DSP

Syntax:

```
KHMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 number contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then right-shift 15 bits to turn the Q30 result into a Q15 number again and saturate the Q15 result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then right- shifted 15-bits and saturated into a Q15 value. The Q15 value is then sing-extended and written into Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KHMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KHMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KHMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
}
Rd = SE32(res[15:0]); // Rv32
Rd = SE64(res[15:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KHMTT(unsigned int a, unsigned int b)

KHMTT (Signed Saturating Half Multiply T16 x T16)

Type: DSP

Syntax:

```
KHMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 number contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then right-shift 15 bits to turn the Q30 result into a Q15 number again and saturate the Q15 result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then right- shifted 15-bits and saturated into a Q15 value. The Q15 value is then sing-extended and written into Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KHMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KHMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KHMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
Rd = SE32(res[15:0]); // Rv32
Rd = SE64(res[15:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KSUBH(int a, int b)`

KSUBH (Signed Subtraction with Q15 Saturation)

Type: DSP

Syntax:

```
KSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the signed lower 32-bit content of two registers with Q15 saturation.

Description: The signed lower 32-bit content of Rs2 is subtracted from the signed lower 32-bit content of Rs1. And the result is saturated to the 16-bit signed integer range of [-2^15, 2^15-1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^15)-1) {
  res = (2^15)-1;
  OV = 1;
} else if (tmp < -2^15) {
  res = -2^15;
  OV = 1
} else {
  res = tmp;
}
Rd = SE(res[15:0]);
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKADDH(unsigned int a, unsigned int b)`

UKADDH (Unsigned Addition with U16 Saturation)

Type: DSP

Syntax:

```
UKADDH Rd, Rs1, Rs2
```

Purpose: Add the unsigned lower 32-bit content of two registers with U16 saturation.

Description: The unsigned lower 32-bit content of Rs1 is added with the unsigned lower 32-bit content of Rs2. And the result is saturated to the 16-bit unsigned integer range of [0, 2^16-1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^16)-1) {
  tmp = (2^16)-1;
  OV = 1;
}
Rd = SE(tmp[15:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKSUBH(unsigned int a, unsigned int b)

UKSUBH (Unsigned Subtraction with U16 Saturation)

Type: DSP

Syntax:

```
UKSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the unsigned lower 32-bit content of two registers with U16 saturation.

Description: The unsigned lower 32-bit content of Rs2 is subtracted from the unsigned lower 32-bit content of Rs1. And the result is saturated to the 16-bit unsigned integer range of [0, 2^16-1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^16)-1) {
  tmp = (2^16)-1;
  OV = 1;
}
else if (tmp < 0) {
  tmp = 0;
  OV = 1;
}
Rd = SE(tmp[15:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Non-SIMD Q31 saturation ALU Instructions

__STATIC_FORCEINLINE unsigned long __RV_KABSW(signed long a)

__STATIC_FORCEINLINE long __RV_KADDW(int a, int b)

__STATIC_FORCEINLINE long __RV_KDMBB(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KDMBT(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KDMTT(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KDMABB(long t, unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KDMABT(long t, unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KDMATT(long t, unsigned int a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KSLLW(long a, unsigned int b)

__STATIC_FORCEINLINE long __RV_KSLRAW(int a, int b)

__STATIC_FORCEINLINE long __RV_KSLRAW_U(int a, int b)

__STATIC_FORCEINLINE long __RV_KSUBW(int a, int b)

__STATIC_FORCEINLINE unsigned long __RV_UKADDW(unsigned int a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_UKSUBW(unsigned int a, unsigned int b)

__RV_KSLLIW (a, b)

group **NMSIS_Core_DSP_Intrinsic_NON_SIMD_Q31_SAT_ALU**

Non-SIMD Q31 saturation ALU Instructions.

there are Non-SIMD Q31 saturation ALU Instructions

Defines

__RV_KSLLIW (a, b)

KSLLIW (Saturating Shift Left Logical Immediate for Word)

Type: DSP

Syntax:

```
KSLLIW Rd, Rs1, imm5u
```

Purpose: Do logical left shift operation with saturation on a 32-bit word. The shift amount is an immediate value.

Description: The first word data in Rs1 is left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant. Any shifted value greater than 2^31-1 is saturated to 2^31-1. Any shifted value smaller than -2^31 is saturated to -2^31. And the saturated result is sign-extended and written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm5u;
res[(31+sa):0] = Rs1.W[0] << sa;
if (res > (2^31)-1) {
  res = 0x7fffffff; OV = 1;
} else if (res < -2^31) {
  res = 0x80000000; OV = 1;
}
Rd[31:0] = res[31:0]; // RV32
Rd[63:0] = SE(res[31:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

`__STATIC_FORCEINLINE unsigned long __RV_KABSW(signed long a)`

KABSW (Scalar 32-bit Absolute Value with Saturation)

Type: DSP

Syntax:

```
KABSW Rd, Rs1
```

Purpose: Get the absolute value of a signed 32-bit integer in a general register.

Description: This instruction calculates the absolute value of a signed 32-bit integer stored in Rs1. The result is sign-extended (for RV64) and written to Rd. This instruction with the minimum negative integer input of 0x80000000 will produce a saturated output of maximum positive integer of 0x7fffffff and the OV flag will be set to 1.

Operations:

```
if (Rs1.W[0] >= 0) {
  res = Rs1.W[0];
} else {
  If (Rs1.W[0] == 0x80000000) {
    res = 0x7fffffff;
    OV = 1;
  } else {
    res = -Rs1.W[0];
  }
}
Rd = SE32(res);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: signed long type of value stored in a

`__STATIC_FORCEINLINE long __RV_KADDW(int a, int b)`

KADDW (Signed Addition with Q31 Saturation)

Type: DSP

Syntax:

```
KADDW Rd, Rs1, Rs2
```

Purpose: Add the lower 32-bit signed content of two registers with Q31 saturation.

Description: The lower 32-bit signed content of Rs1 is added with the lower 32-bit signed content of Rs2. And the result is saturated to the 32-bit signed integer range of [-2^31, 2^31-1] and then sign- extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^31)-1) {
  res = (2^31)-1;
  OV = 1;
} else if (tmp < -2^31) {
  res = -2^31;
  OV = 1
} else {
  res = tmp;
}
Rd = res[31:0]; // RV32
Rd = SE(res[31:0]) // RV64
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KDMBB(unsigned int a, unsigned int b)`

KDMBB (Signed Saturating Double Multiply B16 x B16)

Type: DSP

Syntax:

```
KDMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result. The result is written into the destination register for RV32 or sign-extended to 64-bits and written into the destination register for RV64. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then written into Rd (sign-extended in RV64). When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
} else {
  resQ31 = 0x7FFFFFFF;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
  OV = 1;
}
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KDMBT(unsigned int a, unsigned int b)

KDMBT (Signed Saturating Double Multiply B16 x T16)

Type: DSP

Syntax:

```
KDMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result. The result is written into the destination register for RV32 or sign-extended to 64-bits and written into the destination register for RV64. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then written into Rd (sign-extended in RV64). When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
} else {
  resQ31 = 0x7FFFFFFF;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
  OV = 1;
}
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KDMTT(unsigned int a, unsigned int b)

KDMTT (Signed Saturating Double Multiply T16 x T16)

Type: DSP

Syntax:

```
KDMxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result. The result is written into the destination register for RV32 or sign-extended to 64-bits and written into the destination register for RV64. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then written into Rd (sign-extended in RV64). When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMTT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
} else {
  resQ31 = 0x7FFFFFFF;
  Rd = resQ31; // RV32
  Rd = SE(resQ31); // RV64
  OV = 1;
}
```

Return value stored in long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KDMABB(long t, unsigned int a, unsigned int b)`

KDMABB (Signed Saturating Double Multiply Addition B16 x B16)

Type: DSP

Syntax:

```
KDMAxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result, add the result with the sign-extended lower 32-bit chunk destination register and write the saturated addition result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then added with the content of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV flag is set to 1. The result after saturation is written to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMABB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMABT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMATT
```

(continues on next page)

(continued from previous page)

```
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
} else {
  resQ31 = 0x7FFFFFFF;
  OV = 1;
}
resadd = Rd + resQ31; // RV32
resadd = Rd.W[0] + resQ31; // RV64
if (resadd > (2^31)-1) {
  resadd = (2^31)-1;
  OV = 1;
} else if (resadd < -2^31) {
  resadd = -2^31;
  OV = 1;
}
Rd = resadd; // RV32
Rd = SE(resadd); // RV64
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KDMABT(long t, unsigned int a, unsigned int b)`

KDMABT (Signed Saturating Double Multiply Addition B16 x T16)

Type: DSP

Syntax:

```
KDMAxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result, add the result with the sign-extended lower 32-bit chunk destination register and write the saturated addition result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then added with the content of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV flag is set to 1. The result after saturation is written to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMABB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMABT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMATT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
```

(continues on next page)

(continued from previous page)

```
} else {
  resQ31 = 0x7FFFFFFF;
  OV = 1;
}
resadd = Rd + resQ31; // RV32
resadd = Rd.W[0] + resQ31; // RV64
if (resadd > (2^31)-1) {
  resadd = (2^31)-1;
  OV = 1;
} else if (resadd < -2^31) {
  resadd = -2^31;
  OV = 1;
}
Rd = resadd; // RV32
Rd = SE(resadd); // RV64
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KDMATT(long t, unsigned int a, unsigned int b)

KDMATT (Signed Saturating Double Multiply Addition T16 x T16)

Type: DSP

Syntax:

```
KDMAxy Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the lower 32-bit chunk in registers and then double and saturate the Q31 result, add the result with the sign-extended lower 32-bit chunk destination register and write the saturated addition result into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs1 with the top or bottom 16-bit Q15 content of the lower 32-bit portion in Rs2. The Q30 result is then doubled and saturated into a Q31 value. The Q31 value is then added with the content of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV flag is set to 1. The result after saturation is written to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
aop = Rs1.H[0]; bop = Rs2.H[0]; // KDMABB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KDMABT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KDMATT
If (0x8000 != aop | 0x8000 != bop) {
  Mresult = aop * bop;
  resQ31 = Mresult << 1;
} else {
  resQ31 = 0x7FFFFFFF;
  OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
resadd = Rd + resQ31; // RV32
resadd = Rd.W[0] + resQ31; // RV64
if (resadd > (2^31)-1) {
  resadd = (2^31)-1;
  OV = 1;
} else if (resadd < -2^31) {
  resadd = -2^31;
  OV = 1;
}
Rd = resadd; // RV32
Rd = SE(resadd); // RV64
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KSLLW(long a, unsigned int b)`

KSLLW (Saturating Shift Left Logical for Word)

Type: DSP

Syntax:

```
KSLLW Rd, Rs1, Rs2
```

Purpose: Do logical left shift operation with saturation on a 32-bit word. The shift amount is a variable from a GPR.

Description: The first word data in Rs1 is left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register. Any shifted value greater than 2^31-1 is saturated to 2^31-1. Any shifted value smaller than -2^31 is saturated to -2^31. And the saturated result is sign-extended and written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[4:0];
res[(31+sa):0] = Rs1.W[0] << sa;
if (res > (2^31)-1) {
  res = 0x7fffffff; OV = 1;
} else if (res < -2^31) {
  res = 0x80000000; OV = 1;
}
Rd[31:0] = res[31:0]; // RV32
Rd[63:0] = SE(res[31:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE long __RV_KSLRAW(int a, int b)

KSLRAW (Shift Left Logical with Q31 Saturation or Shift Right Arithmetic)

Type: DSP

Syntax:

```
KSLRAW Rd, Rs1, Rs2
```

Purpose: Perform a logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift on a 32-bit data.

Description: The lower 32-bit content of Rs1 is left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of [-25, 25-1]. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0] clamped to the actual shift range of [0, 31]. The left-shifted result is saturated to the 32-bit signed integer range of [-2^31, 2^31-1]. After the shift operation, the final result is bit-31 sign-extended and written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affected the operation of this instruction.

Operations:

```
if (Rs2[5:0] < 0) {
  sa = -Rs2[5:0];
  sa = (sa == 32)? 31 : sa;
  res[31:0] = Rs1.W[0] >>(arith) sa;
} else {
  sa = Rs2[5:0];
  tmp = Rs1.W[0] <<(logic) sa;
  if (tmp > (2^31)-1) {
    res[31:0] = (2^31)-1;
    OV = 1;
  } else if (tmp < -2^31) {
    res[31:0] = -2^31;
    OV = 1
  } else {
    res[31:0] = tmp[31:0];
  }
}
Rd = res[31:0]; // RV32
Rd = SE64(res[31:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE long __RV_KSLRAW_U(int a, int b)

KSLRAW.u (Shift Left Logical with Q31 Saturation or Rounding Shift Right Arithmetic)

Type: DSP

Syntax:

```
KSLRAW.u Rd, Rs1, Rs2
```

Purpose: Perform a logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift and a rounding up operation for the right shift on a 32-bit data.

Description: The lower 32-bit content of Rs1 is left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of [-25, 25-1]. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0] clamped to the actual shift range of [0, 31]. The left-shifted result is saturated to the 32-bit signed integer range of [-2^31, 2^31-1]. The right-shifted result is added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final result is bit-31 sign-extended and written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect the operation of this instruction.

Operations:

```
if (Rs2[5:0] < 0) {
  sa = -Rs2[5:0];
  sa = (sa == 32)? 31 : sa;
  res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
  rst[31:0] = res[31:0];
} else {
  sa = Rs2[5:0];
  tmp = Rs1.W[0] <<(logic) sa;
  if (tmp > (2^31)-1) {
    rst[31:0] = (2^31)-1;
    OV = 1;
  } else if (tmp < -2^31) {
    rst[31:0] = -2^31;
    OV = 1
  } else {
    rst[31:0] = tmp[31:0];
  }
}
Rd = rst[31:0]; // RV32
Rd = SE64(rst[31:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE long __RV_KSUBW(int a, int b)`

KSUBW (Signed Subtraction with Q31 Saturation)

Type: DSP

Syntax:

```
KSUBW Rd, Rs1, Rs2
```

Purpose: Subtract the signed lower 32-bit content of two registers with Q31 saturation.

Description: The signed lower 32-bit content of Rs2 is subtracted from the signed lower 32-bit content of Rs1. And the result is saturated to the 32-bit signed integer range of [-2^31, 2^31-1] and then sign-extened and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp > (2^31)-1) {
  res = (2^31)-1;
```

(continues on next page)

(continued from previous page)

```
  OV = 1;
} else if (tmp < -2^31) {
res = -2^31;
  OV = 1
} else {
  res = tmp;
}
Rd = res[31:0]; // RV32
Rd = SE(res[31:0]); // RV64
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKADDW(unsigned int a, unsigned int b)`

UKADDW (Unsigned Addition with U32 Saturation)

Type: DSP

Syntax:

```
UKADDW Rd, Rs1, Rs2
```

Purpose: Add the unsigned lower 32-bit content of two registers with U32 saturation.

Description: The unsigned lower 32-bit content of Rs1 is added with the unsigned lower 32-bit content of Rs2. And the result is saturated to the 32-bit unsigned integer range of [0, 2^32-1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] + Rs2.W[0];
if (tmp > (2^32)-1) {
  tmp[31:0] = (2^32)-1;
  OV = 1;
}
Rd = tmp[31:0]; // RV32
Rd = SE(tmp[31:0]); // RV64
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKSUBW(unsigned int a, unsigned int b)`

UKSUBW (Unsigned Subtraction with U32 Saturation)

Type: DSP

Syntax:

```
UKSUBW Rd, Rs1, Rs2
```

Purpose: Subtract the unsigned lower 32-bit content of two registers with unsigned 32-bit saturation.

Description: The unsigned lower 32-bit content of Rs2 is subtracted from the unsigned lower 32-bit content of Rs1. And the result is saturated to the 32-bit unsigned integer range of [0, 2^32-1] and then sign-extended and written to Rd. If saturation happens, this instruction sets the OV flag.

Operations:

```
tmp = Rs1.W[0] - Rs2.W[0];
if (tmp < 0) {
  tmp[31:0] = 0;
  OV = 1;
}
Rd = tmp[31:0]; // RV32
Rd = SE(tmp[31:0]); // RV64
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

32-bit Computation Instructions

`__STATIC_FORCEINLINE long __RV_MAXW(int a, int b)`

`__STATIC_FORCEINLINE long __RV_MINW(int a, int b)`

`__STATIC_FORCEINLINE unsigned long long __RV_MULR64(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE long long __RV_MULSR64(long a, long b)`

`__STATIC_FORCEINLINE long __RV_RADDW(int a, int b)`

`__STATIC_FORCEINLINE long __RV_RSUBW(int a, int b)`

`__STATIC_FORCEINLINE unsigned long __RV_URADDW(unsigned int a, unsigned int b)`

`__STATIC_FORCEINLINE unsigned long __RV_URSUBW(unsigned int a, unsigned int b)`

group **`NMSIS_Core_DSP_Intrinsic_32B_COMPUTATION`**

32-bit Computation Instructions

there are 8 32-bit Computation Instructions

Functions

`__STATIC_FORCEINLINE long __RV_MAXW(int a, int b)`

MAXW (32-bit Signed Word Maximum)

Type: DSP

Syntax:

```
MAXW Rd, Rs1, Rs2
```

Purpose: Get the larger value from the 32-bit contents of two general registers.

Description: This instruction compares two signed 32-bit integers stored in Rs1 and Rs2, picks the larger value as the result, and writes the result to Rd.

Operations:

```
if (Rs1.W[0] >= Rs2.W[0]) {
  Rd = SE(Rs1.W[0]);
} else {
  Rd = SE(Rs2.W[0]);
}
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE long __RV_MINW(int a, int b)`

MINW (32-bit Signed Word Minimum)

Type: DSP

Syntax:

```
MINW Rd, Rs1, Rs2
```

Purpose: Get the smaller value from the 32-bit contents of two general registers.

Description: This instruction compares two signed 32-bit integers stored in Rs1 and Rs2, picks the smaller value as the result, and writes the result to Rd.

Operations:

```
if (Rs1.W[0] >= Rs2.W[0]) { Rd = SE(Rs2.W[0]); } else { Rd = SE(Rs1.W[0]); }
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_MULR64(unsigned long a, unsigned long b)`

MULR64 (Multiply Word Unsigned to 64-bit Data)

Type: DSP

Syntax:

```
MULR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned integer contents of two registers and write the 64-bit result.

RV32 Description: This instruction multiplies the 32-bit content of Rs1 with that of Rs2 and writes the 64-bit multiplication result to an even/odd pair of registers containing Rd. Rd(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result. The lower 32-bit contents of Rs1 and Rs2 are treated as unsigned integers.

RV64 Description: This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2 and writes the 64-bit multiplication result to Rd. The lower 32-bit contents of Rs1 and Rs2 are treated as unsigned integers.

Operations:

```
RV32:
Mresult = CONCAT(1`b0,Rs1) u* CONCAT(1`b0,Rs2);
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
RV64:
Rd = Mresult[63:0];
Mresult = CONCAT(1`b0,Rs1.W[0]) u* CONCAT(1`b0,Rs2.W[0]);
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long long __RV_MULSR64(long a, long b)`

MULSR64 (Multiply Word Signed to 64-bit Data)

Type: DSP

Syntax:

```
MULSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed integer contents of two registers and write the 64-bit result.

RV32 Description: This instruction multiplies the lower 32-bit content of Rs1 with the lower 32-bit content of Rs2 and writes the 64-bit multiplication result to an even/odd pair of registers containing Rd. Rd(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result. The lower 32-bit contents of Rs1 and Rs2 are treated as signed integers.

RV64 Description: This instruction multiplies the lower 32-bit content of Rs1 with the lower 32-bit content of Rs2 and writes the 64-bit multiplication result to Rd. The lower 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
RV32:
Mresult = Ra s* Rb;
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
RV64:
Mresult = Ra.W[0] s* Rb.W[0];
Rd = Mresult[63:0];
```

Return value stored in long long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE long __RV_RADDW(int a, int b)

RADDW (32-bit Signed Halving Addition)

Type: DSP

Syntax:

```
RADDW Rd, Rs1, Rs2
```

Purpose: Add 32-bit signed integers and the results are halved to avoid overflow or saturation.

Description: This instruction adds the first 32-bit signed integer in Rs1 with the first 32-bit signed integer in Rs2. The result is first arithmetically right-shifted by 1 bit and then sign-extended and written to Rd.

Examples:

```
* Rs1 = 0x7FFFFFFF, Rs2 = 0x7FFFFFFF, Rd = 0x7FFFFFFF
* Rs1 = 0x80000000, Rs2 = 0x80000000, Rd = 0x80000000
* Rs1 = 0x40000000, Rs2 = 0x80000000, Rd = 0xE0000000
```

Operations:

```
RV32:
Rd[31:0] = (Rs1[31:0] + Rs2[31:0]) s>> 1;
RV64:
resw[31:0] = (Rs1[31:0] + Rs2[31:0]) s>> 1;
Rd[63:0] = SE(resw[31:0]);
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE long __RV_RSUBW(int a, int b)

RSUBW (32-bit Signed Halving Subtraction)

Type: DSP

Syntax:

```
RSUBW Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit signed integers and the result is halved to avoid overflow or saturation.

Description: This instruction subtracts the first 32-bit signed integer in Rs2 from the first 32-bit signed integer in Rs1. The result is first arithmetically right-shifted by 1 bit and then sign-extended and written to Rd.

Examples:

```
* Rs1 = 0x7FFFFFFF, Rs2 = 0x80000000, Rd = 0x7FFFFFFF
* Rs1 = 0x80000000, Rs2 = 0x7FFFFFFF, Rd = 0x80000000
* Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0xA0000000
```

Operations:

```
RV32:
Rd[31:0] = (Rs1[31:0] - Rs2[31:0]) s>> 1;
RV64:
resw[31:0] = (Rs1[31:0] - Rs2[31:0]) s>> 1;
Rd[63:0] = SE(resw[31:0]);
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URADDW(unsigned int a, unsigned int b)

URADDW (32-bit Unsigned Halving Addition)

Type: DSP

Syntax:

```
URADDW Rd, Rs1, Rs2
```

Purpose: Add 32-bit unsigned integers and the results are halved to avoid overflow or saturation.

Description: This instruction adds the first 32-bit unsigned integer in Rs1 with the first 32-bit unsigned integer in Rs2. The result is first logically right-shifted by 1 bit and then sign-extended and written to Rd.

Examples:

```
* Ra = 0x7FFFFFFF, Rb = 0x7FFFFFFF Rt = 0x7FFFFFFF
* Ra = 0x80000000, Rb = 0x80000000 Rt = 0x80000000
* Ra = 0x40000000, Rb = 0x80000000 Rt = 0x60000000
```

Operations:

```
* RV32:
Rd[31:0] = (Rs1[31:0] + Rs2[31:0]) u>> 1;
* RV64:
resw[31:0] = (Rs1[31:0] + Rs2[31:0]) u>> 1;
Rd[63:0] = SE(resw[31:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSUBW(unsigned int a, unsigned int b)

URSUBW (32-bit Unsigned Halving Subtraction)

Type: DSP

Syntax:

```
URSUBW Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit unsigned integers and the result is halved to avoid overflow or saturation.

Description: This instruction subtracts the first 32-bit signed integer in Rs2 from the first 32-bit signed integer in Rs1. The result is first logically right-shifted by 1 bit and then sign-extended and written to Rd.

Examples:

```
* Ra = 0x7FFFFFFF, Rb = 0x80000000 Rt = 0xFFFFFFFF
* Ra = 0x80000000, Rb = 0x7FFFFFFF Rt = 0x00000000
* Ra = 0x80000000, Rb = 0x40000000 Rt = 0x20000000
```

Operations:

```
* RV32:
Rd[31:0] = (Rs1[31:0] - Rs2[31:0]) u>> 1;
* RV64:
resw[31:0] = (Rs1[31:0] - Rs2[31:0]) u>> 1;
Rd[63:0] = SE(resw[31:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

OV (Overflow) flag Set/Clear Instructions

__STATIC_FORCEINLINE void __RV_CLROV(void)

__STATIC_FORCEINLINE unsigned long __RV_RDOV(void)

group **NMSIS_Core_DSP_Intrinsic_OV_FLAG_SC**

OV (Overflow) flag Set/Clear Instructions.

The following table lists the user instructions related to Overflow (OV) flag manipulation. there are 2 OV (Overflow) flag Set/Clear Instructions

Functions

__STATIC_FORCEINLINE void __RV_CLROV(void)

CLROV (Clear OV flag)

Type: DSP

Syntax:

```
CLROV # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias to `CSRRCI x0, ucode, 1` instruction.

__STATIC_FORCEINLINE unsigned long __RV_RDOV(void)

RDOV (Read OV flag)

Type: DSP

Syntax:

```
RDOV Rd  # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias to `CSRR Rd, ucode` instruction which maps to the real instruction of `CSRRS Rd, ucode, x0`.

Return value stored in unsigned long type

Non-SIMD Miscellaneous Instructions

__STATIC_FORCEINLINE long __RV_AVE(long a, long b)

__STATIC_FORCEINLINE unsigned long __RV_BITREV(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_BPICK(unsigned long a, unsigned long b, unsigned lo

__STATIC_FORCEINLINE unsigned long __RV_MADDR32(unsigned long t, unsigned long a, unsigned

__STATIC_FORCEINLINE unsigned long __RV_MSUBR32(unsigned long t, unsigned long a, unsigned

__STATIC_FORCEINLINE long __RV_SRA_U(long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SWAP8(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SWAP16(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_WEXT(long long a, unsigned int b)

__RV_BITREVI (a, b)

__RV_INSB (t, a, b)

__RV_SRAI_U (a, b)

__RV_WEXTI (a, b)

group **NMSIS_Core_DSP_Intrinsic_NON_SIMD_MISC**

Non-SIMD Miscellaneous Instructions.

There are 13 Miscellaneous Instructions here.

Defines

__RV_BITREVI (a, b)

BITREVI (Bit Reverse Immediate)

Type: DSP

Syntax:

```
(RV32) BITREVI Rd, Rs1, imm[4:0]
(RV64) BITREVI Rd, Rs1, imm[5:0]
```

Purpose: Reverse the bit positions of the source operand within a specified width starting from bit 0. The reversed width is an immediate value.

Description: This instruction reverses the bit positions of the content of Rs1. The reversed bit width is calculated as imm[4:0]+1 (RV32) or imm[5:0]+1 (RV64). The upper bits beyond the reversed width are filled with zeros. After the bit reverse operation, the result is written to Rd.

Operations:

```
msb = imm[4:0]; (RV32)
msb = imm[5:0]; (RV64)
rev[0:msb] = Rs1[msb:0];
Rd = ZE(rev[msb:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__RV_INSB (t, a, b)

INSB (Insert Byte)

Type: DSP

Syntax:

```
(RV32) INSB Rd, Rs1, imm[1:0]
(RV64) INSB Rd, Rs1, imm[2:0]
```

Purpose: Insert byte 0 of a 32-bit or 64-bit register into one of the byte elements of another register.

Description: This instruction inserts byte 0 of Rs1 into byte `imm[1:0]` (RV32) or `imm[2:0]` (RV64) of Rd.

Operations:

```
bpos = imm[1:0]; (RV32)
bpos = imm[2:0]; (RV64)
Rd.B[bpos] = Rs1.B[0]
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__RV_SRAI_U (a, b)

SRAI.u (Rounding Shift Right Arithmetic Immediate)

Type: DSP

Syntax:

```
SRAI.u Rd, Rs1, imm6u[4:0] (RV32)
SRAI.u Rd, Rs1, imm6u[5:0] (RV64)
```

Purpose: Perform an arithmetic right shift operation with rounding. The shift amount is an immediate value.

Description: This instruction right-shifts the content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit and the shift amount is specified by the imm6u[4:0] (RV32) or imm6u[5:0] (RV64) constant . For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is written to Rd.

Operations:

```
* RV32:
sa = imm6u[4:0];
if (sa > 0) {
  res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
  Rd = res[31:0];
} else {
  Rd = Rs1;
}
* RV64:
sa = imm6u[5:0];
if (sa > 0) {
  res[63:-1] = SE65(Rs1[63:(sa-1)]) + 1;
  Rd = res[63:0];
} else {
  Rd = Rs1;
}
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_WEXTI (a, b)

WEXTI (Extract Word from 64-bit Immediate)

Type: DSP

Syntax:

```
WEXTI Rd, Rs1, #LSBloc
```

Purpose: Extract a 32-bit word from a 64-bit value stored in an even/odd pair of registers (RV32) or a register (RV64) starting from a specified immediate LSB bit position.

RV32 Description: This instruction extracts a 32-bit word from a 64-bit value of an even/odd pair of registers specified by Rs1(4,1) starting from a specified immediate LSB bit position, #LSBloc. The extracted word is written to Rd. Rs1(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the 64-bit value and the even `2d` register of the pair contains the low 32-bit of the 64-bit value.

RV64 Description: This instruction extracts a 32-bit word from a 64-bit value in Rs1 starting from a specified immediate LSB bit position, #LSBloc. The extracted word is sign-extended and written to lower 32- bit of Rd.

Operations:

```
* RV32:
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs2(4,1),1'b1);
src[63:0] = Concat(R[Idx1], R[Idx0]);
Rd = src[31+LSBloc:LSBloc];
* RV64:
ExtractW = Rs1[31+LSBloc:LSBloc];
Rd = SE(ExtractW)
```

Return value stored in unsigned long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

`__STATIC_FORCEINLINE long __RV_AVE(long a, long b)`

AVE (Average with Rounding)

Type: DSP

Syntax:

```
AVE Rd, Rs1, Rs2
```

Purpose: Calculate the average of the contents of two general registers.

Description: This instruction calculates the average value of two signed integers stored in Rs1 and Rs2, rounds up a half-integer result to the nearest integer, and writes the result to Rd.

Operations:

```
Sum = CONCAT(Rs1[MSB],Rs1[MSB:0]) + CONCAT(Rs2[MSB],Rs2[MSB:0]) + 1;
Rd = Sum[(MSB+1):1];
for RV32: MSB=31,
for RV64: MSB=63
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_BITREV(unsigned long a, unsigned long b)`

BITREV (Bit Reverse)

Type: DSP

Syntax:

```
BITREV Rd, Rs1, Rs2
```

Purpose: Reverse the bit positions of the source operand within a specified width starting from bit 0. The reversed width is a variable from a GPR.

Description: This instruction reverses the bit positions of the content of Rs1. The reversed bit width is calculated as Rs2[4:0]+1 (RV32) or Rs2[5:0]+1 (RV64). The upper bits beyond the reversed width are filled with zeros. After the bit reverse operation, the result is written to Rd.

Operations:

```
msb = Rs2[4:0]; (for RV32)
msb = Rs2[5:0]; (for RV64)
rev[0:msb] = Rs1[msb:0];
Rd = ZE(rev[msb:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_BPICK(unsigned long a, unsigned long b, unsign

BPICK (Bit-wise Pick)

Type: DSP

Syntax:

```
BPICK Rd, Rs1, Rs2, Rc
```

Purpose: Select from two source operands based on a bit mask in the third operand.

Description: This instruction selects individual bits from Rs1 or Rs2, based on the bit mask value in Rc. If a bit in Rc is 1, the corresponding bit is from Rs1; otherwise, the corresponding bit is from Rs2. The selection results are written to Rd.

Operations:

```
Rd[x] = Rc[x]? Rs1[x] : Rs2[x];
for RV32, x=31...0
for RV64, x=63...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b
- `[in] c`: unsigned long type of value stored in c

__STATIC_FORCEINLINE unsigned long __RV_MADDR32(unsigned long t, unsigned long a, unsi

MADDR32 (Multiply and Add to 32-Bit Word)

Type: DSP

Syntax:

```
MADDR32 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit contents of two registers and add the lower 32-bit multiplication result to the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2. It adds the lower 32-bit multiplication result to the lower 32-bit content of Rd and writes the final result (RV32) or sign-extended result (RV64) back to Rd. The contents of Rs1 and Rs2 can be either signed or unsigned integers.

Operations:

```
RV32:
Mresult = Rs1 * Rs2;
Rd = Rd + Mresult.W[0];
RV64:
Mresult = Rs1.W[0] * Rs2.W[0];
tres[31:0] = Rd.W[0] + Mresult.W[0];
Rd = SE64(tres[31:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_MSUBR32(unsigned long t, unsigned long a, unsi

MSUBR32 (Multiply and Subtract from 32-Bit Word)

Type: DSP

Syntax:

```
MSUBR32 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit contents of two registers and subtract the lower 32-bit multiplication result from the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the lower 32-bit content of Rs1 with that of Rs2, subtracts the lower 32-bit multiplication result from the lower 32-bit content of Rd, then writes the final result (RV32) or sign-extended result (RV64) back to Rd. The contents of Rs1 and Rs2 can be either signed or unsigned integers.

Operations:

```
RV32:
Mresult = Rs1 * Rs2;
Rd = Rd - Mresult.W[0];
RV64:
Mresult = Rs1.W[0] * Rs2.W[0];
tres[31:0] = Rd.W[0] - Mresult.W[0];
Rd = SE64(tres[31:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SRA_U(long a, unsigned int b)

SRA.u (Rounding Shift Right Arithmetic)

Type: DSP

Syntax:

```
SRA.u Rd, Rs1, Rs2
```

Purpose: Perform an arithmetic right shift operation with rounding. The shift amount is a variable from a GPR.

Description: This instruction right-shifts the content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit and the shift amount is specified by the low-order 5-bits (RV32) or 6-bits (RV64) of the Rs2 register. For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is written to Rd.

Operations:

```
* RV32:
sa = Rs2[4:0];
if (sa > 0) {
  res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
  Rd = res[31:0];
} else {
  Rd = Rs1;
}
* RV64:
sa = Rs2[5:0];
if (sa > 0) {
  res[63:-1] = SE65(Rs1[63:(sa-1)]) + 1;
  Rd = res[63:0];
} else {
  Rd = Rs1;
}
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SWAP8(unsigned long a)`

SWAP8 (Swap Byte within Halfword)

Type: DSP

Syntax:

```
SWAP8 Rd, Rs1
```

Purpose: Swap the bytes within each halfword of a register.

Description: This instruction swaps the bytes within each halfword of Rs1 and writes the result to Rd.

Operations:

```
Rd.H[x] = CONCAT(Rs1.H[x][7:0],Rs1.H[x][15:8]);
for RV32: x=1...0,
for RV64: x=3...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_SWAP16(unsigned long a)`

SWAP16 (Swap Halfword within Word)

Type: DSP

Syntax:

```
SWAP16 Rd, Rs1
```

Purpose: Swap the 16-bit halfwords within each word of a register.

Description: This instruction swaps the 16-bit halfwords within each word of Rs1 and writes the result to Rd.

Operations:

```
Rd.W[x] = CONCAT(Rs1.W[x][15:0],Rs1.H[x][31:16]);
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_WEXT(long long a, unsigned int b)`

WEXT (Extract Word from 64-bit)

Type: DSP

Syntax:

```
WEXT Rd, Rs1, Rs2
```

Purpose: Extract a 32-bit word from a 64-bit value stored in an even/odd pair of registers (RV32) or a register (RV64) starting from a specified LSB bit position in a register.

RV32 Description: This instruction extracts a 32-bit word from a 64-bit value of an even/odd pair of registers specified by Rs1(4,1) starting from a specified LSB bit position, specified in Rs2[4:0]. The extracted word is written to Rd. Rs1(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the 64-bit value and the even `2d` register of the pair contains the low 32-bit of the 64-bit value.

Operations:

```
* RV32:
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs1(4,1),1'b1);
src[63:0] = Concat(R[Idx1], R[Idx0]);
LSBloc = Rs2[4:0];
Rd = src[31+LSBloc:LSBloc];
* RV64:
LSBloc = Rs2[4:0];
ExtractW = Rs1[31+LSBloc:LSBloc];
Rd = SE(ExtractW)
```

Return value stored in unsigned long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

group **`NMSIS_Core_DSP_Intrinsic_NON_SIMD`**

Non-SIMD Instructions.

Partial-SIMD Data Processing Instructions

SIMD 16-bit Packing Instructions

__STATIC_FORCEINLINE unsigned long __RV_PKBB16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKBT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKTT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKTB16(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIMD_16B_PACK**

SIMD 16-bit Packing Instructions.

there are 4 SIMD16-bit Packing Instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_PKBB16(unsigned long a, unsigned long b)

PKBB16 (Pack Two 16-bit Data from Both Bottom Half)

Type: DSP

Syntax:

```
PKBB16 Rd, Rs1, Rs2
PKBT16 Rd, Rs1, Rs2
PKTT16 Rd, Rs1, Rs2
PKTB16 Rd, Rs1, Rs2
```

Purpose: Pack 16-bit data from 32-bit chunks in two registers.

- PKBB16: bottom.bottom
- PKBT16 bottom.top
- PKTT16 top.top
- PKTB16 top.bottom

Description: (PKBB16) moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0]. (PKBT16) moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTT16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTB16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]); // PKBB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]); // PKBT16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]); // PKTB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]); // PKTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_PKBT16(unsigned long a, unsigned long b)`

PKBT16 (Pack Two 16-bit Data from Bottom and Top Half)

Type: DSP

Syntax:

```
PKBB16 Rd, Rs1, Rs2
PKBT16 Rd, Rs1, Rs2
PKTT16 Rd, Rs1, Rs2
PKTB16 Rd, Rs1, Rs2
```

Purpose: Pack 16-bit data from 32-bit chunks in two registers.

- PKBB16: bottom.bottom
- PKBT16 bottom.top
- PKTT16 top.top
- PKTB16 top.bottom

Description: (PKBB16) moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0]. (PKBT16) moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTT16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTB16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]); // PKBB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]); // PKBT16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]); // PKTB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]); // PKTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_PKTT16(unsigned long a, unsigned long b)`

PKTT16 (Pack Two 16-bit Data from Both Top Half)

Type: DSP

Syntax:

```
PKBB16 Rd, Rs1, Rs2
PKBT16 Rd, Rs1, Rs2
PKTT16 Rd, Rs1, Rs2
PKTB16 Rd, Rs1, Rs2
```

Purpose: Pack 16-bit data from 32-bit chunks in two registers.

- PKBB16: bottom.bottom
- PKBT16 bottom.top

- PKTT16 top.top
- PKTB16 top.bottom

Description: (PKBB16) moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0]. (PKBT16) moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTT16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTB16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]); // PKBB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]); // PKBT16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]); // PKTB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]); // PKTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_PKTB16(unsigned long a, unsigned long b)`

PKTB16 (Pack Two 16-bit Data from Top and Bottom Half)

Type: DSP

Syntax:

```
PKBB16 Rd, Rs1, Rs2
PKBT16 Rd, Rs1, Rs2
PKTT16 Rd, Rs1, Rs2
PKTB16 Rd, Rs1, Rs2
```

Purpose: Pack 16-bit data from 32-bit chunks in two registers.

- PKBB16: bottom.bottom
- PKBT16 bottom.top
- PKTT16 top.top
- PKTB16 top.bottom

Description: (PKBB16) moves Rs1.W[x][15:0] to Rd.W[x][31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0]. (PKBT16) moves Rs1.W[x] [15:0] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTT16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [31:16] to Rd.W[x] [15:0]. (PKTB16) moves Rs1.W[x] [31:16] to Rd.W[x] [31:16] and moves Rs2.W[x] [15:0] to Rd.W[x] [15:0].

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][15:0]); // PKBB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][15:0], Rs2.W[x][31:16]); // PKBT16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][15:0]); // PKTB16
Rd.W[x][31:0] = CONCAT(Rs1.W[x][31:16], Rs2.W[x][31:16]); // PKTT16
```

(continues on next page)

(continued from previous page)

```
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a
- [in] b: unsigned long type of value stored in b

Signed MSW 32x32 Multiply and Add Instructions

__STATIC_FORCEINLINE long __RV_KMMAC(long t, long a, long b)

__STATIC_FORCEINLINE long __RV_KMMAC_U(long t, long a, long b)

__STATIC_FORCEINLINE long __RV_KMMSB(long t, long a, long b)

__STATIC_FORCEINLINE long __RV_KMMSB_U(long t, long a, long b)

__STATIC_FORCEINLINE long __RV_KWMMUL(long a, long b)

__STATIC_FORCEINLINE long __RV_KWMMUL_U(long a, long b)

__STATIC_FORCEINLINE long __RV_SMMUL(long a, long b)

__STATIC_FORCEINLINE long __RV_SMMUL_U(long a, long b)

group **NMSIS_Core_DSP_Intrinsic_SIGNED_MSW_32X32_MAC**

Signed MSW 32x32 Multiply and Add Instructions.

there are 8 Signed MSW 32x32 Multiply and Add Instructions

Functions

__STATIC_FORCEINLINE long __RV_KMMAC(long t, long a, long b)

KMMAC (SIMD Saturating MSW Signed Multiply Word and Add)

Type: SIMD

Syntax:

```
KMMAC Rd, Rs1, Rs2
KMMAC.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and add the most significant 32-bit results with the signed 32-bit integer elements of a third register. The addition results are saturated first and then written back to the third register. The `.u` form performs an additional rounding up operation on the multiplication results before adding the most significant 32-bit part of the results.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The `.u` form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][63:32];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMMAC_U(long t, long a, long b)`

KMMAC.u (SIMD Saturating MSW Signed Multiply Word and Add with Rounding)

Type: SIMD

Syntax:

```
KMMAC Rd, Rs1, Rs2
KMMAC.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and add the most significant 32-bit results with the signed 32-bit integer elements of a third register. The addition results are saturated first and then written back to the third register. The `.u` form performs an additional rounding up operation on the multiplication results before adding the most significant 32-bit part of the results.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The `.u` form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][63:32];
}
```

(continues on next page)

(continued from previous page)

```
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMMSB(long t, long a, long b)`

KMMSB (SIMD Saturating MSW Signed Multiply Word and Subtract)

Type: SIMD

Syntax:

```
KMMSB Rd, Rs1, Rs2
KMMSB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and subtract the most significant 32-bit results from the signed 32-bit elements of a third register. The subtraction results are written to the third register. The `.u` form performs an additional rounding up operation on the multiplication results before subtracting the most significant 32-bit part of the results.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and subtracts the most significant 32-bit multiplication results from the signed 32-bit elements of Rd. If the subtraction result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The `.u` form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  res[x] = Rd.W[x] - Round[x][32:1];
} else {
  res[x] = Rd.W[x] - Mres[x][63:32];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- [in] t: long type of value stored in t
- [in] a: long type of value stored in a
- [in] b: long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMSB_U(long t, long a, long b)

KMMSB.u (SIMD Saturating MSW Signed Multiply Word and Subtraction with Rounding)

Type: SIMD

Syntax:

```
KMMSB Rd, Rs1, Rs2
KMMSB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers and subtract the most significant 32-bit results from the signed 32-bit elements of a third register. The subtraction results are written to the third register. The .u form performs an additional rounding up operation on the multiplication results before subtracting the most significant 32-bit part of the results.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed 32-bit elements of Rs2 and subtracts the most significant 32-bit multiplication results from the signed 32-bit elements of Rd. If the subtraction result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The .u form of the instruction additionally rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  res[x] = Rd.W[x] - Round[x][32:1];
} else {
  res[x] = Rd.W[x] - Mres[x][63:32];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- [in] t: long type of value stored in t
- [in] a: long type of value stored in a
- [in] b: long type of value stored in b

__STATIC_FORCEINLINE long __RV_KWMMUL(long a, long b)

KWMMUL (SIMD Saturating MSW Signed Multiply Word & Double)

Type: SIMD

Syntax:

```
KWMMUL Rd, Rs1, Rs2
KWMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers, shift the results left 1-bit, saturate, and write the most significant 32-bit results to a register. The .u form additionally rounds up the multiplication results from the most signification discarded bit.

Description: This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2. It then shifts the multiplication results one bit to the left and takes the most significant 32-bit results. If the shifted result is greater than 2^31-1, it is saturated to 2^31-1 and the OV flag is set to 1. The final element result is written to Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The .u form of the instruction additionally rounds up the 64-bit multiplication results by adding a 1 to bit 30 before the shift and saturation operations.

Operations:

```
if ((0x80000000 != Rs1.W[x]) | (0x80000000 != Rs2.W[x])) {
  Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
  if (`.u` form) {
    Round[x][33:0] = Mres[x][63:30] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][62:31];
  }
} else {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- [in] a: long type of value stored in a
- [in] b: long type of value stored in b

__STATIC_FORCEINLINE long __RV_KWMMUL_U(long a, long b)

KWMMUL.u (SIMD Saturating MSW Signed Multiply Word & Double with Rounding)

Type: SIMD

Syntax:

```
KWMMUL Rd, Rs1, Rs2
KWMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of two registers, shift the results left 1-bit, saturate, and write the most significant 32-bit results to a register. The `.u` form additionally rounds up the multiplication results from the most signification discarded bit.

Description: This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2. It then shifts the multiplication results one bit to the left and takes the most significant 32-bit results. If the shifted result is greater than 2^31-1, it is saturated to 2^31-1 and the OV flag is set to 1. The final element result is written to Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The `.u` form of the instruction additionally rounds up the 64-bit multiplication results by adding a 1 to bit 30 before the shift and saturation operations.

Operations:

```
if ((0x80000000 != Rs1.W[x]) | (0x80000000 != Rs2.W[x])) {
  Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
  if (`.u` form) {
    Round[x][33:0] = Mres[x][63:30] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][62:31];
  }
} else {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

`__STATIC_FORCEINLINE long __RV_SMMUL(long a, long b)`

SMMUL (SIMD MSW Signed Multiply Word)

Type: SIMD

Syntax:

```
SMMUL Rd, Rs1, Rs2
SMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed integer elements of two registers and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form performs an additional rounding up operation on the multiplication results before taking the most significant 32-bit part of the results.

Description: This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The `.u` form of the instruction rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

- For `smmul/RV32` instruction, it is an alias to `mulh/RV32` instruction.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][63:32];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMMUL_U(long a, long b)

SMMUL.u (SIMD MSW Signed Multiply Word with Rounding)

Type: SIMD

Syntax:

```
SMMUL Rd, Rs1, Rs2
SMMUL.u Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed integer elements of two registers and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form performs an additional rounding up operation on the multiplication results before taking the most significant 32-bit part of the results.

Description: This instruction multiplies the 32-bit elements of Rs1 with the 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The 32-bit elements of Rs1 and Rs2 are treated as signed integers. The `.u` form of the instruction rounds up the most significant 32-bit of the 64-bit multiplication results by adding a 1 to bit 31 of the results.

- For `smmul/RV32` instruction, it is an alias to `mulh/RV32` instruction.

Operations:

```
Mres[x][63:0] = Rs1.W[x] * Rs2.W[x];
if (`.u` form) {
  Round[x][32:0] = Mres[x][63:31] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][63:32];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

Signed MSW 32x16 Multiply and Add Instructions

__STATIC_FORCEINLINE long __RV_KMMAWB(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWB_U(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWB2(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWB2_U(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWT(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWT_U(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWT2(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMAWT2_U(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMWB2(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMWB2_U(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMWT2(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMMWT2_U(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMMWB(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMMWB_U(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMMWT(long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMMWT_U(long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIGNED_MSW_32X16_MAC**

Signed MSW 32x16 Multiply and Add Instructions.

there are 15 Signed MSW 32x16 Multiply and Add Instructions

Functions

__STATIC_FORCEINLINE long __RV_KMMAWB(long t, unsigned long a, unsigned long b)

KMMAWB (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add)

Type: SIMD

Syntax:

```
KMMAWB Rd, Rs1, Rs2
KMMAWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][47:16];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMMAWB_U(long t, unsigned long a, unsigned long b)`

KMMAWB.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add with Rounding)

Type: SIMD

Syntax:

```
KMMAWB Rd, Rs1, Rs2
KMMAWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][47:16];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWB2(long t, unsigned long a, unsigned long b)

KMMAWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add)

Type: SIMD

Syntax:

```
KMMAWB2 Rd, Rs1, Rs2
KMMAWB2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and add the saturated most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The saturated addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {
  addop.W[x] = 0x7fffffff;
  OV = 1;
} else {
```

(continues on next page)

(continued from previous page)

```
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];
  if (`.u` form) {
    Mres[x][47:14] = Mres[x][47:14] + 1;
  }
  addop.W[x] = Mres[x][46:15]; // doubling
}
res[x] = Rd.W[x] + addop.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWB2_U(long t, unsigned long a, unsigned long b)

KMMAWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 and Add with Rounding)

Type: SIMD

Syntax:

```
KMMAWB2 Rd, Rs1, Rs2
KMMAWB2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and add the saturated most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The saturated addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {
  addop.W[x] = 0x7fffffff;
```

(continues on next page)

(continued from previous page)

```
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];
  if (`.u` form) {
    Mres[x][47:14] = Mres[x][47:14] + 1;
  }
  addop.W[x] = Mres[x][46:15]; // doubling
}
res[x] = Rd.W[x] + addop.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWT(long t, unsigned long a, unsigned long b)

KMMAWT (SIMD Saturating MSW Signed Multiply Word and Top Half and Add)

Type: SIMD

Syntax:

```
KMMAWT Rd, Rs1, Rs2
KMMAWT.u Rd Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the signed top 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition results are written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed top 16-bit of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (`.u` form) {
```

(continues on next page)

(continued from previous page)

```
  Round[x][32:0] = Mres[x][47:15] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][47:16];
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWT_U(long t, unsigned long a, unsigned long b)

KMMAWT.u (SIMD Saturating MSW Signed Multiply Word and Top Half and Add with Rounding)

Type: SIMD

Syntax:

```
KMMAWT Rd, Rs1, Rs2
KMMAWT.u Rd Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the signed top 16-bit of the corresponding 32-bit elements of another register and add the most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The addition results are written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed top 16-bit of the corresponding 32-bit elements of Rs2 and adds the most significant 32-bit multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the result before the addition operations.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  res[x] = Rd.W[x] + Round[x][32:1];
} else {
  res[x] = Rd.W[x] + Mres[x][47:16];
```

(continues on next page)

(continued from previous page)

```
}
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWT2(long t, unsigned long a, unsigned long b)

KMMAWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add)

Type: SIMD

Syntax:

```
KMMAWT2 Rd, Rs1, Rs2
KMMAWT2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and add the saturated most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The saturated addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
  addop.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
  if (`.u` form) {
    Mres[x][47:14] = Mres[x][47:14] + 1;
  }
  addop.W[x] = Mres[x][46:15]; // doubling
```

(continues on next page)

(continued from previous page)

```
}
res[x] = Rd.W[x] + addop.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMAWT2_U(long t, unsigned long a, unsigned long b)

KMMAWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 and Add with Rounding)

Type: SIMD

Syntax:

```
KMMAWT2 Rd, Rs1, Rs2
KMMAWT2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and add the saturated most significant 32-bit results with the corresponding signed 32-bit elements of a third register. The saturated addition result is written to the corresponding 32-bit elements of the third register. The `.u` form rounds up the multiplication results from the most significant discarded bit before the addition operations.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and adds the saturated most significant 32-bit Q31 multiplication results with the corresponding signed 32-bit elements of Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the result before the addition operations.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
  addop.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
  if (`.u` form) {
    Mres[x][47:14] = Mres[x][47:14] + 1;
  }
```

(continues on next page)

(continued from previous page)

```
  addop.W[x] = Mres[x][46:15]; // doubling
}
res[x] = Rd.W[x] + addop.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMWB2(long a, unsigned long b)

KMMWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2)

Type: SIMD

Syntax:

```
KMMWB2 Rd, Rs1, Rs2
KMMWB2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and write the saturated most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];
  if (`.u` form) {
    Round[x][32:0] = Mres[x][46:14] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][46:15];
  }
```

(continues on next page)

(continued from previous page)

```
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMWB2_U(long a, unsigned long b)

KMMWB2.u (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 2 with Rounding)

Type: SIMD

Syntax:

```
KMMWB2 Rd, Rs1, Rs2
KMMWB2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and write the saturated most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed bottom 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[0] == 0x8000)) {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];
  if (`.u` form) {
    Round[x][32:0] = Mres[x][46:14] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][46:15];
  }
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMMWT2(long a, unsigned long b)

KMMWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 2)

Type: SIMD

Syntax:

```
KMMWT2 Rd, Rs1, Rs2
KMMWT2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and write the saturated most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
  if (`.u` form) {
    Round[x][32:0] = Mres[x][46:14] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][46:15];
  }
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMMWT2_U(long a, unsigned long b)`

KMMWT2.u (SIMD Saturating MSW Signed Multiply Word and Top Half & 2 with Rounding)

Type: SIMD

Syntax:

```
KMMWT2 Rd, Rs1, Rs2
KMMWT2.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, double the multiplication results and write the saturated most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit Q31 elements of Rs1 with the signed top 16-bit Q15 content of the corresponding 32-bit elements of Rs2, doubles the Q46 results to Q47 numbers and writes the saturated most significant 32-bit Q31 multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit Q47 multiplication results by adding a 1 to bit 15 (i.e., bit 14 before doubling) of the results.

Operations:

```
if ((Rs1.W[x] == 0x80000000) & (Rs2.W[x].H[1] == 0x8000)) {
  Rd.W[x] = 0x7fffffff;
  OV = 1;
} else {
  Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[1];
  if (`.u` form) {
    Round[x][32:0] = Mres[x][46:14] + 1;
    Rd.W[x] = Round[x][32:1];
  } else {
    Rd.W[x] = Mres[x][46:15];
  }
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_SMMWB(long a, unsigned long b)`

SMMWB (SIMD MSW Signed Multiply Word and Bottom Half)

Type: SIMD

Syntax:

```
SMMWB Rd, Rs1, Rs2
SMMWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMMWB_U(long a, unsigned long b)

SMMWB.u (SIMD MSW Signed Multiply Word and Bottom Half with Rounding)

Type: SIMD

Syntax:

```
SMMWB Rd, Rs1, Rs2
SMMWB.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the bottom 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the signed bottom 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[0];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMMWT(long a, unsigned long b)

SMMWT (SIMD MSW Signed Multiply Word and Top Half)

Type: SIMD

Syntax:

```
SMMWT Rd, Rs1, Rs2
SMMWT.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the top signed 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_SMMWT_U(long a, unsigned long b)`

SMMWT.u (SIMD MSW Signed Multiply Word and Top Half with Rounding)

Type: SIMD

Syntax:

```
SMMWT Rd, Rs1, Rs2
SMMWT.u Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit integer elements of one register and the top 16-bit of the corresponding 32-bit elements of another register, and write the most significant 32-bit results to the corresponding 32-bit elements of a register. The `.u` form rounds up the results from the most significant discarded bit.

Description: This instruction multiplies the signed 32-bit elements of Rs1 with the top signed 16-bit content of the corresponding 32-bit elements of Rs2 and writes the most significant 32-bit multiplication results to the corresponding 32-bit elements of Rd. The `.u` form of the instruction rounds up the most significant 32-bit of the 48-bit multiplication results by adding a 1 to bit 15 of the results.

Operations:

```
Mres[x][47:0] = Rs1.W[x] * Rs2.W[x].H[1];
if (`.u` form) {
  Round[x][32:0] = Mres[x][47:15] + 1;
  Rd.W[x] = Round[x][32:1];
} else {
  Rd.W[x] = Mres[x][47:16];
}
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

Signed 16-bit Multiply 32-bit Add/Subtract Instructions

__STATIC_FORCEINLINE long __RV_KMABB(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMABT(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMATT(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMADA(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMAXDA(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMADS(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMADRS(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMAXDS(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMDA(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMXDA(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMSDA(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMSXDA(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMBB16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMBT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMTT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMDS(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMDRS(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMXDS(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIGNED_16B_MULT_32B_ADDSUB**

Signed 16-bit Multiply 32-bit Add/Subtract Instructions.

there are 18 Signed 16-bit Multiply 32-bit Add/Subtract Instructions

Functions

__STATIC_FORCEINLINE long __RV_KMABB(long t, unsigned long a, unsigned long b)

KMABB (SIMD Saturating Signed Multiply Bottom Halfs & Add)

Type: SIMD

Syntax:

```
KMABB Rd, Rs1, Rs2
KMABT Rd, Rs1, Rs2
KMATT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of 32-bit elements in a register with the 16-bit content of 32-bit elements in another register and add the result to the content of 32-bit elements in the third register. The addition result may be saturated and is written to the third register.

- KMABB: rd.W[x] + bottom*bottom (per 32-bit element)

- KMABT rd.W[x] + bottom*top (per 32-bit element)
- KMATT rd.W[x] + top*top (per 32-bit element)

Description: For the `KMABB` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. For the `KMABT` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMATT` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. The multiplication result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to

1. The results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]); // KMABB
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[1]); // KMABT
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]); // KMATT
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMABT(long t, unsigned long a, unsigned long b)`

KMABT (SIMD Saturating Signed Multiply Bottom & Top Halfs & Add)

Type: SIMD

Syntax:

```
KMABB Rd, Rs1, Rs2
KMABT Rd, Rs1, Rs2
KMATT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of 32-bit elements in a register with the 16-bit content of 32-bit elements in another register and add the result to the content of 32-bit elements in the third register. The addition result may be saturated and is written to the third register.

- KMABB: rd.W[x] + bottom*bottom (per 32-bit element)
- KMABT rd.W[x] + bottom*top (per 32-bit element)
- KMATT rd.W[x] + top*top (per 32-bit element)

Description: For the KMABB instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. For the KMABT instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the KMATT instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. The multiplication result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to

1. The results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]); // KMABB
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[1]); // KMABT
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]); // KMATT
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMATT(long t, unsigned long a, unsigned long b)

KMATT (SIMD Saturating Signed Multiply Top Halfs & Add)

Type: SIMD

Syntax:

```
KMABB Rd, Rs1, Rs2
KMABT Rd, Rs1, Rs2
KMATT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of 32-bit elements in a register with the 16-bit content of 32-bit elements in another register and add the result to the content of 32-bit elements in the third register. The addition result may be saturated and is written to the third register.

- KMABB: rd.W[x] + bottom*bottom (per 32-bit element)
- KMABT rd.W[x] + bottom*top (per 32-bit element)
- KMATT rd.W[x] + top*top (per 32-bit element)

Description: For the KMABB instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. For the KMABT instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the KMATT instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. The multiplication result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to

1. The results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]); // KMABB
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[1]); // KMABT
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]); // KMATT
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMADA(long t, unsigned long a, unsigned long b)

KMADA (SIMD Saturating Signed Multiply Two Halfs and Two Adds)

Type: SIMD

Syntax:

```
KMADA Rd, Rs1, Rs2
KMAXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then adds the two 32-bit results and 32-bit elements in a third register together. The addition result may be saturated.

- KMADA: rd.W[x] + top*top + bottom*bottom (per 32-bit element)
- KMAXDA: rd.W[x] + top*bottom + bottom*top (per 32-bit element)

Description: For the ‘KMADA instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMAXDA` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. The result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADA
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMAXDA
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) + (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMAXDA(long t, unsigned long a, unsigned long b)

KMAXDA (SIMD Saturating Signed Crossed Multiply Two Halfs and Two Adds)

Type: SIMD

Syntax:

```
KMADA Rd, Rs1, Rs2
KMAXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then adds the two 32-bit results and 32-bit elements in a third register together. The addition result may be saturated.

- KMADA: rd.W[x] + top*top + bottom*bottom (per 32-bit element)
- KMAXDA: rd.W[x] + top*bottom + bottom*top (per 32-bit element)

Description: For the ‘KMADA instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMAXDA` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. The result is added to the content of 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADA
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMAXDA
```

(continues on next page)

(continued from previous page)

```
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) + (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMADS(long t, unsigned long a, unsigned long b)

KMADS (SIMD Saturating Signed Multiply Two Halfs & Subtract & Add)

Type: SIMD

Syntax:

```
KMADS Rd, Rs1, Rs2
KMADRS Rd, Rs1, Rs2
KMAXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the corresponding 32-bit elements in a third register. The addition result may be saturated.

- KMADS: rd.W[x] + (top*top - bottom*bottom) (per 32-bit element)
- KMADRS: rd.W[x] + (bottom*bottom - top*top) (per 32-bit element)
- KMAXDS: rd.W[x] + (top*bottom - bottom*top) (per 32-bit element)

Description: For the `KMADS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMADRS` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32- bit elements in Rs2. For the `KMAXDS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. The subtraction result is then added to the content of the corresponding 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMADRS
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.
↪W[x].H[1]);
// KMAXDS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMADRS(long t, unsigned long a, unsigned long b)`

KMADRS (SIMD Saturating Signed Multiply Two Halfs & Reverse Subtract & Add)

Type: SIMD

Syntax:

```
KMADS Rd, Rs1, Rs2
KMADRS Rd, Rs1, Rs2
KMAXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the corresponding 32-bit elements in a third register. The addition result may be saturated.

- KMADS: rd.W[x] + (top*top - bottom*bottom) (per 32-bit element)
- KMADRS: rd.W[x] + (bottom*bottom - top*top) (per 32-bit element)
- KMAXDS: rd.W[x] + (top*bottom - bottom*top) (per 32-bit element)

Description: For the `KMADS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMADRS` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32- bit elements in Rs2. For the `KMAXDS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. The subtraction result is then added to the content of the corresponding 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMADRS
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.
↪W[x].H[1]);
// KMAXDS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMAXDS(long t, unsigned long a, unsigned long b)

KMAXDS (SIMD Saturating Signed Crossed Multiply Two Halfs & Subtract & Add)

Type: SIMD

Syntax:

```
KMADS Rd, Rs1, Rs2
KMADRS Rd, Rs1, Rs2
KMAXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the corresponding 32-bit elements in a third register. The addition result may be saturated.

- KMADS: rd.W[x] + (top*top - bottom*bottom) (per 32-bit element)
- KMADRS: rd.W[x] + (bottom*bottom - top*top) (per 32-bit element)
- KMAXDS: rd.W[x] + (top*bottom - bottom*top) (per 32-bit element)

Description: For the `KMADS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2. For the `KMADRS` instruction, it multiplies the top 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32- bit elements in Rs2. For the `KMAXDS` instruction, it multiplies the bottom 16-bit content of 32-bit elements in Rs1 with the top 16-bit content of 32-bit elements in Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of 32-bit elements in Rs1 with the bottom 16-bit content of 32-bit elements in Rs2. The subtraction result is then added to the content of the corresponding 32-bit elements in Rd. If the addition result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The 32-bit results after saturation are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
// KMADS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMADRS
res[x] = Rd.W[x] + (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.
↪W[x].H[1]);
// KMAXDS
res[x] = Rd.W[x] + (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMDA(unsigned long a, unsigned long b)

KMDA (SIMD Signed Multiply Two Halfs and Add)

Type: SIMD

Syntax:

```
KMDA Rd, Rs1, Rs2
KMXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results together. The addition result may be saturated.

- KMDA: top*top + bottom*bottom (per 32-bit element)
- KMXDA: top*bottom + bottom*top (per 32-bit element)

Description: For the `KMDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32- bit elements of Rs2. For the `KMXDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The addition result is checked for saturation. If saturation happens, the result is saturated to 2^31-1. The final results are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
if Rs1.W[x] != 0x80008000) or (Rs2.W[x] != 0x80008000 { // KMDA Rd.
↪W[x] = Rs1.W[x].H[1] *
Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.W[x].H[0]; // KMXDA Rd.W[x] = Rs1.W[x].
↪H[1] * Rs2.W[x].H[0])
+ (Rs1.W[x].H[0] * Rs2.W[x].H[1]; } else { Rd.W[x] = 0x7fffffff; OV␣
↪ = 1; } for RV32: x=0 for RV64:
x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMXDA(unsigned long a, unsigned long b)`

KMXDA (SIMD Signed Crossed Multiply Two Halfs and Add)

Type: SIMD

Syntax:

```
KMDA Rd, Rs1, Rs2
KMXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results together. The addition result may be saturated.

- KMDA: top*top + bottom*bottom (per 32-bit element)
- KMXDA: top*bottom + bottom*top (per 32-bit element)

Description: For the `KMDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32- bit elements of Rs2. For the `KMXDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The addition result is checked for saturation. If saturation happens, the result is saturated to 2^31-1. The final results are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
if Rs1.W[x] != 0x80008000) or (Rs2.W[x] != 0x80008000 { // KMDA Rd.
↪W[x] = Rs1.W[x].H[1] *
Rs2.W[x].H[1]) + (Rs1.W[x].H[0] * Rs2.W[x].H[0]; // KMXDA Rd.W[x] = Rs1.W[x].
↪H[1] * Rs2.W[x].H[0])
+ (Rs1.W[x].H[0] * Rs2.W[x].H[1]; } else { Rd.W[x] = 0x7fffffff; OV␣
↪ = 1; } for RV32: x=0 for RV64:
x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMSDA(long t, unsigned long a, unsigned long b)

KMSDA (SIMD Saturating Signed Multiply Two Halfs & Add & Subtract)

Type: SIMD

Syntax:

```
KMSDA Rd, Rs1, Rs2
KMSXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the corresponding 32-bit elements of a third register. The subtraction result may be saturated.

- KMSDA: rd.W[x] - top*top - bottom*bottom (per 32-bit element)
- KMSXDA: rd.W[x] - top*bottom - bottom*top (per 32-bit element)

Description: For the `KMSDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `KMSXDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The two 32-bit multiplication results are then subtracted from the content of the corresponding 32- bit elements of Rd. If the subtraction result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
// KMSDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMSXDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMSXDA(long t, unsigned long a, unsigned long b)

KMSXDA (SIMD Saturating Signed Crossed Multiply Two Halfs & Add & Subtract)

Type: SIMD

Syntax:

```
KMSDA Rd, Rs1, Rs2
KMSXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the corresponding 32-bit elements of a third register. The subtraction result may be saturated.

- KMSDA: rd.W[x] - top*top - bottom*bottom (per 32-bit element)
- KMSXDA: rd.W[x] - top*bottom - bottom*top (per 32-bit element)

Description: For the `KMSDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `KMSXDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The two 32-bit multiplication results are then subtracted from the content of the corresponding 32- bit elements of Rd. If the subtraction result is beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), it is saturated to the range and the OV bit is set to 1. The results after saturation are written to Rd. The 16-bit contents are treated as signed integers.

Operations:

```
// KMSDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[0]);
// KMSXDA
res[x] = Rd.W[x] - (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.
↪W[x].H[1]);
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMBB16(unsigned long a, unsigned long b)

SMBB16 (SIMD Signed Multiply Bottom Half & Bottom Half)

Type: SIMD

Syntax:

```
SMBB16 Rd, Rs1, Rs2
SMBT16 Rd, Rs1, Rs2
SMTT16 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16- bit content of the 32-bit elements of another register and write the result to a third register.

- SMBB16: W[x].bottom*W[x].bottom
- SMBT16: W[x].bottom *W[x].top
- SMTT16: W[x].top * W[x].top

Description: For the `SMBB16` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMBT16` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMTT16` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[0]; // SMBB16
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[1]; // SMBT16
Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[1]; // SMTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_SMBT16(unsigned long a, unsigned long b)`

SMBT16 (SIMD Signed Multiply Bottom Half & Top Half)

Type: SIMD

Syntax:

```
SMBB16 Rd, Rs1, Rs2
SMBT16 Rd, Rs1, Rs2
SMTT16 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16- bit content of the 32-bit elements of another register and write the result to a third register.

- SMBB16: W[x].bottom*W[x].bottom
- SMBT16: W[x].bottom *W[x].top
- SMTT16: W[x].top * W[x].top

Description: For the `SMBB16` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMBT16` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the SMTT16 instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[0]; // SMBB16
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[1]; // SMBT16
Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[1]; // SMTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMTT16(unsigned long a, unsigned long b)
SMTT16 (SIMD Signed Multiply Top Half & Top Half)

Type: SIMD

Syntax:

```
SMBB16 Rd, Rs1, Rs2
SMBT16 Rd, Rs1, Rs2
SMTT16 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the signed 16- bit content of the 32-bit elements of another register and write the result to a third register.

- SMBB16: W[x].bottom*W[x].bottom
- SMBT16: W[x].bottom *W[x].top
- SMTT16: W[x].top * W[x].top

Description: For the SMBB16 instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the SMBT16 instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the SMTT16 instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[0]; // SMBB16
Rd.W[x] = Rs1.W[x].H[0] * Rs2.W[x].H[1]; // SMBT16
Rd.W[x] = Rs1.W[x].H[1] * Rs2.W[x].H[1]; // SMTT16
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMDS(unsigned long a, unsigned long b)
SMDS (SIMD Signed Multiply Two Halfs and Subtract)

Type: SIMD

Syntax:

```
SMDS Rd, Rs1, Rs2
SMDRS Rd, Rs1, Rs2
SMXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results.

- SMDS: top*top - bottom*bottom (per 32-bit element)
- SMDRS: bottom*bottom - top*top (per 32-bit element)
- SMXDS: top*bottom - bottom*top (per 32-bit element)

Description: For the `SMDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMXDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

```
* SMDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
* SMDRS:
Rd.W[x] = (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);
* SMXDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMDRS(unsigned long a, unsigned long b)
SMDRS (SIMD Signed Multiply Two Halfs and Reverse Subtract)

Type: SIMD

Syntax:

```
SMDS Rd, Rs1, Rs2
SMDRS Rd, Rs1, Rs2
SMXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results.

- SMDS: top*top - bottom*bottom (per 32-bit element)
- SMDRS: bottom*bottom - top*top (per 32-bit element)
- SMXDS: top*bottom - bottom*top (per 32-bit element)

Description: For the `SMDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMXDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

```
* SMDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
* SMDRS:
Rd.W[x] = (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);
* SMXDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_SMXDS(unsigned long a, unsigned long b)`

SMXDS (SIMD Signed Crossed Multiply Two Halfs and Subtract)

Type: SIMD

Syntax:

```
SMDS Rd, Rs1, Rs2
SMDRS Rd, Rs1, Rs2
SMXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results.

- SMDS: top*top - bottom*bottom (per 32-bit element)
- SMDRS: bottom*bottom - top*top (per 32-bit element)
- SMXDS: top*bottom - bottom*top (per 32-bit element)

Description: For the `SMDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMXDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction result is written to the corresponding 32-bit element of Rd. The 16-bit contents of multiplication are treated as signed integers.

Operations:

```
* SMDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[1]) - (Rs1.W[x].H[0] * Rs2.W[x].H[0]);
* SMDRS:
Rd.W[x] = (Rs1.W[x].H[0] * Rs2.W[x].H[0]) - (Rs1.W[x].H[1] * Rs2.W[x].H[1]);
* SMXDS:
Rd.W[x] = (Rs1.W[x].H[1] * Rs2.W[x].H[0]) - (Rs1.W[x].H[0] * Rs2.W[x].H[1]);
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

Partial-SIMD Miscellaneous Instructions

`__STATIC_FORCEINLINE unsigned long __RV_CLRS32(unsigned long a)`

`__STATIC_FORCEINLINE unsigned long __RV_CLO32(unsigned long a)`

`__STATIC_FORCEINLINE unsigned long __RV_CLZ32(unsigned long a)`

`__STATIC_FORCEINLINE unsigned long __RV_PBSAD(unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_PBSADA(unsigned long t, unsigned long a, unsigned l`

`__RV_SCLIP32` (a, b)

`__RV_UCLIP32` (a, b)

group **`NMSIS_Core_DSP_Intrinsic_PART_SIMD_MISC`**

Partial-SIMD Miscellaneous Instructions.

there are 7 Partial-SIMD Miscellaneous Instructions

Defines

`__RV_SCLIP32` (a, b)

SCLIP32 (SIMD 32-bit Signed Clip Value)

Type: DSP

Syntax:

```
SCLIP32 Rd, Rs1, imm5u[4:0]
```

Purpose: Limit the 32-bit signed integer elements of a register into a signed range simultaneously.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 into a signed integer range between 2imm5u-1 and -2imm5u, and writes the limited results to Rd. For example, if imm5u is 3, the 32-bit input values should be saturated between 7 and -8. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.W[x];
if (src > (2^imm5u)-1) {
  src = (2^imm5u)-1;
  OV = 1;
} else if (src < -2^imm5u) {
  src = -2^imm5u;
  OV = 1;
}
Rd.W[x] = src
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] a`: long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__RV_UCLIP32 (a, b)

UCLIP32 (SIMD 32-bit Unsigned Clip Value)

Type: SIMD

Syntax:

```
UCLIP32 Rd, Rs1, imm5u[4:0]
```

Purpose: Limit the 32-bit signed integer elements of a register into an unsigned range simultaneously.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 into an unsigned integer range between 2imm5u-1 and 0, and writes the limited results to Rd. For example, if imm5u is 3, the 32-bit input values should be saturated between 7 and 0. If saturation is performed, set OV bit to 1.

Operations:

```
src = Rs1.W[x];
if (src > (2^imm5u)-1) {
  src = (2^imm5u)-1;
  OV = 1;
} else if (src < 0) {
  src = 0;
  OV = 1;
}
Rd.W[x] = src
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

Functions

__STATIC_FORCEINLINE unsigned long __RV_CLRS32(unsigned long a)

CLRS32 (SIMD 32-bit Count Leading Redundant Sign)

Type: SIMD

Syntax:

```
CLRS32 Rd, Rs1
```

Purpose: Count the number of redundant sign bits of the 32-bit elements of a general register.

Description: Starting from the bits next to the sign bits of the 32-bit elements of Rs1, this instruction counts the number of redundant sign bits and writes the result to the corresponding 32- bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 30 to 0) {
  if (snum[x](i) == snum[x](31)) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_CLO32(unsigned long a)

CLO32 (SIMD 32-bit Count Leading One)

Type: SIMD

Syntax:

```
CLO32 Rd, Rs1
```

Purpose: Count the number of leading one bits of the 32-bit elements of a general register.

Description: Starting from the most significant bits of the 32-bit elements of Rs1, this instruction counts the number of leading one bits and writes the results to the corresponding 32-bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 31 to 0) {
  if (snum[x](i) == 1) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
```

(continues on next page)

(continued from previous page)

```
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_CLZ32(unsigned long a)

CLZ32 (SIMD 32-bit Count Leading Zero)

Type: SIMD

Syntax:

```
CLZ32 Rd, Rs1
```

Purpose: Count the number of leading zero bits of the 32-bit elements of a general register.

Description: Starting from the most significant bits of the 32-bit elements of Rs1, this instruction counts the number of leading zero bits and writes the results to the corresponding 32-bit elements of Rd.

Operations:

```
snum[x] = Rs1.W[x];
cnt[x] = 0;
for (i = 31 to 0) {
  if (snum[x](i) == 0) {
    cnt[x] = cnt[x] + 1;
  } else {
    break;
  }
}
Rd.W[x] = cnt[x];
for RV32: x=0
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_PBSAD(unsigned long a, unsigned long b)

PBSAD (Parallel Byte Sum of Absolute Difference)

Type: DSP

Syntax:

```
PBSAD Rd, Rs1, Rs2
```

Purpose: Calculate the sum of absolute difference of unsigned 8-bit data elements.

Description: This instruction subtracts the un-signed 8-bit elements of Rs2 from those of Rs1. Then it adds the absolute value of each difference together and writes the result to Rd.

Operations:

```
absdiff[x] = ABS(Rs1.B[x] - Rs2.B[x]);
Rd = SUM(absdiff[x]);
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_PBSADA(unsigned long t, unsigned long a, unsig`

PBSADA (Parallel Byte Sum of Absolute Difference Accum)

Type: DSP

Syntax:

```
PBSADA Rd, Rs1, Rs2
```

Purpose: Calculate the sum of absolute difference of four unsigned 8-bit data elements and accumulate it into a register.

Description: This instruction subtracts the un-signed 8-bit elements of Rs2 from those of Rs1. It then adds the absolute value of each difference together along with the content of Rd and writes the accumulated result back to Rd.

Operations:

```
absdiff[x] = ABS(Rs1.B[x] - Rs2.B[x]);
Rd = Rd + SUM(absdiff[x]);
for RV32: x=3...0,
for RV64: x=7...0
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

8-bit Multiply with 32-bit Add Instructions

`__STATIC_FORCEINLINE long __RV_SMAQA(long t, unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE long __RV_SMAQA_SU(long t, unsigned long a, unsigned long b)`

`__STATIC_FORCEINLINE unsigned long __RV_UMAQA(unsigned long t, unsigned long a, unsigned lo`

group **`NMSIS_Core_DSP_Intrinsic_8B_MULT_32B_ADD`**

8-bit Multiply with 32-bit Add Instructions

there are 3 8-bit Multiply with 32-bit Add Instructions

Functions

__STATIC_FORCEINLINE long __RV_SMAQA(long t, unsigned long a, unsigned long b)

SMAQA (Signed Multiply Four Bytes with 32-bit Adds)

Type: Partial-SIMD (Reduction)

Syntax:

```
SMAQA Rd, Rs1, Rs2
```

Purpose: Do four signed 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description: This instruction multiplies the four signed 8-bit elements of 32-bit chunks of Rs1 with the four signed 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the signed content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] +
    (Rs1.W[x].B[3] s* Rs2.W[x].B[3]) + (Rs1.W[x].B[2] s* Rs2.W[x].B[2]) +
    (Rs1.W[x].B[1] s* Rs2.W[x].B[1]) + (Rs1.W[x].B[0] s* Rs2.W[x].B[0]);
Rd.W[x] = res[x];
for RV32: x=0,
for RV64: x=1,0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMAQA_SU(long t, unsigned long a, unsigned long b)

SMAQA.SU (Signed and Unsigned Multiply Four Bytes with 32-bit Adds)

Type: Partial-SIMD (Reduction)

Syntax:

```
SMAQA.SU Rd, Rs1, Rs2
```

Purpose: Do four `signed x unsigned` 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description: This instruction multiplies the four signed 8-bit elements of 32-bit chunks of Rs1 with the four unsigned 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the signed content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] +
    (Rs1.W[x].B[3] su* Rs2.W[x].B[3]) + (Rs1.W[x].B[2] su* Rs2.W[x].B[2]) +
    (Rs1.W[x].B[1] su* Rs2.W[x].B[1]) + (Rs1.W[x].B[0] su* Rs2.W[x].B[0]);
Rd.W[x] = res[x];
```

(continues on next page)

(continued from previous page)

```
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMAQA(unsigned long t, unsigned long a, unsign

UMAQA (Unsigned Multiply Four Bytes with 32- bit Adds)

Type: DSP

Syntax:

```
UMAQA Rd, Rs1, Rs2
```

Purpose: Do four unsigned 8-bit multiplications from 32-bit chunks of two registers; and then adds the four 16-bit results and the content of corresponding 32-bit chunks of a third register together.

Description: This instruction multiplies the four unsigned 8-bit elements of 32-bit chunks of Rs1 with the four unsigned 8-bit elements of 32-bit chunks of Rs2 and then adds the four results together with the unsigned content of the corresponding 32-bit chunks of Rd. The final results are written back to the corresponding 32-bit chunks in Rd.

Operations:

```
res[x] = Rd.W[x] + (Rs1.W[x].B[3] u* Rs2.W[x].B[3]) +
         (Rs1.W[x].B[2] u* Rs2.W[x].B[2]) + (Rs1.W[x].B[1] u* Rs2.W[x].B[1]) +
         (Rs1.W[x].B[0] u* Rs2.W[x].B[0]);
Rd.W[x] = res[x];
for RV32: x=0,
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

group **NMSIS_Core_DSP_Intrinsic_PART_SIMD_DATA_PROCESS**

Partial-SIMD Data Processing Instructions.

64-bit Profile Instructions

64-bit Addition & Subtraction Instructions

__STATIC_FORCEINLINE unsigned long long __RV_ADD64(unsigned long long a, unsigned long long

__STATIC_FORCEINLINE long long __RV_KADD64(long long a, long long b)

__STATIC_FORCEINLINE long long __RV_KSUB64(long long a, long long b)

__STATIC_FORCEINLINE long long __RV_RADD64(long long a, long long b)

__STATIC_FORCEINLINE long long __RV_RSUB64(long long a, long long b)

__STATIC_FORCEINLINE unsigned long long __RV_SUB64(unsigned long long a, unsigned long long

__STATIC_FORCEINLINE unsigned long long __RV_UKADD64(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_UKSUB64(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_URADD64(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_URSUB64(unsigned long long a, unsigned long lo

group **NMSIS_Core_DSP_Intrinsic_64B_ADDSUB**

64-bit Addition & Subtraction Instructions

there are 10 64-bit Addition & Subtraction Instructions.

Functions

__STATIC_FORCEINLINE unsigned long long __RV_ADD64(unsigned long long a, unsigned long

ADD64 (64-bit Addition)

Type: 64-bit Profile

Syntax:

```
ADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed or unsigned integers.

RV32 Description: This instruction adds the 64-bit integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit integer of an even/odd pair of registers specified by Rs2(4,1), and then writes the 64-bit result to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., value d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction has the same behavior as the ADD instruction in RV64I.

Note: This instruction can be used for either signed or unsigned addition.

Operations:

```
RV32:
 t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
 a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
 b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
 R[t_H].R[t_L] = R[a_H].R[a_L] + R[b_H].R[b_L];
RV64:
 Rd = Rs1 + Rs2;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE long long __RV_KADD64(long long a, long long b)

KADD64 (64-bit Signed Saturating Addition)

Type: DSP (64-bit Profile)

Syntax:

```
KADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed integers. The result is saturated to the Q63 range.

RV32 Description: This instruction adds the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1). If the 64-bit result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., value d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit signed integer in Rs1 with the 64-bit signed integer in Rs2. If the result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written to Rd.

Operations:

```
RV32:
 t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
 a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
 b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
 result = R[a_H].R[a_L] + R[b_H].R[b_L];
 if (result > (2^63)-1) {
   result = (2^63)-1; OV = 1;
 } else if (result < -2^63) {
   result = -2^63; OV = 1;
 }
 R[t_H].R[t_L] = result;
RV64:
 result = Rs1 + Rs2;
 if (result > (2^63)-1) {
   result = (2^63)-1; OV = 1;
 } else if (result < -2^63) {
   result = -2^63; OV = 1;
 }
 Rd = result;
```

Return value stored in long long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: long long type of value stored in b

__STATIC_FORCEINLINE long long __RV_KSUB64(long long a, long long b)

KSUB64 (64-bit Signed Saturating Subtraction)

Type: DSP (64-bit Profile)

Syntax:

```
KSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is saturated to the Q63 range.

RV32 Description: This instruction subtracts the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1). If the 64-bit result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit signed integer of Rs2 from the 64-bit signed integer of Rs1. If the 64-bit result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to Rd.

Operations:

```
RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
result = R[a_H].R[a_L] - R[b_H].R[b_L];
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;
RV64:
result = Rs1 - Rs2;
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
Rd = result;
```

Return value stored in long long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: long long type of value stored in b

__STATIC_FORCEINLINE long long __RV_RADD64(long long a, long long b)

RADD64 (64-bit Signed Halving Addition)

Type: DSP (64-bit Profile)

Syntax:

```
RADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit signed integers. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction adds the 64-bit signed integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit signed integer of an even/odd pair of registers specified by Rs2(4,1). The 64-bit addition result is first arithmetically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., value d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit signed integer in Rs1 with the 64-bit signed integer in Rs2. The 64-bit addition result is first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] + R[b_H].R[b_L]) s>> 1;
RV64:
Rd = (Rs1 + Rs2) s>> 1;
```

Return value stored in long long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: long long type of value stored in b

`__STATIC_FORCEINLINE long long __RV_RSUB64(long long a, long long b)`

RSUB64 (64-bit Signed Halving Subtraction)

Type: DSP (64-bit Profile)

Syntax:

```
RSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction subtracts the 64-bit signed integer of an even/odd pair of registers specified by Rb(4,1) from the 64-bit signed integer of an even/odd pair of registers specified by Ra(4,1). The subtraction result is first arithmetically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rt(4,1). Rx(4,1), i.e., value d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction subtracts the 64-bit signed integer in Rs2 from the 64-bit signed integer in Rs1. The 64-bit subtraction result is first arithmetically right-shifted by 1 bit and then written to Rd.

Operations:

```
RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] - R[b_H].R[b_L]) s>> 1;
RV64:
Rd = (Rs1 - Rs2) s>> 1;
```

Return value stored in long long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_SUB64(unsigned long long a, unsigned long

SUB64 (64-bit Subtraction)

Type: DSP (64-bit Profile)

Syntax:

```
SUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed or unsigned integer subtraction.

RV32 Description: This instruction subtracts the 64-bit integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit integer of an even/odd pair of registers specified by Rs1(4,1), and then writes the 64-bit result to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit integer of Rs2 from the 64-bit integer of Rs1, and then writes the 64-bit result to Rd.

Note: This instruction can be used for either signed or unsigned subtraction.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
R[t_H].R[t_L] = R[a_H].R[a_L] - R[b_H].R[b_L];
* RV64:
Rd = Rs1 - Rs2;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UKADD64(unsigned long long a, unsigned lo

UKADD64 (64-bit Unsigned Saturating Addition)

Type: DSP (64-bit Profile)

Syntax:

```
UKADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit unsigned integers. The result is saturated to the U64 range.

RV32 Description: This instruction adds the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1). If the 64-bit result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit unsigned integer in Rs1 with the 64-bit unsigned integer in Rs2. If the 64-bit result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rt(4,1),1'b0); t_H = CONCAT(Rt(4,1),1'b1);
a_L = CONCAT(Ra(4,1),1'b0); a_H = CONCAT(Ra(4,1),1'b1);
b_L = CONCAT(Rb(4,1),1'b0); b_H = CONCAT(Rb(4,1),1'b1);
result = R[a_H].R[a_L] + R[b_H].R[b_L];
if (result > (2^64)-1) {
  result = (2^64)-1; OV = 1;
}
R[t_H].R[t_L] = result;
* RV64:
result = Rs1 + Rs2;
if (result > (2^64)-1) {
  result = (2^64)-1; OV = 1;
}
Rd = result;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UKSUB64(unsigned long long a, unsigned lo

UKSUB64 (64-bit Unsigned Saturating Subtraction)

Type: DSP (64-bit Profile)

Syntax:

```
UKSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is saturated to the U64 range.

RV32 Description: This instruction subtracts the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1). If the 64-bit result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction subtracts the 64-bit unsigned integer of Rs2 from the 64-bit unsigned integer of an even/odd pair of Rs1. If the 64-bit result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is then written to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
```

(continues on next page)

(continued from previous page)

```
a_L = CONCAT(Rs1(4,1),1'b0); a_H = CONCAT(Rs1(4,1),1'b1);
b_L = CONCAT(Rs2(4,1),1'b0); b_H = CONCAT(Rs2(4,1),1'b1);
result = R[a_H].R[a_L] - R[b_H].R[b_L];
if (result < 0) {
  result = 0; OV = 1;
}
R[t_H].R[t_L] = result;
* RV64
result = Rs1 - Rs2;
if (result < 0) {
  result = 0; OV = 1;
}
Rd = result;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_URADD64(unsigned long long a, unsigned lo

URADD64 (64-bit Unsigned Halving Addition)

Type: DSP (64-bit Profile)

Syntax:

```
URADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit unsigned integers. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction adds the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1) with the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1). The 64-bit addition result is first logically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction adds the 64-bit unsigned integer in Rs1 with the 64-bit unsigned integer Rs2. The 64-bit addition result is first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rt(4,1),1'b0); t_H = CONCAT(Rt(4,1),1'b1);
a_L = CONCAT(Ra(4,1),1'b0); a_H = CONCAT(Ra(4,1),1'b1);
b_L = CONCAT(Rb(4,1),1'b0); b_H = CONCAT(Rb(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] + R[b_H].R[b_L]) u>> 1;
* RV64:
Rd = (Rs1 + Rs2) u>> 1;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_URSUB64(unsigned long long a, unsigned lo

URSUB64 (64-bit Unsigned Halving Subtraction)

Type: DSP (64-bit Profile)

Syntax:

```
URSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit unsigned integer subtraction. The result is halved to avoid overflow or saturation.

RV32 Description: This instruction subtracts the 64-bit unsigned integer of an even/odd pair of registers specified by Rs2(4,1) from the 64-bit unsigned integer of an even/odd pair of registers specified by Rs1(4,1). The subtraction result is first logically right-shifted by 1 bit and then written to an even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction subtracts the 64-bit unsigned integer in Rs2 from the 64-bit unsigned integer in Rs1. The subtraction result is first logically right-shifted by 1 bit and then written to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rt(4,1),1'b0); t_H = CONCAT(Rt(4,1),1'b1);
a_L = CONCAT(Ra(4,1),1'b0); a_H = CONCAT(Ra(4,1),1'b1);
b_L = CONCAT(Rb(4,1),1'b0); b_H = CONCAT(Rb(4,1),1'b1);
R[t_H].R[t_L] = (R[a_H].R[a_L] - R[b_H].R[b_L]) u>> 1;
* RV64:
Rd = (Rs1 - Rs2) u>> 1;
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

32-bit Multiply with 64-bit Add/Subtract Instructions

__STATIC_FORCEINLINE long long __RV_KMAR64(long long t, long a, long b)

__STATIC_FORCEINLINE long long __RV_KMSR64(long long t, long a, long b)

__STATIC_FORCEINLINE long long __RV_SMAR64(long long t, long a, long b)

__STATIC_FORCEINLINE long long __RV_SMSR64(long long t, long a, long b)

__STATIC_FORCEINLINE unsigned long long __RV_UKMAR64(unsigned long long t, unsigned long a,

__STATIC_FORCEINLINE unsigned long long __RV_UKMSR64(unsigned long long t, unsigned long a,

__STATIC_FORCEINLINE unsigned long long __RV_UMAR64(unsigned long long t, unsigned long a,

__STATIC_FORCEINLINE unsigned long long __RV_UMSR64(unsigned long long t, unsigned long a,

group **NMSIS_Core_DSP_Intrinsic_32B_MULT_64B_ADDSUB**

32-bit Multiply with 64-bit Add/Subtract Instructions

there are 32-bit Multiply 64-bit Add/Subtract Instructions

Functions

`__STATIC_FORCEINLINE long long __RV_KMAR64(long long t, long a, long b)`

KMAR64 (Signed Multiply and Saturating Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
KMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and add the 64-bit multiplication results to the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is saturated to the Q63 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., value d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit signed data of Rd with unlimited precision. If the 64-bit addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

```
RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] + (Rs1 * Rs2);
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;
RV64:
// `result` has unlimited precision
result = Rd + (Rs1.W[0] * Rs2.W[0]) + (Rs1.W[1] * Rs2.W[1]);
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
Rd = result;
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE long long __RV_KMSR64(long long t, long a, long b)

KMSR64 (Signed Multiply and Saturating Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
KMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and subtract the 64-bit multiplication results from the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is saturated to the Q63 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit subtraction result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit signed data in Rd with unlimited precision. If the 64-bit subtraction result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

```
RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] - (Rs1 * Rs2);
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
R[t_H].R[t_L] = result;
RV64:
// `result` has unlimited precision
result = Rd - (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]);
if (result > (2^63)-1) {
  result = (2^63)-1; OV = 1;
} else if (result < -2^63) {
  result = -2^63; OV = 1;
}
Rd = result;
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMAR64(long long t, long a, long b)

SMAR64 (Signed Multiply and Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
SMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and add the 64-bit multiplication result to the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1). The addition result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit signed data of Rd. The addition result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] + (Rs1 * Rs2);
* RV64:
Rd = Rd + (Rs1.W[0] * Rs2.W[0]) + (Rs1.W[1] * Rs2.W[1]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMSR64(long long t, long a, long b)

SMSR64 (Signed Multiply and Subtract from 64- Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
SMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit signed elements in two registers and subtract the 64-bit multiplication results from the 64-bit signed data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit signed data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit signed data of an even/odd pair of registers specified by Rd(4,1). The subtraction result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit signed elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit signed data of Rd. The subtraction result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] - (Rs1 * Rs2);
* RV64:
Rd = Rd - (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: long type of value stored in a
- `[in] b`: long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UKMAR64(unsigned long long t, unsigned lo

UKMAR64 (Unsigned Multiply and Saturating Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
UKMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and add the 64-bit multiplication results to the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is saturated to the U64 range and written back to the pair of registers (RV32) or the register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit addition result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit unsigned data in Rd with unlimited precision. If the 64-bit addition result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] + (Rs1 * Rs2);
if (result > (2^64)-1) {
  result = (2^64)-1; OV = 1;
}
R[t_H].R[t_L] = result;
* RV64:
// `result` has unlimited precision
result = Rd + (Rs1.W[0] u* Rs2.W[0]) + (Rs1.W[1] u* Rs2.W[1]);
if (result > (2^64)-1) {
  result = (2^64)-1; OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
Rd = result;
```

Return value stored in unsigned long long type

Parameters

- `[in] t`: unsigned long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UKMSR64(unsigned long long t, unsigned lo

UKMSR64 (Unsigned Multiply and Saturating Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
UKMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and subtract the 64-bit multiplication results from the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is saturated to the U64 range and written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1) with unlimited precision. If the 64-bit subtraction result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit unsigned data of Rd with unlimited precision. If the 64-bit subtraction result is beyond the U64 number range (0 <= U64 <= 2^64-1), it is saturated to the range and the OV bit is set to 1. The saturated result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
result = R[t_H].R[t_L] - (Rs1 u* Rs2);
if (result < 0) {
  result = 0; OV = 1;
}
R[t_H].R[t_L] = result;
* RV64:
// `result` has unlimited precision
result = Rd - (Rs1.W[0] u* Rs2.W[0]) - (Rs1.W[1] u* Rs2.W[1]);
if (result < 0) {
  result = 0; OV = 1;
}
Rd = result;
```

Return value stored in unsigned long long type

Parameters

- `[in] t`: unsigned long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UMAR64(unsigned long long t, unsigned lon

UMAR64 (Unsigned Multiply and Add to 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
UMAR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and add the 64-bit multiplication results to the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It adds the 64-bit multiplication result to the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1). The addition result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It adds the 64-bit multiplication results to the 64-bit unsigned data of Rd. The addition result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] + (Rs1 * Rs2);
* RV64:
Rd = Rd + (Rs1.W[0] u* Rs2.W[0]) + (Rs1.W[1] u* Rs2.W[1]);
```

Return value stored in unsigned long long type

Parameters

- `[in] t`: unsigned long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_UMSR64(unsigned long long t, unsigned lon

UMSR64 (Unsigned Multiply and Subtract from 64-Bit Data)

Type: DSP (64-bit Profile)

Syntax:

```
UMSR64 Rd, Rs1, Rs2
```

Purpose: Multiply the 32-bit unsigned elements in two registers and subtract the 64-bit multiplication results from the 64-bit unsigned data of a pair of registers (RV32) or a register (RV64). The result is written back to the pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the 32-bit unsigned data of Rs1 with that of Rs2. It subtracts the 64-bit multiplication result from the 64-bit unsigned data of an even/odd pair of registers specified by Rd(4,1). The subtraction result is written back to the even/odd pair of registers specified by Rd(4,1). Rx(4,1), i.e., d, determines the even/odd pair group of two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: This instruction multiplies the 32-bit unsigned elements of Rs1 with that of Rs2. It subtracts the 64-bit multiplication results from the 64-bit unsigned data of Rd. The subtraction result is written back to Rd.

Operations:

```
* RV32:
t_L = CONCAT(Rd(4,1),1'b0); t_H = CONCAT(Rd(4,1),1'b1);
R[t_H].R[t_L] = R[t_H].R[t_L] - (Rs1 * Rs2);
* RV64:
Rd = Rd - (Rs1.W[0] u* Rs2.W[0]) - (Rs1.W[1] u* Rs2.W[1]);
```

Return value stored in unsigned long long type

Parameters

- `[in] t`: unsigned long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

Signed 16-bit Multiply 64-bit Add/Subtract Instructions

__STATIC_FORCEINLINE long long __RV_SMAL(long long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALBB(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALBT(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALTT(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALDA(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALXDA(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALDS(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALDRS(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMALXDS(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMSLDA(long long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long long __RV_SMSLXDA(long long t, unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_SIGNED_16B_MULT_64B_ADDSUB**

Signed 16-bit Multiply 64-bit Add/Subtract Instructions.

Signed 16-bit Multiply with 64-bit Add/Subtract Instructions.

there is Signed 16-bit Multiply 64-bit Add/Subtract Instructions

there are 10 Signed 16-bit Multiply with 64-bit Add/Subtract Instructions

Functions

__STATIC_FORCEINLINE long long __RV_SMAL(long long a, unsigned long b)

SMAL (Signed Multiply Halfs & Add 64-bit)

Type: Partial-SIMD

Syntax:

```
SMAL Rd, Rs1, Rs2
```

Purpose: Multiply the signed bottom 16-bit content of the 32-bit elements of a register with the top 16-bit content of the same 32-bit elements of the same register, and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to another even/odd pair of registers (RV32) or a register (RV64).

RV32 Description: This instruction multiplies the bottom 16-bit content of the lower 32-bit of Rs2 with the top 16-bit content of the lower 32-bit of Rs2 and adds the result with the 64-bit value of an even/odd pair of registers specified by Rs1(4,1). The 64-bit addition result is written back to an even/odd pair of registers specified by Rd(4,1). The 16-bit values of Rs2, and the 64-bit value of the Rs1(4,1) register- pair are treated as signed integers. Rx(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: This instruction multiplies the bottom 16-bit content of the 32-bit elements of Rs2 with the top 16-bit content of the same 32-bit elements of Rs2 and adds the results with the 64-bit value of Rs1. The 64- bit addition result is written back to Rd. The 16-bit values of Rs2, and the 64-bit value of Rs1 are treated as signed integers.

Operations:

```
RV32:
Mres[31:0] = Rs2.H[1] * Rs2.H[0];
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs1(4,1),1'b1); +
Idx2 = CONCAT(Rd(4,1),1'b0); Idx3 = CONCAT(Rd(4,1),1'b1);
R[Idx3].R[Idx2] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
RV64:
Mres[0][31:0] = Rs2.W[0].H[1] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs2.W[1].H[1] * Rs2.W[1].H[0];
Rd = Rs1 + SE64(Mres[1][31:0]) + SE64(Mres[0][31:0]);
```

Return value stored in long long type

Parameters

- `[in] a`: long long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMALBB(long long t, unsigned long a, unsigned long

SMALBB (Signed Multiply Bottom Halfs & Add 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALBB Rd, Rs1, Rs2
SMALBT Rd, Rs1, Rs2
SMALTT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

- SMALBB: rt pair + bottom*bottom (all 32-bit elements)
- SMALBT rt pair + bottom*top (all 32-bit elements)
- SMALTT rt pair + top*top (all 32-bit elements)

RV32 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2. The multiplication result is added with the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
RV32:
Mres[31:0] = Rs1.H[0] * Rs2.H[0]; // SMALBB
Mres[31:0] = Rs1.H[0] * Rs2.H[1]; // SMALBT
Mres[31:0] = Rs1.H[1] * Rs2.H[1]; // SMALTT
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
RV64:
// SMALBB
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[0];
// SMALBT
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[1];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[1];
// SMALTT
Mres[0][31:0] = Rs1.W[0].H[1] * Rs2.W[0].H[1];
Mres[1][31:0] = Rs1.W[1].H[1] * Rs2.W[1].H[1];
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long long __RV_SMALBT(long long t, unsigned long a, unsigned long`

SMALBT (Signed Multiply Bottom Half & Top Half & Add 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALBB Rd, Rs1, Rs2
SMALBT Rd, Rs1, Rs2
SMALTT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

- SMALBB: rt pair + bottom*bottom (all 32-bit elements)
- SMALBT rt pair + bottom*top (all 32-bit elements)
- SMALTT rt pair + top*top (all 32-bit elements)

RV32 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2. The multiplication result is added with the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
RV32:
Mres[31:0] = Rs1.H[0] * Rs2.H[0]; // SMALBB
Mres[31:0] = Rs1.H[0] * Rs2.H[1]; // SMALBT
Mres[31:0] = Rs1.H[1] * Rs2.H[1]; // SMALTT
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
RV64:
// SMALBB
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[0];
// SMALBT
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[1];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[1];
// SMALTT
Mres[0][31:0] = Rs1.W[0].H[1] * Rs2.W[0].H[1];
Mres[1][31:0] = Rs1.W[1].H[1] * Rs2.W[1].H[1];
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMALTT(long long t, unsigned long a, unsigned long

SMALTT (Signed Multiply Top Halfs & Add 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALBB Rd, Rs1, Rs2
SMALBT Rd, Rs1, Rs2
SMALTT Rd, Rs1, Rs2
```

Purpose: Multiply the signed 16-bit content of the 32-bit elements of a register with the 16-bit content of the corresponding 32-bit elements of another register and add the results with a 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair (RV32) or the register (RV64).

- SMALBB: rt pair + bottom*bottom (all 32-bit elements)
- SMALBT rt pair + bottom*top (all 32-bit elements)
- SMALTT rt pair + top*top (all 32-bit elements)

RV32 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2. The multiplication result is added with the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALBB` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALBT` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALTT` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The multiplication results are added with the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
RV32:
Mres[31:0] = Rs1.H[0] * Rs2.H[0]; // SMALBB
Mres[31:0] = Rs1.H[0] * Rs2.H[1]; // SMALBT
Mres[31:0] = Rs1.H[1] * Rs2.H[1]; // SMALTT
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
RV64:
// SMALBB
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[0];
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[0];
// SMALBT
Mres[0][31:0] = Rs1.W[0].H[0] * Rs2.W[0].H[1];
```

(continues on next page)

(continued from previous page)

```
Mres[1][31:0] = Rs1.W[1].H[0] * Rs2.W[1].H[1];
// SMALTT
Mres[0][31:0] = Rs1.W[0].H[1] * Rs2.W[0].H[1];
Mres[1][31:0] = Rs1.W[1].H[1] * Rs2.W[1].H[1];
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMALDA(long long t, unsigned long a, unsigned long

SMALDA (Signed Multiply Two Halfs and Two Adds 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALDA Rd, Rs1, Rs2
SMALXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results and the 64-bit value of an even/odd pair of registers together.

- SMALDA: rt pair+ top*top + bottom*bottom (all 32-bit elements)
- SMALXDA: rt pair+ top*bottom + bottom*top (all 32-bit elements)

RV32 Description: For the `SMALDA` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision. For the `SMALXDA` instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision. The result is added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64- bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64- bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32- bit elements of Rs2 with unlimited precision. For the `SMALXDA` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 with unlimited precision. The results are added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
RV32:
// SMALDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
// SMALXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres0[31:0]) + SE64(Mres1[31:0]);
RV64:
// SMALDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
// SMALXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd + SE64(Mres0[0][31:0]) + SE64(Mres1[0][31:0]) + SE64(Mres0[1][31:0]) +
SE64(Mres1[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMALXDA(long long t, unsigned long a, unsigned lon

SMALXDA (Signed Crossed Multiply Two Halfs and Two Adds 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALDA Rd, Rs1, Rs2
SMALXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then adds the two 32-bit results and the 64-bit value of an even/odd pair of registers together.

- SMALDA: rt pair+ top*top + bottom*bottom (all 32-bit elements)
- SMALXDA: rt pair+ top*bottom + bottom*top (all 32-bit elements)

RV32 Description: For the `SMALDA` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision. For the `SMALXDA` instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 with unlimited precision. The result is added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64- bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64- bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32- bit elements of Rs2 with unlimited precision. For the `SMALXDA` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 with unlimited precision. The results are added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
RV32:
// SMALDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
// SMALXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres0[31:0]) + SE64(Mres1[31:0]);
RV64:
// SMALDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
// SMALXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd + SE64(Mres0[0][31:0]) + SE64(Mres1[0][31:0]) + SE64(Mres0[1][31:0]) +
SE64(Mres1[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long long __RV_SMALDS(long long t, unsigned long a, unsigned long`

SMALDS (Signed Multiply Two Halfs & Subtract & Add 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALDS Rd, Rs1, Rs2
SMALDRS Rd, Rs1, Rs2
SMALXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair.

- SMALDS: rt pair + (top*top - bottom*bottom) (all 32-bit elements)
- SMALDRS: rt pair + (bottom*bottom - top*top) (all 32-bit elements)
- SMALXDS: rt pair + (top*bottom - bottom*top) (all 32-bit elements)

RV32 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALXDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2. The subtraction result is then added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALXDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction results are then added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
* RV32:
Mres[31:0] = (Rs1.H[1] * Rs2.H[1]) - (Rs1.H[0] * Rs2.H[0]); // SMALDS
Mres[31:0] = (Rs1.H[0] * Rs2.H[0]) - (Rs1.H[1] * Rs2.H[1]); // SMALDRS
Mres[31:0] = (Rs1.H[1] * Rs2.H[0]) - (Rs1.H[0] * Rs2.H[1]); // SMALXDS
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
* RV64:
// SMALDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[0]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[1]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[0]);
// SMALDRS
Mres[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]) - (Rs1.W[0].H[1] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[0].H[0]) - (Rs1.W[1].H[1] * Rs2.W[1].
↪H[1]);
// SMALXDS
```

(continues on next page)

(continued from previous page)

```
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[0]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[1]);
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMALDRS(long long t, unsigned long a, unsigned lon

SMALDRS (Signed Multiply Two Halfs & Reverse Subtract & Add 64- bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALDS Rd, Rs1, Rs2
SMALDRS Rd, Rs1, Rs2
SMALXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair.

- SMALDS: rt pair + (top*top - bottom*bottom) (all 32-bit elements)
- SMALDRS: rt pair + (bottom*bottom - top*top) (all 32-bit elements)
- SMALXDS: rt pair + (top*bottom - bottom*top) (all 32-bit elements)

RV32 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALXDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2. The subtraction result is then added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALXDS` instruction, it multiplies the bottom

16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction results are then added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
* RV32:
Mres[31:0] = (Rs1.H[1] * Rs2.H[1]) - (Rs1.H[0] * Rs2.H[0]); // SMALDS
Mres[31:0] = (Rs1.H[0] * Rs2.H[0]) - (Rs1.H[1] * Rs2.H[1]); // SMALDRS
Mres[31:0] = (Rs1.H[1] * Rs2.H[0]) - (Rs1.H[0] * Rs2.H[1]); // SMALXDS
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
* RV64:
// SMALDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[0]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[1]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[0]);
// SMALDRS
Mres[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]) - (Rs1.W[0].H[1] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[0].H[0]) - (Rs1.W[1].H[1] * Rs2.W[1].
↪H[1]);
// SMALXDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[0]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[1]);
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long long __RV_SMALXDS(long long t, unsigned long a, unsigned lon`

SMALXDS (Signed Crossed Multiply Two Halfs & Subtract & Add 64- bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMALDS Rd, Rs1, Rs2
SMALDRS Rd, Rs1, Rs2
SMALXDS Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then perform a subtraction operation between the two 32-bit results. Then add the subtraction result to the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The addition result is written back to the register-pair.

- SMALDS: rt pair + (top*top - bottom*bottom) (all 32-bit elements)
- SMALDRS: rt pair + (bottom*bottom - top*top) (all 32-bit elements)

- SMALXDS: rt pair + (top*bottom - bottom*top) (all 32-bit elements)

RV32 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of Rs1 with the bottom 16-bit content of Rs2. For the `SMALXDS` instruction, it multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2. The subtraction result is then added to the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit addition result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the operand and the even `2d` register of the pair contains the low 32-bit of the operand.

RV64 Description: For the `SMALDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMALDRS` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. For the `SMALXDS` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2 and then subtracts the result from the result of multiplying the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2. The subtraction results are then added to the 64-bit value of Rd. The 64-bit addition result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
* RV32:
Mres[31:0] = (Rs1.H[1] * Rs2.H[1]) - (Rs1.H[0] * Rs2.H[0]); // SMALDS
Mres[31:0] = (Rs1.H[0] * Rs2.H[0]) - (Rs1.H[1] * Rs2.H[1]); // SMALDRS
Mres[31:0] = (Rs1.H[1] * Rs2.H[0]) - (Rs1.H[0] * Rs2.H[1]); // SMALXDS
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
* RV64:
// SMALDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[0]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[1]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[0]);
// SMALDRS
Mres[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]) - (Rs1.W[0].H[1] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[0].H[0]) - (Rs1.W[1].H[1] * Rs2.W[1].
↪H[1]);
// SMALXDS
Mres[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]) - (Rs1.W[0].H[0] * Rs2.W[0].
↪H[1]);
Mres[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[0].H[0]) - (Rs1.W[1].H[0] * Rs2.W[1].
↪H[1]);
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMSLDA(long long t, unsigned long a, unsigned long

SMSLDA (Signed Multiply Two Halfs & Add & Subtract 64-bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMSLDA Rd, Rs1, Rs2
SMSLXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The subtraction result is written back to the register-pair.

- SMSLDA: rd pair - top*top - bottom*bottom (all 32-bit elements)
- SMSLXDA: rd pair - top*bottom - bottom*top (all 32-bit elements)

RV32 Description: For the `SMSLDA` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content Rs2 and multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMSLXDA` instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2. The two multiplication results are subtracted from the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit subtraction result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: For the `SMSLDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMSLXDA` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The four multiplication results are subtracted from the 64-bit value of Rd. The 64-bit subtraction result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
* RV32:
// SMSLDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
// SMSLXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] - SE64(Mres0[31:0]) - SE64(Mres1[31:0]);
* RV64:
// SMSLDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
```

(continues on next page)

(continued from previous page)

```
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
// SMSLXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd - SE64(Mres0[0][31:0]) - SE64(Mres1[0][31:0]) - SE64(Mres0[1][31:0]) -
SE64(Mres1[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long long __RV_SMSLXDA(long long t, unsigned long a, unsigned long

SMSLXDA (Signed Crossed Multiply Two Halfs & Add & Subtract 64- bit)

Type: DSP (64-bit Profile)

Syntax:

```
SMSLDA Rd, Rs1, Rs2
SMSLXDA Rd, Rs1, Rs2
```

Purpose: Do two signed 16-bit multiplications from the 32-bit elements of two registers; and then subtracts the two 32-bit results from the 64-bit value of an even/odd pair of registers (RV32) or a register (RV64). The subtraction result is written back to the register-pair.

- SMSLDA: rd pair - top*top - bottom*bottom (all 32-bit elements)
- SMSLXDA: rd pair - top*bottom - bottom*top (all 32-bit elements)

RV32 Description: For the `SMSLDA` instruction, it multiplies the bottom 16-bit content of Rs1 with the bottom 16-bit content Rs2 and multiplies the top 16-bit content of Rs1 with the top 16-bit content of Rs2. For the `SMSLXDA` instruction, it multiplies the top 16-bit content of Rs1 with the bottom 16-bit content of Rs2 and multiplies the bottom 16-bit content of Rs1 with the top 16-bit content of Rs2. The two multiplication results are subtracted from the 64-bit value of an even/odd pair of registers specified by Rd(4,1). The 64-bit subtraction result is written back to the register-pair. The 16-bit values of Rs1 and Rs2, and the 64-bit value of the register-pair are treated as signed integers. Rd(4,1), i.e., d, determines the even/odd pair group of the two registers. Specifically, the register pair includes register 2d and 2d+1. The odd `2d+1` register of the pair contains the high 32-bit of the result and the even `2d` register of the pair contains the low 32-bit of the result.

RV64 Description: For the `SMSLDA` instruction, it multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the top 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. For the `SMSLXDA` instruction, it multiplies the top 16-bit content of the 32-bit elements of Rs1 with the bottom 16-bit content of the 32-bit elements of Rs2 and multiplies the bottom 16-bit content of the 32-bit elements of Rs1 with the top 16-bit content of the 32-bit elements of Rs2. The four multiplication results are subtracted from the 64-bit value of Rd. The 64-bit subtraction result is written back to Rd. The 16-bit values of Rs1 and Rs2, and the 64-bit value of Rd are treated as signed integers.

Operations:

```
* RV32:
// SMSLDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[1]);
// SMSLXDA
Mres0[31:0] = (Rs1.H[0] * Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] * Rs2.H[0]);
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] - SE64(Mres0[31:0]) - SE64(Mres1[31:0]);
* RV64:
// SMSLDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[1]);
// SMSLXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] * Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] * Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] * Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] * Rs2.W[1].H[0]);
Rd = Rd - SE64(Mres0[0][31:0]) - SE64(Mres1[0][31:0]) - SE64(Mres0[1][31:0]) -
SE64(Mres1[1][31:0]);
```

Return value stored in long long type

Parameters

- `[in] t`: long long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

group **NMSIS_Core_DSP_Intrinsic_64B_PROFILE**
64-bit Profile Instructions

RV64 Only Instructions

(RV64 Only) SIMD 32-bit Add/Subtract Instructions

__STATIC_FORCEINLINE unsigned long __RV_ADD32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_CRAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_CRSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KADD32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KCRAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KCRSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KSTAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KSTSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KSUB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RADD32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RCRAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RCRSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RSTAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RSTSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_RSUB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_STAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_STSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SUB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKADD32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKCRAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKCRSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKSTAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKSTSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UKSUB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URADD32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URCRAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URCRSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URSTAS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URSTSA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_URSUB32(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_SIMD_32B_ADDSUB**

(RV64 Only) SIMD 32-bit Add/Subtract Instructions

The following tables list instructions that are only present in RV64. There are 30 SIMD 32-bit addition or subtraction instructions.there are 4 SIMD16-bit Packing Instructions.

Functions

__STATIC_FORCEINLINE unsigned long __RV_ADD32(unsigned long a, unsigned long b)

ADD32 (SIMD 32-bit Addition)

Type: SIMD (RV64 Only)

Syntax:

```
ADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element additions simultaneously.

Description: This instruction adds the 32-bit integer elements in Rs1 with the 32-bit integer elements in Rs2, and then writes the 32-bit element results to Rd.

Note: This instruction can be used for either signed or unsigned addition.

Operations:

```
Rd.W[x] = Rs1.W[x] + Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_CRAS32(unsigned long a, unsigned long b)

CRAS32 (SIMD 32-bit Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
CRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [63:32] of Rd; at the same time, it subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[1] = Rs1.W[1] + Rs2.W[0];
Rd.W[0] = Rs1.W[0] - Rs2.W[1];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_CRSA32(unsigned long a, unsigned long b)

CRSA32 (SIMD 32-bit Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
CRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. *Description: * This instruction subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2, and writes the result to [31:0] of Rd

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[1] = Rs1.W[1] - Rs2.W[0];
Rd.W[0] = Rs1.W[0] + Rs2.W[1];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KADD32(unsigned long a, unsigned long b)

KADD32 (SIMD 32-bit Signed Saturating Addition)

Type: SIMD (RV64 Only)

Syntax:

```
KADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] + Rs2.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KCRAS32(unsigned long a, unsigned long b)

KCRAS32 (SIMD 32-bit Signed Saturating Cross Addition & Subtraction)

Type: SIM (RV64 Only)

Syntax:

```
KCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [31:0] of Rs2; at the same time, it subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [31:0] of Rs1. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[0];
res[0] = Rs1.W[0] - Rs2.W[1];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KCRSA32(unsigned long a, unsigned long b)

KCRSA32 (SIMD 32-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
KCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. *Description: * This instruction subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [63:32] of Rs2. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[0];
res[0] = Rs1.W[0] + Rs2.W[1];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSTAS32(unsigned long a, unsigned long b)

KSTAS32 (SIMD 32-bit Signed Saturating Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
KSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [63:32] of Rs2; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [31:0] of Rs1. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res[1] = Rs1.W[1] + Rs2.W[1];
res[0] = Rs1.W[0] - Rs2.W[0];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSTSA32(unsigned long a, unsigned long b)

KSTSA32 (SIMD 32-bit Signed Saturating Straight Subtraction & Addition)

Type: SIM (RV64 Only)

Syntax:

```
KSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. *Description: * This instruction subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2. If any of the results are beyond the Q31 number range (- 231 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res[1] = Rs1.W[1] - Rs2.W[1];
res[0] = Rs1.W[0] + Rs2.W[0];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[1] = res[1];
Rd.W[0] = res[0];
for RV64, x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KSUB32(unsigned long a, unsigned long b)`

KSUB32 (SIMD 32-bit Signed Saturating Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
KSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. If any of the results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] - Rs2.W[x];
if (res[x] > (2^31)-1) {
  res[x] = (2^31)-1;
  OV = 1;
} else if (res[x] < -2^31) {
  res[x] = -2^31;
  OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_RADD32(unsigned long a, unsigned long b)`

RADD32 (SIMD 32-bit Signed Halving Addition)

Type: SIMD (RV64 Only)

Syntax:

```
RADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Rs1 = 0x7FFFFFFF, Rs2 = 0x7FFFFFFF Rd = 0x7FFFFFFF
* Rs1 = 0x80000000, Rs2 = 0x80000000 Rd = 0x80000000
* Rs1 = 0x40000000, Rs2 = 0x80000000 Rd = 0xE0000000
```

Operations:

```
Rd.W[x] = (Rs1.W[x] + Rs2.W[x]) s>> 1;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RCRAS32(unsigned long a, unsigned long b)

RCRAS32 (SIMD 32-bit Signed Halving Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
RCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2, and subtracts the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Examples:

```
Please see `RADD32` and `RSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[0]) s>> 1;
Rd.W[0] = (Rs1.W[0] - Rs2.W[1]) s>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RCRSA32(unsigned long a, unsigned long b)

RCRSA32 (SIMD 32-bit Signed Halving Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
RCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer element in [31:0] of Rs2 from the 32-bit signed integer element in [63:32] of Rs1, and adds the 32-bit signed element integer in [31:0] of Rs1 with the 32-bit signed integer element in [63:32] of Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Examples:

```
Please see `RADD32` and `RSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[0]) s>> 1;
Rd.W[0] = (Rs1.W[0] + Rs2.W[1]) s>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSTAS32(unsigned long a, unsigned long b)

RSTAS32 (SIMD 32-bit Signed Halving Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
RSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit signed integer element in [63:32] of Rs1 with the 32-bit signed integer element in [63:32] of Rs2, and subtracts the 32-bit signed integer element in [31:0] of Rs2 from the 32-bit signed integer element in [31:0] of Rs1. The element results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Examples:

```
Please see `RADD32` and `RSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[1]) s>> 1;
Rd.W[0] = (Rs1.W[0] - Rs2.W[0]) s>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSTSA32(unsigned long a, unsigned long b)

RSTSA32 (SIMD 32-bit Signed Halving Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
RSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer element in [63:32] of Rs2 from the 32-bit signed integer element in [63:32] of Rs1, and adds the 32-bit signed element integer in [31:0] of Rs1 with the 32-bit signed integer element in [31:0] of Rs2. The two results are first arithmetically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Examples:

```
Please see `RADD32` and `RSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[1]) s>> 1;
Rd.W[0] = (Rs1.W[0] + Rs2.W[0]) s>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_RSUB32(unsigned long a, unsigned long b)

RSUB32 (SIMD 32-bit Signed Halving Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
RSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit signed integer elements in Rs2 from the 32-bit signed integer elements in Rs1. The results are first arithmetically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFFFFFF, Rb = 0x80000000 Rt = 0x7FFFFFFF
* Ra = 0x80000000, Rb = 0x7FFFFFFF Rt = 0x80000000
* Ra = 0x80000000, Rb = 0x40000000 Rt = 0xA0000000
```

Operations:

```
Rd.W[x] = (Rs1.W[x] - Rs2.W[x]) s>> 1;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_STAS32(unsigned long a, unsigned long b)

STAS32 (SIMD 32-bit Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
STAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit integer element in [63:32] of Rs1 with the 32-bit integer element in [63:32] of Rs2, and writes the result to [63:32] of Rd; at the same time, it subtracts the 32-bit integer element in [31:0] of Rs2 from the 32-bit integer element in [31:0] of Rs1, and writes the result to [31:0] of Rd.

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[1] = Rs1.W[1] + Rs2.W[1];
Rd.W[0] = Rs1.W[0] - Rs2.W[0];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_STSA32(unsigned long a, unsigned long b)

STSA32 (SIMD 32-bit Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
STSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. *Description: * This instruction subtracts the 32-bit integer element in [63:32] of Rs2 from the 32-bit integer element in [63:32] of Rs1, and writes the result to [63:32] of Rd; at the same time, it adds the 32-bit integer element in [31:0] of Rs1 with the 32-bit integer element in [31:0] of Rs2, and writes the result to [31:0] of Rd

Note: This instruction can be used for either signed or unsigned operations.

Operations:

```
Rd.W[1] = Rs1.W[1] - Rs2.W[1];
Rd.W[0] = Rs1.W[0] + Rs2.W[0];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SUB32(unsigned long a, unsigned long b)

SUB32 (SIMD 32-bit Subtraction)

Type: DSP (RV64 Only)

Syntax:

```
SUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit integer element subtractions simultaneously.

Description: This instruction subtracts the 32-bit integer elements in Rs2 from the 32-bit integer elements in Rs1, and then writes the results to Rd.

Note: This instruction can be used for either signed or unsigned subtraction.

Operations:

```
Rd.W[x] = Rs1.W[x] - Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKADD32(unsigned long a, unsigned long b)

UKADD32 (SIMD 32-bit Unsigned Saturating Addition)

Type: SIMD (RV64 Only)

Syntax:

```
UKADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element saturating additions simultaneously.

Description: This instruction adds the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] + Rs2.W[x];
if (res[x] > (2^32)-1) {
  res[x] = (2^32)-1;
  OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKCRAS32(unsigned long a, unsigned long b)

UKCRAS32 (SIMD 32-bit Unsigned Saturating Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
UKCRAS32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32- bit unsigned integer element in [31:0] of Rs2; at the same time, it subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [31:0] Rs1. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res1 = Rs1.W[1] + Rs2.W[0];
res2 = Rs1.W[0] - Rs2.W[1];
if (res1 > (2^32)-1) {
  res1 = (2^32)-1;
  OV = 1;
}
if (res2 < 0) {
  res2 = 0;
  OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKCRSA32(unsigned long a, unsigned long b)

UKCRSA32 (SIMD 32-bit Unsigned Saturating Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
UKCRSA32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements.

Description: This instruction subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1; at the same time, it adds the 32-bit unsigned integer element in [63:32] of Rs2 with the 32-bit unsigned integer element in [31:0] Rs1. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res1 = Rs1.W[1] - Rs2.W[0];
res2 = Rs1.W[0] + Rs2.W[1];
if (res1 < 0) {
  res1 = 0;
  OV = 1;
} else if (res2 > (2^32)-1) {
  res2 = (2^32)-1;
  OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKSTAS32(unsigned long a, unsigned long b)

UKSTAS32 (SIMD 32-bit Unsigned Saturating Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
UKSTAS32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32-bit unsigned integer element in [63:32] of Rs2; at the same time, it subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [31:0] Rs1. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res1 = Rs1.W[1] + Rs2.W[1];
res2 = Rs1.W[0] - Rs2.W[0];
if (res1 > (2^32)-1) {
```

(continues on next page)

(continued from previous page)

```
  res1 = (2^32)-1;
  OV = 1;
}
if (res2 < 0) {
  res2 = 0;
  OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_UKSTSA32(unsigned long a, unsigned long b)`

UKSTSA32 (SIMD 32-bit Unsigned Saturating Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
UKSTSA32 Rd, Rs1, Rs2
```

Purpose: Do one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements.

Description: This instruction subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1; at the same time, it adds the 32-bit unsigned integer element in [31:0] of Rs2 with the 32-bit unsigned integer element in [31:0] Rs1. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res1 = Rs1.W[1] - Rs2.W[1];
res2 = Rs1.W[0] + Rs2.W[0];
if (res1 < 0) {
  res1 = 0;
  OV = 1;
} else if (res2 > (2^32)-1) {
  res2 = (2^32)-1;
  OV = 1;
}
Rd.W[1] = res1;
Rd.W[0] = res2;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UKSUB32(unsigned long a, unsigned long b)
UKSUB32 (SIMD 32-bit Unsigned Saturating Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
UKSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 32-bit unsigned integer elements in Rs2 from the 32-bit unsigned integer elements in Rs1. If any of the results are beyond the 32-bit unsigned number range (0 <= RES <= 2^32-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.W[x] - Rs2.W[x];
if (res[x] < 0) {
  res[x] = 0;
  OV = 1;
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URADD32(unsigned long a, unsigned long b)
URADD32 (SIMD 32-bit Unsigned Halving Addition)

Type: SIMD (RV64 Only)

Syntax:

```
URADD32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element additions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFFFFFF, Rb = 0x7FFFFFFF Rt = 0x7FFFFFFF
* Ra = 0x80000000, Rb = 0x80000000 Rt = 0x80000000
* Ra = 0x40000000, Rb = 0x80000000 Rt = 0x60000000
```

Operations:

```
Rd.W[x] = (Rs1.W[x] + Rs2.W[x]) u>> 1;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URCRAS32(unsigned long a, unsigned long b)

URCRAS32 (SIMD 32-bit Unsigned Halving Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
URCRAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32- bit unsigned integer element in [31:0] of Rs2, and subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. The element results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Examples:

```
Please see `URADD32` and `URSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[0]) u>> 1;
Rd.W[0] = (Rs1.W[0] - Rs2.W[1]) u>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URCRSA32(unsigned long a, unsigned long b)

URCRSA32 (SIMD 32-bit Unsigned Halving Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
URCRSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk simultaneously. Operands are from crossed 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1, and adds the 32-bit unsigned element integer in [31:0] of Rs1 with the 32-bit unsigned integer element in [63:32] of Rs2. The two results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Examples:

```
Please see `URADD32` and `URSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[0]) u>> 1;
Rd.W[0] = (Rs1.W[0] + Rs2.W[1]) u>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSTAS32(unsigned long a, unsigned long b)

URSTAS32 (SIMD 32-bit Unsigned Halving Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
URSTAS32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction adds the 32-bit unsigned integer element in [63:32] of Rs1 with the 32- bit unsigned integer element in [63:32] of Rs2, and subtracts the 32-bit unsigned integer element in [31:0] of Rs2 from the 32-bit unsigned integer element in [31:0] of Rs1. The element results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Examples:

```
Please see `URADD32` and `URSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] + Rs2.W[1]) u>> 1;
Rd.W[0] = (Rs1.W[0] - Rs2.W[0]) u>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_URSTSA32(unsigned long a, unsigned long b)

URSTSA32 (SIMD 32-bit Unsigned Halving Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Syntax:

```
URSTSA32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk simultaneously. Operands are from corresponding 32-bit elements. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer element in [63:32] of Rs2 from the 32-bit unsigned integer element in [63:32] of Rs1, and adds the 32-bit unsigned element integer in [31:0] of Rs1 with the 32-bit unsigned integer element in [31:0] of Rs2. The two results are first logically right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Examples:

```
Please see `URADD32` and `URSUB32` instructions.
```

Operations:

```
Rd.W[1] = (Rs1.W[1] - Rs2.W[1]) u>> 1;
Rd.W[0] = (Rs1.W[0] + Rs2.W[0]) u>> 1;
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_URSUB32(unsigned long a, unsigned long b)`

URSUB32 (SIMD 32-bit Unsigned Halving Subtraction)

Type: SIMD (RV64 Only)

Syntax:

```
URSUB32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer element subtractions simultaneously. The results are halved to avoid overflow or saturation.

Description: This instruction subtracts the 32-bit unsigned integer elements in Rs2 from the 32-bit unsigned integer elements in Rs1. The results are first logically right-shifted by 1 bit and then written to Rd.

Examples:

```
* Ra = 0x7FFFFFFF, Rb = 0x80000000, Rt = 0xFFFFFFFF
* Ra = 0x80000000, Rb = 0x7FFFFFFF, Rt = 0x00000000
* Ra = 0x80000000, Rb = 0x40000000, Rt = 0x20000000
```

Operations:

```
Rd.W[x] = (Rs1.W[x] - Rs2.W[x]) u>> 1;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) SIMD 32-bit Shift Instructions

__STATIC_FORCEINLINE unsigned long __RV_KSLL32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_KSLLI32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_KSLRA32(unsigned long a, int b)

__STATIC_FORCEINLINE unsigned long __RV_KSLRA32_U(unsigned long a, int b)

__STATIC_FORCEINLINE unsigned long __RV_SLL32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SLLI32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRA32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRA32_U(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRAI32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRAI32_U(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRL32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRL32_U(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRLI32(unsigned long a, unsigned int b)

__STATIC_FORCEINLINE unsigned long __RV_SRLI32_U(unsigned long a, unsigned int b)

group **NMSIS_Core_DSP_Intrinsic_RV64_SIMD_32B_SHIFT**

(RV64 Only) SIMD 32-bit Shift Instructions

there are 14 (RV64 Only) SIMD 32-bit Shift Instructions

Functions

__STATIC_FORCEINLINE unsigned long __RV_KSLL32(unsigned long a, unsigned int b)

KSLL32 (SIMD 32-bit Saturating Shift Left Logical)

Type: SIMD (RV64 Only)

Syntax:

```
KSLL32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left shift operations with saturation simultaneously. The shift amount is a variable from a GPR.

Description: The 32-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register. Any shifted value greater than 2^31-1 is saturated to 2^31-1. Any shifted value smaller than -2^31 is saturated to -2^31. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = Rs2[4:0];
if (sa != 0) {
  res[(31+sa):0] = Rs1.W[x] << sa;
  if (res > (2^31)-1) {
    res = 0x7fffffff; OV = 1;
  } else if (res < -2^31) {
    res = 0x80000000; OV = 1;
```

(continues on next page)

(continued from previous page)

```
  }
  Rd.W[x] = res[31:0];
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSLLI32(unsigned long a, unsigned int b)

KSLLI32 (SIMD 32-bit Saturating Shift Left Logical Immediate)

Type: SIMD (RV64 Only)

Syntax:

```
KSLLI32 Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements logical left shift operations with saturation simultaneously. The shift amount is an immediate value.

Description: The 32-bit data elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant. Any shifted value greater than 2^31-1 is saturated to 2^31-1. Any shifted value smaller than -2^31 is saturated to -2^31. And the saturated results are written to Rd. If any saturation is performed, set OV bit to 1.

Operations:

```
sa = imm5u[4:0];
if (sa != 0) {
  res[(31+sa):0] = Rs1.W[x] << sa;
  if (res > (2^31)-1) {
    res = 0x7fffffff; OV = 1;
  } else if (res < -2^31) {
    res = 0x80000000; OV = 1;
  }
  Rd.W[x] = res[31:0];
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KSLRA32(unsigned long a, int b)

KSLRA32 (SIMD 32-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Syntax:

```
KSLRA32 Rd, Rs1, Rs2
KSLRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 32-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of [-25, 25-1]. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0]. However, the behavior of `Rs2[5:0]==-25 (0x20)` is defined to be equivalent to the behavior of `Rs2[5:0]==-(25-1) (0x21)`. The left-shifted results are saturated to the 32-bit signed integer range of [-2^31, 2^31-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect this instruction.

Operations:

```
if (Rs2[5:0] < 0) {
  sa = -Rs2[5:0];
  sa = (sa == 32)? 31 : sa;
  if (`.u` form) {
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  } else {
    Rd.W[x] = SE32(Rs1.W[x][31:sa]);
  }
} else {
  sa = Rs2[4:0];
  res[(31+sa):0] = Rs1.W[x] <<(logic) sa;
  if (res > (2^31)-1) {
    res[31:0] = 0x7fffffff; OV = 1;
  } else if (res < -2^31) {
    res[31:0] = 0x80000000; OV = 1;
  }
  Rd.W[x] = res[31:0];
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KSLRA32_U(unsigned long a, int b)`

KSLRA32.u (SIMD 32-bit Shift Left Logical with Saturation or Rounding Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Syntax:

```
KSLRA32 Rd, Rs1, Rs2
KSLRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q31 saturation for the left shift. The `.u` form performs additional rounding up operations for the right shift.

Description: The 32-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[5:0]. Rs2[5:0] is in the signed range of [-25, 25-1]. A positive Rs2[5:0] means logical left shift and a negative Rs2[5:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[5:0]. However, the behavior of `Rs2[5:0]==-25 (0x20)` is defined to be equivalent to the behavior of `Rs2[5:0]==-(25-1) (0x21)`. The left-shifted results are saturated to the 32-bit signed integer range of [-2^31, 2^31-1]. For the `.u` form of the instruction, the right-shifted results are added a 1 to the most significant discarded bit position for rounding effect. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:6] will not affect this instruction.

Operations:

```
if (Rs2[5:0] < 0) {
  sa = -Rs2[5:0];
  sa = (sa == 32)? 31 : sa;
  if (`.u` form) {
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  } else {
    Rd.W[x] = SE32(Rs1.W[x][31:sa]);
  }
} else {
  sa = Rs2[4:0];
  res[(31+sa):0] = Rs1.W[x] <<(logic) sa;
  if (res > (2^31)-1) {
    res[31:0] = 0x7fffffff; OV = 1;
  } else if (res < -2^31) {
    res[31:0] = 0x80000000; OV = 1;
  }
  Rd.W[x] = res[31:0];
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SLL32(unsigned long a, unsigned int b)`

SLL32 (SIMD 32-bit Shift Left Logical)

Type: SIMD (RV64 Only)

Syntax:

```
SLL32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit elements logical left shift operations simultaneously. The shift amount is a variable from a GPR.

Description: The 32-bit elements in Rs1 are left-shifted logically. And the results are written to Rd. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the value in the Rs2 register.

Operations:

```
sa = Rs2[4:0];
Rd.W[x] = Rs1.W[x] << sa;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SLLI32(unsigned long a, unsigned int b)`

SLLI32 (SIMD 32-bit Shift Left Logical Immediate)

Type: SIMD (RV64 Only)

Syntax:

```
SLLI32 Rd, Rs1, imm5u[4:0]
```

Purpose: Do 32-bit element logical left shift operations simultaneously. The shift amount is an immediate value.

Description: The 32-bit elements in Rs1 are left-shifted logically. The shifted out bits are filled with zero and the shift amount is specified by the imm5u[4:0] constant. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
Rd.W[x] = Rs1.W[x] << sa;
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRA32(unsigned long a, unsigned int b)`

SRA32 (SIMD 32-bit Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Syntax:

```
SRA32 Rd, Rs1, Rs2
SRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[4:0];
if (sa > 0) {
  if (`.u` form) { // SRA32.u
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRA32
    Rd.W[x] = SE32(Rs1.W[x][31:sa])
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRA32_U(unsigned long a, unsigned int b)

SRA32.u (SIMD 32-bit Rounding Shift Right Arithmetic)

Type: SIMD (RV64 Only)

Syntax:

```
SRA32 Rd, Rs1, Rs2
SRA32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element arithmetic right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the data elements. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[4:0];
if (sa > 0) {
  if (`.u` form) { // SRA32.u
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRA32
    Rd.W[x] = SE32(Rs1.W[x][31:sa])
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRAI32(unsigned long a, unsigned int b)

SRAI32 (SIMD 32-bit Shift Right Arithmetic Immediate)

Type: DSP (RV64 Only)

Syntax:

```
SRAI32 Rd, Rs1, imm5u
SRAI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 32-bit data elements. The shift amount is specified by the imm5u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
  if (sa > 0) {
  if (`.u` form) { // SRAI32.u
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRAI32
    Rd.W[x] = SE32(Rs1.W[x][31:sa]);
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRAI32_U(unsigned long a, unsigned int b)

SRAI32.u (SIMD 32-bit Rounding Shift Right Arithmetic Immediate)

Type: DSP (RV64 Only)

Syntax:

```
SRAI32 Rd, Rs1, imm5u
SRAI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements arithmetic right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit of the 32-bit data elements. The shift amount is specified by the imm5u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
  if (sa > 0) {
  if (`.u` form) { // SRAI32.u
    res[31:-1] = SE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRAI32
    Rd.W[x] = SE32(Rs1.W[x][31:sa]);
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRL32(unsigned long a, unsigned int b)

SRL32 (SIMD 32-bit Shift Right Logical)

Type: SIMD (RV64 Only)

Syntax:

```
SRL32 Rd, Rs1, Rs2
SRL32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[4:0];
if (sa > 0) {
  if (`.u` form) { // SRA32.u
    res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRA32
    Rd.W[x] = ZE32(Rs1.W[x][31:sa])
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRL32_U(unsigned long a, unsigned int b)

SRL32.u (SIMD 32-bit Rounding Shift Right Logical)

Type: SIMD (RV64 Only)

Syntax:

```
SRL32 Rd, Rs1, Rs2
SRL32.u Rd, Rs1, Rs2
```

Purpose: Do 32-bit element logical right shift operations simultaneously. The shift amount is a variable from a GPR. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 5-bits of the value in the Rs2 register. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data element to calculate the final results. And the results are written to Rd.

Operations:

```
sa = Rs2[4:0];
if (sa > 0) {
  if (`.u` form) { // SRA32.u
    res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRA32
    Rd.W[x] = ZE32(Rs1.W[x][31:sa])
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SRLI32(unsigned long a, unsigned int b)

SRLI32 (SIMD 32-bit Shift Right Logical Immediate)

Type: SIMD (RV64 Only)

Syntax:

```
SRLI32 Rd, Rs1, imm5u
SRLI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm5u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
if (sa > 0) {
  if (`.u` form) { // SRLI32.u
    res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRLI32
    Rd.W[x] = ZE32(Rs1.W[x][31:sa]);
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned int type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_SRLI32_U(unsigned long a, unsigned int b)`

SRLI32.u (SIMD 32-bit Rounding Shift Right Logical Immediate)

Type: SIMD (RV64 Only)

Syntax:

```
SRLI32 Rd, Rs1, imm5u
SRLI32.u Rd, Rs1, imm5u
```

Purpose: Do 32-bit elements logical right shift operations simultaneously. The shift amount is an immediate value. The `.u` form performs additional rounding up operations on the shifted results.

Description: The 32-bit data elements in Rs1 are right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm5u constant. For the rounding operation of the `.u` form, a value of 1 is added to the most significant discarded bit of each 32-bit data to calculate the final results. And the results are written to Rd.

Operations:

```
sa = imm5u[4:0];
if (sa > 0) {
  if (`.u` form) { // SRLI32.u
    res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
    Rd.W[x] = res[31:0];
  else { // SRLI32
    Rd.W[x] = ZE32(Rs1.W[x][31:sa]);
  }
} else {
  Rd = Rs1;
}
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned int type of value stored in b

(RV64 Only) SIMD 32-bit Miscellaneous Instructions

__STATIC_FORCEINLINE unsigned long __RV_KABS32(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_SMAX32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_SMIN32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMAX32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_UMIN32(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_SIMD_32B_MISC**

(RV64 Only) SIMD 32-bit Miscellaneous Instructions

there are 5 (RV64 Only) SIMD 32-bit Miscellaneous Instructions

Functions

__STATIC_FORCEINLINE unsigned long __RV_KABS32(unsigned long a)

KABS32 (Scalar 32-bit Absolute Value with Saturation)

Type: DSP (RV64 Only) 24 20 19 15 14 12 11 7 KABS32 10010 Rs1 000 Rd 6 0 GE80B 1111111

Syntax:

```
KABS32 Rd, Rs1
```

Purpose: Get the absolute value of signed 32-bit integer elements in a general register.

Description: This instruction calculates the absolute value of signed 32-bit integer elements stored in Rs1. The results are written to Rd. This instruction with the minimum negative integer input of 0x80000000 will produce a saturated output of maximum positive integer of 0x7fffffff and the OV flag will be set to 1.

Operations:

```
if (Rs1.W[x] >= 0) {
  res[x] = Rs1.W[x];
} else {
  If (Rs1.W[x] == 0x80000000) {
    res[x] = 0x7fffffff;
    OV = 1;
  } else {
    res[x] = -Rs1.W[x];
  }
}
Rd.W[x] = res[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

__STATIC_FORCEINLINE unsigned long __RV_SMAX32(unsigned long a, unsigned long b)

SMAX32 (SIMD 32-bit Signed Maximum)

Type: SIMD (RV64 Only)

Syntax:

```
SMAX32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements finding maximum operations simultaneously.

Description: This instruction compares the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] > Rs2.W[x])? Rs1.W[x] : Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_SMIN32(unsigned long a, unsigned long b)

SMIN32 (SIMD 32-bit Signed Minimum)

Type: SIMD (RV64 Only)

Syntax:

```
SMIN32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit signed integer elements finding minimum operations simultaneously.

Description: This instruction compares the 32-bit signed integer elements in Rs1 with the 32-bit signed integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] < Rs2.W[x])? Rs1.W[x] : Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMAX32(unsigned long a, unsigned long b)

UMAX32 (SIMD 32-bit Unsigned Maximum)

Type: SIMD (RV64 Only)

Syntax:

```
UMAX32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements finding maximum operations simultaneously.

Description: This instruction compares the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2 and selects the numbers that is greater than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] u> Rs2.W[x])? Rs1.W[x] : Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_UMIN32(unsigned long a, unsigned long b)

UMIN32 (SIMD 32-bit Unsigned Minimum)

Type: SIMD (RV64 Only)

Syntax:

```
UMIN32 Rd, Rs1, Rs2
```

Purpose: Do 32-bit unsigned integer elements finding minimum operations simultaneously.

Description: This instruction compares the 32-bit unsigned integer elements in Rs1 with the 32-bit unsigned integer elements in Rs2 and selects the numbers that is less than the other one. The selected results are written to Rd.

Operations:

```
Rd.W[x] = (Rs1.W[x] <u Rs2.W[x])? Rs1.W[x] : Rs2.W[x];
for RV64: x=1...0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) SIMD Q15 Saturating Multiply Instructions

__STATIC_FORCEINLINE unsigned long __RV_KDMBB16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KDMBT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KDMTT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KDMABB16(unsigned long t, unsigned long a, unsigned

__STATIC_FORCEINLINE unsigned long __RV_KDMABT16(unsigned long t, unsigned long a, unsigned

__STATIC_FORCEINLINE unsigned long __RV_KDMATT16(unsigned long t, unsigned long a, unsigned

__STATIC_FORCEINLINE unsigned long __RV_KHMBB16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KHMBT16(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_KHMTT16(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_SIMD_Q15_SAT_MULT**

(RV64 Only) SIMD Q15 Saturating Multiply Instructions

there are 9 (RV64 Only) SIMD Q15 saturating Multiply Instructions

Functions

__STATIC_FORCEINLINE unsigned long __RV_KDMBB16(unsigned long a, unsigned long b)

KDMBB16 (SIMD Signed Saturating Double Multiply B16 x B16)

Type: SIMD (RV64 only)

Syntax:

```
KDMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results into the 32-bit chunks in the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
// KDMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
Rd.W[z] = resQ31[z];
```

Return value stored in unsigned long type

Parameters

- [in] a: unsigned long type of value stored in a
- [in] b: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KDMBT16(unsigned long a, unsigned long b)

KDMBT16 (SIMD Signed Saturating Double Multiply B16 x T16)

Type: SIMD (RV64 only)

Syntax:

```
KDMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results into the 32-bit chunks in the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
// KDMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
Rd.W[z] = resQ31[z];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KDMTT16(unsigned long a, unsigned long b)`

KDMTT16 (SIMD Signed Saturating Double Multiply T16 x T16)

Type: SIMD (RV64 only)

Syntax:

```
KDMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results into the 32-bit chunks in the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFFFFFF and the overflow flag OV will be set.

Operations:

```
// KDMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMTT16: (x,y,z)=(1,1,0),(3,3,1)
```

(continues on next page)

(continued from previous page)

```
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
Rd.W[z] = resQ31[z];
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KDMABB16(unsigned long t, unsigned long a, uns`

KDMABB16 (SIMD Signed Saturating Double Multiply Addition B16 x B16)

Type: SIMD (RV64 only)

Syntax:

```
KDMAxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results, add the results with the values of the corresponding 32-bit chunks from the destination register and write the saturated addition results back into the corresponding 32-bit chunks of the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the corresponding 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then added with the content of the corresponding 32-bit portions of Rd. If the addition results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV flag is set to 1. The results after saturation are written back to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
// KDMABB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMABT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMATT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
resadd[z] = Rd.W[z] + resQ31[z];
if (resadd[z] > (2^31)-1) {
  resadd[z] = (2^31)-1;
  OV = 1;
} else if (resadd[z] < -2^31) {
```

(continues on next page)

(continued from previous page)

```
  resadd[z] = -2^31;
  OV = 1;
}
Rd.W[z] = resadd[z];
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KDMABT16(unsigned long t, unsigned long a, uns

KDMABT16 (SIMD Signed Saturating Double Multiply Addition B16 x T16)

Type: SIMD (RV64 only)

Syntax:

```
KDMAxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results, add the results with the values of the corresponding 32-bit chunks from the destination register and write the saturated addition results back into the corresponding 32-bit chunks of the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the corresponding 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then added with the content of the corresponding 32-bit portions of Rd. If the addition results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV flag is set to 1. The results after saturation are written back to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
// KDMABB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMABT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMATT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
resadd[z] = Rd.W[z] + resQ31[z];
if (resadd[z] > (2^31)-1) {
  resadd[z] = (2^31)-1;
  OV = 1;
} else if (resadd[z] < -2^31) {
  resadd[z] = -2^31;
  OV = 1;
}
Rd.W[z] = resadd[z];
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KDMATT16(unsigned long t, unsigned long a, uns

KDMATT16 (SIMD Signed Saturating Double Multiply Addition T16 x T16)

Type: SIMD (RV64 only)

Syntax:

```
KDMAxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then double and saturate the Q31 results, add the results with the values of the corresponding 32-bit chunks from the destination register and write the saturated addition results back into the corresponding 32-bit chunks of the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the corresponding 32-bit portions in Rs2. The Q30 results are then doubled and saturated into Q31 values. The Q31 values are then added with the content of the corresponding 32-bit portions of Rd. If the addition results are beyond the Q31 number range (-2^31 <= Q31 <= 2^31-1), they are saturated to the range and the OV flag is set to 1. The results after saturation are written back to Rd. When both the two Q15 inputs are 0x8000, saturation will happen and the overflow flag OV will be set.

Operations:

```
// KDMABB16: (x,y,z)=(0,0,0),(2,2,1)
// KDMABT16: (x,y,z)=(0,1,0),(2,3,1)
// KDMATT16: (x,y,z)=(1,1,0),(3,3,1)
aop[z] = Rs1.H[x]; bop[z] = Rs2.H[y];
If (0x8000 != aop[z] | 0x8000 != bop[z]) {
  Mresult[z] = aop[z] * bop[z];
  resQ31[z] = Mresult[z] << 1;
} else {
  resQ31[z] = 0x7FFFFFFF;
  OV = 1;
}
resadd[z] = Rd.W[z] + resQ31[z];
if (resadd[z] > (2^31)-1) {
  resadd[z] = (2^31)-1;
  OV = 1;
} else if (resadd[z] < -2^31) {
  resadd[z] = -2^31;
  OV = 1;
}
Rd.W[z] = resadd[z];
```

Return value stored in unsigned long type

Parameters

- `[in] t`: unsigned long type of value stored in t
- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KHMBB16(unsigned long a, unsigned long b)
KHMBB16 (SIMD Signed Saturating Half Multiply B16 x B16)

Type: SIMD (RV64 Only)

Syntax:

```
KHMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then right-shift 15 bits to turn the Q30 results into Q15 numbers again and saturate the Q15 results into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portion in Rs2. The Q30 results are then right-shifted 15- bits and saturated into Q15 values. The 32-bit Q15 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
// KHMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KHMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KHMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop = Rs1.H[x]; bop = Rs2.H[y];
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
}
Rd.W[z] = SE32(res[15:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_KHMBT16(unsigned long a, unsigned long b)
KHMBT16 (SIMD Signed Saturating Half Multiply B16 x T16)

Type: SIMD (RV64 Only)

Syntax:

```
KHMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then right-shift 15 bits to turn the Q30 results into Q15 numbers again and saturate the Q15 results into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portion in Rs2. The Q30 results are then right-shifted 15- bits and saturated into Q15 values. The 32-bit Q15 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
// KHMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KHMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KHMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop = Rs1.H[x]; bop = Rs2.H[y];
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
}
Rd.W[z] = SE32(res[15:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_KHMTT16(unsigned long a, unsigned long b)`

KHMTT16 (SIMD Signed Saturating Half Multiply T16 x T16)

Type: SIMD (RV64 Only)

Syntax:

```
KHMxy16 Rd, Rs1, Rs2 (xy = BB, BT, TT)
```

Purpose: Multiply the signed Q15 integer contents of two 16-bit data in the corresponding portion of the 32-bit chunks in registers and then right-shift 15 bits to turn the Q30 results into Q15 numbers again and saturate the Q15 results into the destination register. If saturation happens, an overflow flag OV will be set.

Description: Multiply the top or bottom 16-bit Q15 content of the 32-bit portions in Rs1 with the top or bottom 16-bit Q15 content of the 32-bit portion in Rs2. The Q30 results are then right-shifted 15- bits and saturated into Q15 values. The 32-bit Q15 values are then written into the 32-bit chunks in Rd. When both the two Q15 inputs are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
// KHMBB16: (x,y,z)=(0,0,0),(2,2,1)
// KHMBT16: (x,y,z)=(0,1,0),(2,3,1)
// KHMTT16: (x,y,z)=(1,1,0),(3,3,1)
aop = Rs1.H[x]; bop = Rs2.H[y];
If (0x8000 != aop | 0x8000 != bop) {
  Mresult[31:0] = aop * bop;
  res[15:0] = Mresult[30:15];
} else {
  res[15:0] = 0x7FFF;
  OV = 1;
}
Rd.W[z] = SE32(res[15:0]);
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) 32-bit Multiply Instructions

__STATIC_FORCEINLINE long __RV_SMBB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMBT32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMTT32(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_32B_MULT**

(RV64 Only) 32-bit Multiply Instructions

there is 3 RV64 Only) 32-bit Multiply Instructions

Functions

__STATIC_FORCEINLINE long __RV_SMBB32(unsigned long a, unsigned long b)

SMBB32 (Signed Multiply Bottom Word & Bottom Word)

Type: DSP (RV64 Only)

Syntax:

```
SMBB32 Rd, Rs1, Rs2
SMBT32 Rd, Rs1, Rs2
SMTT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

- SMBB32: bottom*bottom
- SMBT32: bottom*top
- SMTT32: top*top

Description: For the `SMBB32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. It is actually an alias of `MULSR64` instruction. For the `SMBT32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMTT32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[0] * Rs2.W[0]; // SMBB32 res = Rs1.W[0] * Rs2.w[1]; // SMBT32 res␣
↪= Rs1.W[1] * Rs2.W[1];
// SMTT32 Rd = res;
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMBT32(unsigned long a, unsigned long b)

SMBT32 (Signed Multiply Bottom Word & Top Word)

Type: DSP (RV64 Only)

Syntax:

```
SMBB32 Rd, Rs1, Rs2
SMBT32 Rd, Rs1, Rs2
SMTT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

- SMBB32: bottom*bottom
- SMBT32: bottom*top
- SMTT32: top*top

Description: For the `SMBB32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. It is actually an alias of `MULSR64` instruction. For the `SMBT32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMTT32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[0] * Rs2.W[0]; // SMBB32 res = Rs1.W[0] * Rs2.w[1]; // SMBT32 res␣
↪= Rs1.W[1] * Rs2.W[1];
// SMTT32 Rd = res;
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMTT32(unsigned long a, unsigned long b)

SMTT32 (Signed Multiply Top Word & Top Word)

Type: DSP (RV64 Only)

Syntax:

```
SMBB32 Rd, Rs1, Rs2
SMBT32 Rd, Rs1, Rs2
SMTT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element of a register with the signed 32-bit element of another register and write the 64-bit result to a third register.

- SMBB32: bottom*bottom
- SMBT32: bottom*top
- SMTT32: top*top

Description: For the `SMBB32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. It is actually an alias of `MULSR64` instruction. For the `SMBT32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMTT32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. The 64-bit multiplication result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rs1.W[0] * Rs2.W[0]; // SMBB32 res = Rs1.W[0] * Rs2.w[1]; // SMBT32 res␣
↪= Rs1.W[1] * Rs2.W[1];
// SMTT32 Rd = res;
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) 32-bit Multiply & Add Instructions

__STATIC_FORCEINLINE long __RV_KMABB32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMABT32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMATT32(long t, unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_32B_MULT_ADD**

(RV64 Only) 32-bit Multiply & Add Instructions

there are 3 (RV64 Only) 32-bit Multiply & Add Instructions

Functions

__STATIC_FORCEINLINE long __RV_KMABB32(long t, unsigned long a, unsigned long b)

KMABB32 (Saturating Signed Multiply Bottom Words & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMABB32 Rd, Rs1, Rs2
KMABT32 Rd, Rs1, Rs2
KMATT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

- KMABB32: rd + bottom*bottom
- KMABT32: rd + bottom*top
- KMATT32: rd + top*top

Description: For the `KMABB32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMABT32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMATT32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2. The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[0]); // KMABB32
 res = Rd + (Rs1.W[0] * Rs2.W[1]); // KMABT32
 res = Rd + (Rs1.W[1] * Rs2.W[1]); // KMATT32
 if (res > (2^63)-1) {
   res = (2^63)-1;
   OV = 1;
 } else if (res < -2^63) {
   res = -2^63;
   OV = 1;
 }
 Rd = res;
*Exceptions:* None
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMABT32(long t, unsigned long a, unsigned long b)

KMABT32 (Saturating Signed Multiply Bottom & Top Words & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMABB32 Rd, Rs1, Rs2
KMABT32 Rd, Rs1, Rs2
KMATT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

- KMABB32: rd + bottom*bottom
- KMABT32: rd + bottom*top
- KMATT32: rd + top*top

Description: For the `KMABB32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMABT32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMATT32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2. The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[0]); // KMABB32
 res = Rd + (Rs1.W[0] * Rs2.W[1]); // KMABT32
 res = Rd + (Rs1.W[1] * Rs2.W[1]); // KMATT32
 if (res > (2^63)-1) {
   res = (2^63)-1;
   OV = 1;
 } else if (res < -2^63) {
   res = -2^63;
   OV = 1;
 }
 Rd = res;
*Exceptions:* None
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMATT32(long t, unsigned long a, unsigned long b)

KMATT32 (Saturating Signed Multiply Top Words & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMABB32 Rd, Rs1, Rs2
KMABT32 Rd, Rs1, Rs2
KMATT32 Rd, Rs1, Rs2
```

Purpose: Multiply the signed 32-bit element in a register with the 32-bit element in another register and add the result to the content of 64-bit data in the third register. The addition result may be saturated and is written to the third register.

- KMABB32: rd + bottom*bottom
- KMABT32: rd + bottom*top
- KMATT32: rd + top*top

Description: For the `KMABB32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMABT32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMATT32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2. The multiplication result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[0] * Rs2.W[0]); // KMABB32
 res = Rd + (Rs1.W[0] * Rs2.W[1]); // KMABT32
 res = Rd + (Rs1.W[1] * Rs2.W[1]); // KMATT32
 if (res > (2^63)-1) {
   res = (2^63)-1;
   OV = 1;
```

(continues on next page)

(continued from previous page)

```
 } else if (res < -2^63) {
   res = -2^63;
   OV = 1;
 }
 Rd = res;
*Exceptions:* None
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) 32-bit Parallel Multiply & Add Instructions

__STATIC_FORCEINLINE long __RV_KMADA32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMAXDA32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMDA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMXDA32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMADS32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMADRS32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMAXDS32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMSDA32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_KMSXDA32(long t, unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMDS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMDRS32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE long __RV_SMXDS32(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_32B_PARALLEL_MAC**

(RV64 Only) 32-bit Parallel Multiply & Add Instructions

there are 12 (RV64 Only) 32-bit Parallel Multiply & Add Instructions

Functions

__STATIC_FORCEINLINE long __RV_KMADA32(long t, unsigned long a, unsigned long b)

KMADA32 (Saturating Signed Multiply Two Words and Two Adds)

Type: DSP (RV64 Only)

Syntax:

```
KMADA32 Rd, Rs1, Rs2
KMAXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit data in two registers; and then adds the two 64-bit results and 64-bit data in a third register together. The addition result may be saturated.

- KMADA32: rd + top*top + bottom*bottom
- KMAXDA32: rd + top*bottom + bottom*top

Description: For the `KMADA32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32- bit element in Rs2 and then adds the result to the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. It is actually an alias of the `KMAR64` instruction. For the `KMAXDA32` instruction, it multiplies the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then adds the result to the result of multiplying the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2. The result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The 64-bit result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.w[1]) + (Rs1.W[0] * Rs2.W[0]); // KMADA32
res = Rd + (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMAXDA32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMAXDA32(long t, unsigned long a, unsigned long b)`

KMAXDA32 (Saturating Signed Crossed Multiply Two Words and Two Adds)

Type: DSP (RV64 Only)

Syntax:

```
KMADA32 Rd, Rs1, Rs2
KMAXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit data in two registers; and then adds the two 64-bit results and 64-bit data in a third register together. The addition result may be saturated.

- KMADA32: rd + top*top + bottom*bottom
- KMAXDA32: rd + top*bottom + bottom*top

Description: For the `KMADA32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32- bit element in Rs2 and then adds the result to the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. It is actually an alias of the `KMAR64` instruction. For the `KMAXDA32` instruction, it multiplies the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then adds the result to the result of multiplying the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2. The result is added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The 64-bit result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.w[1]) + (Rs1.W[0] * Rs2.W[0]); // KMADA32
res = Rd + (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMAXDA32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMDA32(unsigned long a, unsigned long b)

KMDA32 (Signed Multiply Two Words and Add)

Type: DSP (RV64 Only)

Syntax:

```
KMDA32 Rd, Rs1, Rs2
KMXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then adds the two 64-bit results together. The addition result may be saturated.

- KMDA32: top*top + bottom*bottom
- KMXDA32: top*bottom + bottom*top

Description: For the `KMDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `KMXDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The addition result is checked for saturation. If saturation happens, the result is saturated to 2^63-1. The final result is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
if ((Rs1 != 0x8000000080000000) or (Rs2 != 0x8000000080000000)) {
  Rd = (Rs1.W[1] * Rs2.W[1]) + (Rs1.W[0] * Rs2.W[0]); // KMDA32
  Rd = (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMXDA32
} else {
  Rd = 0x7fffffffffffffff;
  OV = 1;
}
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMXDA32(unsigned long a, unsigned long b)

KMXDA32 (Signed Crossed Multiply Two Words and Add)

Type: DSP (RV64 Only)

Syntax:

```
KMDA32 Rd, Rs1, Rs2
KMXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then adds the two 64-bit results together. The addition result may be saturated.

- KMDA32: top*top + bottom*bottom
- KMXDA32: top*bottom + bottom*top

Description: For the `KMDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `KMXDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then adds the result to the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The addition result is checked for saturation. If saturation happens, the result is saturated to 2^63-1. The final result is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
if ((Rs1 != 0x8000000080000000) or (Rs2 != 0x8000000080000000)) {
  Rd = (Rs1.W[1] * Rs2.W[1]) + (Rs1.W[0] * Rs2.W[0]); // KMDA32
  Rd = (Rs1.W[1] * Rs2.W[0]) + (Rs1.W[0] * Rs2.W[1]); // KMXDA32
} else {
  Rd = 0x7fffffffffffffff;
  OV = 1;
}
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMADS32(long t, unsigned long a, unsigned long b)

KMADS32 (Saturating Signed Multiply Two Words & Subtract & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMADS32 Rd, Rs1, Rs2
KMADRS32 Rd, Rs1, Rs2
KMAXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 64-bit results. Then add the subtraction result to 64-bit data in a third register. The addition result may be saturated.

- KMADS32: rd + (top*top - bottom*bottom)
- KMADRS32: rd + (bottom*bottom - top*top)
- KMAXDS32: rd + (top*bottom - bottom*top)

Description: For the `KMADS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMADRS32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMAXDS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2. The subtraction result is then added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to

1. The 64-bit result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMADS32
res = Rd + (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // KMADRS32
res = Rd + (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMAXDS32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMADRS32(long t, unsigned long a, unsigned long b)`

KMADRS32 (Saturating Signed Multiply Two Words & Reverse Subtract & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMADS32 Rd, Rs1, Rs2
KMADRS32 Rd, Rs1, Rs2
KMAXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 64-bit results. Then add the subtraction result to 64-bit data in a third register. The addition result may be saturated.

- KMADS32: rd + (top*top - bottom*bottom)
- KMADRS32: rd + (bottom*bottom - top*top)
- KMAXDS32: rd + (top*bottom - bottom*top)

Description: For the `KMADS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMADRS32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMAXDS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2. The subtraction result is then added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to

1. The 64-bit result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMADS32
res = Rd + (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // KMADRS32
res = Rd + (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMAXDS32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

`__STATIC_FORCEINLINE long __RV_KMAXDS32(long t, unsigned long a, unsigned long b)`

KMAXDS32 (Saturating Signed Crossed Multiply Two Words & Subtract & Add)

Type: DSP (RV64 Only)

Syntax:

```
KMADS32 Rd, Rs1, Rs2
KMADRS32 Rd, Rs1, Rs2
KMAXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from 32-bit elements in two registers; and then perform a subtraction operation between the two 64-bit results. Then add the subtraction result to 64-bit data in a third register. The addition result may be saturated.

- KMADS32: rd + (top*top - bottom*bottom)
- KMADRS32: rd + (bottom*bottom - top*top)

- KMAXDS32: rd + (top*bottom - bottom*top)

Description: For the `KMADS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the top 32-bit element in Rs2. For the `KMADRS32` instruction, it multiplies the top 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element in Rs1 with the bottom 32-bit element in Rs2. For the `KMAXDS32` instruction, it multiplies the bottom 32-bit element in Rs1 with the top 32-bit element in Rs2 and then subtracts the result from the result of multiplying the top 32-bit element in Rs1 with the bottom 32-bit element in Rs2. The subtraction result is then added to the content of 64-bit data in Rd. If the addition result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to

1. The 64-bit result after saturation is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
res = Rd + (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMADS32
res = Rd + (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // KMADRS32
res = Rd + (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMAXDS32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMSDA32(long t, unsigned long a, unsigned long b)
KMSDA32 (Saturating Signed Multiply Two Words & Add & Subtract)

Type: DSP (RV64 Only)

Syntax:

```
KMSDA32 Rd, Rs1, Rs2
KMSXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then subtracts the two 64-bit results from a third register. The subtraction result may be saturated.

- KMSDA: rd - top*top - bottom*bottom
- KMSXDA: rd - top*bottom - bottom*top

Description: For the `KMSDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `KMSXDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The two 64-bit multiplication results are then subtracted from the content of Rd. If the subtraction result is beyond the

Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
res = Rd - (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMSDA32
res = Rd - (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMSXDA32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_KMSXDA32(long t, unsigned long a, unsigned long b)

KMSXDA32 (Saturating Signed Crossed Multiply Two Words & Add & Subtract)

Type: DSP (RV64 Only)

Syntax:

```
KMSDA32 Rd, Rs1, Rs2
KMSXDA32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the 32-bit element of two registers; and then subtracts the two 64-bit results from a third register. The subtraction result may be saturated.

- KMSDA: rd - top*top - bottom*bottom
- KMSXDA: rd - top*bottom - bottom*top

Description: For the `KMSDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `KMSXDA32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and multiplies the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The two 64-bit multiplication results are then subtracted from the content of Rd. If the subtraction result is beyond the Q63 number range (-2^63 <= Q63 <= 2^63-1), it is saturated to the range and the OV bit is set to 1. The result after saturation is written to Rd. The 32-bit contents are treated as signed integers.

Operations:

```
res = Rd - (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // KMSDA32
res = Rd - (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // KMSXDA32
if (res > (2^63)-1) {
  res = (2^63)-1;
  OV = 1;
} else if (res < -2^63) {
  res = -2^63;
  OV = 1;
```

(continues on next page)

(continued from previous page)

```
}
Rd = res;
```

Return value stored in long type

Parameters

- `[in] t`: long type of value stored in t
- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMDS32(unsigned long a, unsigned long b)

SMDS32 (Signed Multiply Two Words and Subtract)

Type: DSP (RV64 Only)

Syntax:

```
SMDS32 Rd, Rs1, Rs2
SMDRS32 Rd, Rs1, Rs2
SMXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the l 32-bit element of two registers; and then perform a subtraction operation between the two 64-bit results.

- SMDS32: top*top - bottom*bottom
- SMDRS32: bottom*bottom - top*top
- SMXDS32: top*bottom - bottom*top

Description: For the `SMDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMDRS32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. For the `SMXDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The subtraction result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rt = (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // SMDS32
Rt = (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // SMDRS32
Rt = (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // SMXDS32
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMDRS32(unsigned long a, unsigned long b)

SMDRS32 (Signed Multiply Two Words and Reverse Subtract)

Type: DSP (RV64 Only)

Syntax:

```
SMDS32 Rd, Rs1, Rs2
SMDRS32 Rd, Rs1, Rs2
SMXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the l 32-bit element of two registers; and then perform a subtraction operation between the two 64-bit results.

- SMDS32: top*top - bottom*bottom
- SMDRS32: bottom*bottom - top*top
- SMXDS32: top*bottom - bottom*top

Description: For the `SMDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMDRS32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. For the `SMXDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The subtraction result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rt = (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // SMDS32
Rt = (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // SMDRS32
Rt = (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // SMXDS32
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE long __RV_SMXDS32(unsigned long a, unsigned long b)

SMXDS32 (Signed Crossed Multiply Two Words and Subtract)

Type: DSP (RV64 Only)

Syntax:

```
SMDS32 Rd, Rs1, Rs2
SMDRS32 Rd, Rs1, Rs2
SMXDS32 Rd, Rs1, Rs2
```

Purpose: Do two signed 32-bit multiplications from the l 32-bit element of two registers; and then perform a subtraction operation between the two 64-bit results.

- SMDS32: top*top - bottom*bottom
- SMDRS32: bottom*bottom - top*top
- SMXDS32: top*bottom - bottom*top

Description: For the `SMDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the top 32-bit element of Rs2. For the `SMDRS32` instruction, it multiplies the top 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the bottom 32-bit element of Rs1 with the bottom 32-bit element of Rs2. For the `SMXDS32` instruction, it multiplies the bottom 32-bit element of Rs1 with the top 32-bit element of Rs2 and then subtracts the result from the result of multiplying the top 32-bit element of Rs1 with the bottom 32-bit element of Rs2. The subtraction result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

Operations:

```
Rt = (Rs1.W[1] * Rs2.W[1]) - (Rs1.W[0] * Rs2.W[0]); // SMDS32
Rt = (Rs1.W[0] * Rs2.W[0]) - (Rs1.W[1] * Rs2.W[1]); // SMDRS32
Rt = (Rs1.W[1] * Rs2.W[0]) - (Rs1.W[0] * Rs2.W[1]); // SMXDS32
```

Return value stored in long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

(RV64 Only) Non-SIMD 32-bit Shift Instructions

__STATIC_FORCEINLINE long __RV_SRAIW_U(int a, unsigned int b)

group **NMSIS_Core_DSP_Intrinsic_RV64_NON_SIMD_32B_SHIFT**

(RV64 Only) Non-SIMD 32-bit Shift Instructions

there are 1 (RV64 Only) Non-SIMD 32-bit Shift Instructions

Functions

__STATIC_FORCEINLINE long __RV_SRAIW_U(int a, unsigned int b)

SRAIW.u (Rounding Shift Right Arithmetic Immediate Word)

Type: DSP (RV64 only)

Syntax:

```
SRAIW.u Rd, Rs1, imm5u
```

Purpose: Perform a 32-bit arithmetic right shift operation with rounding. The shift amount is an immediate value.

Description: This instruction right-shifts the lower 32-bit content of Rs1 arithmetically. The shifted out bits are filled with the sign-bit Rs1(31) and the shift amount is specified by the imm5u constant. For the rounding operation, a value of 1 is added to the most significant discarded bit of the data to calculate the final result. And the result is sign-extended and written to Rd.

Operations:

```
sa = imm5u;
if (sa != 0) {
  res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
  Rd = SE32(res[31:0]);
} else {
  Rd = SE32(Rs1.W[0]);
}
```

Return value stored in long type

Parameters

- `[in] a`: int type of value stored in a
- `[in] b`: unsigned int type of value stored in b

32-bit Packing Instructions

__STATIC_FORCEINLINE unsigned long __RV_PKBB32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKBT32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKTT32(unsigned long a, unsigned long b)

__STATIC_FORCEINLINE unsigned long __RV_PKTB32(unsigned long a, unsigned long b)

group **NMSIS_Core_DSP_Intrinsic_RV64_32B_PACK**

32-bit Packing Instructions

There are four 32-bit packing instructions here

Functions

__STATIC_FORCEINLINE unsigned long __RV_PKBB32(unsigned long a, unsigned long b)

PKBB32 (Pack Two 32-bit Data from Both Bottom Half)

Type: DSP (RV64 Only)

Syntax:

```
PKBB32 Rd, Rs1, Rs2
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2
PKTB32 Rd, Rs1, Rs2
```

Purpose: Pack 32-bit data from 64-bit chunks in two registers.

- PKBB32: bottom.bottom
- PKBT32: bottom.top
- PKTT32: top.top
- PKTB32: top.bottom

Description: (PKBB32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0]. (PKBT32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTT32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTB32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*0*_]); // PKBB32
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*1*_]); // PKBT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*1*_]); // PKTT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*0*_]); // PKTB32
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_PKBT32(unsigned long a, unsigned long b)

PKBT32 (Pack Two 32-bit Data from Bottom and Top Half)

Type: DSP (RV64 Only)

Syntax:

```
PKBB32 Rd, Rs1, Rs2
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2
PKTB32 Rd, Rs1, Rs2
```

Purpose: Pack 32-bit data from 64-bit chunks in two registers.

- PKBB32: bottom.bottom
- PKBT32: bottom.top
- PKTT32: top.top
- PKTB32: top.bottom

Description: (PKBB32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0]. (PKBT32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTT32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTB32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*0*_]); // PKBB32
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*1*_]); // PKBT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*1*_]); // PKTT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*0*_]); // PKTB32
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_PKTT32(unsigned long a, unsigned long b)

PKTT32 (Pack Two 32-bit Data from Both Top Half)

Type: DSP (RV64 Only)

Syntax:

```
PKBB32 Rd, Rs1, Rs2
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2
PKTB32 Rd, Rs1, Rs2
```

Purpose: Pack 32-bit data from 64-bit chunks in two registers.

- PKBB32: bottom.bottom
- PKBT32: bottom.top
- PKTT32: top.top
- PKTB32: top.bottom

Description: (PKBB32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0]. (PKBT32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTT32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTB32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*0*_]); // PKBB32
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*1*_]); // PKBT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*1*_]); // PKTT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*0*_]); // PKTB32
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

__STATIC_FORCEINLINE unsigned long __RV_PKTB32(unsigned long a, unsigned long b)

PKTB32 (Pack Two 32-bit Data from Top and Bottom Half)

Type: DSP (RV64 Only)

Syntax:

```
PKBB32 Rd, Rs1, Rs2
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2
PKTB32 Rd, Rs1, Rs2
```

Purpose: Pack 32-bit data from 64-bit chunks in two registers.

- PKBB32: bottom.bottom
- PKBT32: bottom.top
- PKTT32: top.top
- PKTB32: top.bottom

Description: (PKBB32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0]. (PKBT32) moves Rs1.W[0] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTT32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[1] to Rd.W[0]. (PKTB32) moves Rs1.W[1] to Rd.W[1] and moves Rs2.W[0] to Rd.W[0].

Operations:

```
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*0*_]); // PKBB32
Rd = CONCAT(Rs1.W[_*0*_], Rs2.W[_*1*_]); // PKBT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*1*_]); // PKTT32
Rd = CONCAT(Rs1.W[_*1*_], Rs2.W[_*0*_]); // PKTB32
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a
- `[in] b`: unsigned long type of value stored in b

group **NMSIS_Core_DSP_Intrinsic_RV64_ONLY**

RV64 Only Instructions.

Nuclei Customized DSP Instructions

__STATIC_FORCEINLINE unsigned long long __RV_DKHM8(unsigned long long a, unsigned long long

__STATIC_FORCEINLINE unsigned long long __RV_DKHM16(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_DKABS8(unsigned long long a)

__STATIC_FORCEINLINE unsigned long long __RV_DKABS16(unsigned long long a)

__STATIC_FORCEINLINE unsigned long long __RV_DKSLRA8(unsigned long long a, int b)

__STATIC_FORCEINLINE unsigned long long __RV_DKSLRA16(unsigned long long a, int b)

__STATIC_FORCEINLINE unsigned long long __RV_DKADD8(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_DKADD16(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_DKSUB8(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long long __RV_DKSUB16(unsigned long long a, unsigned long lo

__STATIC_FORCEINLINE unsigned long __RV_EXPD80(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_EXPD81(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_EXPD82(unsigned long a)

__STATIC_FORCEINLINE unsigned long __RV_EXPD83(unsigned long a)

group **NMSIS_Core_DSP_Intrinsic_NUCLEI_CUSTOM**

(RV32 only)Nuclei Customized DSP Instructions

This is Nuclei customized DSP instructions only for RV32

Functions

__STATIC_FORCEINLINE unsigned long long __RV_DKHM8(unsigned long long a, unsigned long

DKHM8 (64-bit SIMD Signed Saturating Q7 Multiply)

Type: SIMD

Syntax:

```
DKHM8 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description: For the `DKHM8` instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2.

The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```
op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top
op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x80 != aop | 0x80 != bop) {
    res = (aop s* bop) >> 7;
  } else {
    res= 0x7F;
    OV = 1;
  }
}
Rd.H[x/2] = concat(rest, resb);
for RV32, x=0,2,4,6
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_DKHM16(unsigned long long a, unsigned lon`

DKHM16 (64-bit SIMD Signed Saturating Q15 Multiply)

Type: SIMD

Syntax:

```
DKHM16 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to Q15 numbers again.

Description: For the `DKHM16` instruction, multiply the top 16-bit Q15 content of 32-bit chunks in Rs1 with the top 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the bottom 16-bit Q15 content of 32-bit chunks in Rs1 with the bottom 16-bit Q15 content of 32-bit chunks in Rs2.

The Q30 results are then right-shifted 15-bits and saturated into Q15 values. The Q15 results are then written into Rd. When both the two Q15 inputs of a multiplication are 0x8000, saturation will happen. The result will be saturated to 0x7FFF and the overflow flag OV will be set.

Operations:

```
op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top
op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom
for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x8000 != aop | 0x8000 != bop) {
    res = (aop s* bop) >> 15;
  } else {
    res= 0x7FFF;
    OV = 1;
  }
}
Rd.W[x/2] = concat(rest, resb);
for RV32: x=0, 2
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_DKABS8(unsigned long long a)

DKABS8 (64-bit SIMD 8-bit Saturating Absolute)

Type: SIMD

Syntax:

```
DKABS8 Rd, Rs1
# Rd, Rs1 are all even/odd pair of registers
```

Purpose: Get the absolute value of 8-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 8-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x80, this instruction generates 0x7f as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src == 0x80) {
  src = 0x7f;
  OV = 1;
} else if (src[7] == 1)
  src = -src;
}
Rd.B[x] = src;
for RV32: x=7...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a

__STATIC_FORCEINLINE unsigned long long __RV_DKABS16(unsigned long long a)

DKABS16 (64-bit SIMD 16-bit Saturating Absolute)

Type: SIMD

Syntax:

```
DKABS16 Rd, Rs1
# Rd, Rs1 are all even/odd pair of registers
```

Purpose: Get the absolute value of 16-bit signed integer elements simultaneously.

Description: This instruction calculates the absolute value of 16-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000, this instruction generates 0x7fff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src == 0x8000) {
  src = 0x7fff;
  OV = 1;
} else if (src[15] == 1)
```

(continues on next page)

(continued from previous page)

```
  src = -src;
}
Rd.H[x] = src;
for RV32: x=3...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a

__STATIC_FORCEINLINE unsigned long long __RV_DKSLRA8(unsigned long long a, int b)

DKSLRA8 (64-bit SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Syntax:

```
DKSLRA8 Rd, Rs1, Rs2
# Rd, Rs1 are all even/odd pair of registers
```

Purpose: Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift.

Description: The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of [-2^3, 2^3-1]. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of `Rs2[3:0]==-2^3 (0x8)` is defined to be equivalent to the behavior of `Rs2[3:0]==-(2^3-1) (0x9)`. The left-shifted results are saturated to the 8-bit signed integer range of [-2^7, 2^7-1]. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:4] will not affect this instruction.

Operations:

```
if (Rs2[3:0] < 0) {
  sa = -Rs2[3:0];
  sa = (sa == 8)? 7 : sa;
  Rd.B[x] = SE8(Rs1.B[x][7:sa]);
} else {
  sa = Rs2[2:0];
  res[(7+sa):0] = Rs1.B[x] <<(logic) sa;
  if (res > (2^7)-1) {
    res[7:0] = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res[7:0] = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
}
for RV32: x=7...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_DKSLRA16(unsigned long long a, int b)
DKSLRA16 (64-bit SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Syntax:

```
DKSLRA16 Rd, Rs1, Rs2
# Rd, Rs1 are all even/odd pair of registers
```

Purpose: Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift.

Description: The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of [-2^4, 2^4-1]. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of `Rs2[4:0]==-2^4 (0x10)` is defined to be equivalent to the behavior of `Rs2[4:0]==-(2^4-1) (0x11)`. The left-shifted results are saturated to the 16-bit signed integer range of [-2^15, 2^15-1]. After the shift, saturation, or rounding, the final results are written to Rd. If any saturation happens, this instruction sets the OV flag. The value of Rs2[31:5] will not affect this instruction.

Operations:

```
if (Rs2[4:0] < 0) {
  sa = -Rs2[4:0];
  sa = (sa == 16)? 15 : sa;
  Rd.H[x] = SE16(Rs1.H[x][15:sa]);
} else {
  sa = Rs2[3:0];
  res[(15+sa):0] = Rs1.H[x] <<(logic) sa;
  if (res > (2^15)-1) {
    res[15:0] = 0x7fff; OV = 1;
  } else if (res < -2^15) {
    res[15:0] = 0x8000; OV = 1;
  }
  d.H[x] = res[15:0];
}
for RV32: x=3...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: int type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_DKADD8(unsigned long long a, unsigned lon
DKADD8 (64-bit SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Syntax:

```
DKADD8 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do 8-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 2^7-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > 127) {
  res[x] = 127;
  OV = 1;
} else if (res[x] < -128) {
  res[x] = -128;
  OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=7...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

`__STATIC_FORCEINLINE unsigned long long __RV_DKADD16(unsigned long long a, unsigned lo`

DKADD16 (64-bit SIMD 16-bit Signed Saturating Addition)

Type: SIMD

Syntax:

```
DKADD16 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do 16-bit signed integer element saturating additions simultaneously.

Description: This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > 32767) {
  res[x] = 32767;
  OV = 1;
} else if (res[x] < -32768) {
  res[x] = -32768;
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=3...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_DKSUB8(unsigned long long a, unsigned long

DKSUB8 (64-bit SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Syntax:

```
DKSUB8 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do 8-bit signed elements saturating subtractions simultaneously.

Description: This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. If any of the results are beyond the Q7 number range (-2^7 <= Q7 <= 2^7-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] > (2^7)-1) {
  res[x] = (2^7)-1;
  OV = 1;
} else if (res[x] < -2^7) {
  res[x] = -2^7;
  OV = 1;
}
Rd.B[x] = res[x];
for RV32: x=7...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

__STATIC_FORCEINLINE unsigned long long __RV_DKSUB16(unsigned long long a, unsigned lo

DKSUB16 (64-bit SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Syntax:

```
DKSUB16 Rd, Rs1, Rs2
# Rd, Rs1, Rs2 are all even/odd pair of registers
```

Purpose: Do 16-bit signed integer elements saturating subtractions simultaneously.

Description: This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. If any of the results are beyond the Q15 number range (-2^15 <= Q15 <= 2^15-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] > (2^15)-1) {
  res[x] = (2^15)-1;
  OV = 1;
} else if (res[x] < -2^15) {
  res[x] = -2^15;
```

(continues on next page)

(continued from previous page)

```
  OV = 1;
}
Rd.H[x] = res[x];
for RV32: x=3...0,
```

Return value stored in unsigned long long type

Parameters

- `[in] a`: unsigned long long type of value stored in a
- `[in] b`: unsigned long long type of value stored in b

`__STATIC_FORCEINLINE unsigned long __RV_EXPD80(unsigned long a)`

EXPD80 (Expand and Copy Byte 0 to 32bit)

Type: DSP

Syntax:

```
EXPD80 Rd, Rs1
```

Purpose: Copy 8-bit data from 32-bit chunks into 4 bytes in a register.

Description: Moves Rs1.B[0][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.
↪B[0][7:0]);
for RV32: x=0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_EXPD81(unsigned long a)`

EXPD81 (Expand and Copy Byte 1 to 32bit)

Type: DSP

Syntax:

```
EXPD81 Rd, Rs1
```

Purpose: Copy 8-bit data from 32-bit chunks into 4 bytes in a register.

Description: Moves Rs1.B[1][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.
↪B[1][7:0]);
for RV32: x=0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_EXPD82(unsigned long a)`

EXPD82 (Expand and Copy Byte 2 to 32bit)

Type: DSP

Syntax:

```
EXPD82 Rd, Rs1
```

Purpose: Copy 8-bit data from 32-bit chunks into 4 bytes in a register.

Description: Moves Rs1.B[2][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.
↪B[2][7:0]);
for RV32: x=0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

`__STATIC_FORCEINLINE unsigned long __RV_EXPD83(unsigned long a)`

EXPD83 (Expand and Copy Byte 3 to 32bit)

Type: DSP

Syntax:

```
EXPD83 Rd, Rs1
```

Purpose: Copy 8-bit data from 32-bit chunks into 4 bytes in a register.

Description: Moves Rs1.B[3][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.
↪B[3][7:0]);
for RV32: x=0
```

Return value stored in unsigned long type

Parameters

- `[in] a`: unsigned long type of value stored in a

group **`NMSIS_Core_DSP_Intrinsic`**

Functions that generate RISC-V DSP SIMD instructions.

The following functions generate specified RISC-V SIMD instructions that cannot be directly accessed by compiler.

- **DSP ISA Extension Instruction Summary**
  - **Shorthand Definitions**
    - r.H == rH1: r[31:16], r.L == r.H0: r[15:0]

- r.B3: r[31:24], r.B2: r[23:16], r.B1: r[15:8], r.B0: r[7:0]
- r.B[x]: r[(x*8+7):(x*8+0)]
- r.H[x]: r[(x*16+7):(x*16+0)]
- r.W[x]: r[(x*32+31):(x*32+0)]
- r[xU]: the upper 32-bit of a 64-bit number; xU represents the GPR number that contains this upper part 32-bit value.
- r[xL]: the lower 32-bit of a 64-bit number; xL represents the GPR number that contains this lower part 32-bit value.
- r[xU].r[xL]: a 64-bit number that is formed from a pair of GPRs.
- s>>: signed arithmetic right shift:
- u>>: unsigned logical right shift
- SAT.Qn(): Saturate to the range of [-2^n, 2^n-1], if saturation happens, set PSW.OV.
- SAT.Um(): Saturate to the range of [0, 2^m-1], if saturation happens, set PSW.OV.
- RUND(): Indicate `rounding`, i.e., add 1 to the most significant discarded bit for right shift or MSW-type multiplication instructions.
- Sign or Zero Extending functions:
  - SEm(data): Sign-Extend data to m-bit.:
  - ZEm(data): Zero-Extend data to m-bit.
- ABS(x): Calculate the absolute value of `x`.
- CONCAT(x,y): Concatinate `x` and `y` to form a value.
- u<: Unsinged less than comparison.
- u<=: Unsinged less than & equal comparison.
- u>: Unsinged greater than comparison.
- s*: Signed multiplication.
- u*: Unsigned multiplication.

2.5.12 PMP Functions

`__STATIC_INLINE uint8_t __get_PMPxCFG(uint32_t idx)`

`__STATIC_INLINE void __set_PMPxCFG(uint32_t idx, uint8_t pmpxcfg)`

`__STATIC_INLINE rv_csr_t __get_PMPCFGx(uint32_t idx)`

`__STATIC_INLINE void __set_PMPCFGx(uint32_t idx, rv_csr_t pmpcfg)`

`__STATIC_INLINE rv_csr_t __get_PMPADDRx(uint32_t idx)`

`__STATIC_INLINE void __set_PMPADDRx(uint32_t idx, rv_csr_t pmpaddr)`

group **`NMSIS_Core_PMP_Functions`**

Functions that related to the RISCV Phyiscal Memory Protection.

Optional physical memory protection (PMP) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region.

The PMP can supports region access control settings as small as four bytes.

Functions

`__STATIC_INLINE uint8_t __get_PMPxCFG(uint32_t idx)`

Get 8bit PMPxCFG Register by PMP entry index.

Return the content of the PMPxCFG Register.

Return PMPxCFG Register value

Parameters

- `[in] idx`: PMP region index(0-15)

`__STATIC_INLINE void __set_PMPxCFG(uint32_t idx, uint8_t pmpxcfg)`

Set 8bit PMPxCFG by pmp entry index.

Set the given pmpxcfg value to the PMPxCFG Register.

Parameters

- `[in] idx`: PMPx region index(0-15)
- `[in] pmpxcfg`: PMPxCFG register value to set

`__STATIC_INLINE rv_csr_t __get_PMPCFGx(uint32_t idx)`

Get PMPCFGx Register by index.

Return the content of the PMPCFGx Register.

Return PMPCFGx Register value

Remark

- For RV64, only idx = 0 and idx = 2 is allowed. pmpcfg0 and pmpcfg2 hold the configurations for the 16 PMP entries, pmpcfg1 and pmpcfg3 are illegal
- For RV32, pmpcfg0–pmpcfg3, hold the configurations pmp0cfg–pmp15cfg for the 16 PMP entries

Parameters

- `[in] idx`: PMPCFG CSR index(0-3)

`__STATIC_INLINE void __set_PMPCFGx(uint32_t idx, rv_csr_t pmpcfg)`

Set PMPCFGx by index.

Write the given value to the PMPCFGx Register.

Remark

- For RV64, only idx = 0 and idx = 2 is allowed. pmpcfg0 and pmpcfg2 hold the configurations for the 16 PMP entries, pmpcfg1 and pmpcfg3 are illegal
- For RV32, pmpcfg0–pmpcfg3, hold the configurations pmp0cfg–pmp15cfg for the 16 PMP entries

Parameters

- `[in] idx`: PMPCFG CSR index(0-3)
- `[in] pmpcfg`: PMPCFGx Register value to set

`__STATIC_INLINE rv_csr_t __get_PMPADDRx(uint32_t idx)`

Get PMPADDRx Register by index.

Return the content of the PMPADDRx Register.

Return PMPADDRx Register value

Parameters

- `[in] idx`: PMP region index(0-15)

__STATIC_INLINE void __set_PMPADDRx(uint32_t idx, rv_csr_t pmpaddr)

Set PMPADDRx by index.

Write the given value to the PMPADDRx Register.

Parameters

- `[in] idx`: PMP region index(0-15)
- `[in] pmpaddr`: PMPADDRx Register value to set

2.5.13 Cache Functions

I-Cache Functions

__STATIC_FORCEINLINE void EnableICache(void)

__STATIC_FORCEINLINE void DisableICache(void)

__STATIC_FORCEINLINE void MInvalICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MInvalICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SInvalICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SInvalICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UInvalICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UInvalICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long MLockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long MLockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long SLockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long SLockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long ULockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long ULockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MUnlockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MUnlockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SUnlockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SUnlockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UUnlockICacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UUnlockICacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MInvalICache(void)

__STATIC_FORCEINLINE void SInvalICache(void)

__STATIC_FORCEINLINE void UInvalICache(void)

group **NMSIS_Core_ICache**

Functions that configure Instruction Cache.

Functions

__STATIC_FORCEINLINE void EnableICache(void)

Enable ICache.

This function enable I-Cache

Remark

- This function can be called in M-Mode only.
- This CSR_MCACHE_CTL register control I Cache enable.

See

- DisableICache

__STATIC_FORCEINLINE void DisableICache(void)

Disable ICache.

This function Disable I-Cache

Remark

- This function can be called in M-Mode only.
- This CSR_MCACHE_CTL register control I Cache enable.

See

- EnableICache

__STATIC_FORCEINLINE void MInvalICacheLine(unsigned long addr)

Invalidate one I-Cache line specified by address in M-Mode.

This function unlock and invalidate one I-Cache line specified by the address. Command CCM_IC_INVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

__STATIC_FORCEINLINE void MInvalICacheLines(unsigned long addr, unsigned long cnt)

Invalidate several I-Cache lines specified by address in M-Mode.

This function unlock and invalidate several I-Cache lines specified by the address and line count. Command CCM_IC_INVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be invalidated
- `[in] cnt`: count of cache lines to be invalidated

__STATIC_FORCEINLINE void SInvalICacheLine(unsigned long addr)

Invalidate one I-Cache line specified by address in S-Mode.

This function unlock and invalidate one I-Cache line specified by the address. Command CCM_IC_INVAL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- [in] addr: start address to be invalidated

__STATIC_FORCEINLINE void SInvalICacheLines(unsigned long addr, unsigned long cnt)

Invalidate several I-Cache lines specified by address in S-Mode.

This function unlock and invalidate several I-Cache lines specified by the address and line count. Command CCM_IC_INVAL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- [in] addr: start address to be invalidated
- [in] cnt: count of cache lines to be invalidated

__STATIC_FORCEINLINE void UInvalICacheLine(unsigned long addr)

Invalidate one I-Cache line specified by address in U-Mode.

This function unlock and invalidate one I-Cache line specified by the address. Command CCM_IC_INVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- [in] addr: start address to be invalidated

__STATIC_FORCEINLINE void UInvalICacheLines(unsigned long addr, unsigned long cnt)

Invalidate several I-Cache lines specified by address in U-Mode.

This function unlock and invalidate several I-Cache lines specified by the address and line count. Command CCM_IC_INVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- [in] addr: start address to be invalidated
- [in] cnt: count of cache lines to be invalidated

__STATIC_FORCEINLINE unsigned long MLockICacheLine(unsigned long addr)

Lock one I-Cache line specified by address in M-Mode.

This function lock one I-Cache line specified by the address. Command CCM_IC_LOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- [in] addr: start address to be locked

__STATIC_FORCEINLINE unsigned long MLockICacheLines(unsigned long addr, unsigned long c

Lock several I-Cache lines specified by address in M-Mode.

This function lock several I-Cache lines specified by the address and line count. Command CCM_IC_LOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

`__STATIC_FORCEINLINE unsigned long SLockICacheLine(unsigned long addr)`

Lock one I-Cache line specified by address in S-Mode.

This function lock one I-Cache line specified by the address. Command CCM_IC_LOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked

`__STATIC_FORCEINLINE unsigned long SLockICacheLines(unsigned long addr, unsigned long c`

Lock several I-Cache lines specified by address in S-Mode.

This function lock several I-Cache lines specified by the address and line count. Command CCM_IC_LOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

`__STATIC_FORCEINLINE unsigned long ULockICacheLine(unsigned long addr)`

Lock one I-Cache line specified by address in U-Mode.

This function lock one I-Cache line specified by the address. Command CCM_IC_LOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked

`__STATIC_FORCEINLINE unsigned long ULockICacheLines(unsigned long addr, unsigned long c`

Lock several I-Cache lines specified by address in U-Mode.

This function lock several I-Cache lines specified by the address and line count. Command CCM_IC_LOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

`__STATIC_FORCEINLINE void MUnlockICacheLine(unsigned long addr)`

Unlock one I-Cache line specified by address in M-Mode.

This function unlock one I-Cache line specified by the address. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

`__STATIC_FORCEINLINE void MUnlockICacheLines(unsigned long addr, unsigned long cnt)`

Unlock several I-Cache lines specified by address in M-Mode.

This function unlock several I-Cache lines specified by the address and line count. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be unlocked
- `[in] cnt`: count of cache lines to be unlocked

`__STATIC_FORCEINLINE void SUnlockICacheLine(unsigned long addr)`

Unlock one I-Cache line specified by address in S-Mode.

This function unlock one I-Cache line specified by the address. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

`__STATIC_FORCEINLINE void SUnlockICacheLines(unsigned long addr, unsigned long cnt)`

Unlock several I-Cache lines specified by address in S-Mode.

This function unlock several I-Cache lines specified by the address and line count. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be unlocked
- `[in] cnt`: count of cache lines to be unlocked

`__STATIC_FORCEINLINE void UUnlockICacheLine(unsigned long addr)`

Unlock one I-Cache line specified by address in U-Mode.

This function unlock one I-Cache line specified by the address. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

`__STATIC_FORCEINLINE void UUnlockICacheLines(unsigned long addr, unsigned long cnt)`

Unlock several I-Cache lines specified by address in U-Mode.

This function unlock several I-Cache lines specified by the address and line count. Command CCM_IC_UNLOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- [in] addr: start address to be unlocked
- [in] cnt: count of cache lines to be unlocked

__STATIC_FORCEINLINE void MInvalICache(void)

Invalidate all I-Cache lines in M-Mode.

This function invalidate all I-Cache lines. Command CCM_IC_INVAL_ALL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- [in] addr: start address to be invalidated

__STATIC_FORCEINLINE void SInvalICache(void)

Invalidate all I-Cache lines in S-Mode.

This function invalidate all I-Cache lines. Command CCM_IC_INVAL_ALL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- [in] addr: start address to be invalidated

__STATIC_FORCEINLINE void UInvalICache(void)

Invalidate all I-Cache lines in U-Mode.

This function invalidate all I-Cache lines. Command CCM_IC_INVAL_ALL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- [in] addr: start address to be invalidated

D-Cache Functions

__STATIC_FORCEINLINE void EnableDCache(void)

__STATIC_FORCEINLINE void DisableDCache(void)

__STATIC_FORCEINLINE void MInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MFlushDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MFlushDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SFlushDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SFlushDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UFlushDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UFlushDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MFlushInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MFlushInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SFlushInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SFlushInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UFlushInvalDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UFlushInvalDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long MLockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long MLockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long SLockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long SLockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE unsigned long ULockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE unsigned long ULockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MUnlockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void MUnlockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void SUnlockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void SUnlockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void UUnlockDCacheLine(unsigned long addr)

__STATIC_FORCEINLINE void UUnlockDCacheLines(unsigned long addr, unsigned long cnt)

__STATIC_FORCEINLINE void MInvalDCache(void)

__STATIC_FORCEINLINE void SInvalDCache(void)

__STATIC_FORCEINLINE void UInvalDCache(void)

__STATIC_FORCEINLINE void MFlushDCache(void)

__STATIC_FORCEINLINE void SFlushDCache(void)

__STATIC_FORCEINLINE void UFlushDCache(void)

__STATIC_FORCEINLINE void MFlushInvalDCache(void)

__STATIC_FORCEINLINE void SFlushInvalDCache(void)

__STATIC_FORCEINLINE void UFlushInvalDCache(void)

group **NMSIS_Core_DCache**

Functions that configure Data Cache.

Functions

__STATIC_FORCEINLINE void EnableDCache(void)

Enable DCache.

This function enable D-Cache

Remark

- This function can be called in M-Mode only.
- This CSR_MCACHE_CTL register control D Cache enable.

See

- DisableDCache

`__STATIC_FORCEINLINE void DisableDCache(void)`

Disable DCache.

This function Disable D-Cache

Remark

- This function can be called in M-Mode only.
- This CSR_MCACHE_CTL register control D Cache enable.

See

- EnableDCache

`__STATIC_FORCEINLINE void MInvalDCacheLine(unsigned long addr)`

Invalidate one D-Cache line specified by address in M-Mode.

This function unlock and invalidate one D-Cache line specified by the address. Command CCM_DC_INVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

`__STATIC_FORCEINLINE void MInvalDCacheLines(unsigned long addr, unsigned long cnt)`

Invalidate several D-Cache lines specified by address in M-Mode.

This function unlock and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_INVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be invalidated
- `[in] cnt`: count of cache lines to be invalidated

`__STATIC_FORCEINLINE void SInvalDCacheLine(unsigned long addr)`

Invalidate one D-Cache line specified by address in S-Mode.

This function unlock and invalidate one D-Cache line specified by the address. Command CCM_DC_INVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

`__STATIC_FORCEINLINE void SInvalDCacheLines(unsigned long addr, unsigned long cnt)`

Invalidate several D-Cache lines specified by address in S-Mode.

This function unlock and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_INVAL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be invalidated
- `[in] cnt`: count of cache lines to be invalidated

`__STATIC_FORCEINLINE void UInvalDCacheLine(unsigned long addr)`

Invalidate one D-Cache line specified by address in U-Mode.

This function unlock and invalidate one D-Cache line specified by the address. Command CCM_DC_INVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

`__STATIC_FORCEINLINE void UInvalDCacheLines(unsigned long addr, unsigned long cnt)`

Invalidate several D-Cache lines specified by address in U-Mode.

This function unlock and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_INVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be invalidated
- `[in] cnt`: count of cache lines to be invalidated

`__STATIC_FORCEINLINE void MFlushDCacheLine(unsigned long addr)`

Flush one D-Cache line specified by address in M-Mode.

This function flush one D-Cache line specified by the address. Command CCM_DC_WB is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed

`__STATIC_FORCEINLINE void MFlushDCacheLines(unsigned long addr, unsigned long cnt)`

Flush several D-Cache lines specified by address in M-Mode.

This function flush several D-Cache lines specified by the address and line count. Command CCM_DC_WB is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed
- `[in] cnt`: count of cache lines to be flushed

`__STATIC_FORCEINLINE void SFlushDCacheLine(unsigned long addr)`

Flush one D-Cache line specified by address in S-Mode.

This function flush one D-Cache line specified by the address. Command CCM_DC_WB is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed

`__STATIC_FORCEINLINE void SFlushDCacheLines(unsigned long addr, unsigned long cnt)`

Flush several D-Cache lines specified by address in S-Mode.

This function flush several D-Cache lines specified by the address and line count. Command CCM_DC_WB is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed
- `[in] cnt`: count of cache lines to be flushed

`__STATIC_FORCEINLINE void UFlushDCacheLine(unsigned long addr)`

Flush one D-Cache line specified by address in U-Mode.

This function flush one D-Cache line specified by the address. Command CCM_DC_WB is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed

`__STATIC_FORCEINLINE void UFlushDCacheLines(unsigned long addr, unsigned long cnt)`

Flush several D-Cache lines specified by address in U-Mode.

This function flush several D-Cache lines specified by the address and line count. Command CCM_DC_WB is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed
- `[in] cnt`: count of cache lines to be flushed

`__STATIC_FORCEINLINE void MFlushInvalDCacheLine(unsigned long addr)`

Flush and invalidate one D-Cache line specified by address in M-Mode.

This function flush and invalidate one D-Cache line specified by the address. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated

`__STATIC_FORCEINLINE void MFlushInvalDCacheLines(unsigned long addr, unsigned long cnt`

Flush and invalidate several D-Cache lines specified by address in M-Mode.

This function flush and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated
- `[in] cnt`: count of cache lines to be flushed and invalidated

__STATIC_FORCEINLINE void SFlushInvalDCacheLine(unsigned long addr)

Flush and invalidate one D-Cache line specified by address in S-Mode.

This function flush and invalidate one D-Cache line specified by the address. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated

__STATIC_FORCEINLINE void SFlushInvalDCacheLines(unsigned long addr, unsigned long cnt

Flush and invalidate several D-Cache lines specified by address in S-Mode.

This function flush and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated
- `[in] cnt`: count of cache lines to be flushed and invalidated

__STATIC_FORCEINLINE void UFlushInvalDCacheLine(unsigned long addr)

Flush and invalidate one D-Cache line specified by address in U-Mode.

This function flush and invalidate one D-Cache line specified by the address. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated

__STATIC_FORCEINLINE void UFlushInvalDCacheLines(unsigned long addr, unsigned long cnt

Flush and invalidate several D-Cache lines specified by address in U-Mode.

This function flush and invalidate several D-Cache lines specified by the address and line count. Command CCM_DC_WBINVAL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed and invalidated
- `[in] cnt`: count of cache lines to be flushed and invalidated

__STATIC_FORCEINLINE unsigned long MLockDCacheLine(unsigned long addr)

Lock one D-Cache line specified by address in M-Mode.

This function lock one D-Cache line specified by the address. Command CCM_DC_LOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked

__STATIC_FORCEINLINE unsigned long MLockDCacheLines(unsigned long addr, unsigned long c

Lock several D-Cache lines specified by address in M-Mode.

This function lock several D-Cache lines specified by the address and line count. Command CCM_DC_LOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

__STATIC_FORCEINLINE unsigned long SLockDCacheLine(unsigned long addr)

Lock one D-Cache line specified by address in S-Mode.

This function lock one D-Cache line specified by the address. Command CCM_DC_LOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked

__STATIC_FORCEINLINE unsigned long SLockDCacheLines(unsigned long addr, unsigned long c

Lock several D-Cache lines specified by address in S-Mode.

This function lock several D-Cache lines specified by the address and line count. Command CCM_DC_LOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

__STATIC_FORCEINLINE unsigned long ULockDCacheLine(unsigned long addr)

Lock one D-Cache line specified by address in U-Mode.

This function lock one D-Cache line specified by the address. Command CCM_DC_LOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked

__STATIC_FORCEINLINE unsigned long ULockDCacheLines(unsigned long addr, unsigned long c

Lock several D-Cache lines specified by address in U-Mode.

This function lock several D-Cache lines specified by the address and line count. Command CCM_DC_LOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Return result of CCM lock operation, see enum CCM_OP_FINFO

Parameters

- `[in] addr`: start address to be locked
- `[in] cnt`: count of cache lines to be locked

`__STATIC_FORCEINLINE void MUnlockDCacheLine(unsigned long addr)`

Unlock one D-Cache line specified by address in M-Mode.

This function unlock one D-Cache line specified by the address. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

`__STATIC_FORCEINLINE void MUnlockDCacheLines(unsigned long addr, unsigned long cnt)`

Unlock several D-Cache lines specified by address in M-Mode.

This function unlock several D-Cache lines specified by the address and line count. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be unlocked
- `[in] cnt`: count of cache lines to be unlocked

`__STATIC_FORCEINLINE void SUnlockDCacheLine(unsigned long addr)`

Unlock one D-Cache line specified by address in S-Mode.

This function unlock one D-Cache line specified by the address. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

`__STATIC_FORCEINLINE void SUnlockDCacheLines(unsigned long addr, unsigned long cnt)`

Unlock several D-Cache lines specified by address in S-Mode.

This function unlock several D-Cache lines specified by the address and line count. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be unlocked
- `[in] cnt`: count of cache lines to be unlocked

`__STATIC_FORCEINLINE void UUnlockDCacheLine(unsigned long addr)`

Unlock one D-Cache line specified by address in U-Mode.

This function unlock one D-Cache line specified by the address. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be unlocked

__STATIC_FORCEINLINE void UUnlockDCacheLines(unsigned long addr, unsigned long cnt)

Unlock several D-Cache lines specified by address in U-Mode.

This function unlock several D-Cache lines specified by the address and line count. Command CCM_DC_UNLOCK is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be unlocked
- `[in] cnt`: count of cache lines to be unlocked

__STATIC_FORCEINLINE void MInvalDCache(void)

Invalidate all D-Cache lines in M-Mode.

This function invalidate all D-Cache lines. Command CCM_DC_INVAL_ALL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

__STATIC_FORCEINLINE void SInvalDCache(void)

Invalidate all D-Cache lines in S-Mode.

This function invalidate all D-Cache lines. Command CCM_DC_INVAL_ALL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

__STATIC_FORCEINLINE void UInvalDCache(void)

Invalidate all D-Cache lines in U-Mode.

This function invalidate all D-Cache lines. In U-Mode, this operation will be automatically translated to flush and invalidate operations by hardware. Command CCM_DC_INVAL_ALL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be invalidated

__STATIC_FORCEINLINE void MFlushDCache(void)

Flush all D-Cache lines in M-Mode.

This function flush all D-Cache lines. Command CCM_DC_WB_ALL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed

__STATIC_FORCEINLINE void SFlushDCache(void)

Flush all D-Cache lines in S-Mode.

This function flush all D-Cache lines. Command CCM_DC_WB_ALL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed

__STATIC_FORCEINLINE void UFlushDCache(void)

Flush all D-Cache lines in U-Mode.

This function flush all D-Cache lines. Command CCM_DC_WB_ALL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed

__STATIC_FORCEINLINE void MFlushInvalDCache(void)

Flush and invalidate all D-Cache lines in M-Mode.

This function flush and invalidate all D-Cache lines. Command CCM_DC_WBINVAL_ALL is written to CSR CSR_CCM_MCOMMAND.

Remark This function must be executed in M-Mode only.

Parameters

- `[in] addr`: start address to be flushed and locked

__STATIC_FORCEINLINE void SFlushInvalDCache(void)

Flush and invalidate all D-Cache lines in S-Mode.

This function flush and invalidate all D-Cache lines. Command CCM_DC_WBINVAL_ALL is written to CSR CSR_CCM_SCOMMAND.

Remark This function must be executed in M/S-Mode only.

Parameters

- `[in] addr`: start address to be flushed and locked

__STATIC_FORCEINLINE void UFlushInvalDCache(void)

Flush and invalidate all D-Cache lines in U-Mode.

This function flush and invalidate all D-Cache lines. Command CCM_DC_WBINVAL_ALL is written to CSR CSR_CCM_UCOMMAND.

Remark This function must be executed in M/S/U-Mode only.

Parameters

- `[in] addr`: start address to be flushed and locked

enum NMSIS_Core_Cache::**CCM_OP_FINFO_Type**

Values:

CCM_OP_SUCCESS = 0x0

CCM_OP_EXCEED_ERR = 0x1

CCM_OP_PERM_CHECK_ERR = 0x2

CCM_OP_REFILL_BUS_ERR = 0x3

CCM_OP_ECC_ERR = 0x4

enum NMSIS_Core_Cache::**CCM_CMD_Type**

Values:

CCM_DC_INVAL = 0x0

CCM_DC_WB = 0x1

CCM_DC_WBINVAL = 0x2

CCM_DC_LOCK = 0x3

CCM_DC_UNLOCK = 0x4

CCM_DC_WBINVAL_ALL = 0x6

CCM_DC_WB_ALL = 0x7

CCM_DC_INVAL_ALL = 0x17

CCM_IC_INVAL = 0x8

CCM_IC_LOCK = 0xb

CCM_IC_UNLOCK = 0xc

CCM_IC_INVAL_ALL = 0xd

__STATIC_FORCEINLINE void EnableSUCCM(void)

__STATIC_FORCEINLINE void DisableSUCCM(void)

__STATIC_FORCEINLINE void FlushPipeCCM(void)

CCM_SUEN_SUEN_Pos 0U

CCM_SUEN_SUEN_Msk (1UL << CCM_SUEN_SUEN_Pos)

group **NMSIS_Core_Cache**

Functions that configure Instruction and Data Cache.

Nuclei provide Cache Control and Maintainence(CCM) for software to control and maintain the internal L1 I/D Cache of the RISC-V Core, software can manage the cache flexibly to meet the actual application scenarios.

The CCM operations have 3 types: by single address, by all and flush pipeline. The CCM operations are done via CSR registers, M/S/U mode has its own CSR registers to do CCM operations. By default, CCM operations are not allowed in S/U mode, you can execute EnableSUCCM in M-Mode to enable it.

- API names started with M<operations>, such as MInvalICacheLine must be called in M-Mode only.
- API names started with S<operations>, such as SInvalICacheLine should be called in S-Mode.
- API names started with U<operations>, such as UInvalICacheLine should be called in U-Mode.

Defines

CCM_SUEN_SUEN_Pos 0U

CSR CCM_SUEN: SUEN bit Position.

CCM_SUEN_SUEN_Msk (1UL << CCM_SUEN_SUEN_Pos)

CSR CCM_SUEN: SUEN Mask.

Enums

enum CCM_OP_FINFO_Type
Cache CCM Operation Fail Info.

Values:

CCM_OP_SUCCESS = 0x0
Lock Succeed.

CCM_OP_EXCEED_ERR = 0x1
Exceed the the number of lockable ways(N-Way I/D-Cache, lockable is N-1)

CCM_OP_PERM_CHECK_ERR = 0x2
PMP/sPMP/Page-Table X(I-Cache)/R(D-Cache) permission check failed, or belong to Device/Non-Cacheable address range.

CCM_OP_REFILL_BUS_ERR = 0x3
Refill has Bus Error.

CCM_OP_ECC_ERR = 0x4
ECC Error.

enum CCM_CMD_Type
Cache CCM Command Types.

Values:

CCM_DC_INVAL = 0x0
Unlock and invalidate D-Cache line specified by CSR CCM_XBEGINADDR.

CCM_DC_WB = 0x1
Flush the specific D-Cache line specified by CSR CCM_XBEGINADDR.

CCM_DC_WBINVAL = 0x2
Unlock, flush and invalidate the specific D-Cache line specified by CSR CCM_XBEGINADDR.

CCM_DC_LOCK = 0x3
Lock the specific D-Cache line specified by CSR CCM_XBEGINADDR.

CCM_DC_UNLOCK = 0x4
Unlock the specific D-Cache line specified by CSR CCM_XBEGINADDR.

CCM_DC_WBINVAL_ALL = 0x6
Unlock and flush and invalidate all the valid and dirty D-Cache lines.

CCM_DC_WB_ALL = 0x7
Flush all the valid and dirty D-Cache lines.

CCM_DC_INVAL_ALL = 0x17
Unlock and invalidate all the D-Cache lines.

CCM_IC_INVAL = 0x8
Unlock and invalidate I-Cache line specified by CSR CCM_XBEGINADDR.

CCM_IC_LOCK = 0xb
Lock the specific I-Cache line specified by CSR CCM_XBEGINADDR.

CCM_IC_UNLOCK = 0xc
Unlock the specific I-Cache line specified by CSR CCM_XBEGINADDR.

CCM_IC_INVAL_ALL = 0xd
Unlock and invalidate all the I-Cache lines.

Functions

__STATIC_FORCEINLINE void EnableSUCCM(void)
Enable CCM operation in Supervisor/User Mode.

This function enable CCM operation in Supervisor/User Mode. If enabled, CCM operations in supervisor/user mode will be allowed.

Remark

- This function can be called in M-Mode only.

See

- DisableSUCCM

__STATIC_FORCEINLINE void DisableSUCCM(void)
Disable CCM operation in Supervisor/User Mode.

This function disable CCM operation in Supervisor/User Mode. If not enabled, CCM operations in supervisor/user mode will trigger a *illegal intruction* exception.

Remark

- This function can be called in M-Mode only.

See

- EnableSUCCM

__STATIC_FORCEINLINE void FlushPipeCCM(void)
Flush pipeline after CCM operation.

This function is used to flush pipeline after CCM operations on Cache, it will ensure latest instructions or data can be seen by pipeline.

Remark

- This function can be called in M/S/U-Mode only.

2.5.14 ARM Compatiable Functions

__STATIC_FORCEINLINE uint32_t __REV(uint32_t value)

__STATIC_FORCEINLINE uint32_t __REV16(uint32_t value)

__STATIC_FORCEINLINE int16_t __REVSH(int16_t value)

__STATIC_FORCEINLINE uint32_t __ROR(uint32_t op1, uint32_t op2)

__ISB() __RWMB()

__DSB() __RWMB()

__DMB() __RWMB()

__LDRBT(ptr) __LB((ptr))

__LDRHT(ptr) __LH((ptr))

__LDRT(ptr) __LW((ptr))

__STRBT(val, ptr) __SB((ptr), (val))

__STRHT(val, ptr) __SH((ptr), (val))

__STRT (val, ptr) __SW((ptr), (val))

__SSAT (val, sat) __RV_SCLIP32((val), (sat-1))

__USAT (val, sat) __RV_UCLIP32((val), (sat))

__RBIT (value) __RV_BITREVI((value), 31)

__CLZ (data) __RV_CLZ32(data)

group **NMSIS_Core_ARMCompatiable_Functions**
A few functions that compatiable with ARM CMSIS-Core.

Here we provided a few functions that compatiable with ARM CMSIS-Core, mostly used in the DSP and NN library.

Defines

__ISB () __RWMB()
Instruction Synchronization Barrier, compatiable with ARM.

__DSB () __RWMB()
Data Synchronization Barrier, compatiable with ARM.

__DMB () __RWMB()
Data Memory Barrier, compatiable with ARM.

__LDRBT (ptr) __LB((ptr))
LDRT Unprivileged (8 bit), ARM Compatiable.

__LDRHT (ptr) __LH((ptr))
LDRT Unprivileged (16 bit), ARM Compatiable.

__LDRT (ptr) __LW((ptr))
LDRT Unprivileged (32 bit), ARM Compatiable.

__STRBT (val, ptr) __SB((ptr), (val))
STRT Unprivileged (8 bit), ARM Compatiable.

__STRHT (val, ptr) __SH((ptr), (val))
STRT Unprivileged (16 bit), ARM Compatiable.

__STRT (val, ptr) __SW((ptr), (val))
STRT Unprivileged (32 bit), ARM Compatiable.

__SSAT (val, sat) __RV_SCLIP32((val), (sat-1))
Signed Saturate.

Saturates a signed value.

Return Saturated value

Parameters

- `[in] value`: Value to be saturated
- `[in] sat`: Bit position to saturate to (1..32)

__USAT (val, sat) __RV_UCLIP32((val), (sat))
Unsigned Saturate.

Saturates an unsigned value.

Return Saturated value

Parameters

- `[in] value`: Value to be saturated
- `[in] sat`: Bit position to saturate to (0..31)

__RBIT(value) __RV_BITREVI((value), 31)

Reverse bit order of value.

Reverses the bit order of the given value.

Return Reversed value

Parameters

- `[in] value`: Value to reverse

__CLZ(data) __RV_CLZ32(data)

Count leading zeros.

Counts the number of leading zeros of a data value.

Return number of leading zeros in value

Parameters

- `[in] data`: Value to count the leading zeros

Functions

`__STATIC_FORCEINLINE uint32_t __REV(uint32_t value)`

Reverse byte order (32 bit)

Reverses the byte order in unsigned integer value. For example, 0x12345678 becomes 0x78563412.

Return Reversed value

Parameters

- `[in] value`: Value to reverse

`__STATIC_FORCEINLINE uint32_t __REV16(uint32_t value)`

Reverse byte order (16 bit)

Reverses the byte order within each halfword of a word. For example, 0x12345678 becomes 0x34127856.

Return Reversed value

Parameters

- `[in] value`: Value to reverse

`__STATIC_FORCEINLINE int16_t __REVSH(int16_t value)`

Reverse byte order (16 bit)

Reverses the byte order in a 16-bit value and returns the signed 16-bit result. For example, 0x0080 becomes 0x8000.

Return Reversed value

Parameters

- `[in] value`: Value to reverse

__STATIC_FORCEINLINE uint32_t __ROR(uint32_t op1, uint32_t op2)

Rotate Right in unsigned value (32 bit)

Rotate Right (immediate) provides the value of the contents of a register rotated by a variable number of bits.

Return Rotated value

Parameters

- `[in] op1`: Value to rotate
- `[in] op2`: Number of Bits to rotate(0-31)

CHAPTER

THREE

NMSIS DSP

3.1 Overview

3.1.1 Introduction

This user manual describes the NMSIS DSP software library, a suite of common signal processing functions for use on Nuclei N/NX Class Processors based devices.

The library is divided into a number of functions each covering a specific category:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters
- Matrix functions
- Transform functions
- Motor control functions
- Statistical functions
- Support functions
- Interpolation functions

The library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values.

3.1.2 Using the Library

The library functions are declared in the public file `riscv_math.h` which is placed in the *NMSIS/DSP/Include* folder.

Simply include this file and link the appropriate library in the application and begin calling the library functions.

The Library supports single public header file `riscv_math.h` for Nuclei N/NX Class Processors cores with little endian. Same header file will be used for floating point unit(FPU) variants.

3.1.3 Examples

The library ships with *a number of examples* (page 418) which demonstrate how to use the library functions.

3.1.4 Toolchain Support

The library has been developed and tested with RISCV GCC Toolchain.

The library is being tested in GCC toolchain and updates on this activity will be made available shortly.

3.1.5 Building the Library

The library installer contains a **Makefile** to rebuild libraries on Nuclei RISCV GCC toolchain in the **NMSIS/** folder.

The libraries can be built by run `make gen_dsp_lib`, it will build and install DSP library into **Library/DSP/GCC** folder.

3.1.6 Preprocessor Macros

Each library project have different preprocessor macros.

RISCV_MATH_MATRIX_CHECK: Define macro RISCV_MATH_MATRIX_CHECK for checking on the input and output sizes of matrices

RISCV_MATH_ROUNDING: Define macro RISCV_MATH_ROUNDING for rounding on support functions

RISCV_MATH_LOOPUNROLL: Define macro RISCV_MATH_LOOPUNROLL to enable manual loop unrolling in DSP functions

3.2 Using NMSIS-DSP

Here we will describe how to run the nmsis dsp examples in Nuclei Spike.

3.2.1 Preparation

- Nuclei Modified Spike - `xl_spike`
- Nuclei SDK modified for `xl_spike` branch `dev_xlspike`
- Nuclei RISCV GNU Toolchain
- CMake >= 3.5

3.2.2 Tool Setup

1. Export **PATH** correctly for *xl_spike* and *riscv-nuclei-elf-gcc*

```
export PATH=/path/to/xl_spike/bin:/path/to/riscv-nuclei-elf-gcc/bin/:$PATH
```

3.2.3 Build NMSIS DSP Library

1. Download or clone NMSIS source code into **NMSIS** directory.
2. cd to *NMSIS/NMSIS/* directory
3. Build NMSIS DSP library using `make gen_dsp_lib`

4. Strip debug informations using `make strip_dsp_lib` to make the generated library smaller
5. The dsp library will be generated into `./Library/DSP/GCC` folder
6. The dsp libraries will be look like this:

```
$ ll Library/DSP/GCC/
total 28604
-rw-r--r-- 1 hqfang nucleisys 1847080 Jul 14 14:51 libnmsis_dsp_rv32imac.a
-rw-r--r-- 1 hqfang nucleisys 2515912 Jul 14 14:51 libnmsis_dsp_rv32imacp.a
-rw-r--r-- 1 hqfang nucleisys 1786008 Jul 14 14:51 libnmsis_dsp_rv32imafc.a
-rw-r--r-- 1 hqfang nucleisys 2377420 Jul 14 14:51 libnmsis_dsp_rv32imafcp.a
-rw-r--r-- 1 hqfang nucleisys 1785500 Jul 14 14:51 libnmsis_dsp_rv32imafdc.a
-rw-r--r-- 1 hqfang nucleisys 2367840 Jul 14 14:51 libnmsis_dsp_rv32imafdcp.a
-rw-r--r-- 1 hqfang nucleisys 2374468 Jul 14 14:51 libnmsis_dsp_rv64imac.a
-rw-r--r-- 1 hqfang nucleisys 3369340 Jul 14 14:51 libnmsis_dsp_rv64imacp.a
-rw-r--r-- 1 hqfang nucleisys 2276836 Jul 14 14:51 libnmsis_dsp_rv64imafc.a
-rw-r--r-- 1 hqfang nucleisys 3151172 Jul 14 14:51 libnmsis_dsp_rv64imafcp.a
-rw-r--r-- 1 hqfang nucleisys 2275828 Jul 14 14:51 libnmsis_dsp_rv64imafdc.a
-rw-r--r-- 1 hqfang nucleisys 3140188 Jul 14 14:51 libnmsis_dsp_rv64imafdcp.a
```

7. library name with extra `p` is build with RISCV DSP enabled.
   - `libnmsis_dsp_rv32imac.a`: Build for **RISCV_ARCH=rv32imac** without DSP enabled.
   - `libnmsis_dsp_rv32imacp.a`: Build for **RISCV_ARCH=rv32imac** with DSP enabled.

Note:

- You can also directly build both DSP and NN library using `make gen`
- You can strip the generated DSP and NN library using `make strip`

3.2.4 How to run

1. Set environment variables `NUCLEI_SDK_ROOT` and `NUCLEI_SDK_NMSIS`, and set Nuclei SDK SoC to *xl-spike*

```
export NUCLEI_SDK_ROOT=/path/to/nuclei_sdk
export NUCLEI_SDK_NMSIS=/path/to/NMSIS/NMSIS
export SOC=xlspike
```

2. Let us take `./riscv_class_marks_example/` for example
3. `cd ./riscv_class_marks_example/`
4. Run with RISCV DSP enabled NMSIS-DSP library for CORE `n307`

```
# Clean project
make DSP_ENABLE=ON CORE=n307 clean
# Build project
make DSP_ENABLE=ON CORE=n307 all
# Run application using xl_spike
make DSP_ENABLE=ON CORE=n307 run
```

5. Run with RISCV DSP disabled NMSIS-DSP library for CORE `n307`

```
make DSP_ENABLE=OFF CORE=n307 clean
make DSP_ENABLE=OFF CORE=n307 all
make DSP_ENABLE=OFF CORE=n307 run
```

Note:

- You can easily run this example in your hardware, if you have enough memory to run it, just modify the `SOC` to the one your are using in step 1.

3.3 NMSIS DSP API

If you want to access doxygen generated NMSIS DSP API, please click NMSIS DSP API Doxygen Documentation.

3.3.1 Examples

Bayes Example

group **BayesExample** **Description:**

Demonstrates the use of Bayesian classifier functions. It is complementing the tutorial about classical ML with NMSIS-DSP and python scikit-learn: https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/implement-classical-ml-with-arm-nmsis-dsp-libraries

Class Marks Example

group **ClassMarks**

Refer riscv_class_marks_example_f32.c

Description:

Demonstrates the use the Maximum, Minimum, Mean, Standard Deviation, Variance and Matrix functions to calculate statistical values of marks obtained in a class.

Note This example also demonstrates the usage of static initialization.

Variables Description:

- `testMarks_f32` points to the marks scored by 20 students in 4 subjects
- `max_marks` Maximum of all marks
- `min_marks` Minimum of all marks
- `mean` Mean of all marks
- `var` Variance of the marks
- `std` Standard deviation of the marks
- `numStudents` Total number of students in the class

NMSIS DSP Software Library Functions Used:

- riscv_mat_init_f32()
- riscv_mat_mult_f32()
- riscv_max_f32()
- riscv_min_f32()
- riscv_mean_f32()
- riscv_std_f32()
- riscv_var_f32()

Convolution Example

group **ConvolutionExample**

Refer riscv_convolution_example_f32.c

Description:

Demonstrates the convolution theorem with the use of the Complex FFT, Complex-by-Complex Multiplication, and Support Functions.

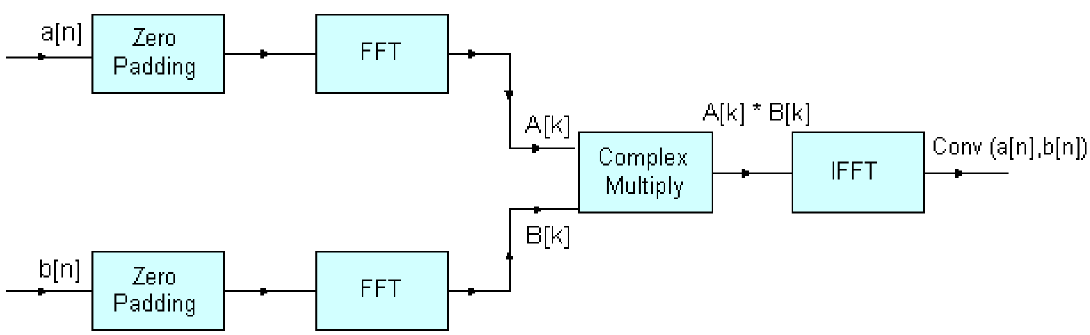
Algorithm:

The convolution theorem states that convolution in the time domain corresponds to multiplication in the frequency domain. Therefore, the Fourier transform of the convoution of two signals is equal to the product of their individual Fourier transforms. The Fourier transform of a signal can be evaluated efficiently using the Fast Fourier Transform (FFT).

Two input signals, `a[n]` and `b[n]`, with lengths `n1` and `n2` respectively, are zero padded so that their lengths become `N`, which is greater than or equal to `(n1+n2-1)` and is a power of 4 as FFT implementation is radix-4. The convolution of `a[n]` and `b[n]` is obtained by taking the FFT of the input signals, multiplying the Fourier transforms of the two signals, and taking the inverse FFT of the multiplied result.

This is denoted by the following equations: where `A[k]` and `B[k]` are the N-point FFTs of the signals `a[n]` and `b[n]` respectively. The length of the convolved signal is `(n1+n2-1)`.

Block Diagram:



Variables Description:

- `testInputA_f32` points to the first input sequence
- `srcALen` length of the first input sequence

- `testInputB_f32` points to the second input sequence
- `srcBLen` length of the second input sequence
- `outLen` length of convolution output sequence, `(srcALen + srcBLen - 1)`
- `AxB` points to the output array where the product of individual FFTs of inputs is stored.

NMSIS DSP Software Library Functions Used:

- riscv_fill_f32()
- riscv_copy_f32()
- riscv_cfft_radix4_init_f32()
- riscv_cfft_radix4_f32()
- riscv_cmplx_mult_cmplx_f32()

Dot Product Example

group **`DotproductExample`**

Refer riscv_dotproduct_example_f32.c

Description:

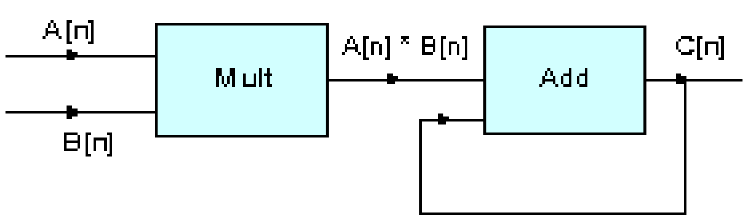
Demonstrates the use of the Multiply and Add functions to perform the dot product. The dot product of two vectors is obtained by multiplying corresponding elements and summing the products.

Algorithm:

The two input vectors `A` and `B` with length `n`, are multiplied element-by-element and then added to obtain dot product.

This is denoted by the following equation:

Block Diagram:



Variables Description:

- `srcA_buf_f32` points to first input vector
- `srcB_buf_f32` points to second input vector
- `testOutput` stores dot product of the two input vectors.

NMSIS DSP Software Library Functions Used:

- riscv_mult_f32()

- riscv_add_f32()

Frequency Bin Example

group **FrequencyBin**

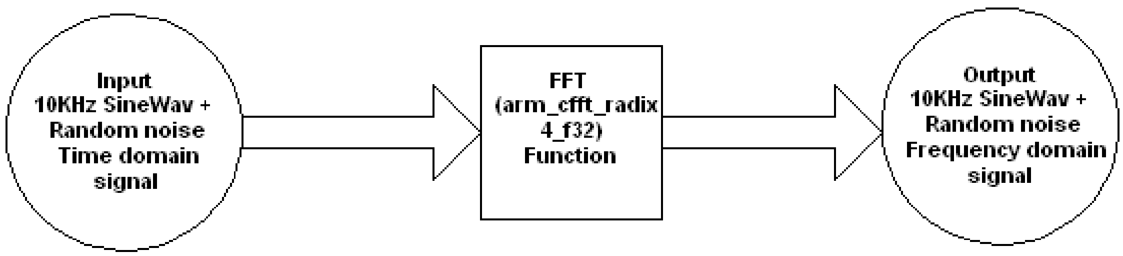
Refer riscv_fft_bin_example_f32.c

Description

Demonstrates the calculation of the maximum energy bin in the frequency domain of the input signal with the use of Complex FFT, Complex Magnitude, and Maximum functions.

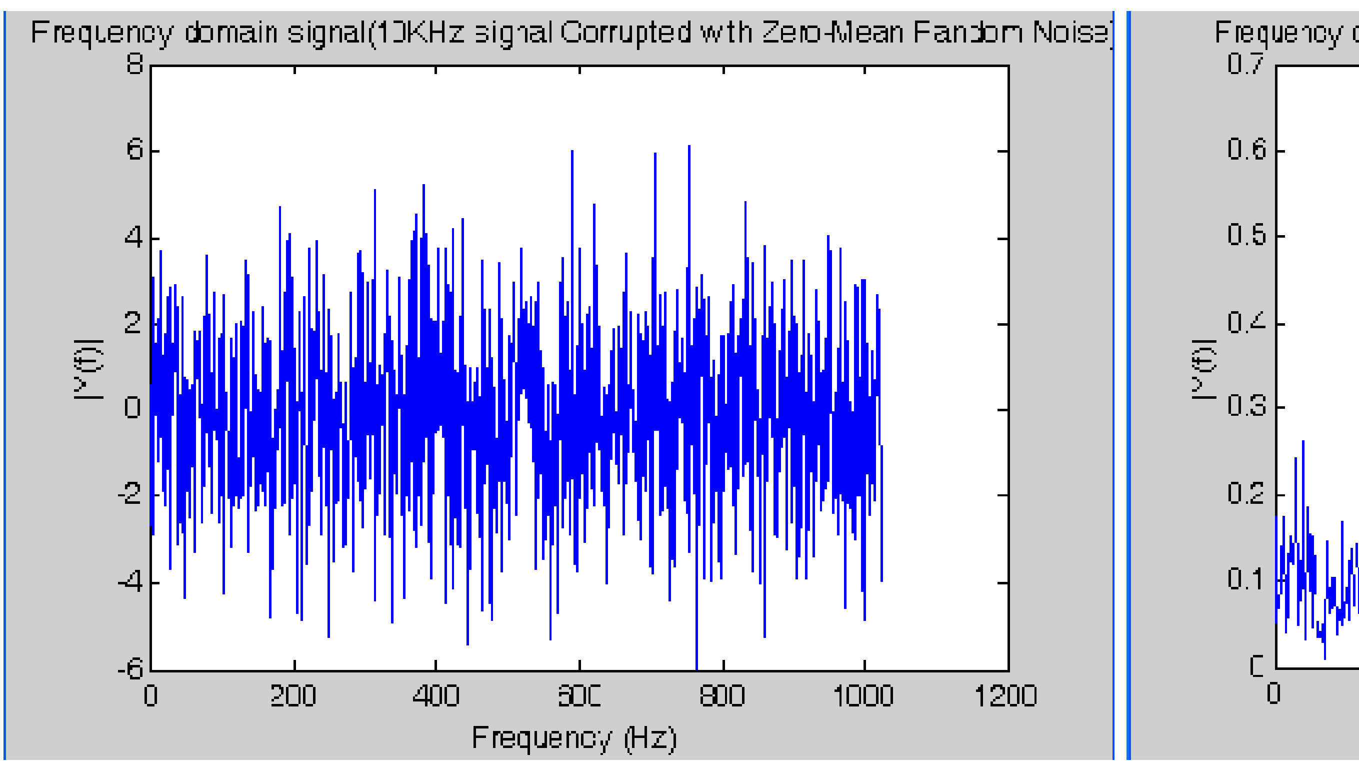
Algorithm:

The input test signal contains a 10 kHz signal with uniformly distributed white noise. Calculating the FFT of the input signal will give us the maximum energy of the bin corresponding to the input frequency of 10 kHz.



Block Diagram:

The figure below shows the time domain signal of 10 kHz signal with uniformly distributed white noise, and the next figure shows the input in the frequency domain. The bin with maximum energy corresponds to 10 kHz signal.



Variables Description:

- `testInput_f32_10khz` points to the input data
- `testOutput` points to the output data
- `fftSize` length of FFT
- `ifftFlag` flag for the selection of CFFT/CIFFT
- `doBitReverse` Flag for selection of normal order or bit reversed order
- `refIndex` reference index value at which maximum energy of bin ocuurs
- `testIndex` calculated index value at which maximum energy of bin ocuurs

NMSIS DSP Software Library Functions Used:

- riscv_cfft_f32()
- riscv_cmplx_mag_f32()
- riscv_max_f32()

FIR Lowpass Filter Example

group **FIRLPF**

Refer riscv_fir_example_f32.c

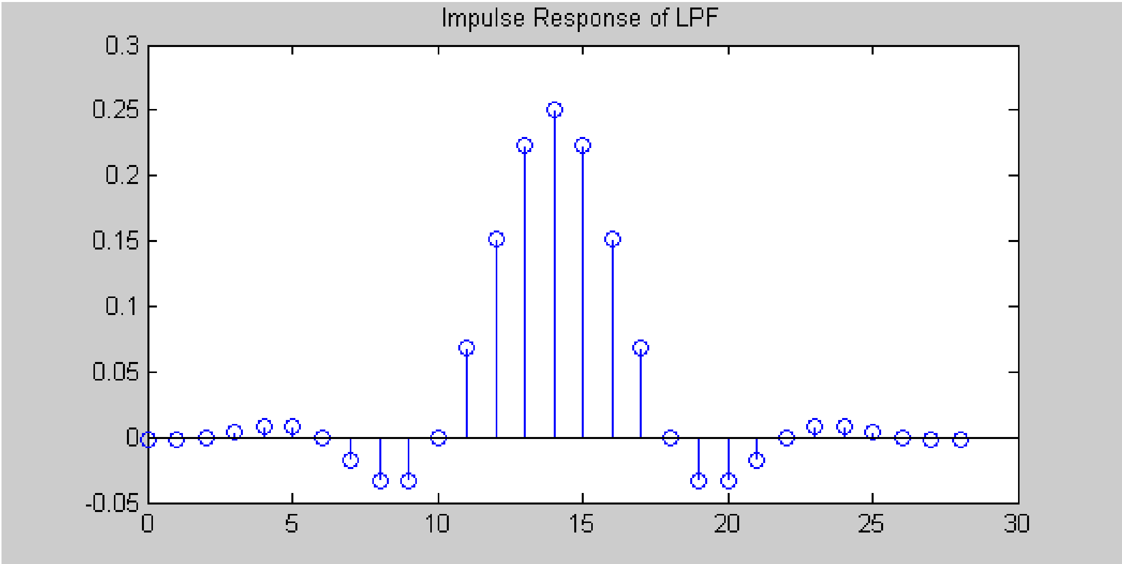
Description:

Removes high frequency signal components from the input using an FIR lowpass filter. The example demonstrates how to configure an FIR filter and then pass data through it in a block-by-block fashion.

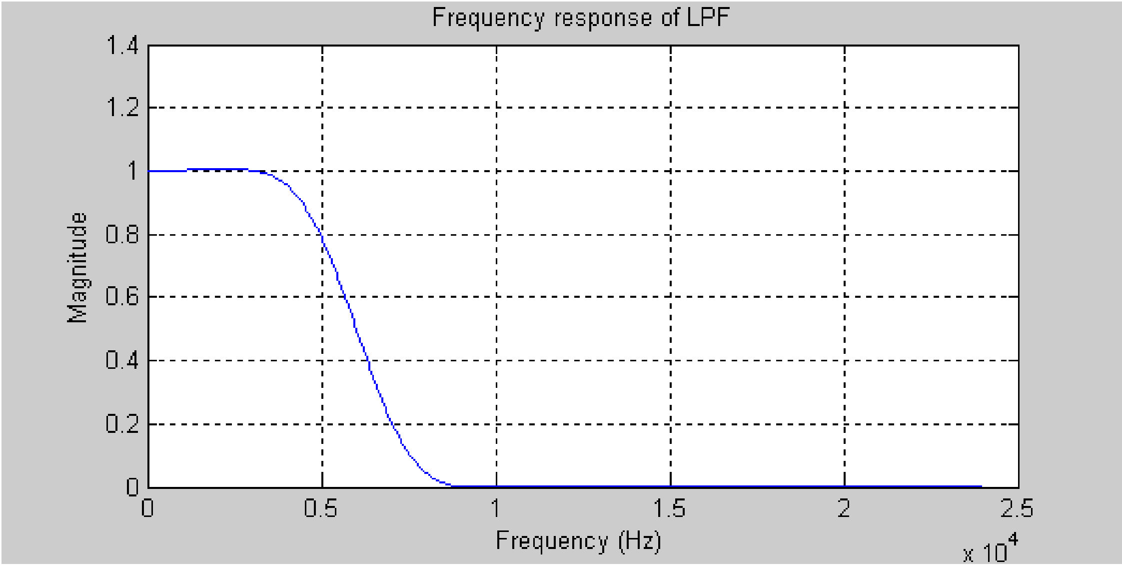
Algorithm:

The input signal is a sum of two sine waves: 1 kHz and 15 kHz. This is processed by an FIR lowpass filter with cutoff frequency 6 kHz. The lowpass filter eliminates the 15 kHz signal leaving only the 1 kHz sine wave at the output.

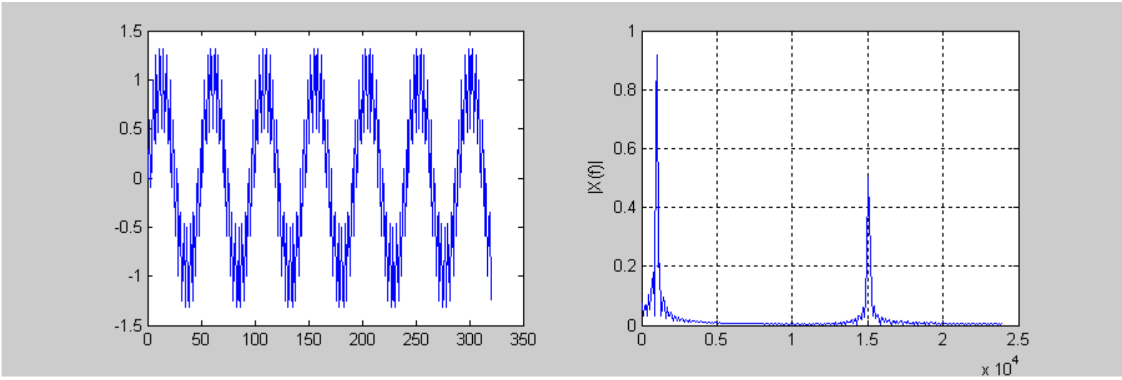
The lowpass filter was designed using MATLAB with a sample rate of 48 kHz and a length of 29 points. The MATLAB code to generate the filter coefficients is shown below: The first argument is the "order" of the filter and is always one less than the desired length. The second argument is the normalized cutoff frequency. This is in the range 0 (DC) to 1.0 (Nyquist). A 6 kHz cutoff with a Nyquist frequency of 24 kHz lies at a normalized frequency of 6/24 = 0.25. The NMSIS FIR filter function requires the coefficients to be in time reversed order. The resulting filter coefficients and are shown below. Note that the filter is symmetric (a property of linear phase FIR filters) and the point of symmetry is sample 14. Thus the filter will have a delay of 14 samples for all frequencies.



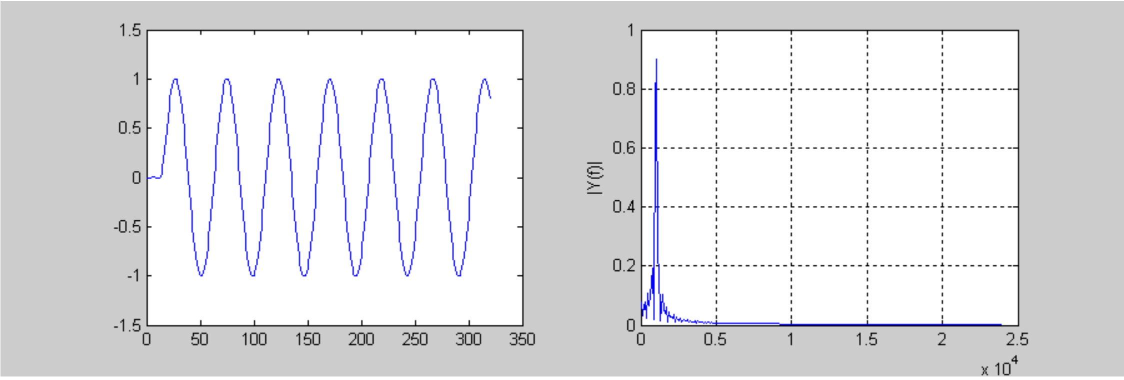
The frequency response of the filter is shown next. The passband gain of the filter is 1.0 and it reaches 0.5 at the cutoff frequency 6 kHz.



The input signal is shown below. The left hand side shows the signal in the time domain while the right hand side is a frequency domain representation. The two sine wave components can be clearly seen.



The output of the filter is shown below. The 15 kHz component has been eliminated.



Variables Description:

- `testInput_f32_1kHz_15kHz` points to the input data
- `refOutput` points to the reference output data
- `testOutput` points to the test output data
- `firStateF32` points to state buffer
- `firCoeffs32` points to coefficient buffer
- `blockSize` number of samples processed at a time
- `numBlocks` number of frames

NMSIS DSP Software Library Functions Used:

- riscv_fir_init_f32()
- riscv_fir_f32()

Graphic Audio Equalizer Example

group **GEQ5Band**

Refer riscv_graphic_equalizer_example_q31.c

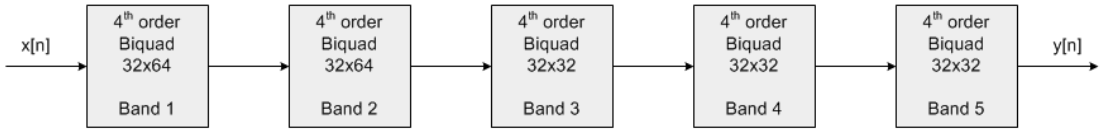
Description:

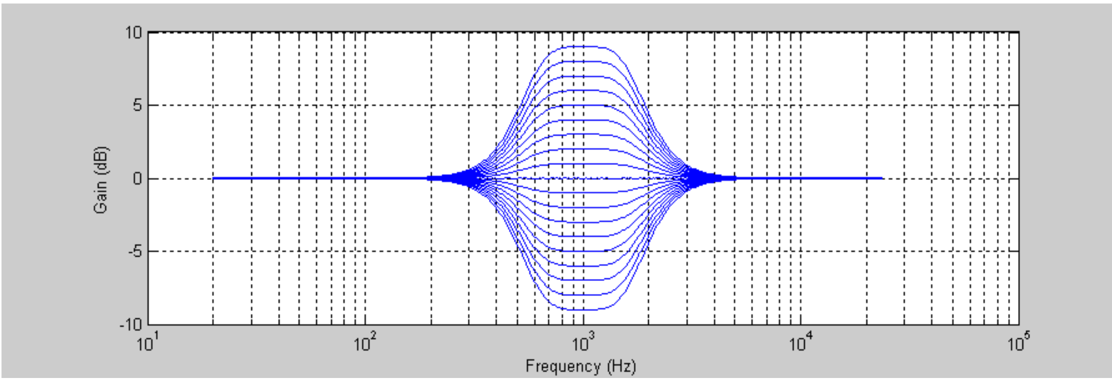
This example demonstrates how a 5-band graphic equalizer can be constructed using the Biquad cascade functions. A graphic equalizer is used in audio applications to vary the tonal quality of the audio.

Block Diagram:

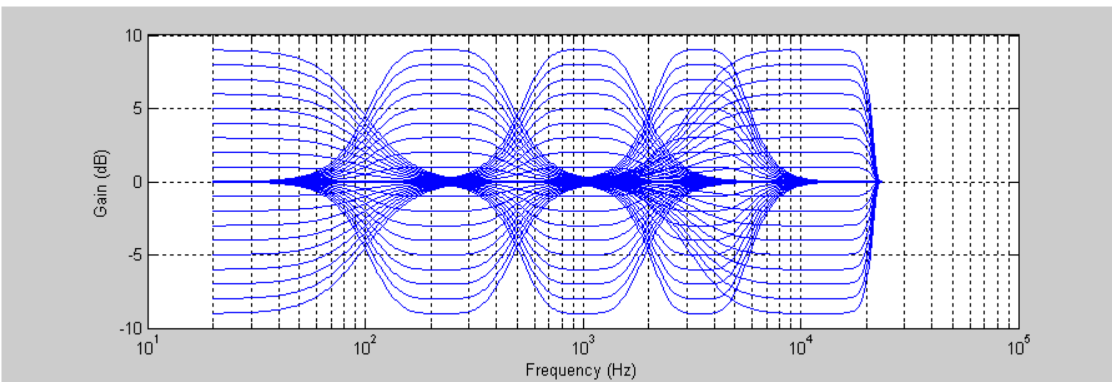
The design is based on a cascade of 5 filter sections. Each filter section is 4th order and consists of a cascade of two Biquads. Each filter has a nominal gain of 0 dB (1.0 in linear units) and boosts or cuts signals within a specific frequency range. The edge frequencies between the 5 bands are 100, 500, 2000, and 6000 Hz. Each band has an adjustable boost or cut in the range of +/-

9 dB. For example, the band that extends from 500 to 2000 Hz has the response shown below:

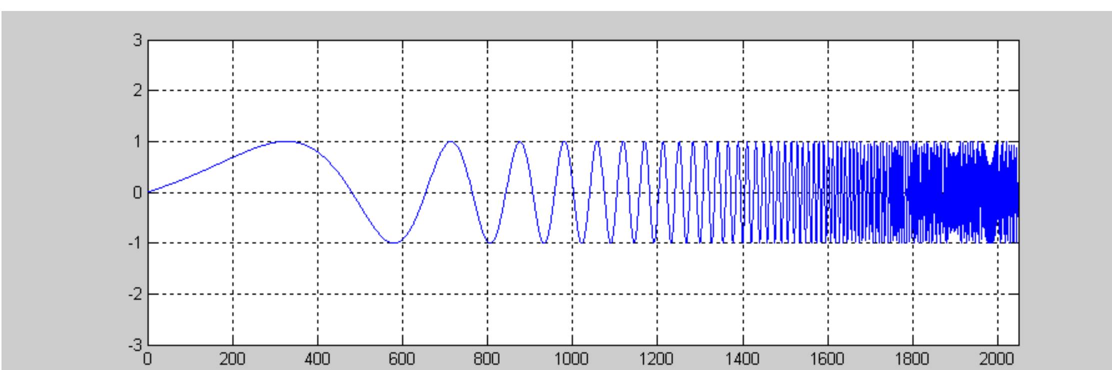




With 1 dB steps, each filter has a total of 19 different settings. The filter coefficients for all possible 19 settings were precomputed in MATLAB and stored in a table. With 5 different tables, there are a total of 5 x 19 = 95 different 4th order filters. All 95 responses are shown below:

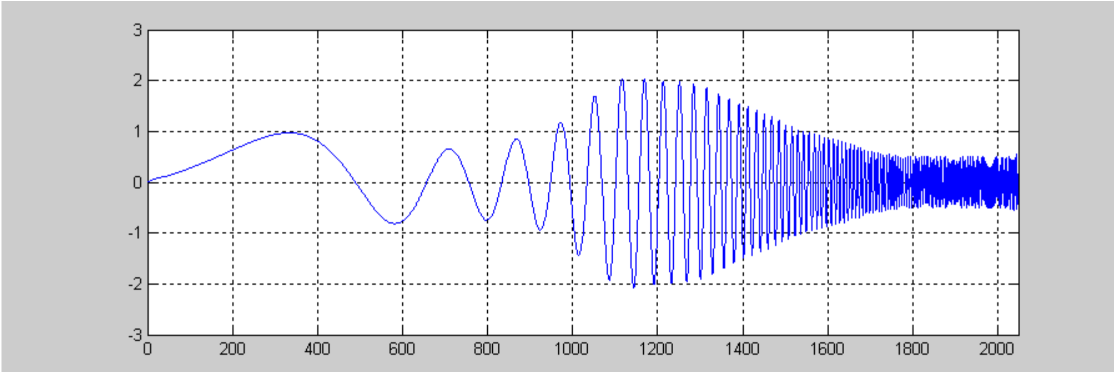


Each 4th order filter has 10 coefficents for a grand total of 950 different filter coefficients that must be tabulated. The input and output data is in Q31 format. For better noise performance, the two low frequency bands are implemented using the high precision 32x64-bit Biquad filters. The remaining 3 high frequency bands use standard 32x32-bit Biquad filters. The input signal used in the example is a logarithmic chirp.



The array `bandGains` specifies the gain in dB to apply in each band. For example, if `bandGains={0,`

`-3, 6, 4, -6};` then the output signal will be:



Note The output chirp signal follows the gain or boost of each band.

Variables Description:

- `testInput_f32` points to the input data
- `testRefOutput_f32` points to the reference output data
- `testOutput` points to the test output data
- `inputQ31` temporary input buffer
- `outputQ31` temporary output buffer
- `biquadStateBand1Q31` points to state buffer for band1
- `biquadStateBand2Q31` points to state buffer for band2
- `biquadStateBand3Q31` points to state buffer for band3
- `biquadStateBand4Q31` points to state buffer for band4
- `biquadStateBand5Q31` points to state buffer for band5
- `coeffTable` points to coefficient buffer for all bands
- `gainDB` gain buffer which has gains applied for all the bands

NMSIS DSP Software Library Functions Used:

- riscv_biquad_cas_df1_32x64_init_q31()
- riscv_biquad_cas_df1_32x64_q31()
- riscv_biquad_cascade_df1_init_q31()
- riscv_biquad_cascade_df1_q31()
- riscv_scale_q31()
- riscv_scale_f32()
- riscv_float_to_q31()
- riscv_q31_to_float()

Linear Interpolate Example

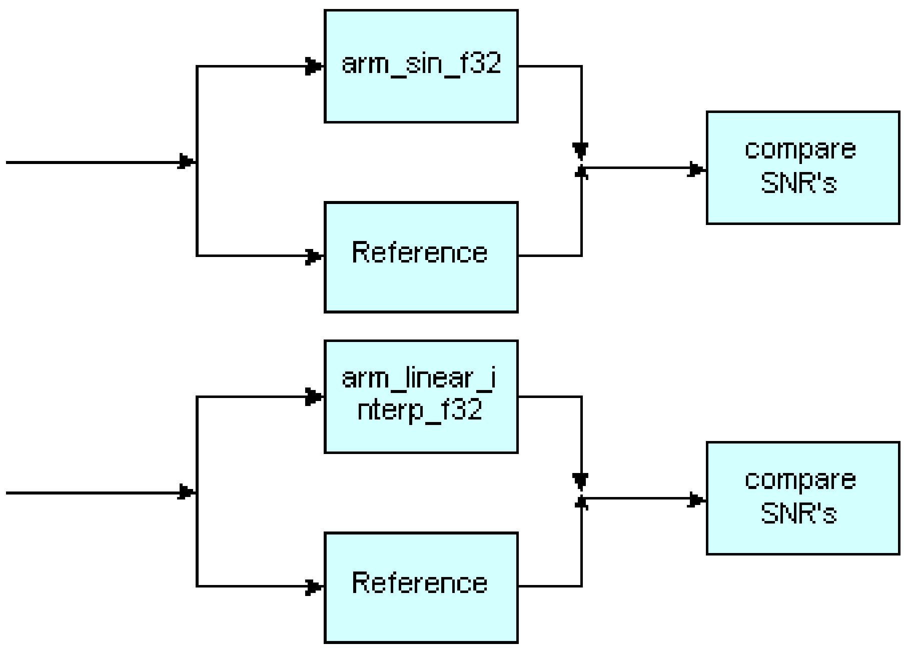
group **LinearInterpExample**

NMSIS DSP Software Library Linear Interpolate Example

Description This example demonstrates usage of linear interpolate modules and fast math modules. Method 1 uses fast math sine function to calculate sine values using cubic interpolation and method 2 uses linear interpolation function and results are compared to reference output. Example shows linear interpolation function can be used to get higher precision compared to fast math sin calculation.

Refer riscv_linear_interp_example_f32.c

Block Diagram:



Variables Description:

- `testInputSin_f32` points to the input values for sine calculation
- `testRefSinOutput32_f32` points to the reference values caculated from sin() matlab function
- `testOutput` points to output buffer calculation from cubic interpolation
- `testLinIntOutput` points to output buffer calculation from linear interpolation
- `snr1` Signal to noise ratio for reference and cubic interpolation output
- `snr2` Signal to noise ratio for reference and linear interpolation output

NMSIS DSP Software Library Functions Used:

- riscv_sin_f32()
- riscv_linear_interp_f32()

Matrix Example

group **MatrixExample**

Refer riscv_matrix_example_f32.c

Description:

Demonstrates the use of Matrix Transpose, Matrix Muliplication, and Matrix Inverse functions to apply least squares fitting to input data. Least squares fitting is the procedure for finding the best-fitting curve that minimizes the sum of the squares of the offsets (least square error) from a given set of data.

Algorithm:

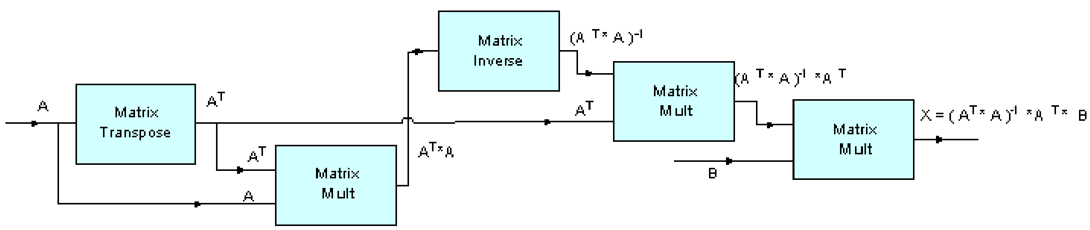
The linear combination of parameters considered is as follows:

`A * X = B`, where `X` is the unknown value and can be estimated from `A` & `B`.

The least squares estimate `X` is given by the following equation:

`X = Inverse(A * A) * A * B`

Block Diagram:



Variables Description:

- `A_f32` input matrix in the linear combination equation
- `B_f32` output matrix in the linear combination equation
- `X_f32` unknown matrix estimated using `A_f32` & `B_f32` matrices

NMSIS DSP Software Library Functions Used:

- riscv_mat_init_f32()
- riscv_mat_trans_f32()
- riscv_mat_mult_f32()
- riscv_mat_inverse_f32()

Signal Convergence Example

group **SignalConvergence**

Refer riscv_signal_converge_example_f32.c

Description:

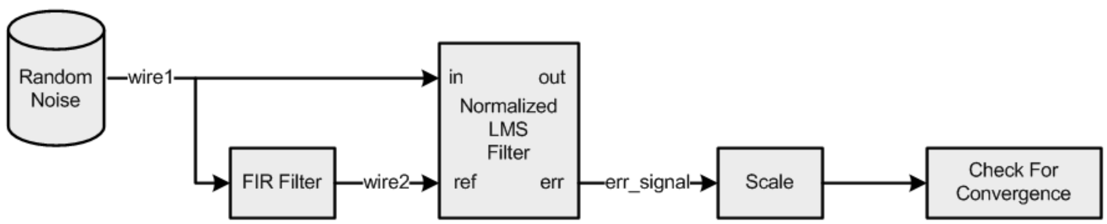
Demonstrates the ability of an adaptive filter to "learn" the transfer function of a FIR lowpass filter using the Normalized LMS Filter, Finite Impulse Response (FIR) Filter, and Basic Math Functions.

Algorithm:

The figure below illustrates the signal flow in this example. Uniformly distributed white noise is passed through an FIR lowpass filter. The output of the FIR filter serves as the reference input of the adaptive filter (normalized LMS filter). The white noise is input to the adaptive filter. The adaptive filter learns the transfer function of the FIR filter. The filter outputs two signals: (1) the output of the internal adaptive FIR filter, and (2) the error signal which is the difference between the adaptive filter and the reference output of the FIR filter. Over time as the adaptive filter learns the transfer function of the FIR filter, the first output approaches the reference output of the FIR filter, and the error signal approaches zero.

The adaptive filter converges properly even if the input signal has a large dynamic range (i.e., varies from small to large values). The coefficients of the adaptive filter are initially zero, and then converge over 1536 samples. The internal function test_signal_converge() implements the stopping condition. The function checks if all of the values of the error signal have a magnitude below a threshold DELTA.

Block Diagram:



Variables Description:

- `testInput_f32` points to the input data
- `firStateF32` points to FIR state buffer
- `lmsStateF32` points to Normalised Least mean square FIR filter state buffer
- `FIRCoeff_f32` points to coefficient buffer
- `lmsNormCoeff_f32` points to Normalised Least mean square FIR filter coefficient buffer
- `wire1`, wir2, wire3 temporary buffers
- `errOutput`, err_signal temporary error buffers

NMSIS DSP Software Library Functions Used:

- riscv_lms_norm_init_f32()
- riscv_fir_init_f32()
- riscv_fir_f32()
- riscv_lms_norm_f32()
- riscv_scale_f32()
- riscv_abs_f32()
- riscv_sub_f32()
- riscv_min_f32()
- riscv_copy_f32()

SineCosine Example

group **SinCosExample**

Refer riscv_sin_cos_example_f32.c

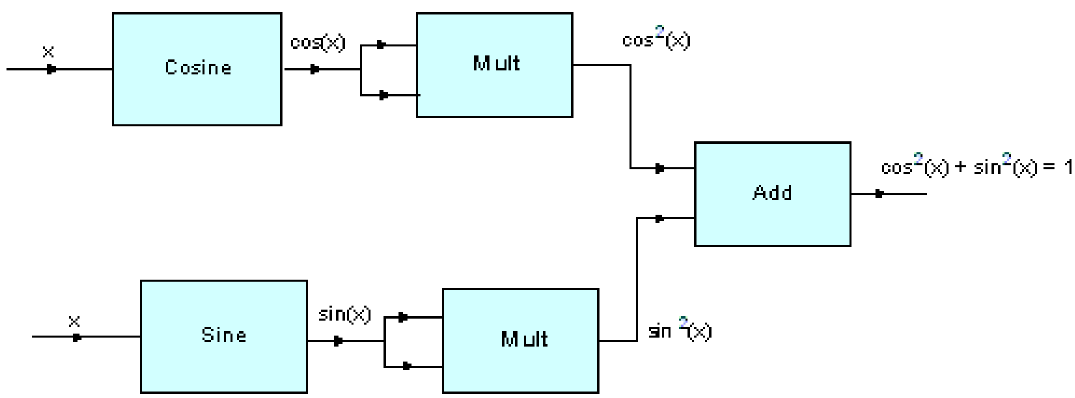
Description:

Demonstrates the Pythagorean trignometric identity with the use of Cosine, Sine, Vector Multiplication, and Vector Addition functions.

Algorithm:

Mathematically, the Pythagorean trignometric identity is defined by the following equation: where `x` is the angle in radians.

Block Diagram:



Variables Description:

- `testInput_f32` array of input angle in radians
- `testOutput` stores sum of the squares of sine and cosine values of input angle

NMSIS DSP Software Library Functions Used:

- riscv_cos_f32()
- riscv_sin_f32()
- riscv_mult_f32()
- riscv_add_f32()

SVM Example

group **SVMExample**

Description:

Demonstrates the use of SVM functions. It is complementing the tutorial about classical ML with NMSIS-DSP and python scikit-learn: https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/implement-classical-ml-with-arm-nmsis-dsp-libraries

Variance Example

group **VarianceExample**

Refer riscv_variance_example_f32.c

Description:

Demonstrates the use of Basic Math and Support Functions to calculate the variance of an input sequence with N samples. Uniformly distributed white noise is taken as input.

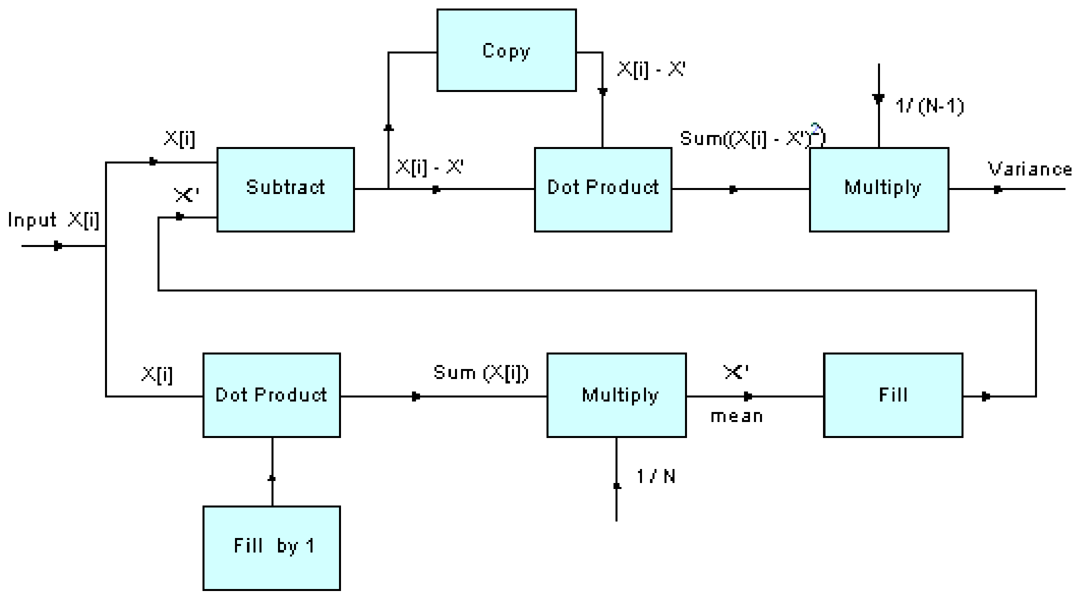
Algorithm:

The variance of a sequence is the mean of the squared deviation of the sequence from its mean.

This is denoted by the following equation: where, `x[n]` is the input sequence, `N` is the number of input samples, and `x'` is the mean value of the input sequence, `x[n]`.

The mean value `x'` is defined as:

Block Diagram:



Variables Description:

- `testInput_f32` points to the input data
- `wire1`, `wir2`, `wire3` temporary buffers
- `blockSize` number of samples processed at a time
- `refVarianceOut` reference variance value

NMSIS DSP Software Library Functions Used:

- riscv_dot_prod_f32()
- riscv_mult_f32()
- riscv_sub_f32()
- riscv_fill_f32()

- riscv_copy_f32()

group **groupExamples**

3.3.2 Basic Math Functions

Vector Absolute Value

void **riscv_abs_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_abs_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_abs_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_abs_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_abs_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicAbs**

Computes the absolute value of a vector on an element-by-element basis.

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_abs_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Floating-point vector absolute value.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_abs_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Floating-point vector absolute value.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_abs_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Q15 vector absolute value.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q15 value -1 (0x8000) will be saturated to the maximum allowable positive value 0x7FFF.

Parameters

- [in] pSrc: points to the input vector
- [out] pDst: points to the output vector
- [in] blockSize: number of samples in each vector

void **riscv_abs_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)
Q31 vector absolute value.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q31 value -1 (0x80000000) will be saturated to the maximum allowable positive value 0x7FFFFFFF.

Parameters

- [in] pSrc: points to the input vector
- [out] pDst: points to the output vector
- [in] blockSize: number of samples in each vector

void **riscv_abs_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Q7 vector absolute value.

Return none

Conditions for optimum performance Input and output buffers should be aligned by 32-bit

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q7 value -1 (0x80) will be saturated to the maximum allowable positive value 0x7F.

Parameters

- [in] pSrc: points to the input vector
- [out] pDst: points to the output vector
- [in] blockSize: number of samples in each vector

Vector Addition

void **riscv_add_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_add_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_add_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_add_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_add_q7** (**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicAdd**
Element-by-element addition of two vectors.

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_add_f16**(**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

Floating-point vector addition.

Return none

Parameters

- `[in] pSrcA`: points to first input vector
- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_add_f32**(**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

Floating-point vector addition.

Return none

Parameters

- `[in] pSrcA`: points to first input vector
- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_add_q15**(**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

Q15 vector addition.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_add_q31**(**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)

Q31 vector addition.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector

- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_add_q7**(**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)
Q7 vector addition.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector bitwise AND

void **riscv_and_u16**(**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)

void **riscv_and_u32**(**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)

void **riscv_and_u8**(**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

group **And**
Compute the logical bitwise AND.

There are separate functions for uint32_t, uint16_t, and uint7_t data types.

Functions

void **riscv_and_u16**(**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise AND of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_and_u32**(**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise AND of two fixed-point vectors.

Return none

Parameters

- [in] pSrcA: points to input vector A
- [in] pSrcB: points to input vector B
- [out] pDst: points to output vector
- [in] blockSize: number of samples in each vector

void **riscv_and_u8** (**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise AND of two fixed-point vectors.

Return none

Parameters

- [in] pSrcA: points to input vector A
- [in] pSrcB: points to input vector B
- [out] pDst: points to output vector
- [in] blockSize: number of samples in each vector

Elementwise clipping

void **riscv_clip_f16** (**const** float16_t **pSrc*, float16_t **pDst*, float16_t *low*, float16_t *high*, uint32_t *numSamples*)

void **riscv_clip_f32** (**const** float32_t **pSrc*, float32_t **pDst*, float32_t *low*, float32_t *high*, uint32_t *numSamples*)

void **riscv_clip_q15** (**const** q15_t **pSrc*, q15_t **pDst*, q15_t *low*, q15_t *high*, uint32_t *numSamples*)

void **riscv_clip_q31** (**const** q31_t **pSrc*, q31_t **pDst*, q31_t *low*, q31_t *high*, uint32_t *numSamples*)

void **riscv_clip_q7** (**const** q7_t **pSrc*, q7_t **pDst*, q7_t *low*, q7_t *high*, uint32_t *numSamples*)

group **BasicClip**

Element-by-element clipping of a value.

The value is constrained between 2 bounds.

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_clip_f16** (**const** float16_t **pSrc*, float16_t **pDst*, float16_t *low*, float16_t *high*, uint32_t *numSamples*)

Elementwise floating-point clipping.

Return none

Parameters

- [in] pSrc: points to input values
- [out] pDst: points to output clipped values
- [in] low: lower bound
- [in] high: higher bound

- `[in] numSamples`: number of samples to clip

void **riscv_clip_f32** (**const** float32_t **pSrc*, float32_t **pDst*, float32_t *low*, float32_t *high*, uint32_t *numSamples*)

Elementwise floating-point clipping.

Return none

Parameters

- `[in] pSrc`: points to input values
- `[out] pDst`: points to output clipped values
- `[in] low`: lower bound
- `[in] high`: higher bound
- `[in] numSamples`: number of samples to clip

void **riscv_clip_q15** (**const** q15_t **pSrc*, q15_t **pDst*, q15_t *low*, q15_t *high*, uint32_t *numSamples*)

Elementwise fixed-point clipping.

Return none

Parameters

- `[in] pSrc`: points to input values
- `[out] pDst`: points to output clipped values
- `[in] low`: lower bound
- `[in] high`: higher bound
- `[in] numSamples`: number of samples to clip

void **riscv_clip_q31** (**const** q31_t **pSrc*, q31_t **pDst*, q31_t *low*, q31_t *high*, uint32_t *numSamples*)

Elementwise fixed-point clipping.

Return none

Parameters

- `[in] pSrc`: points to input values
- `[out] pDst`: points to output clipped values
- `[in] low`: lower bound
- `[in] high`: higher bound
- `[in] numSamples`: number of samples to clip

void **riscv_clip_q7** (**const** q7_t **pSrc*, q7_t **pDst*, q7_t *low*, q7_t *high*, uint32_t *numSamples*)

Elementwise fixed-point clipping.

Return none

Parameters

- `[in] pSrc`: points to input values

- `[out] pDst`: points to output clipped values
- `[in] low`: lower bound
- `[in] high`: higher bound
- `[in] numSamples`: number of samples to clip

Vector Dot Product

void **riscv_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*, float16_t **result*)

void **riscv_dot_prod_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*, float32_t **result*)

void **riscv_dot_prod_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, uint32_t *blockSize*, q63_t **result*)

void **riscv_dot_prod_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, uint32_t *blockSize*, q63_t **result*)

void **riscv_dot_prod_q7** (**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, uint32_t *blockSize*, q31_t **result*)

group **BasicDotProd**

Computes the dot product of two vectors. The vectors are multiplied element-by-element and then summed.

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*, float16_t **result*)

Dot product of floating-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[in] blockSize`: number of samples in each vector.
- `[out] result`: output result returned here.

void **riscv_dot_prod_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*, float32_t **result*)

Dot product of floating-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[in] blockSize`: number of samples in each vector.
- `[out] result`: output result returned here.

void **riscv_dot_prod_q15**(**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, uint32_t *blockSize*, q63_t **result*)

Dot product of Q15 vectors.

Return none

Scaling and Overflow Behavior The intermediate multiplications are in 1.15 x 1.15 = 2.30 format and these results are added to a 64-bit accumulator in 34.30 format. Nonsaturating additions are used and given that there are 33 guard bits in the accumulator there is no risk of overflow. The return result is in 34.30 format.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] blockSize`: number of samples in each vector
- `[out] result`: output result returned here

void **riscv_dot_prod_q31**(**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, uint32_t *blockSize*, q63_t **result*)

Dot product of Q31 vectors.

Return none

Scaling and Overflow Behavior The intermediate multiplications are in 1.31 x 1.31 = 2.62 format and these are truncated to 2.48 format by discarding the lower 14 bits. The 2.48 result is then added without saturation to a 64-bit accumulator in 16.48 format. There are 15 guard bits in the accumulator and there is no risk of overflow as long as the length of the vectors is less than 2^16 elements. The return result is in 16.48 format.

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[in] blockSize`: number of samples in each vector.
- `[out] result`: output result returned here.

void **riscv_dot_prod_q7**(**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, uint32_t *blockSize*, q31_t **result*)

Dot product of Q7 vectors.

Return none

Scaling and Overflow Behavior The intermediate multiplications are in 1.7 x 1.7 = 2.14 format and these results are added to an accumulator in 18.14 format. Nonsaturating additions are used and there is no danger of wrap around as long as the vectors are less than 2^18 elements long. The return result is in 18.14 format.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] blockSize`: number of samples in each vector
- `[out] result`: output result returned here

Vector Multiplication

void **riscv_mult_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_mult_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_mult_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_mult_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_mult_q7** (**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicMult**

Element-by-element multiplication of two vectors.

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_mult_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

Floating-point vector multiplication.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_mult_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

Floating-point vector multiplication.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_mult_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

Q15 vector multiplication.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcA`: points to first input vector

- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_mult_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)
Q31 vector multiplication.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector.
- `[in] pSrcB`: points to the second input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_mult_q7** (**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)
Q7 vector multiplication.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector Negate

void **riscv_negate_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_negate_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_negate_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_negate_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_negate_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicNegate**
Negates the elements of a vector.

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_negate_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)
Negates the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector.
- `[out] pDst`: points to output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_negate_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)
Negates the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector.
- `[out] pDst`: points to output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_negate_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)
Negates the elements of a Q15 vector.

Return none

Conditions for optimum performance Input and output buffers should be aligned by 32-bit

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q15 value -1 (0x8000) is saturated to the maximum allowable positive value 0x7FFF.

Parameters

- `[in] pSrc`: points to the input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_negate_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)
Negates the elements of a Q31 vector.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q31 value -1 (0x80000000) is saturated to the maximum allowable positive value 0x7FFFFFFF.

Parameters

- `[in] pSrc`: points to the input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

void **riscv_negate_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Negates the elements of a Q7 vector.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q7 value -1 (0x80) is saturated to the maximum allowable positive value 0x7F.

Parameters

- `[in] pSrc`: points to the input vector.
- `[out] pDst`: points to the output vector.
- `[in] blockSize`: number of samples in each vector.

Vector bitwise NOT

void **riscv_not_u16** (**const** uint16_t **pSrc*, uint16_t **pDst*, uint32_t *blockSize*)

void **riscv_not_u32** (**const** uint32_t **pSrc*, uint32_t **pDst*, uint32_t *blockSize*)

void **riscv_not_u8** (**const** uint8_t **pSrc*, uint8_t **pDst*, uint32_t *blockSize*)

group **Not**
Compute the logical bitwise NOT.

There are separate functions for uint32_t, uint16_t, and uint8_t data types.

Functions

void **riscv_not_u16** (**const** uint16_t **pSrc*, uint16_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise NOT of a fixed-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_not_u32** (**const** uint32_t **pSrc*, uint32_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise NOT of a fixed-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_not_u8** (**const** uint8_t **pSrc*, uint8_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise NOT of a fixed-point vector.

Return none

Parameters

- [in] pSrc: points to input vector
- [out] pDst: points to output vector
- [in] blockSize: number of samples in each vector

Vector Offset

void **riscv_offset_f16** (**const** float16_t **pSrc*, float16_t *offset*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_offset_f32** (**const** float32_t **pSrc*, float32_t *offset*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_offset_q15** (**const** q15_t **pSrc*, q15_t *offset*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_offset_q31** (**const** q31_t **pSrc*, q31_t *offset*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_offset_q7** (**const** q7_t **pSrc*, q7_t *offset*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicOffset**

Adds a constant offset to each element of a vector.

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer. There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_offset_f16** (**const** float16_t **pSrc*, float16_t *offset*, float16_t **pDst*, uint32_t *blockSize*)

Adds a constant offset to a floating-point vector.

Return none

Parameters

- [in] pSrc: points to the input vector
- [in] offset: is the offset to be added
- [out] pDst: points to the output vector
- [in] blockSize: number of samples in each vector

void **riscv_offset_f32** (**const** float32_t **pSrc*, float32_t *offset*, float32_t **pDst*, uint32_t *blockSize*)

Adds a constant offset to a floating-point vector.

Return none

Parameters

- [in] pSrc: points to the input vector
- [in] offset: is the offset to be added
- [out] pDst: points to the output vector
- [in] blockSize: number of samples in each vector

void **riscv_offset_q15** (**const** q15_t **pSrc*, q15_t *offset*, q15_t **pDst*, uint32_t *blockSize*)

Adds a constant offset to a Q15 vector.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] offset`: is the offset to be added
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_offset_q31** (**const** q31_t **pSrc*, q31_t *offset*, q31_t **pDst*, uint32_t *blockSize*)
Adds a constant offset to a Q31 vector.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] offset`: is the offset to be added
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_offset_q7** (**const** q7_t **pSrc*, q7_t *offset*, q7_t **pDst*, uint32_t *blockSize*)
Adds a constant offset to a Q7 vector.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] offset`: is the offset to be added
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector bitwise inclusive OR

void **riscv_or_u16** (**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)

void **riscv_or_u32** (**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)

void **riscv_or_u8** (**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

group **Or**
Compute the logical bitwise OR.

There are separate functions for uint32_t, uint16_t, and uint8_t data types.

Functions

void **riscv_or_u16** (**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise OR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_or_u32** (**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise OR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_or_u8** (**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise OR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

Vector Scale

void **riscv_scale_f16** (**const** float16_t **pSrc*, float16_t *scale*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_scale_f32** (**const** float32_t **pSrc*, float32_t *scale*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_scale_q15** (**const** q15_t **pSrc*, q15_t *scaleFract*, int8_t *shift*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_scale_q31** (**const** q31_t **pSrc*, q31_t *scaleFract*, int8_t *shift*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_scale_q7** (**const** q7_t **pSrc*, q7_t *scaleFract*, int8_t *shift*, q7_t **pDst*, uint32_t *blockSize*) *group* **BasicScale**

Multiply a vector by a scalar value. For floating-point data, the algorithm used is:

In the fixed-point Q7, Q15, and Q31 functions, `scale` is represented by a fractional multiplication `scaleFract` and an arithmetic shift `shift`. The shift allows the gain of the scaling operation to exceed 1.0. The algorithm used with fixed-point data is:

The overall scale factor applied to the fixed-point data is

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer.

Functions

void **riscv_scale_f16**(**const** float16_t **pSrc*, float16_t *scale*, float16_t **pDst*, uint32_t *blockSize*)

Multiplies a floating-point vector by a scalar.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] scale`: scale factor to be applied
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_scale_f32**(**const** float32_t **pSrc*, float32_t *scale*, float32_t **pDst*, uint32_t *blockSize*)

Multiplies a floating-point vector by a scalar.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] scale`: scale factor to be applied
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_scale_q15**(**const** q15_t **pSrc*, q15_t *scaleFract*, int8_t *shift*, q15_t **pDst*, uint32_t *blockSize*)

Multiplies a Q15 vector by a scalar.

Return none

Scaling and Overflow Behavior The input data `*pSrc` and `scaleFract` are in 1.15 format. These are multiplied to yield a 2.30 intermediate result and this is shifted with saturation to 1.15 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] scaleFract`: fractional portion of the scale value
- `[in] shift`: number of bits to shift the result by
- `[out] pDst`: points to the output vector

- `[in] blockSize`: number of samples in each vector

void **riscv_scale_q31**(**const** q31_t **pSrc*, q31_t *scaleFract*, int8_t *shift*, q31_t **pDst*, uint32_t *blockSize*)

Multiplies a Q31 vector by a scalar.

Return none

Scaling and Overflow Behavior The input data `*pSrc` and `scaleFract` are in 1.31 format. These are multiplied to yield a 2.62 intermediate result and this is shifted with saturation to 1.31 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] scaleFract`: fractional portion of the scale value
- `[in] shift`: number of bits to shift the result by
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_scale_q7**(**const** q7_t **pSrc*, q7_t *scaleFract*, int8_t *shift*, q7_t **pDst*, uint32_t *blockSize*)

Multiplies a Q7 vector by a scalar.

Return none

Scaling and Overflow Behavior The input data `*pSrc` and `scaleFract` are in 1.7 format. These are multiplied to yield a 2.14 intermediate result and this is shifted with saturation to 1.7 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] scaleFract`: fractional portion of the scale value
- `[in] shift`: number of bits to shift the result by
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector Shift

void **riscv_shift_q15**(**const** q15_t **pSrc*, int8_t *shiftBits*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_shift_q31**(**const** q31_t **pSrc*, int8_t *shiftBits*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_shift_q7**(**const** q7_t **pSrc*, int8_t *shiftBits*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicShift**

Shifts the elements of a fixed-point vector by a specified number of bits. There are separate functions for Q7, Q15, and Q31 data types. The underlying algorithm used is:

If `shift` is positive then the elements of the vector are shifted to the left. If `shift` is negative then the elements of the vector are shifted to the right.

The functions support in-place computation allowing the source and destination pointers to reference the same memory buffer.

Functions

void **riscv_shift_q15**(**const** q15_t **pSrc*, int8_t *shiftBits*, q15_t **pDst*, uint32_t *blockSize*)
Shifts the elements of a Q15 vector a specified number of bits.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_shift_q31**(**const** q31_t **pSrc*, int8_t *shiftBits*, q31_t **pDst*, uint32_t *blockSize*)
Shifts the elements of a Q31 vector a specified number of bits.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in the vector

void **riscv_shift_q7**(**const** q7_t **pSrc*, int8_t *shiftBits*, q7_t **pDst*, uint32_t *blockSize*)
Shifts the elements of a Q7 vector a specified number of bits.

Return none

onditions for optimum performance Input and output buffers should be aligned by 32-bit

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] are saturated.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] shiftBits`: number of bits to shift. A positive value shifts left; a negative value shifts right.
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector Subtraction

void **riscv_sub_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_sub_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_sub_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_sub_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_sub_q7** (**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)

group **BasicSub**

Element-by-element subtraction of two vectors.

There are separate functions for floating-point, Q7, Q15, and Q31 data types.

Functions

void **riscv_sub_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *blockSize*)

Floating-point vector subtraction.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_sub_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *blockSize*)

Floating-point vector subtraction.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_sub_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *blockSize*)

Q15 vector subtraction.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector

- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_sub_q31**(**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *blockSize*)
Q31 vector subtraction.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_sub_q7**(**const** q7_t **pSrcA*, **const** q7_t **pSrcB*, q7_t **pDst*, uint32_t *blockSize*)
Q7 vector subtraction.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[out] pDst`: points to the output vector
- `[in] blockSize`: number of samples in each vector

Vector bitwise exclusive OR

void **riscv_xor_u16**(**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)

void **riscv_xor_u32**(**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)

void **riscv_xor_u8**(**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

group **Xor**
Compute the logical bitwise XOR.

There are separate functions for uint32_t, uint16_t, and uint8_t data types.

Functions

void **riscv_xor_u16**(**const** uint16_t **pSrcA*, **const** uint16_t **pSrcB*, uint16_t **pDst*, uint32_t *blockSize*)
Compute the logical bitwise XOR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_xor_u32** (**const** uint32_t **pSrcA*, **const** uint32_t **pSrcB*, uint32_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise XOR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_xor_u8** (**const** uint8_t **pSrcA*, **const** uint8_t **pSrcB*, uint8_t **pDst*, uint32_t *blockSize*)

Compute the logical bitwise XOR of two fixed-point vectors.

Return none

Parameters

- `[in] pSrcA`: points to input vector A
- `[in] pSrcB`: points to input vector B
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

group **groupMath**

3.3.3 Bayesian estimators

uint32_t **riscv_gaussian_naive_bayes_predict_f16** (**const** riscv_gaussian_naive_bayes_instance_f16 **S*, **const** float16_t **in*, float16_t **pOutputProbabilities*, float16_t **pBufferB*)

uint32_t **riscv_gaussian_naive_bayes_predict_f32** (**const** riscv_gaussian_naive_bayes_instance_f32 **S*, **const** float32_t **in*, float32_t **pOutputProbabilities*, float32_t **pBufferB*)

group **groupBayes**

Implement the naive gaussian Bayes estimator. The training must be done from scikit-learn.

The parameters can be easily generated from the scikit-learn object. Some examples are given in DSP/Testing/PatternGeneration/Bayes.py

Functions

uint32_t **riscv_gaussian_naive_bayes_predict_f16**(**const** riscv_gaussian_naive_bayes_instance_f16 **S*, **const** float16_t **in*, float16_t **pOutputProbabilities*, float16_t **pBufferB*)

Naive Gaussian Bayesian Estimator.

Return The predicted class

Parameters

- `[in] *S`: points to a naive bayes instance structure
- `[in] *in`: points to the elements of the input vector.
- `[out] *pOutputProbabilities`: points to a buffer of length numberOfClasses containing estimated probabilities
- `[out] *pBufferB`: points to a temporary buffer of length numberOfClasses

uint32_t **riscv_gaussian_naive_bayes_predict_f32**(**const** riscv_gaussian_naive_bayes_instance_f32 **S*, **const** float32_t **in*, float32_t **pOutputProbabilities*, float32_t **pBufferB*)

Naive Gaussian Bayesian Estimator.

Return The predicted class

Parameters

- `[in] *S`: points to a naive bayes instance structure
- `[in] *in`: points to the elements of the input vector.
- `[out] *pOutputProbabilities`: points to a buffer of length numberOfClasses containing estimated probabilities
- `[out] *pBufferB`: points to a temporary buffer of length numberOfClasses

3.3.4 Complex Math Functions

Complex Conjugate

void **riscv_cmplx_conj_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_conj_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_conj_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_conj_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)

group **cmplx_conj**

Conjugates the elements of a complex data vector.

The `pSrc` points to the source data and `pDst` points to the destination data where the result should be written. `numSamples` specifies the number of complex samples and the data in each array is stored in an interleaved fashion (real, imag, real, imag, …). Each array has a total of `2*numSamples` values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_conj_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)
Floating-point complex conjugate.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_conj_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)
Floating-point complex conjugate.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_conj_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)
Q15 complex conjugate.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q15 value -1 (0x8000) is saturated to the maximum allowable positive value 0x7FFF.

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_conj_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)
Q31 complex conjugate.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. The Q31 value -1 (0x80000000) is saturated to the maximum allowable positive value 0x7FFFFFFF.

Parameters

- `[in] pSrc`: points to the input vector
- `[out] pDst`: points to the output vector
- `[in] numSamples`: number of samples in each vector

Complex Dot Product

void **riscv_cmplx_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *numSamples*, float16_t **realResult*, float16_t **imagResult*)

void **riscv_cmplx_dot_prod_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *numSamples*, float32_t **realResult*, float32_t **imagResult*)

void **riscv_cmplx_dot_prod_q15** (**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, uint32_t *numSamples*, q31_t **realResult*, q31_t **imagResult*)

void **riscv_cmplx_dot_prod_q31** (**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, uint32_t *numSamples*, q63_t **realResult*, q63_t **imagResult*)

group **cmplx_dot_prod**

Computes the dot product of two complex vectors. The vectors are multiplied element-by-element and then summed.

The `pSrcA` points to the first complex input vector and `pSrcB` points to the second complex input vector. `numSamples` specifies the number of complex samples and the data in each array is stored in an interleaved fashion (real, imag, real, imag, . . .). Each array has a total of `2*numSamples` values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *numSamples*, float16_t **realResult*, float16_t **imagResult*)

Floating-point complex dot product.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] numSamples`: number of samples in each vector
- `[out] realResult`: real part of the result returned here
- `[out] imagResult`: imaginary part of the result returned here

void **riscv_cmplx_dot_prod_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *numSamples*, float32_t **realResult*, float32_t **imagResult*)

Floating-point complex dot product.

Return none

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] numSamples`: number of samples in each vector
- `[out] realResult`: real part of the result returned here
- `[out] imagResult`: imaginary part of the result returned here

void **riscv_cmplx_dot_prod_q15** (**const** q15_t *pSrcA*, **const** q15_t *pSrcB*, uint32_t *numSamples*, q31_t **realResult*, q31_t **imagResult*)

Q15 complex dot product.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The intermediate 1.15 by 1.15 multiplications are performed with full precision and yield a 2.30 result. These are accumulated in a 64-bit accumulator with 34.30 precision. As a final step, the accumulators are converted to 8.24 format. The return results `realResult` and `imagResult` are in 8.24 format.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] numSamples`: number of samples in each vector
- `[out] realResult`: real part of the result returned here
- `[out] imagResult`: imaginary part of the result returned her

void **riscv_cmplx_dot_prod_q31** (**const** q31_t *pSrcA*, **const** q31_t *pSrcB*, uint32_t *numSamples*, q63_t **realResult*, q63_t **imagResult*)

Q31 complex dot product.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The intermediate 1.31 by 1.31 multiplications are performed with 64-bit precision and then shifted to 16.48 format. The internal real and imaginary accumulators are in 16.48 format and provide 15 guard bits. Additions are nonsaturating and no overflow will occur as long as `numSamples` is less than 32768. The return results `realResult` and `imagResult` are in 16.48 format. Input down scaling is not required.

Parameters

- `[in] pSrcA`: points to the first input vector
- `[in] pSrcB`: points to the second input vector
- `[in] numSamples`: number of samples in each vector
- `[out] realResult`: real part of the result returned here
- `[out] imagResult`: imaginary part of the result returned here

Complex Magnitude

void **riscv_cmplx_mag_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)

group **cmplx_mag**

Computes the magnitude of the elements of a complex data vector.

The `pSrc` points to the source data and `pDst` points to the where the result should be written. `numSamples` specifies the number of complex samples in the input array and the data is stored in an interleaved fashion (real, imag, real, imag, ...). The input array has a total of `2*numSamples` values; the output array has a total of `numSamples` values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_mag_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)
Floating-point complex magnitude.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)
Floating-point complex magnitude.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)
Q15 complex magnitude.

Return none

Scaling and Overflow Behavior The function implements 1.15 by 1.15 multiplications and finally output is converted into 2.14 format.

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)
Q31 complex magnitude.

Return none

Scaling and Overflow Behavior The function implements 1.31 by 1.31 multiplications and finally output is converted into 2.30 format. Input down scaling is not required.

Parameters

- `[in] pSrc`: points to input vector

- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

Complex Magnitude Squared

void **riscv_cmplx_mag_squared_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_squared_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_squared_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mag_squared_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)

group **cmplx_mag_squared**

Computes the magnitude squared of the elements of a complex data vector.

The `pSrc` points to the source data and `pDst` points to the where the result should be written. `numSamples` specifies the number of complex samples in the input array and the data is stored in an interleaved fashion (real, imag, real, imag, …). The input array has a total of `2*numSamples` values; the output array has a total of `numSamples` values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_mag_squared_f16**(**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *numSamples*)

Floating-point complex magnitude squared.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_squared_f32**(**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *numSamples*)

Floating-point complex magnitude squared.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_squared_q15**(**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *numSamples*)

Q15 complex magnitude squared.

Return none

Scaling and Overflow Behavior The function implements 1.15 by 1.15 multiplications and finally output is converted into 3.13 format.

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mag_squared_q31**(**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *numSamples*)
Q31 complex magnitude squared.

Return none

Scaling and Overflow Behavior The function implements 1.31 by 1.31 multiplications and finally output is converted into 3.29 format. Input down scaling is not required.

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

Complex-by-Complex Multiplication

void **riscv_cmplx_mult_cmplx_f16**(**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_cmplx_f32**(**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_cmplx_q15**(**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_cmplx_q31**(**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *numSamples*)

group **CmplxByCmplxMult**

Multiplies a complex vector by another complex vector and generates a complex result. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, . . .). The parameter `numSamples` represents the number of complex samples processed. The complex arrays have a total of `2*numSamples` real values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_mult_cmplx_f16**(**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, float16_t **pDst*, uint32_t *numSamples*)
Floating-point complex-by-complex multiplication.

Return none

Parameters

- `[in] pSrcA`: points to first input vector

- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_cmplx_f32**(**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, float32_t **pDst*, uint32_t *numSamples*)

Floating-point complex-by-complex multiplication.

Return none

Parameters

- `[in] pSrcA`: points to first input vector
- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_cmplx_q15**(**const** q15_t **pSrcA*, **const** q15_t **pSrcB*, q15_t **pDst*, uint32_t *numSamples*)

Q15 complex-by-complex multiplication.

Return none

Scaling and Overflow Behavior The function implements 1.15 by 1.15 multiplications and finally output is converted into 3.13 format.

Parameters

- `[in] pSrcA`: points to first input vector
- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_cmplx_q31**(**const** q31_t **pSrcA*, **const** q31_t **pSrcB*, q31_t **pDst*, uint32_t *numSamples*)

Q31 complex-by-complex multiplication.

Return none

Scaling and Overflow Behavior The function implements 1.31 by 1.31 multiplications and finally output is converted into 3.29 format. Input down scaling is not required.

Parameters

- `[in] pSrcA`: points to first input vector
- `[in] pSrcB`: points to second input vector
- `[out] pDst`: points to output vector
- `[in] numSamples`: number of samples in each vector

Complex-by-Real Multiplication

void **riscv_cmplx_mult_real_f16** (**const** float16_t **pSrcCmplx*, **const** float16_t **pSrcReal*, float16_t **pCmplxDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_real_f32** (**const** float32_t **pSrcCmplx*, **const** float32_t **pSrcReal*, float32_t **pCmplxDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_real_q15** (**const** q15_t **pSrcCmplx*, **const** q15_t **pSrcReal*, q15_t **pCmplxDst*, uint32_t *numSamples*)

void **riscv_cmplx_mult_real_q31** (**const** q31_t **pSrcCmplx*, **const** q31_t **pSrcReal*, q31_t **pCmplxDst*, uint32_t *numSamples*)

group **CmplxByRealMult**

Multiplies a complex vector by a real vector and generates a complex result. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, …). The parameter `numSamples` represents the number of complex samples processed. The complex arrays have a total of `2*numSamples` real values while the real array has a total of `numSamples` real values.

The underlying algorithm is used:

There are separate functions for floating-point, Q15, and Q31 data types.

Functions

void **riscv_cmplx_mult_real_f16** (**const** float16_t **pSrcCmplx*, **const** float16_t **pSrcReal*, float16_t **pCmplxDst*, uint32_t *numSamples*)

Floating-point complex-by-real multiplication.

Return none

Parameters

- `[in] pSrcCmplx`: points to complex input vector
- `[in] pSrcReal`: points to real input vector
- `[out] pCmplxDst`: points to complex output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_real_f32** (**const** float32_t **pSrcCmplx*, **const** float32_t **pSrcReal*, float32_t **pCmplxDst*, uint32_t *numSamples*)

Floating-point complex-by-real multiplication.

Return none

Parameters

- `[in] pSrcCmplx`: points to complex input vector
- `[in] pSrcReal`: points to real input vector
- `[out] pCmplxDst`: points to complex output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_real_q15** (**const** q15_t **pSrcCmplx*, **const** q15_t **pSrcReal*, q15_t **pCmplxDst*, uint32_t *numSamples*)

Q15 complex-by-real multiplication.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcCmplx`: points to complex input vector
- `[in] pSrcReal`: points to real input vector
- `[out] pCmplxDst`: points to complex output vector
- `[in] numSamples`: number of samples in each vector

void **riscv_cmplx_mult_real_q31** (**const** q31_t **pSrcCmplx*, **const** q31_t **pSrcReal*, q31_t **pCmplxDst*, uint32_t *numSamples*)

Q31 complex-by-real multiplication.

Return none

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcCmplx`: points to complex input vector
- `[in] pSrcReal`: points to real input vector
- `[out] pCmplxDst`: points to complex output vector
- `[in] numSamples`: number of samples in each vector

group **groupCmplxMath**

This set of functions operates on complex data vectors. The data in the complex arrays is stored in an interleaved fashion (real, imag, real, imag, . . .). In the API functions, the number of samples in a complex array refers to the number of complex values; the array contains twice this number of real values.

3.3.5 Controller Functions

PID Motor Control

__STATIC_FORCEINLINE float32_t riscv_pid_f32(riscv_pid_instance_f32 * S, float32_t in)

__STATIC_FORCEINLINE q31_t riscv_pid_q31(riscv_pid_instance_q31 * S, q31_t in)

__STATIC_FORCEINLINE q15_t riscv_pid_q15(riscv_pid_instance_q15 * S, q15_t in)

void **riscv_pid_init_f32** (riscv_pid_instance_f32 **S*, int32_t *resetStateFlag*)

void **riscv_pid_init_q15** (riscv_pid_instance_q15 **S*, int32_t *resetStateFlag*)

void **riscv_pid_init_q31** (riscv_pid_instance_q31 **S*, int32_t *resetStateFlag*)

void **riscv_pid_reset_f32** (riscv_pid_instance_f32 **S*)

void **riscv_pid_reset_q15** (riscv_pid_instance_q15 **S*)

void **riscv_pid_reset_q31** (riscv_pid_instance_q31 **S*)

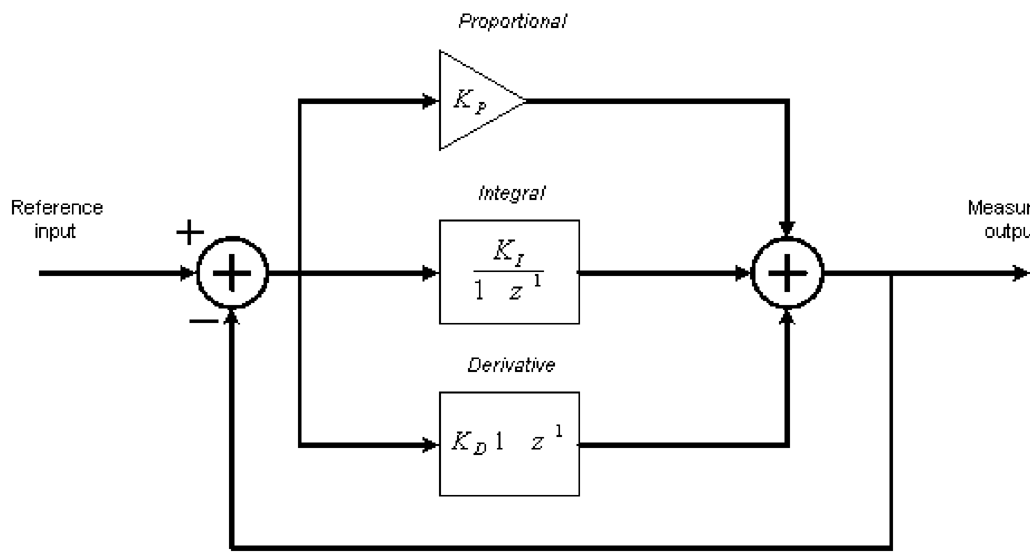
group **PID**
end of SinCos group

A Proportional Integral Derivative (PID) controller is a generic feedback control loop mechanism widely used in industrial control systems. A PID controller is the most commonly used type of feedback controller.

This set of functions implements (PID) controllers for Q15, Q31, and floating-point data types. The functions operate on a single sample of data and each call to the function returns a single processed value. `S` points to an instance of the PID control data structure. `in` is the input sample value. The functions return the output value.

Algorithm:

where `Kp` is proportional constant, `Ki` is Integral constant and `Kd` is Derivative constant



The PID controller calculates an "error" value as the difference between the measured output and the reference input. The controller attempts to minimize the error by adjusting the process control inputs. The proportional value determines the reaction to the current error, the integral value determines the reaction based on the sum of recent errors, and the derivative value determines the reaction based on the rate at which the error has been changing.

Instance Structure The Gains A0, A1, A2 and state variables for a PID controller are stored together in an instance data structure. A separate instance structure must be defined for each PID Controller. There are separate instance structure declarations for each of the 3 supported data types.

Reset Functions There is also an associated reset function for each data type which clears the state array.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Initializes the Gains A0, A1, A2 from Kp,Ki, Kd gains.
- Zeros out the values in the state buffer.

Instance structure cannot be placed into a const data section and it is recommended to use the initialization function.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the PID Controller functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

__STATIC_FORCEINLINE float32_t riscv_pid_f32(riscv_pid_instance_f32 * S, float32_t in

Process function for the floating-point PID Control.

Return processed output sample.

Parameters

- `[inout] S`: is an instance of the floating-point PID Control structure
- `[in] in`: input sample to process

__STATIC_FORCEINLINE q31_t riscv_pid_q31(riscv_pid_instance_q31 * S, q31_t in)

Process function for the Q31 PID Control.

Return processed output sample.

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2 bits as there are four additions. After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

Parameters

- `[inout] S`: points to an instance of the Q31 PID Control structure
- `[in] in`: input sample to process

__STATIC_FORCEINLINE q15_t riscv_pid_q15(riscv_pid_instance_q15 * S, q15_t in)

Process function for the Q15 PID Control.

Return processed output sample.

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both Gains and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Parameters

- `[inout] S`: points to an instance of the Q15 PID Control structure
- `[in] in`: input sample to process

void **riscv_pid_init_f32**(riscv_pid_instance_f32 **S*, int32_t *resetStateFlag*)

Initialization function for the floating-point PID Control.

Return none

Details The `resetStateFlag` specifies whether to set state to zero or not. The function computes the structure fields: `A0`, `A1` `A2` using the proportional gain(`Kp`), integral gain(`Ki`) and derivative gain(`Kd`) also sets the state variables to all zeros.

Parameters

- `[inout] S`: points to an instance of the PID structure

- `[in] resetStateFlag`:
  - value = 0: no change in state
  - value = 1: reset state

void **riscv_pid_init_q15**(riscv_pid_instance_q15 **S*, int32_t *resetStateFlag*)
Initialization function for the Q15 PID Control.

Return none

Details The `resetStateFlag` specifies whether to set state to zero or not. The function computes the structure fields: `A0`, `A1` `A2` using the proportional gain(`Kp`), integral gain(`Ki`) and derivative gain(`Kd`) also sets the state variables to all zeros.

Parameters

- `[inout] S`: points to an instance of the Q15 PID structure
- `[in] resetStateFlag`:
  - value = 0: no change in state
  - value = 1: reset state

void **riscv_pid_init_q31**(riscv_pid_instance_q31 **S*, int32_t *resetStateFlag*)
Initialization function for the Q31 PID Control.

Return none

Details The `resetStateFlag` specifies whether to set state to zero or not. The function computes the structure fields: `A0`, `A1` `A2` using the proportional gain(`Kp`), integral gain(`Ki`) and derivative gain(`Kd`) also sets the state variables to all zeros.

Parameters

- `[inout] S`: points to an instance of the Q31 PID structure
- `[in] resetStateFlag`:
  - value = 0: no change in state
  - value = 1: reset state

void **riscv_pid_reset_f32**(riscv_pid_instance_f32 **S*)
Reset function for the floating-point PID Control.

Return none

Details The function resets the state buffer to zeros.

Parameters

- `[inout] S`: points to an instance of the floating-point PID structure

void **riscv_pid_reset_q15**(riscv_pid_instance_q15 **S*)
Reset function for the Q15 PID Control.

Return none

Details The function resets the state buffer to zeros.

Parameters

- `[inout] S`: points to an instance of the Q15 PID structure

void **riscv_pid_reset_q31** (riscv_pid_instance_q31 **S*)

Reset function for the Q31 PID Control.

Return none

Details The function resets the state buffer to zeros.

Parameters

- `[inout] S`: points to an instance of the Q31 PID structure

Vector Park Transform

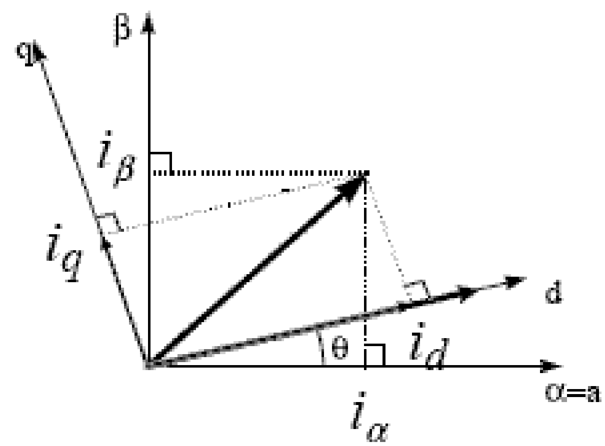
`__STATIC_FORCEINLINE void riscv_park_f32(float32_t Ialpha, float32_t Ibeta, float32_t * pI`

`__STATIC_FORCEINLINE void riscv_park_q31(q31_t Ialpha, q31_t Ibeta, q31_t * pId, q31_t * p`

group **park**

end of PID group

Forward Park transform converts the input two-coordinate vector to flux and torque components. The Park transform can be used to realize the transformation of the `Ialpha` and the `Ibeta` currents from the stationary to the moving reference frame and control the spatial relationship between the stator vector current and rotor flux vector. If we consider the d axis aligned with the rotor flux, the diagram below shows the current vector and the relationship from the two reference frames: The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point



data types.

Algorithm

where `Ialpha` and `Ibeta` are the stator vector components, `pId` and `pIq` are rotor vector components and `cosVal` and `sinVal` are the cosine and sine values of theta (rotor flux position).

$$
\begin{aligned}
pId &= Ialpha * cosVal + Ibeta * sinVal \\
pIq &= - Ialpha\ sinVal + Ibeta * cosVal
\end{aligned}
$$

Fixed-Point Behavior Care must be taken when using the Q31 version of the Park transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

__STATIC_FORCEINLINE void riscv_park_f32(float32_t Ialpha, float32_t Ibeta, float32_t

Floating-point Park transform.

The function implements the forward Park transform.

Return none

Parameters

- `[in] Ialpha`: input two-phase vector coordinate alpha
- `[in] Ibeta`: input two-phase vector coordinate beta
- `[out] pId`: points to output rotor reference frame d
- `[out] pIq`: points to output rotor reference frame q
- `[in] sinVal`: sine value of rotation angle theta
- `[in] cosVal`: cosine value of rotation angle theta

__STATIC_FORCEINLINE void riscv_park_q31(q31_t Ialpha, q31_t Ibeta, q31_t * pId, q31_t

Park transform for Q31 version.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition and subtraction, hence there is no risk of overflow.

Parameters

- `[in] Ialpha`: input two-phase vector coordinate alpha
- `[in] Ibeta`: input two-phase vector coordinate beta
- `[out] pId`: points to output rotor reference frame d
- `[out] pIq`: points to output rotor reference frame q
- `[in] sinVal`: sine value of rotation angle theta
- `[in] cosVal`: cosine value of rotation angle theta

Vector Inverse Park transform

__STATIC_FORCEINLINE void riscv_inv_park_f32(float32_t Id, float32_t Iq, float32_t * pIalp

__STATIC_FORCEINLINE void riscv_inv_park_q31(q31_t Id, q31_t Iq, q31_t * pIalpha, q31_t * p

group **inv_park**

end of park group

Inverse Park transform converts the input flux and torque components to two-coordinate vector.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

where `pIalpha` and `pIbeta` are the stator vector components, `Id` and `Iq` are rotor vector components and `cosVal` and `sinVal` are the cosine and sine values of theta (rotor flux position).

$$
\begin{aligned}
pIalpha &= Id * cosVal - Iq * sinVal \\
pIbeta &= Id * sinVal + Iq * cosVal
\end{aligned}
$$

Fixed-Point Behavior Care must be taken when using the Q31 version of the Park transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

`__STATIC_FORCEINLINE void riscv_inv_park_f32(float32_t Id, float32_t Iq, float32_t * p`

Floating-point Inverse Park transform.

Return none

Parameters

- `[in] Id`: input coordinate of rotor reference frame d
- `[in] Iq`: input coordinate of rotor reference frame q
- `[out] pIalpha`: points to output two-phase orthogonal vector axis alpha
- `[out] pIbeta`: points to output two-phase orthogonal vector axis beta
- `[in] sinVal`: sine value of rotation angle theta
- `[in] cosVal`: cosine value of rotation angle theta

`__STATIC_FORCEINLINE void riscv_inv_park_q31(q31_t Id, q31_t Iq, q31_t * pIalpha, q31_`

Inverse Park transform for Q31 version.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition, hence there is no risk of overflow.

Parameters

- `[in] Id`: input coordinate of rotor reference frame d
- `[in] Iq`: input coordinate of rotor reference frame q
- `[out] pIalpha`: points to output two-phase orthogonal vector axis alpha
- `[out] pIbeta`: points to output two-phase orthogonal vector axis beta
- `[in] sinVal`: sine value of rotation angle theta
- `[in] cosVal`: cosine value of rotation angle theta

Vector Clarke Transform

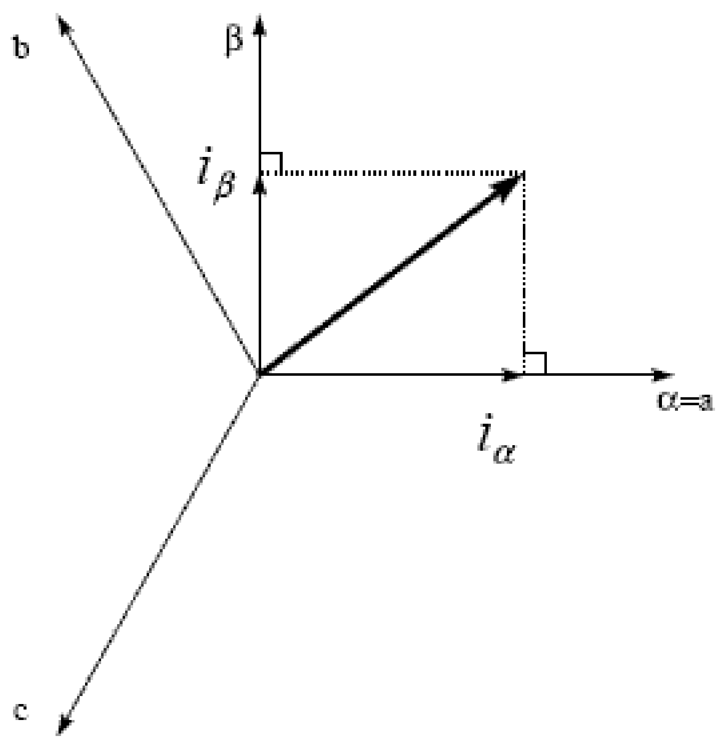
`__STATIC_FORCEINLINE void riscv_clarke_f32(float32_t Ia, float32_t Ib, float32_t * pIalpha,`

`__STATIC_FORCEINLINE void riscv_clarke_q31(q31_t Ia, q31_t Ib, q31_t * pIalpha, q31_t * pIb`

group **`clarke`**

end of Inverse park group

Forward Clarke transform converts the instantaneous stator phases into a two-coordinate time invariant vector. Generally the Clarke transform uses three-phase currents `Ia, Ib and Ic` to calculate currents in the two-phase orthogonal stator axis `Ialpha` and `Ibeta`. When `Ialpha` is superposed with `Ia` as shown in the figure below and `Ia + Ib + Ic = 0`, in this condition `Ialpha` and `Ibeta` can be calculated using only `Ia` and



`Ib`.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

where `Ia` and `Ib` are the instantaneous stator phases and `pIalpha` and `pIbeta` are the two coordinates

$$\text{pIalpha} = \text{Ia}$$

$$\text{pIbeta} = (1/\sqrt{3})\,\text{Ia} + (2/\sqrt{3})\,\text{Ib}$$

of time invariant vector.

Fixed-Point Behavior Care must be taken when using the Q31 version of the Clarke transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

`__STATIC_FORCEINLINE void riscv_clarke_f32(float32_t Ia, float32_t Ib, float32_t * pIa`

Floating-point Clarke transform.

Return none

Parameters

- `[in] Ia`: input three-phase coordinate `a`
- `[in] Ib`: input three-phase coordinate `b`
- `[out] pIalpha`: points to output two-phase orthogonal vector axis alpha
- `[out] pIbeta`: points to output two-phase orthogonal vector axis beta

`__STATIC_FORCEINLINE void riscv_clarke_q31(q31_t Ia, q31_t Ib, q31_t * pIalpha, q31_t`

Clarke transform for Q31 version.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the addition, hence there is no risk of overflow.

Parameters

- `[in] Ia`: input three-phase coordinate `a`
- `[in] Ib`: input three-phase coordinate `b`
- `[out] pIalpha`: points to output two-phase orthogonal vector axis alpha
- `[out] pIbeta`: points to output two-phase orthogonal vector axis beta

Vector Inverse Clarke Transform

`__STATIC_FORCEINLINE void riscv_inv_clarke_f32(float32_t Ialpha, float32_t Ibeta, float32_t`

`__STATIC_FORCEINLINE void riscv_inv_clarke_q31(q31_t Ialpha, q31_t Ibeta, q31_t * pIa, q31_`

group **`inv_clarke`**

end of clarke group

Inverse Clarke transform converts the two-coordinate time invariant vector into instantaneous stator phases.

The function operates on a single sample of data and each call to the function returns the processed output. The library provides separate functions for Q31 and floating-point data types.

Algorithm

where `pIa` and `pIb` are the instantaneous stator phases and `Ialpha` and `Ibeta` are the two coordinates

$$\mathrm{pIa} = \mathrm{Ialpha}$$

$$\mathrm{pIb} = (-1/2)\,\mathrm{Ialpha} + (\sqrt{3}/2)\,\mathrm{Ibeta}$$

of time invariant vector.

Fixed-Point Behavior Care must be taken when using the Q31 version of the Clarke transform. In particular, the overflow and saturation behavior of the accumulator used must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

`__STATIC_FORCEINLINE void riscv_inv_clarke_f32(float32_t Ialpha, float32_t Ibeta, floa`

Floating-point Inverse Clarke transform.

Return none

Parameters

- `[in] Ialpha`: input two-phase orthogonal vector axis alpha
- `[in] Ibeta`: input two-phase orthogonal vector axis beta
- `[out] pIa`: points to output three-phase coordinate `a`
- `[out] pIb`: points to output three-phase coordinate `b`

`__STATIC_FORCEINLINE void riscv_inv_clarke_q31(q31_t Ialpha, q31_t Ibeta, q31_t * pIa,`

Inverse Clarke transform for Q31 version.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The accumulator maintains 1.31 format by truncating lower 31 bits of the intermediate multiplication in 2.62 format. There is saturation on the subtraction, hence there is no risk of overflow.

Parameters

- `[in] Ialpha`: input two-phase orthogonal vector axis alpha
- `[in] Ibeta`: input two-phase orthogonal vector axis beta
- `[out] pIa`: points to output three-phase coordinate `a`
- `[out] pIb`: points to output three-phase coordinate `b`

Sine Cosine

void **riscv_sin_cos_f32** (float32_t *theta*, float32_t **pSinVal*, float32_t **pCosVal*)

void **riscv_sin_cos_q31** (q31_t *theta*, q31_t **pSinVal*, q31_t **pCosVal*)

group **SinCos**

Computes the trigonometric sine and cosine values using a combination of table lookup and linear interpolation. There are separate functions for Q31 and floating-point data types. The input to the floating-point version is in degrees while the fixed-point Q31 have a scaled input with the range [-1 0.9999] mapping to [-180 +180] degrees.

The floating point function also allows values that are out of the usual range. When this happens, the function will take extra time to adjust the input value to the range of [-180 180].

The result is accurate to 5 digits after the decimal point.

The implementation is based on table lookup using 360 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index.
2. Compute the fractional portion (fract) of the input.
3. Fetch the value corresponding to `index` from sine table to `y0` and also value from `index+1` to `y1`.
4. Sine value is computed as `*psinVal = y0 + (fract * (y1 - y0))`.

5. Fetch the value corresponding to `index` from cosine table to `y0` and also value from `index+1` to `y1`.
6. Cosine value is computed as `*pcosVal = y0 + (fract * (y1 - y0))`.

Functions

void **riscv_sin_cos_f32** (float32_t *theta*, float32_t **pSinVal*, float32_t **pCosVal*)
Floating-point sin_cos function.

Return none

Parameters

- `[in] theta`: input value in degrees
- `[out] pSinVal`: points to the processed sine output.
- `[out] pCosVal`: points to the processed cos output.
- `[in] theta`: input value in degrees
- `[out] pSinVal`: points to processed sine output
- `[out] pCosVal`: points to processed cosine output

void **riscv_sin_cos_q31** (q31_t *theta*, q31_t **pSinVal*, q31_t **pCosVal*)
Q31 sin_cos function.

The Q31 input value is in the range [-1 0.999999] and is mapped to a degree value in the range [-180 179].

Return none

Parameters

- `[in] theta`: scaled input value in degrees
- `[out] pSinVal`: points to the processed sine output.
- `[out] pCosVal`: points to the processed cosine output.
- `[in] theta`: scaled input value in degrees
- `[out] pSinVal`: points to processed sine output
- `[out] pCosVal`: points to processed cosine output

group **groupController**

3.3.6 Distance functions

Float Distances

Bray-Curtis distance

float16_t **riscv_braycurtis_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_braycurtis_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **braycurtis**
Bray-Curtis distance between two vectors

Functions

float16_t **riscv_braycurtis_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Bray-Curtis distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_braycurtis_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Bray-Curtis distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Canberra distance

float16_t **riscv_canberra_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_canberra_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **Canberra**

Canberra distance

Functions

float16_t **riscv_canberra_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Canberra distance between two vectors.

This function may divide by zero when samples pA[i] and pB[i] are both zero. The result of the computation will be correct. So the division per zero may be ignored.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_canberra_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Canberra distance between two vectors.

This function may divide by zero when samples pA[i] and pB[i] are both zero. The result of the computation will be correct. So the division per zero may be ignored.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Chebyshev distance

float16_t **riscv_chebyshev_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_chebyshev_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **Chebyshev**

Chebyshev distance

Functions

float16_t **riscv_chebyshev_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Chebyshev distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_chebyshev_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Chebyshev distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Cityblock (Manhattan) distance

float16_t **riscv_cityblock_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_cityblock_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **Manhattan**

Cityblock (Manhattan) distance

Functions

float16_t **riscv_cityblock_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Cityblock (Manhattan) distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_cityblock_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Cityblock (Manhattan) distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Correlation distance

float16_t **riscv_correlation_distance_f16** (float16_t **pA*, float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_correlation_distance_f32** (float32_t **pA*, float32_t **pB*, uint32_t *blockSize*)

group **Correlation**

Correlation distance

Functions

float16_t **riscv_correlation_distance_f16** (float16_t **pA*, float16_t **pB*, uint32_t *blockSize*)

Correlation distance between two vectors.

The input vectors are modified in place !

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_correlation_distance_f32** (float32_t **pA*, float32_t **pB*, uint32_t *blockSize*)

Correlation distance between two vectors.

The input vectors are modified in place !

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Cosine distance

float16_t **riscv_cosine_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_cosine_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **CosineDist**

Cosine distance

Functions

float16_t **riscv_cosine_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Cosine distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_cosine_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Cosine distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector

- `[in] blockSize`: vector length

Euclidean distance

float16_t **riscv_euclidean_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_euclidean_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **Euclidean**
Euclidean distance

Functions

float16_t **riscv_euclidean_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)
Euclidean distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

float32_t **riscv_euclidean_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)
Euclidean distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Jensen-Shannon distance

float16_t **riscv_jensenshannon_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

float32_t **riscv_jensenshannon_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

group **JensenShannon**
Jensen-Shannon distance

Functions

`__STATIC_INLINE float16_t rel_entr(float16_t x, float16_t y)`

float16_t **`riscv_jensenshannon_distance_f16`**(**const** float16_t **pA*, **const** float16_t **pB*, uint32_t *blockSize*)

Jensen-Shannon distance between two vectors.

This function is assuming that elements of second vector are > 0 and 0 only when the corresponding element of first vector is 0. Otherwise the result of the computation does not make sense and for speed reasons, the cases returning NaN or Infinity are not managed.

When the function is computing x log (x / y) with x == 0 and y == 0, it will compute the right result (0) but a division by zero will occur and should be ignored in client code.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

`__STATIC_INLINE float32_t rel_entr(float32_t x, float32_t y)`

float32_t **`riscv_jensenshannon_distance_f32`**(**const** float32_t **pA*, **const** float32_t **pB*, uint32_t *blockSize*)

Jensen-Shannon distance between two vectors.

This function is assuming that elements of second vector are > 0 and 0 only when the corresponding element of first vector is 0. Otherwise the result of the computation does not make sense and for speed reasons, the cases returning NaN or Infinity are not managed.

When the function is computing x log (x / y) with x == 0 and y == 0, it will compute the right result (0) but a division by zero will occur and should be ignored in client code.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] blockSize`: vector length

Minkowski distance

float16_t **`riscv_minkowski_distance_f16`**(**const** float16_t **pA*, **const** float16_t **pB*, int32_t *order*, uint32_t *blockSize*)

float32_t **`riscv_minkowski_distance_f32`**(**const** float32_t **pA*, **const** float32_t **pB*, int32_t *order*, uint32_t *blockSize*)

group **Minkowski**

Minkowski distance

Functions

float16_t **riscv_minkowski_distance_f16** (**const** float16_t **pA*, **const** float16_t **pB*, int32_t *order*, uint32_t *blockSize*)

Minkowski distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] order`: Distance order
- `[in] blockSize`: Number of samples

float32_t **riscv_minkowski_distance_f32** (**const** float32_t **pA*, **const** float32_t **pB*, int32_t *order*, uint32_t *blockSize*)

Minkowski distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector
- `[in] pB`: Second vector
- `[in] order`: Distance order
- `[in] blockSize`: Number of samples

group **FloatDist**

Distances between two vectors of float values.

Boolean Distances

float32_t **riscv_dice_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_hamming_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_jaccard_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_kulsinski_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_rogerstanimoto_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_russellrao_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_sokalmichener_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_sokalsneath_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

float32_t **riscv_yule_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*) *group* **BoolDist**

Distances between two vectors of boolean values.

Booleans are packed in 32 bit words. numberOfBooleans argument is the number of booleans and not the number of words.

Bits are packed in big-endian mode (because of behavior of numpy packbits in in version < 1.17)

Unnamed Group

float32_t **riscv_dice_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Dice distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

Functions

float32_t **riscv_hamming_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Hamming distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_jaccard_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Jaccard distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_kulsinski_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Kulsinski distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_rogerstanimoto_distance**(**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Rogers Tanimoto distance between two vectors.

Roger Stanimoto distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_russellrao_distance**(**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Russell-Rao distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_sokalmichener_distance**(**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Sokal-Michener distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_sokalsneath_distance**(**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Sokal-Sneath distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

float32_t **riscv_yule_distance** (**const** uint32_t **pA*, **const** uint32_t **pB*, uint32_t *numberOfBools*)

Yule distance between two vectors.

Return distance

Parameters

- `[in] pA`: First vector of packed booleans
- `[in] pB`: Second vector of packed booleans
- `[in] numberOfBools`: Number of booleans

group **groupDistance**

Distance functions for use with clustering algorithms. There are distance functions for float vectors and boolean vectors.

3.3.7 Fast Math Functions

Cosine

float32_t **riscv_cos_f32** (float32_t *x*)

q31_t **riscv_cos_q31** (q31_t *x*)

q15_t **riscv_cos_q15** (q15_t *x*)

group **cos**

Computes the trigonometric cosine function using a combination of table lookup and linear interpolation. There are separate functions for Q15, Q31, and floating-point data types. The input to the floating-point version is in radians while the fixed-point Q15 and Q31 have a scaled input with the range [0 +0.9999] mapping to [0 2*pi). The fixed-point range is chosen so that a value of 2*pi wraps around to 0.

The implementation is based on table lookup using 512 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index
2. Compute the fractional portion (fract) of the table index.
3. The final result equals `(1.0f-fract)*a + fract*b;`

where

end of sin group

Functions

float32_t **riscv_cos_f32** (float32_t *x*)

Fast approximation to the trigonometric cosine function for floating-point data.

Return cos(x).

Return cos(x)

Parameters

- `[in] x`: input value in radians.

Parameters

- `[in] x`: input value in radians

q31_t **riscv_cos_q31** (q31_t *x*)

Fast approximation to the trigonometric cosine function for Q31 data.

The Q31 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*PI).

Return cos(x).

Return cos(x)

Parameters

- `[in] x`: Scaled input value in radians.

Parameters

- `[in] x`: Scaled input value in radians

q15_t **riscv_cos_q15** (q15_t *x*)

Fast approximation to the trigonometric cosine function for Q15 data.

The Q15 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*PI).

Return cos(x).

Return cos(x)

Parameters

- `[in] x`: Scaled input value in radians.

Parameters

- `[in] x`: Scaled input value in radians

Fixed point division

riscv_status **riscv_divide_q15** (q15_t *numerator*, q15_t *denominator*, q15_t **quotient*, int16_t **shift*)

group **divide**

Functions

riscv_status **riscv_divide_q15** (q15_t *numerator*, q15_t *denominator*, q15_t **quotient*, int16_t **shift*)

Fixed point division.

When dividing by 0, an error RISCV_MATH_NANINF is returned. And the quotient is forced to the saturated negative or positive value.

Return error status

Parameters

- `[in] numerator`: Numerator
- `[in] denominator`: Denominator
- `[out] quotient`: Quotient value normalized between -1.0 and 1.0
- `[out] shift`: Shift left value to get the unnormalized quotient

Sine

float32_t **riscv_sin_f32** (float32_t *x*)

q31_t **riscv_sin_q31** (q31_t *x*)

q15_t **riscv_sin_q15** (q15_t *x*)

group **sin**

Computes the trigonometric sine function using a combination of table lookup and linear interpolation. There are separate functions for Q15, Q31, and floating-point data types. The input to the floating-point version is in radians while the fixed-point Q15 and Q31 have a scaled input with the range [0 +0.9999] mapping to [0 2*pi). The fixed-point range is chosen so that a value of 2*pi wraps around to 0.

The implementation is based on table lookup using 512 values together with linear interpolation. The steps used are:

1. Calculation of the nearest integer table index
2. Compute the fractional portion (fract) of the table index.
3. The final result equals `(1.0f-fract)*a + fract*b;`

where

Functions

float32_t **riscv_sin_f32** (float32_t *x*)

Fast approximation to the trigonometric sine function for floating-point data.

Return sin(x).

Return sin(x)

Parameters

- `[in] x`: input value in radians.

Parameters

- `[in] x`: input value in radians.

q31_t **riscv_sin_q31** (q31_t *x*)

Fast approximation to the trigonometric sine function for Q31 data.

The Q31 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*PI).

Return sin(x).

Return sin(x)

Parameters

- `[in] x`: Scaled input value in radians.

Parameters

- `[in] x`: Scaled input value in radians

q15_t **riscv_sin_q15** (q15_t *x*)

Fast approximation to the trigonometric sine function for Q15 data.

The Q15 input value is in the range [0 +0.9999] and is mapped to a radian value in the range [0 2*PI).

Return sin(x).

Return sin(x)

Parameters

- `[in] x`: Scaled input value in radians.

Parameters

- `[in] x`: Scaled input value in radians

Square Root

__STATIC_FORCEINLINE riscv_status riscv_sqrt_f32(float32_t in, float32_t * pOut)

riscv_status **riscv_sqrt_q31** (q31_t *in*, q31_t **pOut*)

riscv_status **riscv_sqrt_q15** (q15_t *in*, q15_t **pOut*)

void **riscv_vsqrt_f32** (float32_t **pIn*, float32_t **pOut*, uint16_t *len*)

void **riscv_vsqrt_q31** (q31_t **pIn*, q31_t **pOut*, uint16_t *len*)

void **riscv_vsqrt_q15** (q15_t **pIn*, q15_t **pOut*, uint16_t *len*)

__STATIC_FORCEINLINE riscv_status riscv_sqrt_f16(float16_t in, float16_t * pOut)

group **SQRT**

Computes the square root of a number. There are separate functions for Q15, Q31, and floating-point data types. The square root function is computed using the Newton-Raphson algorithm. This is an iterative algorithm of the form: where `x1` is the current estimate, `x0` is the previous estimate, and `f'(x0)` is the derivative of `f()` evaluated at `x0`. For the square root function, the algorithm reduces to:

Functions

__STATIC_FORCEINLINE riscv_status riscv_sqrt_f32(float32_t in, float32_t * pOut)

Floating-point square root function.

Return execution status

- RISCV_MATH_SUCCESS : input value is positive
- RISCV_MATH_ARGUMENT_ERROR : input value is negative; *pOut is set to 0

Parameters

- `[in] in`: input value
- `[out] pOut`: square root of input value

riscv_status **riscv_sqrt_q31** (q31_t *in*, q31_t **pOut*)

Q31 square root function.

Return execution status

- RISCV_MATH_SUCCESS : input value is positive
- RISCV_MATH_ARGUMENT_ERROR : input value is negative; *pOut is set to 0

Parameters

- `[in] in`: input value. The range of the input value is [0 +1) or 0x00000000 to 0x7FFFFFFF
- `[out] pOut`: points to square root of input value

riscv_status **riscv_sqrt_q15** (q15_t *in*, q15_t **pOut*)

Q15 square root function.

Return execution status

- RISCV_MATH_SUCCESS : input value is positive
- RISCV_MATH_ARGUMENT_ERROR : input value is negative; *pOut is set to 0

Parameters

- `[in] in`: input value. The range of the input value is [0 +1) or 0x0000 to 0x7FFF
- `[out] pOut`: points to square root of input value

void **riscv_vsqrt_f32** (float32_t **pIn*, float32_t **pOut*, uint16_t *len*)

Vector Floating-point square root function.

Return The function returns RISCV_MATH_SUCCESS if input value is positive value or RISCV_MATH_ARGUMENT_ERROR if `in` is negative value and returns zero output for negative values.

Parameters

- `[in] pIn`: input vector.
- `[out] pOut`: vector of square roots of input elements.
- `[in] len`: length of input vector.

void **riscv_vsqrt_q31** (q31_t **pIn*, q31_t **pOut*, uint16_t *len*)

void **riscv_vsqrt_q15** (q15_t **pIn*, q15_t **pOut*, uint16_t *len*)

__STATIC_FORCEINLINE riscv_status riscv_sqrt_f16(float16_t in, float16_t * pOut)

Floating-point square root function.

Return execution status

- RISCV_MATH_SUCCESS : input value is positive
- RISCV_MATH_ARGUMENT_ERROR : input value is negative; *pOut is set to 0

Parameters

- `[in] in`: input value
- `[out] pOut`: square root of input value

group **groupFastMath**

This set of functions provides a fast approximation to sine, cosine, and square root. As compared to most of the other functions in the NMSIS math library, the fast math functions operate on individual values and not arrays. There are separate functions for Q15, Q31, and floating-point data.

3.3.8 Filtering Functions

High Precision Q31 Biquad Cascade Filter

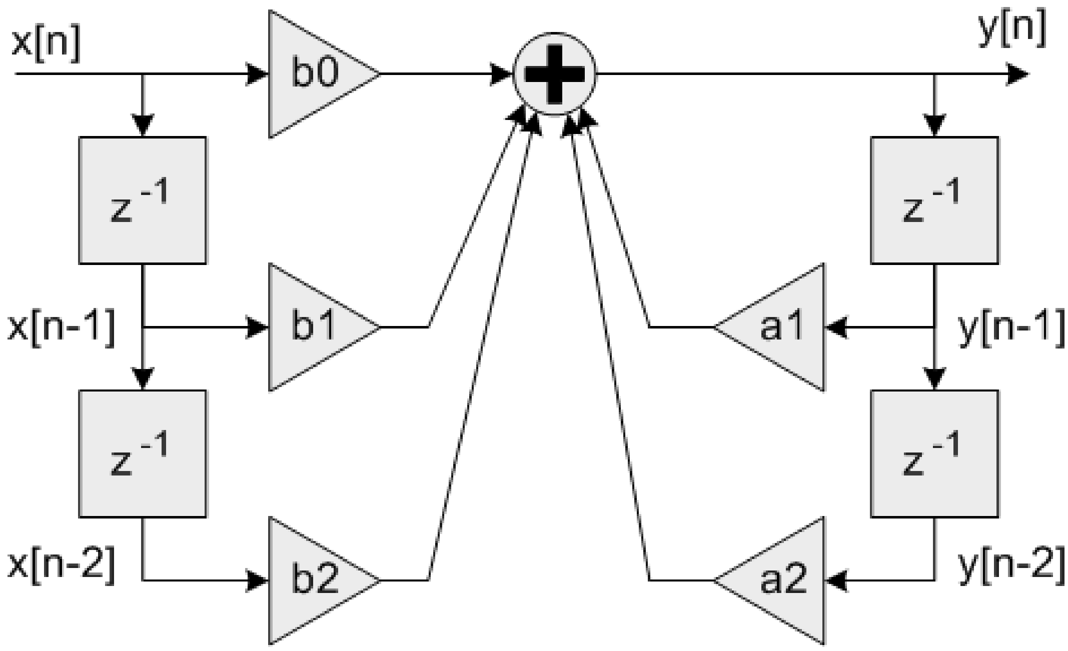
void **riscv_biquad_cas_df1_32x64_init_q31** (riscv_biquad_cas_df1_32x64_ins_q31 **S*, uint8_t *numStages*, **const** q31_t **pCoeffs*, q63_t **pState*, uint8_t *postShift*) void **riscv_biquad_cas_df1_32x64_q31** (**const** riscv_biquad_cas_df1_32x64_ins_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **BiquadCascadeDF1_32x64**

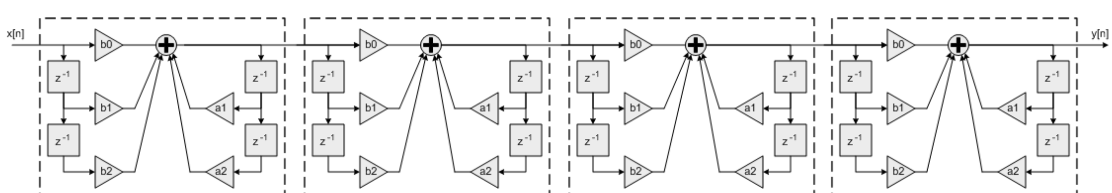
This function implements a high precision Biquad cascade filter which operates on Q31 data values. The filter coefficients are in 1.31 format and the state variables are in 1.63 format. The double precision state variables reduce quantization noise in the filter and provide a cleaner output. These filters are particularly useful when implementing filters in which the singularities are close to the unit circle. This is common for low pass or high pass filters with very low cutoff frequencies.

The function operates on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` and `pDst` points to input and output arrays containing `blockSize` Q31 values.

Algorithm Each Biquad stage implements a second order filter using the difference equation: A Direct Form I algorithm is used with 5 coefficients and 4 state variables per stage. Coefficients `b0, b1 and b2` multiply the input signal `x[n]` and are referred to as the feedforward coefficients. Coefficients `a1` and `a2` multiply the output signal `y[n]` and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools use the difference equation In this case the feedback coefficients `a1` and `a2` must be negated when used with the NMSIS DSP Library.



Higher order filters are realized as a cascade of second order sections. `numStages` refers to the number of second order stages used. For example, an 8th order filter would be realized with `numStages=4` second order stages. A 9th order filter would be realized with `numStages=5` second order stages with the coefficients for one of the stages configured as a first order filter (`b2=0` and `a2=0`).



The `pState` points to state variables array. Each Biquad stage has 4 state variables `x[n-1], x[n-2],` y[n-1], and y[n-2] and each state variable in 1.63 format to improve precision. The state variables are arranged in the array as:

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of 4*numStages values of data in 1.63 format. The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared.

Init Function There is also an associated initialization function which performs the following operations:

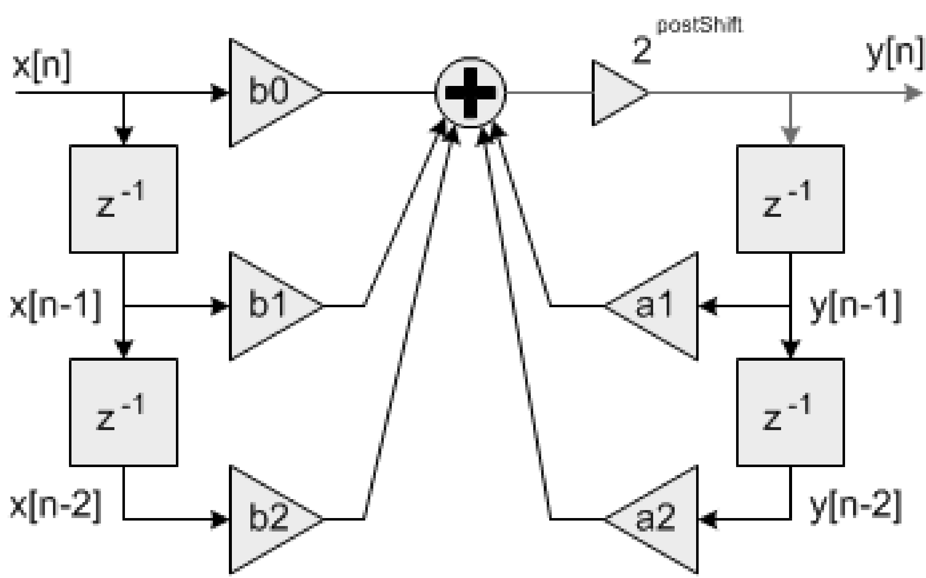
- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, postShift, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. For example, to statically initialize the filter instance structure use where numStages is the number of Biquad stages in the filter; pState is the address of the state buffer; pCoeffs is the address of the coefficient buffer; postShift shift to be applied which is described in detail below.

Fixed-Point Behavior Care must be taken while using Biquad Cascade 32x64 filter function. Following issues must be considered:

- Scaling of coefficients
- Filter gain
- Overflow and saturation

Filter coefficients are represented as fractional values and restricted to lie in the range [-1 +1). The processing function has an additional scaling parameter postShift which allows the filter coefficients to exceed the range [+1 -1). At the output of the filter's accumulator is a shift register which shifts the result by postShift bits. This essentially scales the filter coefficients by 2^postShift. For example, to realize the coefficients set the Coefficient array to: and set postShift=1



The second thing to keep in mind is the gain through the filter. The frequency response of a Biquad filter is a function of its coefficients. It is possible for the gain through the filter to exceed 1.0 meaning that the filter increases the amplitude of certain frequencies. This means that an input signal with amplitude < 1.0 may result in an output > 1.0 and these are saturated or overflowed based on the implementation of the filter. To avoid this behavior the filter needs to be scaled down such that its peak gain < 1.0 or the input signal must be scaled down so that the combination of input and filter are never overflowed.

The third item to consider is the overflow and saturation behavior of the fixed-point Q31 version. This is described in the function specific documentation below.

Functions

void **riscv_biquad_cas_df1_32x64_init_q31**(riscv_biquad_cas_df1_32x64_ins_q31 **S*, uint8_t *numStages*, **const** q31_t **pCoeffs*, q63_t **pState*, uint8_t *postShift*)

Initialization function for the Q31 Biquad cascade 32x64 filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order: where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` points to state variables array and size of each state variable is 1.63 format. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1],` and `y[n-2]`. The state variables are arranged in the state array as: The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the high precision Q31 Biquad cascade filter structure
- `[in] numStages`: number of 2nd order stages in the filter
- `[in] pCoeffs`: points to the filter coefficients
- `[in] pState`: points to the state buffer
- `[in] postShift`: Shift to be applied after the accumulator. Varies according to the coefficients format

void **riscv_biquad_cas_df1_32x64_q31**(**const** riscv_biquad_cas_df1_32x64_ins_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 Biquad cascade 32x64 filter.

Return none

Details The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2 bits and lie in the range [-0.25 +0.25). After all 5 multiply-accumulates are performed, the 2.62 accumulator is shifted by `postShift` bits and the result truncated to 1.31 format by discarding the low 32 bits.

Two related functions are provided in the NMSIS DSP library.

- riscv_biquad_cascade_df1_q31() implements a Biquad cascade with 32-bit coefficients and state variables with a Q63 accumulator.

- riscv_biquad_cascade_df1_fast_q31() implements a Biquad cascade with 32-bit coefficients and state variables with a Q31 accumulator.

Parameters

- `[in] S`: points to an instance of the high precision Q31 Biquad cascade filter
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

Biquad Cascade IIR Filters Using Direct Form I Structure

void **riscv_biquad_cascade_df1_f16**(**const** riscv_biquad_casd_df1_inst_f16 **S*, **const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_biquad_cascade_df1_f32**(**const** riscv_biquad_casd_df1_inst_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_biquad_cascade_df1_fast_q15**(**const** riscv_biquad_casd_df1_inst_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_biquad_cascade_df1_fast_q31**(**const** riscv_biquad_casd_df1_inst_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_biquad_cascade_df1_init_f16**(riscv_biquad_casd_df1_inst_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

void **riscv_biquad_cascade_df1_init_f32**(riscv_biquad_casd_df1_inst_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

void **riscv_biquad_cascade_df1_init_q15**(riscv_biquad_casd_df1_inst_q15 **S*, uint8_t *numStages*, **const** q15_t **pCoeffs*, q15_t **pState*, int8_t *postShift*)

void **riscv_biquad_cascade_df1_init_q31**(riscv_biquad_casd_df1_inst_q31 **S*, uint8_t *numStages*, **const** q31_t **pCoeffs*, q31_t **pState*, int8_t *postShift*)

void **riscv_biquad_cascade_df1_q15**(**const** riscv_biquad_casd_df1_inst_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

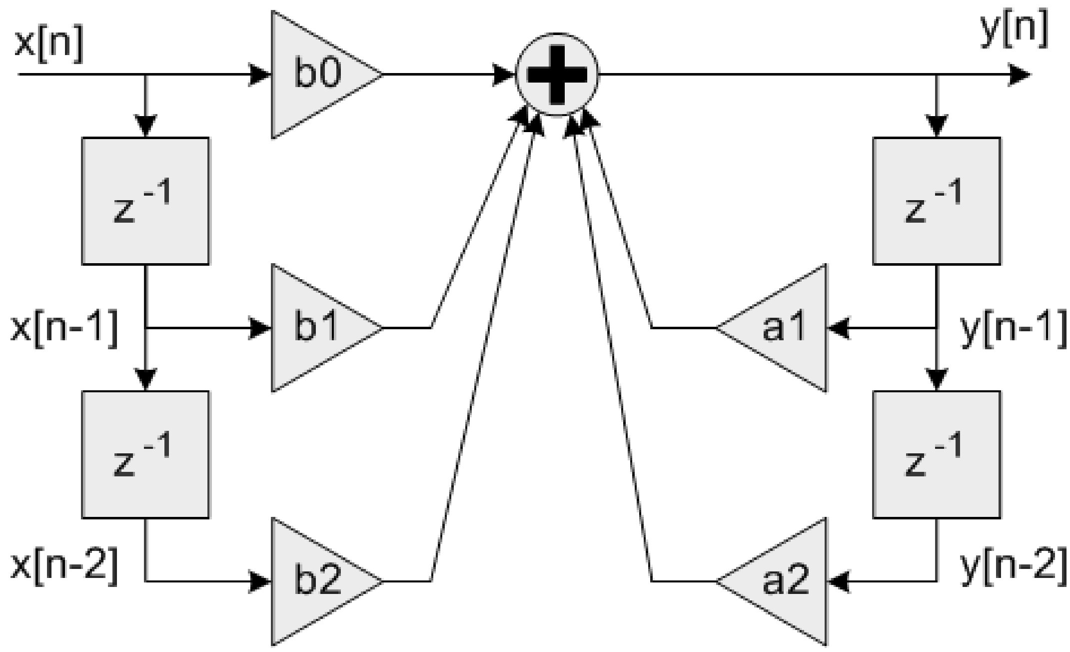
void **riscv_biquad_cascade_df1_q31**(**const** riscv_biquad_casd_df1_inst_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **BiquadCascadeDF1**

This set of functions implements arbitrary order recursive (IIR) filters. The filters are implemented as a cascade of second order Biquad sections. The functions support Q15, Q31 and floating-point data types. Fast version of Q15 and Q31 also available.

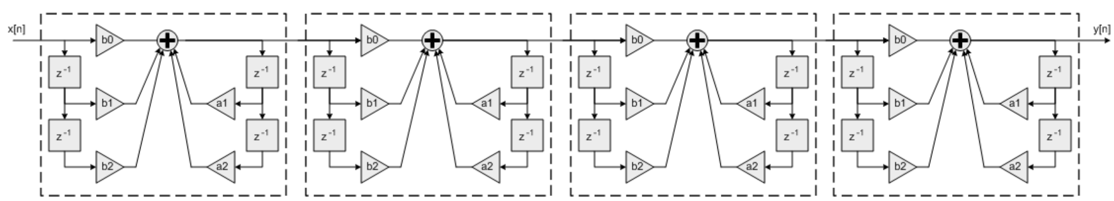
The functions operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` points to the array of input data and `pDst` points to the array of output data. Both arrays contain `blockSize` values.

Algorithm Each Biquad stage implements a second order filter using the difference equation: A Direct Form I algorithm is used with 5 coefficients and 4 state variables per stage. Coefficients `b0, b1 and b2` multiply the input signal `x[n]` and are referred to as the feedforward coefficients. Coefficients `a1` and `a2` multiply the output signal `y[n]` and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools use the difference equation In this case the feedback coefficients `a1` and `a2` must be negated when used with the NMSIS DSP



Library.

Higher order filters are realized as a cascade of second order sections. `numStages` refers to the number of second order stages used. For example, an 8th order filter would be realized with `numStages=4` second order stages. A 9th order filter would be realized with `numStages=5` second order stages with the coefficients for one of the stages configured as a first order filter (`b2=0` and



`a2=0`).

The `pState` points to state variables array. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1],` and `y[n-2]`. The state variables are arranged in the `pState` array as:

The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Init Function There is also an associated initialization function for each data type. The initialization function performs following operations:

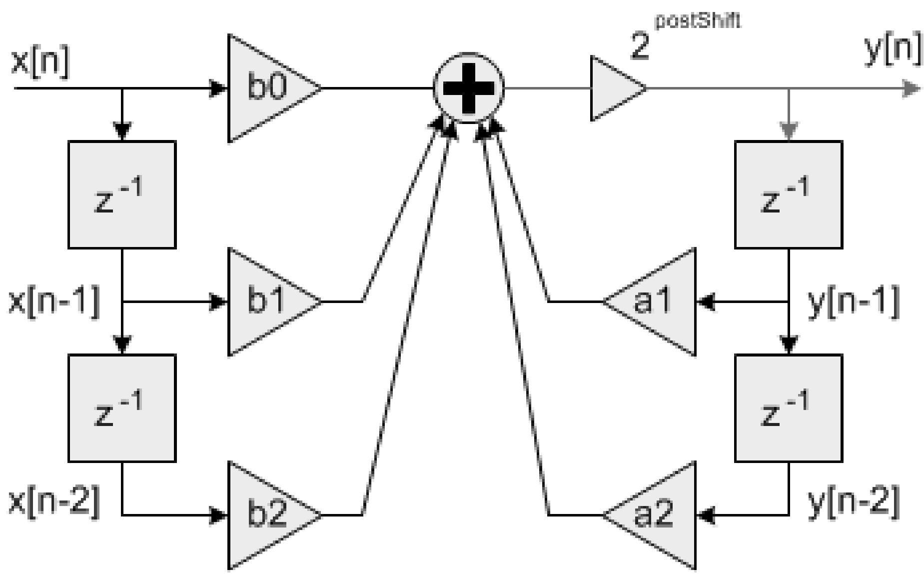
- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 3 different data type filter instance structures where `numStages` is the number of Biquad stages in the filter; `pState` is the address of the state buffer; `pCoeffs` is the address of the coefficient buffer; `postShift` shift to be applied.

Fixed-Point Behavior Care must be taken when using the Q15 and Q31 versions of the Biquad Cascade filter functions. Following issues must be considered:

- Scaling of coefficients
- Filter gain
- Overflow and saturation

Scaling of coefficients Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range `[-1 +1)`. The fixed-point functions have an additional scaling parameter `postShift` which allow the filter coefficients to exceed the range `[+1 -1)`. At the output of the filter's accumulator is a shift register which shifts the result by `postShift` bits. This essentially scales the filter coefficients by `2^postShift`. For example, to realize the coefficients set the pCoeffs array to: and set



`postShift=1`

Filter gain The frequency response of a Biquad filter is a function of its coefficients. It is possible for the gain through the filter to exceed 1.0 meaning that the filter increases the amplitude of certain frequencies. This means that an input signal with amplitude < 1.0 may result in an output > 1.0 and these are saturated or overflowed based on the implementation of the filter. To avoid this behavior the filter needs to be scaled down such that its peak gain < 1.0 or the input signal must be scaled down so that the combination of input and filter are never overflowed.

Overflow and saturation For Q15 and Q31 versions, it is described separately as part of the function specific documentation below.

Functions

void **riscv_biquad_cascade_df1_f16**(**const** riscv_biquad_casd_df1_inst_f16 **S*, **const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Processing function for the floating-point Biquad cascade filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_biquad_cascade_df1_f32**(**const** riscv_biquad_casd_df1_inst_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for the floating-point Biquad cascade filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_biquad_cascade_df1_fast_q15**(**const** riscv_biquad_casd_df1_inst_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 Biquad cascade filter (fast variant).

Fast but less precise processing function for the Q15 Biquad cascade filter for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, 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 two bits and lie in the range [-0.25 +0.25). The 2.30 accumulator is then shifted by `postShift` bits and the result truncated to 1.15 format by discarding the low 16 bits.

Remark Refer to riscv_biquad_cascade_df1_q15() for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use the function riscv_biquad_cascade_df1_init_q15() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q15 Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process per call

void **riscv_biquad_cascade_df1_fast_q31**(**const** riscv_biquad_casd_df1_inst_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 Biquad cascade filter (fast variant).

Fast but less precise processing function for the Q31 Biquad cascade filter for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by two bits and lie in the range [-0.25 +0.25). Use the intialization function riscv_biquad_cascade_df1_init_q31() to initialize filter structure.

Remark Refer to riscv_biquad_cascade_df1_q31() for a slower implementation of this function which uses 64-bit accumulation to provide higher precision. Both the slow and the fast versions use the same instance structure. Use the function riscv_biquad_cascade_df1_init_q31() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q31 Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process per call

void **riscv_biquad_cascade_df1_init_f16** (riscv_biquad_casd_df1_inst_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

Initialization function for the floating-point Biquad cascade filter.

The initialization function which must be used is riscv_biquad_cascade_df1_mve_init_f16.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1], and y[n-2]`. The state variables are arranged in the `pState` array as: The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

For MVE code, an additional buffer of modified coefficients is required. Its size is numStages and each element of this buffer has type riscv_biquad_mod_coef_f16. So, its total size is 96*numStages float16_t elements.

Parameters

- `[inout] S`: points to an instance of the floating-point Biquad cascade structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

void **riscv_biquad_cascade_df1_init_f32** (riscv_biquad_casd_df1_inst_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

Initialization function for the floating-point Biquad cascade filter.

The initialization function which must be used is riscv_biquad_cascade_df1_mve_init_f32.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1],` and `y[n-2]`. The state variables are arranged in the `pState` array as: The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

For MVE code, an additional buffer of modified coefficients is required. Its size is numStages and each element of this buffer has type riscv_biquad_mod_coef_f32. So, its total size is 32*numStages float32_t elements.

Parameters

- `[inout] S`: points to an instance of the floating-point Biquad cascade structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

void **riscv_biquad_cascade_df1_init_q15** (riscv_biquad_casd_df1_inst_q15 **S*, uint8_t *numStages*, **const** q15_t **pCoeffs*, q15_t **pState*, int8_t *postShift*)

Initialization function for the Q15 Biquad cascade filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `6*numStages` values. The zero coefficient between `b1` and `b2` facilities use of 16-bit SIMD instructions on the RISC-V Core with DSP.

The state variables are stored in the array `pState`. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1],` and `y[n-2]`. The state variables are arranged in the `pState` array as: The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the Q15 Biquad cascade structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

- `[in] postShift`: Shift to be applied to the accumulator result. Varies according to the coefficients format

void **riscv_biquad_cascade_df1_init_q31**(riscv_biquad_casd_df1_inst_q31 **S*, uint8_t *numStages*, **const** q31_t **pCoeffs*, q31_t **pState*, int8_t *postShift*)

Initialization function for the Q31 Biquad cascade filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` points to state variables array. Each Biquad stage has 4 state variables `x[n-1], x[n-2], y[n-1],` and `y[n-2]`. The state variables are arranged in the `pState` array as: The 4 state variables for stage 1 are first, then the 4 state variables for stage 2, and so on. The state array has a total length of `4*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the Q31 Biquad cascade structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.
- `[in] postShift`: Shift to be applied after the accumulator. Varies according to the coefficients format

void **riscv_biquad_cascade_df1_q15**(**const** riscv_biquad_casd_df1_inst_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 Biquad cascade filter.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is then shifted by `postShift` bits to truncate the result to 1.15 format by discarding the low 16 bits. Finally, the result is saturated to 1.15 format.

Remark Refer to riscv_biquad_cascade_df1_fast_q15() for a faster but less precise implementation of this filter.

Parameters

- `[in] S`: points to an instance of the Q15 Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the location where the output result is written
- `[in] blockSize`: number of samples to process

void **riscv_biquad_cascade_df1_q31** (**const** riscv_biquad_casd_df1_inst_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 Biquad cascade filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2 bits and lie in the range [-0.25 +0.25). After all 5 multiply-accumulates are performed, the 2.62 accumulator is shifted by `postShift` bits and the result truncated to 1.31 format by discarding the low 32 bits.

Remark Refer to riscv_biquad_cascade_df1_fast_q31() for a faster but less precise implementation of this filter.

Parameters

- `[in] S`: points to an instance of the Q31 Biquad cascade structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

Biquad Cascade IIR Filters Using a Direct Form II Transposed Structure

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f16(const riscv_biquad_cascade_df2T_

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f32(const riscv_biquad_cascade_df2T_

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f64(const riscv_biquad_cascade_df2T_

void **riscv_biquad_cascade_df2T_init_f16** (riscv_biquad_cascade_df2T_instance_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

void **riscv_biquad_cascade_df2T_init_f32** (riscv_biquad_cascade_df2T_instance_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

void **riscv_biquad_cascade_df2T_init_f64** (riscv_biquad_cascade_df2T_instance_f64 **S*, uint8_t *numStages*, **const** float64_t **pCoeffs*, float64_t **pState*)

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_stereo_df2T_f16(const riscv_biquad_cascac

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_stereo_df2T_f32(const riscv_biquad_cascac

void **riscv_biquad_cascade_stereo_df2T_init_f16** (riscv_biquad_cascade_stereo_df2T_instance_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

void **riscv_biquad_cascade_stereo_df2T_init_f32** (riscv_biquad_cascade_stereo_df2T_instance_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

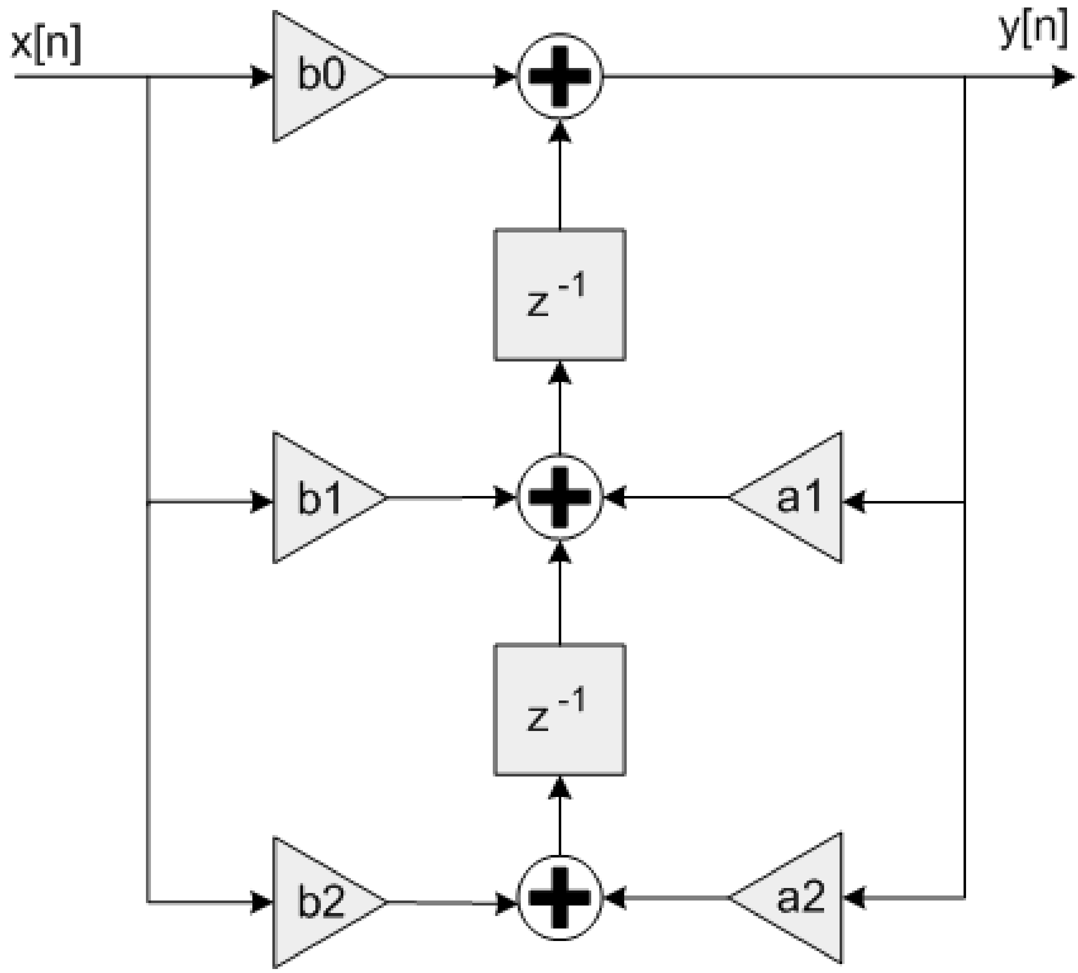
group **BiquadCascadeDF2T**

This set of functions implements arbitrary order recursive (IIR) filters using a transposed direct form II structure. The filters are implemented as a cascade of second order Biquad sections. These functions provide a slight memory savings as compared to the direct form I Biquad filter functions. Only floating-point data is supported.

This function operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` points to the array of input data and `pDst` points to the array of output data. Both arrays contain `blockSize` values.

Algorithm Each Biquad stage implements a second order filter using the difference equation: where d1 and d2 represent the two state values.

A Biquad filter using a transposed Direct Form II structure is shown below. Coefficients `b0, b1, and b2` multiply the input signal `x[n]` and are referred to as the feedforward coefficients. Coefficients `a1` and `a2` multiply the output signal `y[n]` and are referred to as the feedback coefficients. Pay careful attention to the sign of the feedback coefficients. Some design tools flip the sign of the feedback coefficients: In this case the feedback coefficients `a1` and `a2` must be negated when used with the NMSIS DSP Library.



Higher order filters are realized as a cascade of second order sections. `numStages` refers to the number of second order stages used. For example, an 8th order filter would be realized with `numStages=4` second order stages. A 9th order filter would be realized with `numStages=5` second order stages with the coefficients for one of the stages configured as a first order filter (`b2=0` and `a2=0`).

`pState` points to the state variable array. Each Biquad stage has 2 state variables `d1` and `d2`. The state variables are arranged in the `pState` array as: where `d1x` refers to the state variables for the first Biquad and `d2x` refers to the state variables for the second Biquad. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

The NMSIS library contains Biquad filters in both Direct Form I and transposed Direct Form II. The advantage of the Direct Form I structure is that it is numerically more robust for fixed-point data types. That is why the Direct Form I structure supports Q15 and Q31 data types. The transposed Direct Form II structure, on the other hand, requires a wide dynamic range for the state variables `d1` and `d2`. Because of this, the NMSIS library only has a floating-point version of the Direct Form II Biquad. The advantage of the Direct Form II Biquad is that it requires half the number of state variables, 2 rather than 4, per Biquad stage.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared.

Init Functions There is also an associated initialization function. The initialization function performs following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. For example, to statically initialize the instance structure use where `numStages` is the number of Biquad stages in the filter; `pState` is the address of the state buffer. `pCoeffs` is the address of the coefficient buffer;

Functions

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f16(const riscv_biquad_cascade_

Processing function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Parameters

- `[in] S`: points to an instance of the filter data structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f32(const riscv_biquad_cascade_

Processing function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Parameters

- `[in] S`: points to an instance of the filter data structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_df2T_f64(const riscv_biquad_cascade_

Processing function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Parameters

- `[in] S`: points to an instance of the filter data structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_biquad_cascade_df2T_init_f16**(riscv_biquad_cascade_df2T_instance_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

Initialization function for the floating-point transposed direct form II Biquad cascade filter.

For Neon version, this array is bigger. If numstages = 4x + y, then the array has size: 32*x + 5*y and it must be initialized using the function riscv_biquad_cascade_df2T_compute_coefs_f16 which is taking the standard array coefficient as parameters.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order in the not Neon version.

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

Parameters

- `[inout] S`: points to an instance of the filter data structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

But, an array of 8*numstages is a good approximation.

Then, the initialization can be done with:

In this example, neonCoefs is a bigger array of size 8 * numStages. coefs is the standard array:

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1,` and `d2`. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

void **riscv_biquad_cascade_df2T_init_f32**(riscv_biquad_cascade_df2T_instance_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

Initialization function for the floating-point transposed direct form II Biquad cascade filter.

For Neon version, this array is bigger. If numstages = 4x + y, then the array has size: 32*x + 5*y and it must be initialized using the function riscv_biquad_cascade_df2T_compute_coefs_f32 which is taking the standard array coefficient as parameters.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order in the not Neon version.

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

Parameters

- `[inout] S`: points to an instance of the filter data structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

But, an array of 8*numstages is a good approximation.

Then, the initialization can be done with:

In this example, neonCoefs is a bigger array of size 8 * numStages. coefs is the standard array:

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1,` and `d2`. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

void **`riscv_biquad_cascade_df2T_init_f64`**(riscv_biquad_cascade_df2T_instance_f64 **S*, uint8_t *numStages*, **`const`** float64_t **pCoeffs*, float64_t **pState*)

Initialization function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1,` and `d2`. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the filter data structure
- `[in] numStages`: number of 2nd order stages in the filter
- `[in] pCoeffs`: points to the filter coefficients
- `[in] pState`: points to the state buffer

`LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_stereo_df2T_f16(const riscv_biquad_c`

Processing function for the floating-point transposed direct form II Biquad cascade filter.

Processing function for the floating-point transposed direct form II Biquad cascade filter. 2 channels.

Return none

Parameters

- `[in] S`: points to an instance of the filter data structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data

- `[in] blockSize`: number of samples to process

LOW_OPTIMIZATION_ENTER void riscv_biquad_cascade_stereo_df2T_f32(const riscv_biquad_ca

Processing function for the floating-point transposed direct form II Biquad cascade filter.

Processing function for the floating-point transposed direct form II Biquad cascade filter. 2 channels.

Return none

Parameters

- `[in] S`: points to an instance of the filter data structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_biquad_cascade_stereo_df2T_init_f16**(riscv_biquad_cascade_stereo_df2T_instance_f16 **S*, uint8_t *numStages*, **const** float16_t **pCoeffs*, float16_t **pState*)

Initialization function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1,` and `d2` for each channel. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on. The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the filter data structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

void **riscv_biquad_cascade_stereo_df2T_init_f32**(riscv_biquad_cascade_stereo_df2T_instance_f32 **S*, uint8_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

Initialization function for the floating-point transposed direct form II Biquad cascade filter.

Return none

Coefficient and State Ordering The coefficients are stored in the array `pCoeffs` in the following order:

where `b1x` and `a1x` are the coefficients for the first stage, `b2x` and `a2x` are the coefficients for the second stage, and so on. The `pCoeffs` array contains a total of `5*numStages` values.

The `pState` is a pointer to state array. Each Biquad stage has 2 state variables `d1,` and `d2` for each channel. The 2 state variables for stage 1 are first, then the 2 state variables for stage 2, and so on.

The state array has a total length of `2*numStages` values. The state variables are updated after each block of data is processed; the coefficients are untouched.

Parameters

- `[inout] S`: points to an instance of the filter data structure.
- `[in] numStages`: number of 2nd order stages in the filter.
- `[in] pCoeffs`: points to the filter coefficients.
- `[in] pState`: points to the state buffer.

Convolution

void **riscv_conv_f32** (**const** float32_t *pSrcA*, uint32_t *srcALen*, **const** float32_t *pSrcB*, uint32_t *srcBLen*, float32_t *pDst*)

void **riscv_conv_fast_opt_q15** (**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*, q15_t *pScratch1*, q15_t *pScratch2*)

void **riscv_conv_fast_q15** (**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*)

void **riscv_conv_fast_q31** (**const** q31_t *pSrcA*, uint32_t *srcALen*, **const** q31_t *pSrcB*, uint32_t *srcBLen*, q31_t *pDst*)

void **riscv_conv_opt_q15** (**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*, q15_t *pScratch1*, q15_t *pScratch2*)

void **riscv_conv_opt_q7** (**const** q7_t *pSrcA*, uint32_t *srcALen*, **const** q7_t *pSrcB*, uint32_t *srcBLen*, q7_t *pDst*, q15_t *pScratch1*, q15_t *pScratch2*)

void **riscv_conv_q15** (**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*)

void **riscv_conv_q31** (**const** q31_t *pSrcA*, uint32_t *srcALen*, **const** q31_t *pSrcB*, uint32_t *srcBLen*, q31_t *pDst*)

void **riscv_conv_q7** (**const** q7_t *pSrcA*, uint32_t *srcALen*, **const** q7_t *pSrcB*, uint32_t *srcBLen*, q7_t *pDst*)

group **Conv**

Convolution is a mathematical operation that operates on two finite length vectors to generate a finite length output vector. Convolution is similar to correlation and is frequently used in filtering and data analysis. The NMSIS DSP library contains functions for convolving Q7, Q15, Q31, and floating-point data types. The library also provides fast versions of the Q15 and Q31 functions.

Algorithm Let `a[n]` and `b[n]` be sequences of length `srcALen` and `srcBLen` samples respectively. Then the convolution

$$c[n] = \sum_{k=0}^{srcALen} a[k]b[n-k]$$

is defined as

Note that `c[n]` is of length `srcALen + srcBLen - 1` and is defined over the interval `n=0, 1, 2, ..., srcALen + srcBLen - 2`. `pSrcA` points to the first input vector of length `srcALen` and `pSrcB` points to the second input vector of length `srcBLen`. The output result is written to `pDst` and the calling function must allocate `srcALen+srcBLen-1` words for the result.

Conceptually, when two signals `a[n]` and `b[n]` are convolved, the signal `b[n]` slides over `a[n]`. For each offset `n`, the overlapping portions of a[n] and b[n] are multiplied and summed together.

Note that convolution is a commutative operation:

This means that switching the A and B arguments to the convolution functions has no effect.

Fixed-Point Behavior Convolution requires summing up a large number of intermediate products. As such, the Q7, Q15, and Q31 functions run a risk of overflow and saturation. Refer to the function specific documentation below for further details of the particular algorithm used.

Fast Versions Fast versions are supported for Q31 and Q15. Cycles for Fast versions are less compared to Q31 and Q15 of conv and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory (Scratch memory) compared to Q15 and Q7 versions

Functions

void **riscv_conv_f32** (**const** float32_t **pSrcA*, uint32_t *srcALen*, **const** float32_t **pSrcB*, uint32_t *srcBLen*, float32_t **pDst*)

Convolution of floating-point sequences.

Return none

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

void **riscv_conv_fast_opt_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch1*, q15_t **pScratch2*)

Convolution of Q15 sequences (fast version).

Convolution of Q15 sequences (fast version) for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Remark Refer to riscv_conv_q15() for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1
- `[in] pScratch1`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2
- `[in] pScratch2`: points to scratch buffer of size min(srcALen, srcBLen

void **riscv_conv_fast_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

Convolution of Q15 sequences (fast version).

Convolution of Q15 sequences (fast version) for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Remark Refer to riscv_conv_q15() for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1

void **riscv_conv_fast_q31**(**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

Convolution of Q31 sequences (fast version).

Convolution of Q31 sequences (fast version) for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally.

Remark Refer to riscv_conv_q31() for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

Parameters

- `[in] pSrcA`: points to the first input sequence.
- `[in] srcALen`: length of the first input sequence.
- `[in] pSrcB`: points to the second input sequence.
- `[in] srcBLen`: length of the second input sequence.
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

void **riscv_conv_opt_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch1*, q15_t **pScratch2*)

Convolution of Q15 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Remark Refer to riscv_conv_fast_q15() for a faster but less precise version of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.
- `[in] pScratch1`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.
- `[in] pScratch2`: points to scratch buffer of size min(srcALen, srcBLen).

void **riscv_conv_opt_q7**(**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, q15_t **pScratch1*, q15_t **pScratch2*)

Convolution of Q7 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as `max(srcALen, srcBLen)<131072`. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and then saturated to 1.7 format.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.
- `[in] pScratch1`: points to scratch buffer(of type q15_t) of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.
- `[in] pScratch2`: points to scratch buffer (of type q15_t) of size min(srcALen, srcBLen).

void **riscv_conv_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

Convolution of Q15 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Remark Refer to riscv_conv_fast_q15() for a faster but less precise version of this function.

Remark Refer to riscv_conv_opt_q15() for a faster implementation of this function using scratch buffers.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

void **riscv_conv_q31**(**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

Convolution of Q31 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down the inputs by log2(min(srcALen, srcBLen)) (log2 is read as log to the base 2) times to avoid overflows, as maximum of min(srcALen, srcBLen) number of additions are carried internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

Remark Refer to riscv_conv_fast_q31() for a faster but less precise implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

void **riscv_conv_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*)

Convolution of Q7 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as `max(srcALen, srcBLen)<131072`. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and then saturated to 1.7 format.

Remark Refer to riscv_conv_opt_q7() for a faster implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length srcALen+srcBLen-1.

Partial Convolution

riscv_status **riscv_conv_partial_f32** (**const** float32_t **pSrcA*, uint32_t *srcALen*, **const** float32_t **pSrcB*, uint32_t *srcBLen*, float32_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

riscv_status **riscv_conv_partial_fast_opt_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t **pScratch1*, q15_t **pScratch2*)

riscv_status **riscv_conv_partial_fast_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

riscv_status **riscv_conv_partial_fast_q31** (**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*) riscv_status **riscv_conv_partial_opt_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t **pScratch1*, q15_t **pScratch2*)

riscv_status **riscv_conv_partial_opt_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t **pScratch1*, q15_t **pScratch2*)

riscv_status **riscv_conv_partial_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

riscv_status **riscv_conv_partial_q31** (**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

riscv_status **riscv_conv_partial_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

group **PartialConv**

Partial Convolution is equivalent to Convolution except that a subset of the output samples is generated. Each function has two additional arguments. `firstIndex` specifies the starting index of the subset of output samples. `numPoints` is the number of output samples to compute. The function computes the output in the range `[firstIndex, ..., firstIndex+numPoints-1]`. The output array `pDst` contains `numPoints` values.

The allowable range of output indices is [0 srcALen+srcBLen-2]. If the requested subset does not fall in this range then the functions return RISCV_MATH_ARGUMENT_ERROR. Otherwise the functions return RISCV_MATH_SUCCESS.

Note Refer to riscv_conv_f32() for details on fixed point behavior.

Fast Versions Fast versions are supported for Q31 and Q15 of partial convolution. Cycles for Fast versions are less compared to Q31 and Q15 of partial conv and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory (Scratch memory) compared to Q15 and Q7 versions of partial convolution

Functions

riscv_status **riscv_conv_partial_f32** (**const** float32_t **pSrcA*, uint32_t *srcALen*, **const** float32_t **pSrcB*, uint32_t *srcBLen*, float32_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of floating-point sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence

- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

riscv_status **riscv_conv_partial_fast_opt_q15**(**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t *pScratch1*, q15_t *pScratch2*)

Partial convolution of Q15 sequences (fast version).

Partial convolution of Q15 sequences (fast version) for RISC-V Core with DSP enabled.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_q15() for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed
- `[in] pScratch1`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2
- `[in] pScratch2`: points to scratch buffer of size min(srcALen, srcBLen)

riscv_status **riscv_conv_partial_fast_q15**(**const** q15_t *pSrcA*, uint32_t *srcALen*, **const** q15_t *pSrcB*, uint32_t *srcBLen*, q15_t *pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of Q15 sequences (fast version).

Partial convolution of Q15 sequences (fast version) for RISC-V Core with DSP enabled.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_q15() for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

riscv_status **`riscv_conv_partial_fast_q31`**(**`const`** q31_t **pSrcA*, uint32_t *srcALen*, **`const`** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of Q31 sequences (fast version).

Partial convolution of Q31 sequences (fast version) for RISC-V Core with DSP enabled.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_q31() for a slower implementation of this function which uses a 64-bit accumulator to provide higher precision.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

riscv_status **`riscv_conv_partial_opt_q15`**(**`const`** q15_t **pSrcA*, uint32_t *srcALen*, **`const`** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t **pScratch1*, q15_t **pScratch2*)

Partial convolution of Q15 sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_fast_q15() for a faster but less precise version of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed
- `[in] pScratch1`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.
- `[in] pScratch2`: points to scratch buffer of size min(srcALen, srcBLen).

riscv_status **riscv_conv_partial_opt_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*, q15_t **pScratch1*, q15_t **pScratch2*)

Partial convolution of Q7 sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed
- `[in] pScratch1`: points to scratch buffer(of type q15_t) of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.
- `[in] pScratch2`: points to scratch buffer (of type q15_t) of size min(srcALen, srcBLen).

riscv_status **riscv_conv_partial_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of Q15 sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_fast_q15() for a faster but less precise version of this function.

Remark Refer to riscv_conv_partial_opt_q15() for a faster implementation of this function using scratch buffers.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

riscv_status **riscv_conv_partial_q31**(**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of Q31 sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_fast_q31() for a faster but less precise implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

riscv_status **riscv_conv_partial_q7**(**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, uint32_t *firstIndex*, uint32_t *numPoints*)

Partial convolution of Q7 sequences.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : requested subset is not in the range [0 srcALen+srcBLen-2]

Remark Refer to riscv_conv_partial_opt_q7() for a faster implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written
- `[in] firstIndex`: is the first output sample to start with
- `[in] numPoints`: is the number of output points to be computed

Correlation

void **riscv_correlate_f16** (**const** float16_t **pSrcA*, uint32_t *srcALen*, **const** float16_t **pSrcB*, uint32_t *srcBLen*, float16_t **pDst*)

void **riscv_correlate_f32** (**const** float32_t **pSrcA*, uint32_t *srcALen*, **const** float32_t **pSrcB*, uint32_t *srcBLen*, float32_t **pDst*)

void **riscv_correlate_fast_opt_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch*)

void **riscv_correlate_fast_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

void **riscv_correlate_fast_q31** (**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

void **riscv_correlate_opt_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch*)

void **riscv_correlate_opt_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, q15_t **pScratch1*, q15_t **pScratch2*)

void **riscv_correlate_q15** (**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

void **riscv_correlate_q31** (**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

void **riscv_correlate_q7** (**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*)

group **Corr**

Correlation is a mathematical operation that is similar to convolution. As with convolution, correlation uses two signals to produce a third signal. The underlying algorithms in correlation and convolution are identical except that one of the inputs is flipped in convolution. Correlation is commonly used to measure the similarity between two signals. It has applications in pattern recognition, cryptanalysis, and searching. The NMSIS library provides correlation functions for Q7, Q15, Q31 and floating-point data types. Fast versions of the Q15 and Q31 functions are also provided.

Algorithm Let `a[n]` and `b[n]` be sequences of length `srcALen` and `srcBLen` samples respectively. The convolution of the two signals is denoted by In correlation, one of the signals is flipped in time

$$c[n] = \sum_{k=0}^{srcALen} a[k]\, b[k-n]$$

and this is mathematically defined as

The `pSrcA` points to the first input vector of length `srcALen` and `pSrcB` points to the second input vector of length `srcBLen`. The result `c[n]` is of length `2 * max(srcALen, srcBLen) - 1` and is defined over the interval `n=0, 1, 2, ..., (2 * max(srcALen, srcBLen) - 2)`. The output result is written to `pDst` and the calling function must allocate `2 * max(srcALen, srcBLen) - 1` words for the result.

Note The `pDst` should be initialized to all zeros before being used.

Fixed-Point Behavior Correlation requires summing up a large number of intermediate products. As such, the Q7, Q15, and Q31 functions run a risk of overflow and saturation. Refer to the function specific documentation below for further details of the particular algorithm used.

Fast Versions Fast versions are supported for Q31 and Q15. Cycles for Fast versions are less compared to Q31 and Q15 of correlate and the design requires the input signals should be scaled down to avoid intermediate overflows.

Opt Versions Opt versions are supported for Q15 and Q7. Design uses internal scratch buffer for getting good optimisation. These versions are optimised in cycles and consumes more memory (Scratch memory) compared to Q15 and Q7 versions of correlate

Functions

void **riscv_correlate_f16**(**const** float16_t **pSrcA*, uint32_t *srcALen*, **const** float16_t **pSrcB*, uint32_t *srcBLen*, float16_t **pDst*)

Correlation of floating-point sequences.

Return none

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_f32**(**const** float32_t **pSrcA*, uint32_t *srcALen*, **const** float32_t **pSrcB*, uint32_t *srcBLen*, float32_t **pDst*)

Correlation of floating-point sequences.

Return none

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence

- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_fast_opt_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch*)

Correlation of Q15 sequences (fast version).

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen) to avoid overflow since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Remark Refer to riscv_correlate_q15() for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence.
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.
- `[in] pScratch`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

void **riscv_correlate_fast_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

Correlation of Q15 sequences (fast version).

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen) to avoid overflow since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.30 accumulator is right shifted by 15 bits and then saturated to 1.15 format to yield the final result.

Remark Refer to riscv_correlate_q15() for a slower implementation of this function which uses a 64-bit accumulator to avoid wrap around distortion.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence

- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_fast_q31**(**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

Correlation of Q31 sequences (fast version).

Return none

Scaling and Overflow Behavior This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen)to avoid overflows since a maximum of min(srcALen, srcBLen) number of additions is carried internally.

Remark Refer to riscv_correlate_q31() for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_opt_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*, q15_t **pScratch*)

Correlation of Q15 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Remark Refer to riscv_correlate_fast_q15() for a faster but less precise version of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence

- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.
- `[in] pScratch`: points to scratch buffer of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.

void **riscv_correlate_opt_q7**(**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*, q15_t **pScratch1*, q15_t **pScratch2*)

Correlation of Q7 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as `max(srcALen, srcBLen)<131072`. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and then saturated to 1.7 format.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.
- `[in] pScratch1`: points to scratch buffer(of type q15_t) of size max(srcALen, srcBLen) + 2*min(srcALen, srcBLen) - 2.
- `[in] pScratch2`: points to scratch buffer (of type q15_t) of size min(srcALen, srcBLen).

void **riscv_correlate_q15**(**const** q15_t **pSrcA*, uint32_t *srcALen*, **const** q15_t **pSrcB*, uint32_t *srcBLen*, q15_t **pDst*)

Correlation of Q15 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both inputs are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Remark Refer to riscv_correlate_fast_q15() for a faster but less precise version of this function.

Remark Refer to riscv_correlate_opt_q15() for a faster implementation of this function using scratch buffers.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence

- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_q31**(**const** q31_t **pSrcA*, uint32_t *srcALen*, **const** q31_t **pSrcB*, uint32_t *srcBLen*, q31_t **pDst*)

Correlation of Q31 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. Scale down one of the inputs by 1/min(srcALen, srcBLen)to avoid overflows since a maximum of min(srcALen, srcBLen) number of additions is carried internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

Remark Refer to riscv_correlate_fast_q31() for a faster but less precise implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

void **riscv_correlate_q7**(**const** q7_t **pSrcA*, uint32_t *srcALen*, **const** q7_t **pSrcB*, uint32_t *srcBLen*, q7_t **pDst*)

Correlation of Q7 sequences.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both the inputs are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. This approach provides 17 guard bits and there is no risk of overflow as long as `max(srcALen, srcBLen)<131072`. The 18.14 result is then truncated to 18.7 format by discarding the low 7 bits and saturated to 1.7 format.

Remark Refer to riscv_correlate_opt_q7() for a faster implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input sequence
- `[in] srcALen`: length of the first input sequence
- `[in] pSrcB`: points to the second input sequence
- `[in] srcBLen`: length of the second input sequence
- `[out] pDst`: points to the location where the output result is written. Length 2 * max(srcALen, srcBLen) - 1.

Finite Impulse Response (FIR) Decimator

void **riscv_fir_decimate_f32** (**const** riscv_fir_decimate_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_decimate_fast_q15** (**const** riscv_fir_decimate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_decimate_fast_q31** (**const** riscv_fir_decimate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

riscv_status **riscv_fir_decimate_init_f32** (riscv_fir_decimate_instance_f32 **S*, uint16_t *numTaps*, uint8_t *M*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

riscv_status **riscv_fir_decimate_init_q15** (riscv_fir_decimate_instance_q15 **S*, uint16_t *numTaps*, uint8_t *M*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

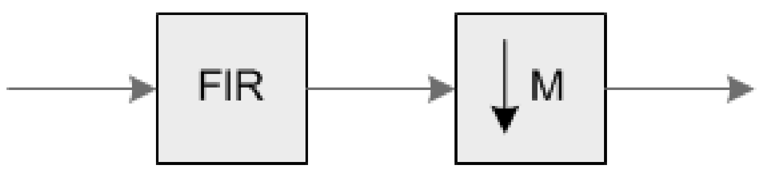
riscv_status **riscv_fir_decimate_init_q31** (riscv_fir_decimate_instance_q31 **S*, uint16_t *numTaps*, uint8_t *M*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

void **riscv_fir_decimate_q15** (**const** riscv_fir_decimate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_decimate_q31** (**const** riscv_fir_decimate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **FIR_decimate**

These functions combine an FIR filter together with a decimator. They are used in multirate systems for reducing the sample rate of a signal without introducing aliasing distortion. Conceptually, the functions are equivalent to the block diagram below: When decimating by a factor of `M`, the signal should be prefiltered by a lowpass filter with a normalized cutoff frequency of `1/M` in order to prevent aliasing distortion. The user of the function is responsible for providing the filter



coefficients.

The FIR decimator functions provided in the NMSIS DSP Library combine the FIR filter and the decimator in an efficient manner. Instead of calculating all of the FIR filter outputs and discarding `M-1` out of every `M`, only the samples output by the decimator are computed. The functions operate on blocks of input and output data. `pSrc` points to an array of `blockSize` input values and `pDst` points to an array of `blockSize/M` output values. In order to have an integer number of output samples `blockSize` must always be a multiple of the decimation factor `M`.

The library provides separate functions for Q15, Q31 and floating-point data types.

Algorithm: The FIR portion of the algorithm uses the standard form filter: where, `b[n]` are the filter coefficients.

The `pCoeffs` points to a coefficient array of size `numTaps`. Coefficients are stored in time reversed order.

`pState` points to a state array of size `numTaps + blockSize - 1`. Samples in the state buffer are stored in the order:

The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable array should be allocated separately. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer.
- Checks to make sure that the size of the input is a multiple of the decimation factor. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, M (decimation factor), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. The code below statically initializes each of the 3 different data type filter instance structures where `M` is the decimation factor; `numTaps` is the number of filter coefficients in the filter; `pCoeffs` is the address of the coefficient buffer; `pState` is the address of the state buffer. Be sure to set the values in the state buffer to zeros when doing static initialization.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the FIR decimate filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_fir_decimate_f32**(**const** riscv_fir_decimate_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for floating-point FIR decimator.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR decimator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_decimate_fast_q15**(**const** riscv_fir_decimate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 FIR decimator (fast variant).

Processing function for the Q15 FIR decimator (fast variant) for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, 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 log2(numTaps) bits (log2 is read as log to the base 2). The 2.30 accumulator is then truncated to 2.15 format and saturated to yield the 1.15 result.

Remark Refer to riscv_fir_decimate_q15() for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use function riscv_fir_decimate_init_q15() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q15 FIR decimator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of input samples to process per call

void **riscv_fir_decimate_fast_q31**(**const** riscv_fir_decimate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 FIR decimator (fast variant).

Processing function for the Q31 FIR decimator (fast variant) for RISC-V Core with DSP enabled.

Return none

Scaling and Overflow Behavior This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits (where log2 is read as log to the base 2).

Remark Refer to riscv_fir_decimate_q31() for a slower implementation of this function which uses a 64-bit accumulator to provide higher precision. Both the slow and the fast versions use the same instance structure. Use function riscv_fir_decimate_init_q31() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q31 FIR decimator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

riscv_status **riscv_fir_decimate_init_f32**(riscv_fir_decimate_instance_f32 **S*, uint16_t *numTaps*, uint8_t *M*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

Initialization function for the floating-point FIR decimator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_LENGTH_ERROR : `blockSize` is not a multiple of `M`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` words where `blockSize` is the number of input samples passed to `riscv_fir_decimate_f32()`. `M` is the decimation factor.

Parameters

- `[inout] S`: points to an instance of the floating-point FIR decimator structure
- `[in] numTaps`: number of coefficients in the filter
- `[in] M`: decimation factor
- `[in] pCoeffs`: points to the filter coefficients
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process per call

riscv_status **`riscv_fir_decimate_init_q15`** (riscv_fir_decimate_instance_q15 **S*, uint16_t *numTaps*, uint8_t *M*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

Initialization function for the Q15 FIR decimator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_LENGTH_ERROR : `blockSize` is not a multiple of `M`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` words where `blockSize` is the number of input samples to the call `riscv_fir_decimate_q15()`. `M` is the decimation factor.

Parameters

- `[inout] S`: points to an instance of the Q15 FIR decimator structure
- `[in] numTaps`: number of coefficients in the filter
- `[in] M`: decimation factor
- `[in] pCoeffs`: points to the filter coefficients
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process

riscv_status **`riscv_fir_decimate_init_q31`** (riscv_fir_decimate_instance_q31 **S*, uint16_t *numTaps*, uint8_t *M*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

Initialization function for the Q31 FIR decimator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_LENGTH_ERROR : `blockSize` is not a multiple of `M`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` words where `blockSize` is the number of input samples passed to `riscv_fir_decimate_q31()`. `M` is the decimation factor.

Parameters

- `[inout] S`: points to an instance of the Q31 FIR decimator structure
- `[in] numTaps`: number of coefficients in the filter
- `[in] M`: decimation factor
- `[in] pCoeffs`: points to the filter coefficients
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process

void **riscv_fir_decimate_q15**(**const** riscv_fir_decimate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 FIR decimator.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Remark Refer to riscv_fir_decimate_fast_q15() for a faster but less precise implementation of this function.

Parameters

- `[in] S`: points to an instance of the Q15 FIR decimator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of input samples to process per call

void **riscv_fir_decimate_q31**(**const** riscv_fir_decimate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 FIR decimator.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits (where log2 is read as log to the base 2). After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

Remark Refer to riscv_fir_decimate_fast_q31() for a faster but less precise implementation of this function.

Parameters

- `[in] S`: points to an instance of the Q31 FIR decimator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data

- `[in] blockSize`: number of samples to process

Finite Impulse Response (FIR) Filters

void **riscv_fir_f16** (**const** riscv_fir_instance_f16 **S*, **const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_f32** (**const** riscv_fir_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_fast_q15** (**const** riscv_fir_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

IAR_ONLY_LOW_OPTIMIZATION_ENTER void riscv_fir_fast_q31(const riscv_fir_instance_q31 * S,

void **riscv_fir_init_f16** (riscv_fir_instance_f16 **S*, uint16_t *numTaps*, **const** float16_t **pCoeffs*, float16_t **pState*, uint32_t *blockSize*)

void **riscv_fir_init_f32** (riscv_fir_instance_f32 **S*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

riscv_status **riscv_fir_init_q15** (riscv_fir_instance_q15 **S*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

void **riscv_fir_init_q31** (riscv_fir_instance_q31 **S*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

void **riscv_fir_init_q7** (riscv_fir_instance_q7 **S*, uint16_t *numTaps*, **const** q7_t **pCoeffs*, q7_t **pState*, uint32_t *blockSize*)

void **riscv_fir_q15** (**const** riscv_fir_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_q31** (**const** riscv_fir_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_q7** (**const** riscv_fir_instance_q7 **S*, **const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

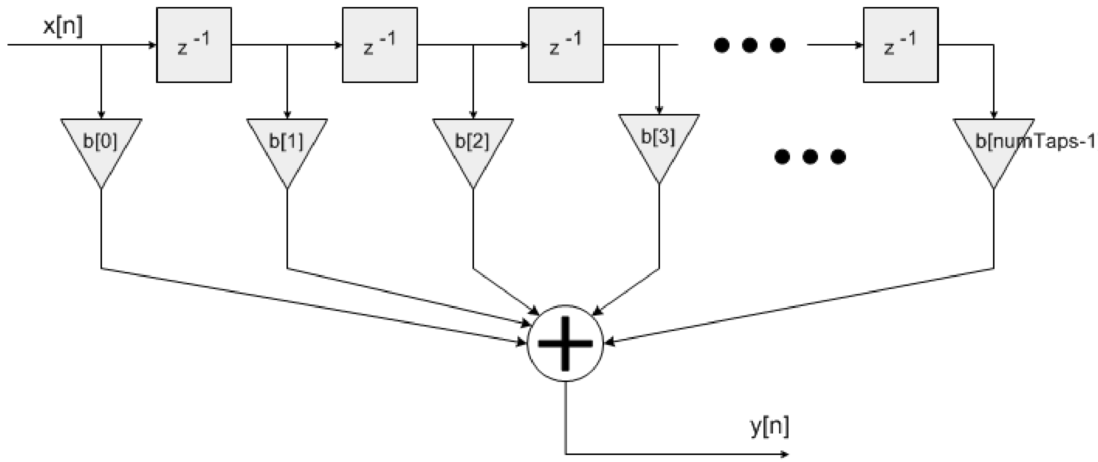
group **FIR**

This set of functions implements Finite Impulse Response (FIR) filters for Q7, Q15, Q31, and floating-point data types. Fast versions of Q15 and Q31 are also provided. The functions operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` and `pDst` points to input and output arrays containing `blockSize` values.

The array length L must be a multiple of x. L = x * a :

- x is 4 for f32
- x is 4 for q31
- x is 4 for f16 (so managed like the f32 version and not like the q15 one)
- x is 8 for q15
- x is 16 for q7

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



`pCoeffs` points to a coefficient array of size `numTaps`. Coefficients are stored in time reversed order.

`pState` points to a state array of size `numTaps + blockSize - 1`. Samples in the state buffer are stored in the following order.

Note that the length of the state buffer exceeds the length of the coefficient array by `blockSize-1`. The increased state buffer length allows circular addressing, which is traditionally used in the FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 4 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 4 different data type filter instance structures where `numTaps` is the number of filter coefficients in the filter; `pState` is the address of the state buffer; `pCoeffs` is the address of the coefficient buffer.

Initialization of Helium version For Helium version the array of coefficients must be padded with zero to contain a full number of lanes.

The additional coefficients (x * a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Helium state buffer The state buffer must contain some additional temporary data used during the computation but which is not the state of the FIR. The first A samples are temporary data. The remaining samples are the state of the FIR filter.

So the state buffer has size `numTaps + A + blockSize - 1 :`

- A is blockSize for f32
- A is 8*ceil(blockSize/8) for f16
- A is 8*ceil(blockSize/4) for q31
- A is 0 for other datatypes (q15 and q7)

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the FIR filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_fir_f16**(**const** riscv_fir_instance_f16 **S*, **const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Processing function for floating-point FIR filter.

Processing function for the floating-point FIR filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_f32**(**const** riscv_fir_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for floating-point FIR filter.

Processing function for the floating-point FIR filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_fast_q15**(**const** riscv_fir_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 FIR filter (fast version).

Processing function for the fast Q15 FIR filter (fast version).

Return none

Scaling and Overflow Behavior This fast version uses a 32-bit accumulator with 2.30 format. The accumulator maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, 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 log2(numTaps) bits. The 2.30 accumulator is then truncated to 2.15 format and saturated to yield the 1.15 result.

Remark Refer to riscv_fir_q15() for a slower implementation of this function which uses 64-bit accumulation to avoid wrap around distortion. Both the slow and the fast versions use the same instance structure. Use function riscv_fir_init_q15() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q15 FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

`IAR_ONLY_LOW_OPTIMIZATION_ENTER void riscv_fir_fast_q31(const riscv_fir_instance_q31`

Processing function for the Q31 FIR filter (fast version).

Processing function for the fast Q31 FIR filter (fast version).

Return none

Scaling and Overflow Behavior This function is optimized for speed at the expense of fixed-point precision and overflow protection. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are added to a 2.30 accumulator. Finally, the accumulator is saturated and converted to a 1.31 result. The fast version has the same overflow behavior as the standard version and provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits.

Remark Refer to riscv_fir_q31() for a slower implementation of this function which uses a 64-bit accumulator to provide higher precision. Both the slow and the fast versions use the same instance structure. Use function riscv_fir_init_q31() to initialize the filter structure.

Parameters

- `[in] S`: points to an instance of the Q31 structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **`riscv_fir_init_f16`**(riscv_fir_instance_f16 **S*, uint16_t *numTaps*, **const** float16_t **pCoeffs*, float16_t **pState*, uint32_t *blockSize*)

Initialization function for the floating-point FIR filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` samples (except for Helium - see below), where `blockSize` is the number of input samples processed by each call to `riscv_fir_f16()`.

Initialization of Helium version For Helium version the array of coefficients must be a multiple of 4 (4a) even if less then 4a coefficients are defined in the FIR. The additional coefficients (4a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Helium state buffer The state buffer must contain some additional temporary data used during the computation but which is not the state of the FIR. The first 8*ceil(blockSize/8) samples are temporary data. The remaining samples are the state of the FIR filter. So the state buffer has size `numTaps + 8*ceil(blockSize/8) + blockSize - 1`

Parameters

- `[inout] S`: points to an instance of the floating-point FIR filter structure
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficients buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of samples processed per call

void **riscv_fir_init_f32**(riscv_fir_instance_f32 **S*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

Initialization function for the floating-point FIR filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables and some working memory for the Helium version. `pState` is of length `numTaps+blockSize-1` samples (except for Helium - see below), where `blockSize` is the number of input samples processed by each call to `riscv_fir_f32()`.

Initialization of Helium version For Helium version the array of coefficients must be a multiple of 4 (4a) even if less then 4a coefficients are defined in the FIR. The additional coefficients (4a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Helium state buffer The state buffer must contain some additional temporary data used during the computation but which is not the state of the FIR. The first blockSize samples are temporary data. The remaining samples are the state of the FIR filter. So the state buffer has size `numTaps + 2 * blockSize - 1`

Parameters

- `[inout] S`: points to an instance of the floating-point FIR filter structure
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficients buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of samples processed per call

riscv_status **riscv_fir_init_q15**(riscv_fir_instance_q15 **S*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

Initialization function for the Q15 FIR filter.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `numTaps` is not greater than or equal to 4 and even

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: Note that `numTaps` must be even and greater than or equal to 4. To implement an odd length filter simply increase `numTaps` by 1 and set the last coefficient to zero. For example, to implement a filter with `numTaps=3` and coefficients set `numTaps=4` and use the coefficients: Similarly, to implement a two point filter set `numTaps=4` and use the coefficients: `pState` points to the array of state variables. `pState` is of length `numTaps+blockSize`, when running on RISC-V Core with DSP enabled and is of length `numTaps+blockSize-1`, when running on RISC-V Core without DSP where `blockSize` is the number of input samples processed by each call to `riscv_fir_q15()`.

Initialization of Helium version For Helium version the array of coefficients must be a multiple of 8 (8a) even if less then 8a coefficients are defined in the FIR. The additional coefficients (8a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Parameters

- `[inout] S`: points to an instance of the Q15 FIR filter structure.
- `[in] numTaps`: number of filter coefficients in the filter. Must be even and greater than or equal to 4.
- `[in] pCoeffs`: points to the filter coefficients buffer.
- `[in] pState`: points to the state buffer.
- `[in] blockSize`: number of samples processed per call.

void **riscv_fir_init_q31**(riscv_fir_instance_q31 **S*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

Initialization function for the Q31 FIR filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: `pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` samples (except for Helium - see below), where `blockSize` is the number of input samples processed by each call to `riscv_fir_q31()`.

Initialization of Helium version For Helium version the array of coefficients must be a multiple of 4 (4a) even if less then 4a coefficients are defined in the FIR. The additional coefficients (4a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Helium state buffer The state buffer must contain some additional temporary data used during the computation but which is not the state of the FIR. The first 2*4*ceil(blockSize/4) samples are temporary data. The remaining samples are the state of the FIR filter. So the state buffer has size `numTaps + 8*ceil(blockSize/4) + blockSize - 1`

Parameters

- `[inout] S`: points to an instance of the Q31 FIR filter structure
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficients buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of samples processed

void **riscv_fir_init_q7**(riscv_fir_instance_q7 **S*, uint16_t *numTaps*, **const** q7_t **pCoeffs*, q7_t **pState*, uint32_t *blockSize*)

Initialization function for the Q7 FIR filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

`pState` points to the array of state variables. `pState` is of length `numTaps+blockSize-1` samples, where `blockSize` is the number of input samples processed by each call to `riscv_fir_q7()`.

Initialization of Helium version For Helium version the array of coefficients must be a multiple of 16 (16a) even if less then 16a coefficients are defined in the FIR. The additional coefficients (16a - numTaps) must be set to 0. numTaps is still set to its right value in the init function. It means that the implementation may require to read more coefficients due to the vectorization and to avoid having to manage too many different cases in the code.

Parameters

- `[inout] S`: points to an instance of the Q7 FIR filter structure
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficients buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of samples processed

void **riscv_fir_q15**(**const** riscv_fir_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Remark Refer to riscv_fir_fast_q15() for a faster but less precise implementation of this function.

Parameters

- `[in] S`: points to an instance of the Q15 FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_q31**(**const** riscv_fir_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for Q31 FIR filter.

Processing function for the Q31 FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. After all multiply-accumulates are performed, the 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

Remark Refer to riscv_fir_fast_q31() for a faster but less precise implementation of this filter.

Parameters

- `[in] S`: points to an instance of the Q31 FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_q7** (**const** riscv_fir_instance_q7 **S*, **const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

Processing function for Q7 FIR filter.

Processing function for the Q7 FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both coefficients and state variables are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is converted to 18.7 format by discarding the low 7 bits. Finally, the result is truncated to 1.7 format.

Parameters

- `[in] S`: points to an instance of the Q7 FIR filter structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

Finite Impulse Response (FIR) Lattice Filters

void **riscv_fir_lattice_f32** (**const** riscv_fir_lattice_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_lattice_init_f32** (riscv_fir_lattice_instance_f32 **S*, uint16_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

void **riscv_fir_lattice_init_q15** (riscv_fir_lattice_instance_q15 **S*, uint16_t *numStages*, **const** q15_t **pCoeffs*, q15_t **pState*)

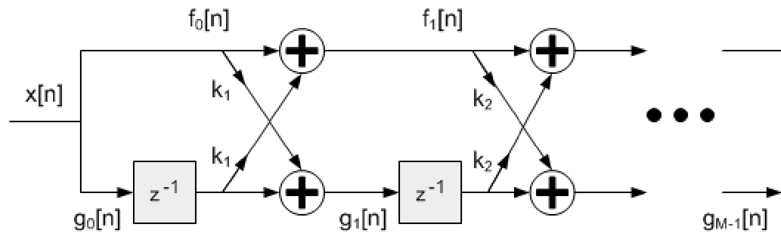
void **riscv_fir_lattice_init_q31** (riscv_fir_lattice_instance_q31 **S*, uint16_t *numStages*, **const** q31_t **pCoeffs*, q31_t **pState*)

void **riscv_fir_lattice_q15** (**const** riscv_fir_lattice_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_lattice_q31** (**const** riscv_fir_lattice_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*) *group* **FIR_Lattice**

This set of functions implements Finite Impulse Response (FIR) lattice filters for Q15, Q31 and floating-point data types. Lattice filters are used in a variety of adaptive filter applications. The filter structure is feedforward and the net impulse response is finite length. The functions operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` and `pDst` point to input and output arrays containing `blockSize` values.

Algorithm



The following difference equation is implemented:

`pCoeffs` points to tha array of reflection coefficients of size `numStages`. Reflection Coefficients are stored in the following order.

where M is number of stages

`pState` points to a state array of size `numStages`. The state variables (g values) hold previous inputs and are stored in the following order. The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros and then manually initialize the instance structure as follows:

where `numStages` is the number of stages in the filter; `pState` is the address of the state buffer; `pCoeffs` is the address of the coefficient buffer.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the FIR Lattice filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_fir_lattice_f32**(**const** riscv_fir_lattice_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for the floating-point FIR lattice filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR lattice structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_lattice_init_f32**(riscv_fir_lattice_instance_f32 **S*, uint16_t *numStages*, **const** float32_t **pCoeffs*, float32_t **pState*)

Initialization function for the floating-point FIR lattice filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR lattice structure
- `[in] numStages`: number of filter stages
- `[in] pCoeffs`: points to the coefficient buffer. The array is of length numStages
- `[in] pState`: points to the state buffer. The array is of length numStages

void **riscv_fir_lattice_init_q15**(riscv_fir_lattice_instance_q15 **S*, uint16_t *numStages*, **const** q15_t **pCoeffs*, q15_t **pState*)

Initialization function for the Q15 FIR lattice filter.

Return none

Parameters

- `[in] S`: points to an instance of the Q15 FIR lattice structure
- `[in] numStages`: number of filter stages
- `[in] pCoeffs`: points to the coefficient buffer. The array is of length numStages
- `[in] pState`: points to the state buffer. The array is of length numStages

void **riscv_fir_lattice_init_q31**(riscv_fir_lattice_instance_q31 **S*, uint16_t *numStages*, **const** q31_t **pCoeffs*, q31_t **pState*)

Initialization function for the Q31 FIR lattice filter.

Return none

Parameters

- `[in] S`: points to an instance of the Q31 FIR lattice structure
- `[in] numStages`: number of filter stages
- `[in] pCoeffs`: points to the coefficient buffer. The array is of length numStages
- `[in] pState`: points to the state buffer. The array is of length numStages

void **riscv_fir_lattice_q15** (**const** riscv_fir_lattice_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for Q15 FIR lattice filter.

Processing function for the Q15 FIR lattice filter.

Return none

Parameters

- `[in] S`: points to an instance of the Q15 FIR lattice structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_lattice_q31** (**const** riscv_fir_lattice_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 FIR lattice filter.

Return none

Scaling and Overflow Behavior In order to avoid overflows the input signal must be scaled down by 2*log2(numStages) bits.

Parameters

- `[in] S`: points to an instance of the Q31 FIR lattice structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

Finite Impulse Response (FIR) Sparse Filters

void **riscv_fir_sparse_f32** (riscv_fir_sparse_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, float32_t **pScratchIn*, uint32_t *blockSize*)

void **riscv_fir_sparse_init_f32** (riscv_fir_sparse_instance_f32 **S*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

void **riscv_fir_sparse_init_q15** (riscv_fir_sparse_instance_q15 **S*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

void **riscv_fir_sparse_init_q31** (riscv_fir_sparse_instance_q31 **S*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

void **riscv_fir_sparse_init_q7** (riscv_fir_sparse_instance_q7 **S*, uint16_t *numTaps*, **const** q7_t **pCoeffs*, q7_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

void **riscv_fir_sparse_q15** (riscv_fir_sparse_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, q15_t **pScratchIn*, q31_t **pScratchOut*, uint32_t *blockSize*)

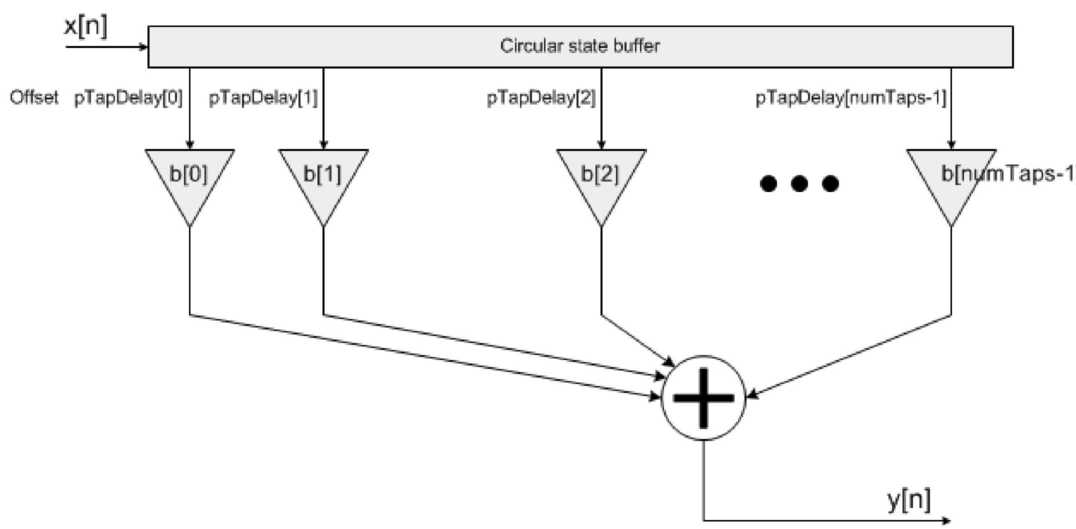
void **riscv_fir_sparse_q31** (riscv_fir_sparse_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, q31_t **pScratchIn*, uint32_t *blockSize*) void **riscv_fir_sparse_q7** (riscv_fir_sparse_instance_q7 **S*, **const** q7_t **pSrc*, q7_t **pDst*, q7_t **pScratchIn*, q31_t **pScratchOut*, uint32_t *blockSize*)

group **FIR_Sparse**

This group of functions implements sparse FIR filters. Sparse FIR filters are equivalent to standard FIR filters except that most of the coefficients are equal to zero. Sparse filters are used for simulating reflections in communications and audio applications.

There are separate functions for Q7, Q15, Q31, and floating-point data types. The functions operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` and `pDst` points to input and output arrays respectively containing `blockSize` values.

Algorithm The sparse filter instant structure contains an array of tap indices `pTapDelay` which specifies the locations of the non-zero coefficients. This is in addition to the coefficient array `b`. The implementation essentially skips the multiplications by zero and leads to an efficient realization.



`pCoeffs` points to a coefficient array of size `numTaps`; `pTapDelay` points to an array of nonzero indices and is also of size `numTaps`; `pState` points to a state array of size `maxDelay + blockSize`, where `maxDelay` is the largest offset value that is ever used in the `pTapDelay` array. Some of the processing functions also require temporary working buffers.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient and offset arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 4 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, pTapDelay, maxDelay, stateIndex, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 4 different data type filter instance structures

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the sparse FIR filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_fir_sparse_f32** (riscv_fir_sparse_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, float32_t **pScratchIn*, uint32_t *blockSize*)

Processing function for the floating-point sparse FIR filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point sparse FIR structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] pScratchIn`: points to a temporary buffer of size blockSize
- `[in] blockSize`: number of input samples to process

void **riscv_fir_sparse_init_f32** (riscv_fir_sparse_instance_f32 **S*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

Initialization function for the floating-point sparse FIR filter.

Return none

Details `pCoeffs` holds the filter coefficients and has length `numTaps`. `pState` holds the filter's state variables and must be of length `maxDelay + blockSize`, where `maxDelay` is the maximum number of delay line values. `blockSize` is the number of samples processed by the `riscv_fir_sparse_f32()` function.

Parameters

- `[inout] S`: points to an instance of the floating-point sparse FIR structure
- `[in] numTaps`: number of nonzero coefficients in the filter
- `[in] pCoeffs`: points to the array of filter coefficients
- `[in] pState`: points to the state buffer
- `[in] pTapDelay`: points to the array of offset times
- `[in] maxDelay`: maximum offset time supported
- `[in] blockSize`: number of samples that will be processed per block

void **riscv_fir_sparse_init_q15** (riscv_fir_sparse_instance_q15 **S*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

Initialization function for the Q15 sparse FIR filter.

Return none

Details `pCoeffs` holds the filter coefficients and has length `numTaps`. `pState` holds the filter's state variables and must be of length `maxDelay + blockSize`, where `maxDelay` is the maximum number of delay line values. `blockSize` is the number of words processed by `riscv_fir_sparse_q15()` function.

Parameters

- `[inout] S`: points to an instance of the Q15 sparse FIR structure
- `[in] numTaps`: number of nonzero coefficients in the filter
- `[in] pCoeffs`: points to the array of filter coefficients
- `[in] pState`: points to the state buffer
- `[in] pTapDelay`: points to the array of offset times
- `[in] maxDelay`: maximum offset time supported
- `[in] blockSize`: number of samples that will be processed per block

void **riscv_fir_sparse_init_q31**(riscv_fir_sparse_instance_q31 **S*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

Initialization function for the Q31 sparse FIR filter.

Return none

Details `pCoeffs` holds the filter coefficients and has length `numTaps`. `pState` holds the filter's state variables and must be of length `maxDelay + blockSize`, where `maxDelay` is the maximum number of delay line values. `blockSize` is the number of words processed by `riscv_fir_sparse_q31()` function.

Parameters

- `[inout] S`: points to an instance of the Q31 sparse FIR structure
- `[in] numTaps`: number of nonzero coefficients in the filter
- `[in] pCoeffs`: points to the array of filter coefficients
- `[in] pState`: points to the state buffer
- `[in] pTapDelay`: points to the array of offset times
- `[in] maxDelay`: maximum offset time supported
- `[in] blockSize`: number of samples that will be processed per block

void **riscv_fir_sparse_init_q7**(riscv_fir_sparse_instance_q7 **S*, uint16_t *numTaps*, **const** q7_t **pCoeffs*, q7_t **pState*, int32_t **pTapDelay*, uint16_t *maxDelay*, uint32_t *blockSize*)

Initialization function for the Q7 sparse FIR filter.

Return none

Details `pCoeffs` holds the filter coefficients and has length `numTaps`. `pState` holds the filter's state variables and must be of length `maxDelay + blockSize`, where `maxDelay` is the maximum number of delay line values. `blockSize` is the number of samples processed by the `riscv_fir_sparse_q7()` function.

Parameters

- `[inout] S`: points to an instance of the Q7 sparse FIR structure

- `[in] numTaps`: number of nonzero coefficients in the filter
- `[in] pCoeffs`: points to the array of filter coefficients
- `[in] pState`: points to the state buffer
- `[in] pTapDelay`: points to the array of offset times
- `[in] maxDelay`: maximum offset time supported
- `[in] blockSize`: number of samples that will be processed per block

void **riscv_fir_sparse_q15** (riscv_fir_sparse_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, q15_t **pScratchIn*, q31_t **pScratchOut*, uint32_t *blockSize*)

Processing function for the Q15 sparse FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The 1.15 x 1.15 multiplications yield a 2.30 result and these are added to a 2.30 accumulator. Thus the full precision of the multiplications is maintained but there is only a single guard bit in the accumulator. If the accumulator result overflows it will wrap around rather than saturate. After all multiply-accumulates are performed, the 2.30 accumulator is truncated to 2.15 format and then saturated to 1.15 format. In order to avoid overflows the input signal or coefficients must be scaled down by log2(numTaps) bits.

Parameters

- `[in] S`: points to an instance of the Q15 sparse FIR structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] pScratchIn`: points to a temporary buffer of size blockSize
- `[in] pScratchOut`: points to a temporary buffer of size blockSize
- `[in] blockSize`: number of input samples to process per call

void **riscv_fir_sparse_q31** (riscv_fir_sparse_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, q31_t **pScratchIn*, uint32_t *blockSize*)

Processing function for the Q31 sparse FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 32-bit accumulator. The 1.31 x 1.31 multiplications are truncated to 2.30 format. This leads to loss of precision on the intermediate multiplications and provides only a single guard bit. If the accumulator result overflows, it wraps around rather than saturate. In order to avoid overflows the input signal or coefficients must be scaled down by log2(numTaps) bits.

Parameters

- `[in] S`: points to an instance of the Q31 sparse FIR structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] pScratchIn`: points to a temporary buffer of size blockSize
- `[in] blockSize`: number of input samples to process

void **riscv_fir_sparse_q7** (riscv_fir_sparse_instance_q7 **S*, **const** q7_t **pSrc*, q7_t **pDst*, q7_t **pScratchIn*, q31_t **pScratchOut*, uint32_t *blockSize*)

Processing function for the Q7 sparse FIR filter.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. Both coefficients and state variables are represented in 1.7 format and multiplications yield a 2.14 result. The 2.14 intermediate results are accumulated in a 32-bit accumulator in 18.14 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. The accumulator is then converted to 18.7 format by discarding the low 7 bits. Finally, the result is truncated to 1.7 format.

Parameters

- `[in] S`: points to an instance of the Q7 sparse FIR structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] pScratchIn`: points to a temporary buffer of size blockSize
- `[in] pScratchOut`: points to a temporary buffer of size blockSize
- `[in] blockSize`: number of input samples to process

Infinite Impulse Response (IIR) Lattice Filters

void **riscv_iir_lattice_f32** (**const** riscv_iir_lattice_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_iir_lattice_init_f32** (riscv_iir_lattice_instance_f32 **S*, uint16_t *numStages*, float32_t **pkCoeffs*, float32_t **pvCoeffs*, float32_t **pState*, uint32_t *blockSize*)

void **riscv_iir_lattice_init_q15** (riscv_iir_lattice_instance_q15 **S*, uint16_t *numStages*, q15_t **pkCoeffs*, q15_t **pvCoeffs*, q15_t **pState*, uint32_t *blockSize*)

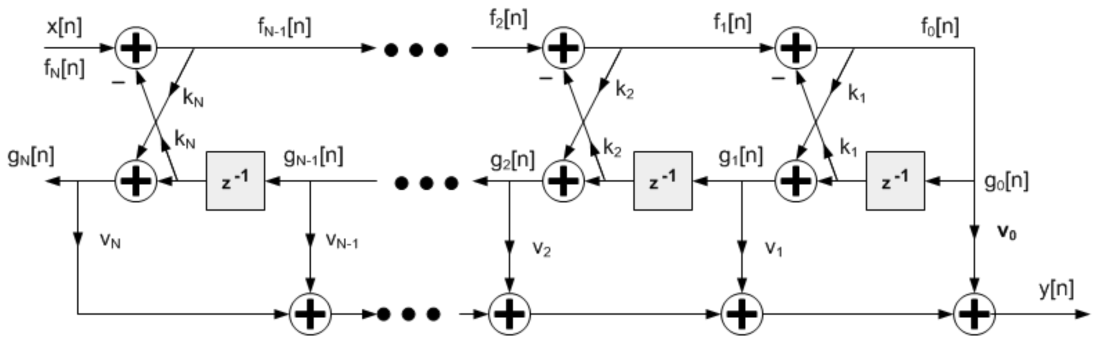
void **riscv_iir_lattice_init_q31** (riscv_iir_lattice_instance_q31 **S*, uint16_t *numStages*, q31_t **pkCoeffs*, q31_t **pvCoeffs*, q31_t **pState*, uint32_t *blockSize*)

void **riscv_iir_lattice_q15** (**const** riscv_iir_lattice_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_iir_lattice_q31** (**const** riscv_iir_lattice_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **IIR_Lattice**

This set of functions implements lattice filters for Q15, Q31 and floating-point data types. Lattice filters are used in a variety of adaptive filter applications. The filter structure has feedforward and feedback components and the net impulse response is infinite length. The functions operate on blocks of input and output data and each call to the function processes `blockSize` samples through the filter. `pSrc` and `pDst` point to input and output arrays containing `blockSize` values.



Algorithm

`pkCoeffs` points to array of reflection coefficients of size `numStages`. Reflection Coefficients are stored in time-reversed order.

`pvCoeffs` points to the array of ladder coefficients of size `(numStages+1)`. Ladder coefficients are stored in time-reversed order.

`pState` points to a state array of size `numStages + blockSize`. The state variables shown in the figure above (the g values) are stored in the `pState` array. The state variables are updated after each block of data is processed; the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable arrays cannot be shared. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numStages, pkCoeffs, pvCoeffs, pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros and then manually initialize the instance structure as follows:

where `numStages` is the number of stages in the filter; `pState` points to the state buffer array; `pkCoeffs` points to array of the reflection coefficients; `pvCoeffs` points to the array of ladder coefficients.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the IIR lattice filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_iir_lattice_f32**(**const** riscv_iir_lattice_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for the floating-point IIR lattice filter.

Return none

Parameters

- [in] S: points to an instance of the floating-point IIR lattice structure
- [in] pSrc: points to the block of input data
- [out] pDst: points to the block of output data
- [in] blockSize: number of samples to process

void **riscv_iir_lattice_init_f32**(riscv_iir_lattice_instance_f32 **S*, uint16_t *numStages*, float32_t **pkCoeffs*, float32_t **pvCoeffs*, float32_t **pState*, uint32_t *blockSize*)

Initialization function for the floating-point IIR lattice filter.

Return none

Parameters

- [in] S: points to an instance of the floating-point IIR lattice structure
- [in] numStages: number of stages in the filter
- [in] pkCoeffs: points to reflection coefficient buffer. The array is of length numStages
- [in] pvCoeffs: points to ladder coefficient buffer. The array is of length numStages+1
- [in] pState: points to state buffer. The array is of length numStages+blockSize
- [in] blockSize: number of samples to process

void **riscv_iir_lattice_init_q15**(riscv_iir_lattice_instance_q15 **S*, uint16_t *numStages*, q15_t **pkCoeffs*, q15_t **pvCoeffs*, q15_t **pState*, uint32_t *blockSize*)

Initialization function for the Q15 IIR lattice filter.

Return none

Parameters

- [in] S: points to an instance of the Q15 IIR lattice structure
- [in] numStages: number of stages in the filter
- [in] pkCoeffs: points to reflection coefficient buffer. The array is of length numStages
- [in] pvCoeffs: points to ladder coefficient buffer. The array is of length numStages+1
- [in] pState: points to state buffer. The array is of length numStages+blockSize
- [in] blockSize: number of samples to process

void **riscv_iir_lattice_init_q31**(riscv_iir_lattice_instance_q31 **S*, uint16_t *numStages*, q31_t **pkCoeffs*, q31_t **pvCoeffs*, q31_t **pState*, uint32_t *blockSize*)

Initialization function for the Q31 IIR lattice filter.

Return none

Parameters

- [in] S: points to an instance of the Q31 IIR lattice structure
- [in] numStages: number of stages in the filter
- [in] pkCoeffs: points to reflection coefficient buffer. The array is of length numStages

- `[in] pvCoeffs`: points to ladder coefficient buffer. The array is of length numStages+1
- `[in] pState`: points to state buffer. The array is of length numStages+blockSize
- `[in] blockSize`: number of samples to process

void **riscv_iir_lattice_q15**(**const** riscv_iir_lattice_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 IIR lattice filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Parameters

- `[in] S`: points to an instance of the Q15 IIR lattice structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_iir_lattice_q31**(**const** riscv_iir_lattice_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 IIR lattice filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by 2*log2(numStages) bits. After all multiply-accumulates are performed, the 2.62 accumulator is saturated to 1.32 format and then truncated to 1.31 format.

Parameters

- `[in] S`: points to an instance of the Q31 IIR lattice structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

Levinson Durbin Algorithm

void **riscv_levinson_durbin_f16**(**const** float16_t **phi*, float16_t **a*, float16_t **err*, int *nbCoefs*)

void **riscv_levinson_durbin_f32**(**const** float32_t **phi*, float32_t **a*, float32_t **err*, int *nbCoefs*)

void **riscv_levinson_durbin_q31**(**const** q31_t **phi*, q31_t **a*, q31_t **err*, int *nbCoefs*)

group **LD**

Functions

void **riscv_levinson_durbin_f16** (**const** float16_t **phi*, float16_t **a*, float16_t **err*, int *nbCoefs*)

Levinson Durbin.

Return none

Parameters

- `[in] phi`: autocovariance vector starting with lag 0 (length is nbCoefs + 1)
- `[out] a`: autoregressive coefficients
- `[out] err`: prediction error (variance)
- `[in] nbCoefs`: number of autoregressive coefficients

void **riscv_levinson_durbin_f32** (**const** float32_t **phi*, float32_t **a*, float32_t **err*, int *nbCoefs*)

Levinson Durbin.

Return none

Parameters

- `[in] phi`: autocovariance vector starting with lag 0 (length is nbCoefs + 1)
- `[out] a`: autoregressive coefficients
- `[out] err`: prediction error (variance)
- `[in] nbCoefs`: number of autoregressive coefficients

void **riscv_levinson_durbin_q31** (**const** q31_t **phi*, q31_t **a*, q31_t **err*, int *nbCoefs*)

Levinson Durbin.

Return none

Parameters

- `[in] phi`: autocovariance vector starting with lag 0 (length is nbCoefs + 1)
- `[out] a`: autoregressive coefficients
- `[out] err`: prediction error (variance)
- `[in] nbCoefs`: number of autoregressive coefficients

Least Mean Square (LMS) Filters

void **riscv_lms_f32** (**const** riscv_lms_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pRef*, float32_t **pOut*, float32_t **pErr*, uint32_t *blockSize*)

void **riscv_lms_init_f32** (riscv_lms_instance_f32 **S*, uint16_t *numTaps*, float32_t **pCoeffs*, float32_t **pState*, float32_t *mu*, uint32_t *blockSize*)

void **riscv_lms_init_q15** (riscv_lms_instance_q15 **S*, uint16_t *numTaps*, q15_t **pCoeffs*, q15_t **pState*, q15_t *mu*, uint32_t *blockSize*, uint32_t *postShift*)

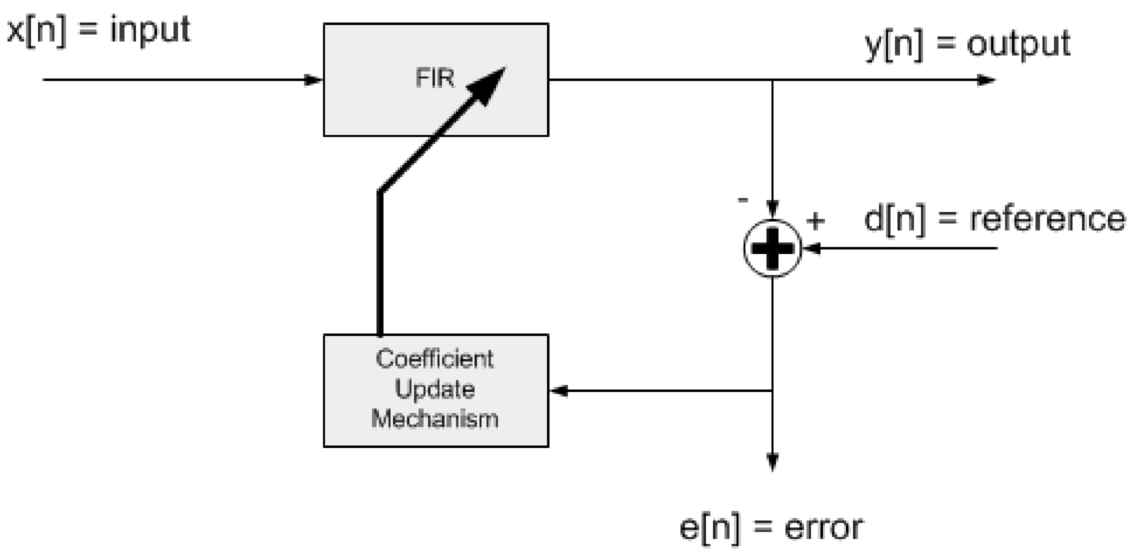
void **riscv_lms_init_q31** (riscv_lms_instance_q31 **S*, uint16_t *numTaps*, q31_t **pCoeffs*, q31_t **pState*, q31_t *mu*, uint32_t *blockSize*, uint32_t *postShift*) void **riscv_lms_q15** (**const** riscv_lms_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pRef*, q15_t **pOut*, q15_t **pErr*, uint32_t *blockSize*)

void **riscv_lms_q31** (**const** riscv_lms_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pRef*, q31_t **pOut*, q31_t **pErr*, uint32_t *blockSize*)

group **LMS**

LMS filters are a class of adaptive filters that are able to "learn" an unknown transfer functions. LMS filters use a gradient descent method in which the filter coefficients are updated based on the instantaneous error signal. Adaptive filters are often used in communication systems, equalizers, and noise removal. The NMSIS DSP Library contains LMS filter functions that operate on Q15, Q31, and floating-point data types. The library also contains normalized LMS filters in which the filter coefficient adaptation is indepedent of the level of the input signal.

An LMS filter consists of two components as shown below. The first component is a standard transversal or FIR filter. The second component is a coefficient update mechanism. The LMS filter has two input signals. The "input" feeds the FIR filter while the "reference input" corresponds to the desired output of the FIR filter. That is, the FIR filter coefficients are updated so that the output of the FIR filter matches the reference input. The filter coefficient update mechanism is based on the difference between the FIR filter output and the reference input. This "error signal" tends towards zero as the filter adapts. The LMS processing functions accept the input and reference input signals and generate the filter output and error signal. The functions operate on blocks of data and each call to the function processes `blockSize` samples through the filter. `pSrc` points to input signal, `pRef` points to reference signal, `pOut` points to output signal and `pErr` points to error signal. All arrays contain `blockSize`



values.

The functions operate on a block-by-block basis. Internally, the filter coefficients `b[n]` are updated on a sample-by-sample basis. The convergence of the LMS filter is slower compared to the normalized LMS algorithm.

Algorithm The output signal `y[n]` is computed by a standard FIR filter:

The error signal equals the difference between the reference signal `d[n]` and the filter output:

After each sample of the error signal is computed, the filter coefficients `b[k]` are updated on a sample-by-sample basis: where `mu` is the step size and controls the rate of coefficient convergence.

In the APIs, `pCoeffs` points to a coefficient array of size `numTaps`. Coefficients are stored in time reversed order.

`pState` points to a state array of size `numTaps + blockSize - 1`. Samples in the state buffer are stored in the order:

Note that the length of the state buffer exceeds the length of the coefficient array by `blockSize-1` samples. The increased state buffer length allows circular addressing, which is traditionally used in FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter and coefficient and state arrays cannot be shared among instances. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, mu, postShift (not for f32), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Set the values in the state buffer to zeros before static initialization. The code below statically initializes each of the 3 different data type filter instance structures where `numTaps` is the number of filter coefficients in the filter; `pState` is the address of the state buffer; `pCoeffs` is the address of the coefficient buffer; `mu` is the step size parameter; and `postShift` is the shift applied to coefficients.

Fixed-Point Behavior Care must be taken when using the Q15 and Q31 versions of the LMS filter. The following issues must be considered:

- Scaling of coefficients
- Overflow and saturation

Scaling of Coefficients Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range `[-1 +1)`. The fixed-point functions have an additional scaling parameter `postShift`. At the output of the filter's accumulator is a shift register which shifts the result by `postShift` bits. This essentially scales the filter coefficients by `2^postShift` and allows the filter coefficients to exceed the range `[+1 -1)`. The value of `postShift` is set by the user based on the expected gain through the system being modeled.

Overflow and Saturation Overflow and saturation behavior of the fixed-point Q15 and Q31 versions are described separately as part of the function specific documentation below.

Functions

void **riscv_lms_f32**(**const** riscv_lms_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pRef*, float32_t **pOut*, float32_t **pErr*, uint32_t *blockSize*)

Processing function for floating-point LMS filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point LMS filter structure

- [in] pSrc: points to the block of input data
- [in] pRef: points to the block of reference data
- [out] pOut: points to the block of output data
- [out] pErr: points to the block of error data
- [in] blockSize: number of samples to process

void **riscv_lms_init_f32**(riscv_lms_instance_f32 **S*, uint16_t *numTaps*, float32_t **pCoeffs*, float32_t **pState*, float32_t *mu*, uint32_t *blockSize*)

Initialization function for floating-point LMS filter.

Return none

Details pCoeffs points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to riscv_lms_f32().

Parameters

- [in] S: points to an instance of the floating-point LMS filter structure
- [in] numTaps: number of filter coefficients
- [in] pCoeffs: points to coefficient buffer
- [in] pState: points to state buffer
- [in] mu: step size that controls filter coefficient updates
- [in] blockSize: number of samples to process

void **riscv_lms_init_q15**(riscv_lms_instance_q15 **S*, uint16_t *numTaps*, q15_t **pCoeffs*, q15_t **pState*, q15_t *mu*, uint32_t *blockSize*, uint32_t *postShift*)

Initialization function for the Q15 LMS filter.

Return none

Details pCoeffs points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. pState points to the array of state variables and size of array is numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to riscv_lms_q15().

Parameters

- [in] S: points to an instance of the Q15 LMS filter structure.
- [in] numTaps: number of filter coefficients.
- [in] pCoeffs: points to coefficient buffer.
- [in] pState: points to state buffer.
- [in] mu: step size that controls filter coefficient updates.
- [in] blockSize: number of samples to process.
- [in] postShift: bit shift applied to coefficients.

void **riscv_lms_init_q31**(riscv_lms_instance_q31 **S*, uint16_t *numTaps*, q31_t **pCoeffs*, q31_t **pState*, q31_t *mu*, uint32_t *blockSize*, uint32_t *postShift*)

Initialization function for Q31 LMS filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. `pState` points to an array of length `numTaps+blockSize-1` samples, where `blockSize` is the number of input samples processed by each call to `riscv_lms_q31()`.

Parameters

- `[in] S`: points to an instance of the Q31 LMS filter structure
- `[in] numTaps`: number of filter coefficients
- `[in] pCoeffs`: points to coefficient buffer
- `[in] pState`: points to state buffer
- `[in] mu`: step size that controls filter coefficient updates
- `[in] blockSize`: number of samples to process
- `[in] postShift`: bit shift applied to coefficients

void **riscv_lms_q15**(**const** riscv_lms_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pRef*, q15_t **pOut*, q15_t **pErr*, uint32_t *blockSize*)

Processing function for Q15 LMS filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

Parameters

- `[in] S`: points to an instance of the Q15 LMS filter structure
- `[in] pSrc`: points to the block of input data
- `[in] pRef`: points to the block of reference data
- `[out] pOut`: points to the block of output data
- `[out] pErr`: points to the block of error data
- `[in] blockSize`: number of samples to process

void **riscv_lms_q31**(**const** riscv_lms_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pRef*, q31_t **pOut*, q31_t **pErr*, uint32_t *blockSize*)

Processing function for Q31 LMS filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clips. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The reference signal should not be scaled down. After all multiply-accumulates are performed, the 2.62 accumulator is shifted and saturated to 1.31 format to yield the final result. The output signal and error signal are in 1.31 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

Parameters

- `[in] S`: points to an instance of the Q31 LMS filter structure.
- `[in] pSrc`: points to the block of input data.
- `[in] pRef`: points to the block of reference data.
- `[out] pOut`: points to the block of output data.
- `[out] pErr`: points to the block of error data.
- `[in] blockSize`: number of samples to process.

Normalized LMS Filters

void **riscv_lms_norm_f32** (riscv_lms_norm_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pRef*, float32_t **pOut*, float32_t **pErr*, uint32_t *blockSize*)

void **riscv_lms_norm_init_f32** (riscv_lms_norm_instance_f32 **S*, uint16_t *numTaps*, float32_t **pCoeffs*, float32_t **pState*, float32_t *mu*, uint32_t *blockSize*)

void **riscv_lms_norm_init_q15** (riscv_lms_norm_instance_q15 **S*, uint16_t *numTaps*, q15_t **pCoeffs*, q15_t **pState*, q15_t *mu*, uint32_t *blockSize*, uint8_t *postShift*)

void **riscv_lms_norm_init_q31** (riscv_lms_norm_instance_q31 **S*, uint16_t *numTaps*, q31_t **pCoeffs*, q31_t **pState*, q31_t *mu*, uint32_t *blockSize*, uint8_t *postShift*)

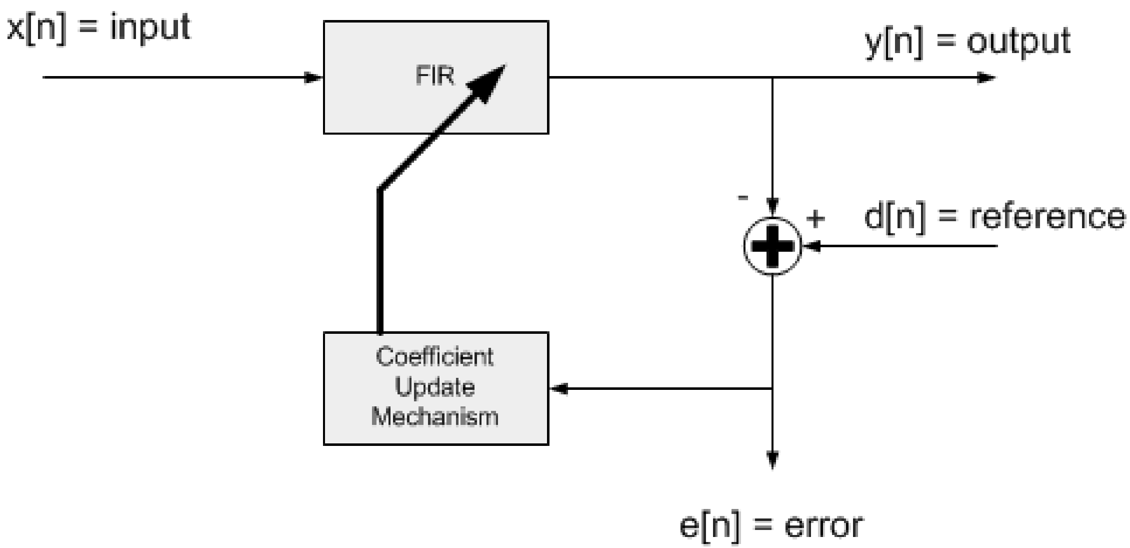
void **riscv_lms_norm_q15** (riscv_lms_norm_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pRef*, q15_t **pOut*, q15_t **pErr*, uint32_t *blockSize*)

void **riscv_lms_norm_q31** (riscv_lms_norm_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pRef*, q31_t **pOut*, q31_t **pErr*, uint32_t *blockSize*)

group **LMS_NORM**

This set of functions implements a commonly used adaptive filter. It is related to the Least Mean Square (LMS) adaptive filter and includes an additional normalization factor which increases the adaptation rate of the filter. The NMSIS DSP Library contains normalized LMS filter functions that operate on Q15, Q31, and floating-point data types.

A normalized least mean square (NLMS) filter consists of two components as shown below. The first component is a standard transversal or FIR filter. The second component is a coefficient update mechanism. The NLMS filter has two input signals. The “input” feeds the FIR filter while the “reference input” corresponds to the desired output of the FIR filter. That is, the FIR filter coefficients are updated so that the output of the FIR filter matches the reference input. The filter coefficient update mechanism is based on the difference between the FIR filter output and the reference input. This “error signal” tends towards zero as the filter adapts. The NLMS processing functions accept the input and reference input signals and generate the filter output and error signal. The functions operate on blocks of data and each call to the function processes `blockSize` samples through the filter. `pSrc` points to input signal, `pRef` points to reference signal, `pOut` points to output signal and `pErr` points to error signal. All arrays contain `blockSize`



values.

The functions operate on a block-by-block basis. Internally, the filter coefficients `b[n]` are updated on a sample-by-sample basis. The convergence of the LMS filter is slower compared to the normalized LMS algorithm.

Algorithm The output signal `y[n]` is computed by a standard FIR filter:

The error signal equals the difference between the reference signal `d[n]` and the filter output:

After each sample of the error signal is computed the instanteous energy of the filter state variables is calculated: The filter coefficients `b[k]` are then updated on a sample-by-sample basis: where `mu` is the step size and controls the rate of coefficient convergence.

In the APIs, `pCoeffs` points to a coefficient array of size `numTaps`. Coefficients are stored in time reversed order.

`pState` points to a state array of size `numTaps + blockSize - 1`. Samples in the state buffer are stored in the order:

Note that the length of the state buffer exceeds the length of the coefficient array by `blockSize-1` samples. The increased state buffer length allows circular addressing, which is traditionally used in FIR filters, to be avoided and yields a significant speed improvement. The state variables are updated after each block of data is processed.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter and coefficient and state arrays cannot be shared among instances. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer. To do this manually without calling the init function, assign the follow subfields of the instance structure: numTaps, pCoeffs, mu, energy, x0, pState. Also set all of the values in pState to zero. For Q7, Q15, and Q31 the following fields must also be initialized; recipTable, postShift

Instance structure cannot be placed into a const data section and it is recommended to use the initialization function.

Fixed-Point Behavior Care must be taken when using the Q15 and Q31 versions of the normalised LMS filter. The following issues must be considered:

- Scaling of coefficients
- Overflow and saturation

Scaling of Coefficients (fixed point versions) Filter coefficients are represented as fractional values and coefficients are restricted to lie in the range `[-1 +1)`. The fixed-point functions have an additional scaling parameter `postShift`. At the output of the filter's accumulator is a shift register which shifts the result by `postShift` bits. This essentially scales the filter coefficients by `2^postShift` and allows the filter coefficients to exceed the range `[+1 -1)`. The value of `postShift` is set by the user based on the expected gain through the system being modeled.

Overflow and Saturation (fixed point versions) Overflow and saturation behavior of the fixed-point Q15 and Q31 versions are described separately as part of the function specific documentation below.

Functions

void **riscv_lms_norm_f32**(riscv_lms_norm_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pRef*, float32_t **pOut*, float32_t **pErr*, uint32_t *blockSize*)

Processing function for floating-point normalized LMS filter.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point normalized LMS filter structure
- `[in] pSrc`: points to the block of input data
- `[in] pRef`: points to the block of reference data
- `[out] pOut`: points to the block of output data
- `[out] pErr`: points to the block of error data
- `[in] blockSize`: number of samples to process

void **riscv_lms_norm_init_f32**(riscv_lms_norm_instance_f32 **S*, uint16_t *numTaps*, float32_t **pCoeffs*, float32_t **pState*, float32_t *mu*, uint32_t *blockSize*)

Initialization function for floating-point normalized LMS filter.

Return none

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. `pState` points to an array of length `numTaps+blockSize-1` samples, where `blockSize` is the number of input samples processed by each call to `riscv_lms_norm_f32()`.

Parameters

- `[in] S`: points to an instance of the floating-point LMS filter structure
- `[in] numTaps`: number of filter coefficients
- `[in] pCoeffs`: points to coefficient buffer
- `[in] pState`: points to state buffer

- [in] mu: step size that controls filter coefficient updates
- [in] blockSize: number of samples to process

void **riscv_lms_norm_init_q15** (riscv_lms_norm_instance_q15 **S*, uint16_t *numTaps*, q15_t **pCoeffs*, q15_t **pState*, q15_t *mu*, uint32_t *blockSize*, uint8_t *postShift*)

Initialization function for Q15 normalized LMS filter.

Return none

Details pCoeffs points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. pState points to the array of state variables and size of array is numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to riscv_lms_norm_q15().

Parameters

- [in] S: points to an instance of the Q15 normalized LMS filter structure.
- [in] numTaps: number of filter coefficients.
- [in] pCoeffs: points to coefficient buffer.
- [in] pState: points to state buffer.
- [in] mu: step size that controls filter coefficient updates.
- [in] blockSize: number of samples to process.
- [in] postShift: bit shift applied to coefficients.

void **riscv_lms_norm_init_q31** (riscv_lms_norm_instance_q31 **S*, uint16_t *numTaps*, q31_t **pCoeffs*, q31_t **pState*, q31_t *mu*, uint32_t *blockSize*, uint8_t *postShift*)

Initialization function for Q31 normalized LMS filter.

Return none

Details pCoeffs points to the array of filter coefficients stored in time reversed order: The initial filter coefficients serve as a starting point for the adaptive filter. pState points to an array of length numTaps+blockSize-1 samples, where blockSize is the number of input samples processed by each call to riscv_lms_norm_q31().

Parameters

- [in] S: points to an instance of the Q31 normalized LMS filter structure.
- [in] numTaps: number of filter coefficients.
- [in] pCoeffs: points to coefficient buffer.
- [in] pState: points to state buffer.
- [in] mu: step size that controls filter coefficient updates.
- [in] blockSize: number of samples to process.
- [in] postShift: bit shift applied to coefficients.

void **riscv_lms_norm_q15** (riscv_lms_norm_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pRef*, q15_t **pOut*, q15_t **pErr*, uint32_t *blockSize*)

Processing function for Q15 normalized LMS filter.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

Parameters

- `[in] S`: points to an instance of the Q15 normalized LMS filter structure
- `[in] pSrc`: points to the block of input data
- `[in] pRef`: points to the block of reference data
- `[out] pOut`: points to the block of output data
- `[out] pErr`: points to the block of error data
- `[in] blockSize`: number of samples to process

void **riscv_lms_norm_q31**(riscv_lms_norm_instance_q31 *S*, **const** q31_t **pSrc*, q31_t **pRef*, q31_t **pOut*, q31_t **pErr*, uint32_t *blockSize*)

Processing function for Q31 normalized LMS filter.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by log2(numTaps) bits. The reference signal should not be scaled down. After all multiply-accumulates are performed, the 2.62 accumulator is shifted and saturated to 1.31 format to yield the final result. The output signal and error signal are in 1.31 format.

In this filter, filter coefficients are updated for each sample and the updation of filter cofficients are saturted.

Parameters

- `[in] S`: points to an instance of the Q31 normalized LMS filter structure
- `[in] pSrc`: points to the block of input data
- `[in] pRef`: points to the block of reference data
- `[out] pOut`: points to the block of output data
- `[out] pErr`: points to the block of error data
- `[in] blockSize`: number of samples to process

Finite Impulse Response (FIR) Interpolator

void **riscv_fir_interpolate_f32**(**const** riscv_fir_interpolate_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*) riscv_status **riscv_fir_interpolate_init_f32** (riscv_fir_interpolate_instance_f32 **S*, uint8_t *L*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

riscv_status **riscv_fir_interpolate_init_q15** (riscv_fir_interpolate_instance_q15 **S*, uint8_t *L*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

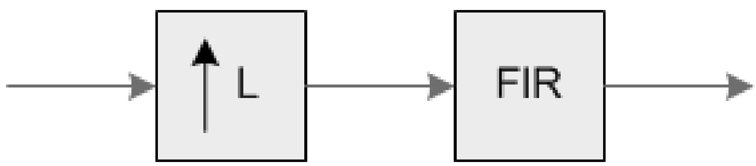
riscv_status **riscv_fir_interpolate_init_q31** (riscv_fir_interpolate_instance_q31 **S*, uint8_t *L*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

void **riscv_fir_interpolate_q15** (**const** riscv_fir_interpolate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fir_interpolate_q31** (**const** riscv_fir_interpolate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **FIR_Interpolate**

These functions combine an upsampler (zero stuffer) and an FIR filter. They are used in multirate systems for increasing the sample rate of a signal without introducing high frequency images. Conceptually, the functions are equivalent to the block diagram below: After upsampling by a factor of `L`, the signal should be filtered by a lowpass filter with a normalized cutoff frequency of `1/L` in order to eliminate high frequency copies of the spectrum. The user of the function is responsible for providing the filter



coefficients.

The FIR interpolator functions provided in the NMSIS DSP Library combine the upsampler and FIR filter in an efficient manner. The upsampler inserts `L-1` zeros between each sample. Instead of multiplying by these zero values, the FIR filter is designed to skip them. This leads to an efficient implementation without any wasted effort. The functions operate on blocks of input and output data. `pSrc` points to an array of `blockSize` input values and `pDst` points to an array of `blockSize*L` output values.

The library provides separate functions for Q15, Q31, and floating-point data types.

Algorithm The functions use a polyphase filter structure: This approach is more efficient than straightforward upsample-then-filter algorithms. With this method the computation is reduced by a factor of `1/L` when compared to using a standard FIR filter.

`pCoeffs` points to a coefficient array of size `numTaps`. `numTaps` must be a multiple of the interpolation factor `L` and this is checked by the initialization functions. Internally, the function divides the FIR filter's impulse response into shorter filters of length `phaseLength=numTaps/L`. Coefficients are stored in time reversed order.

`pState` points to a state array of size `blockSize + phaseLength - 1`. Samples in the state buffer are stored in the order:

The state variables are updated after each block of data is processed, the coefficients are untouched.

Instance Structure The coefficients and state variables for a filter are stored together in an instance data structure. A separate instance structure must be defined for each filter. Coefficient arrays may be shared among several instances while state variable array should be allocated separately. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Zeros out the values in the state buffer.
- Checks to make sure that the length of the filter is a multiple of the interpolation factor. To do this manually without calling the init function, assign the follow subfields of the instance structure: L (interpolation factor), pCoeffs, phaseLength (numTaps / L), pState. Also set all of the values in pState to zero.

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. The code below statically initializes each of the 3 different data type filter instance structures

where `L` is the interpolation factor; `phaseLength=numTaps/L` is the length of each of the shorter FIR filters used internally, `pCoeffs` is the address of the coefficient buffer; `pState` is the address of the state buffer. Be sure to set the values in the state buffer to zeros when doing static initialization.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the FIR interpolate filter functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_fir_interpolate_f32**(**const** riscv_fir_interpolate_instance_f32 **S*, **const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for floating-point FIR interpolator.

Processing function for the floating-point FIR interpolator.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point FIR interpolator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

riscv_status **riscv_fir_interpolate_init_f32**(riscv_fir_interpolate_instance_f32 **S*, uint8_t *L*, uint16_t *numTaps*, **const** float32_t **pCoeffs*, float32_t **pState*, uint32_t *blockSize*)

Initialization function for the floating-point FIR interpolator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : filter length `numTaps` is not a multiple of the interpolation factor `L`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order:

The length of the filter `numTaps` must be a multiple of the interpolation factor `L`.

`pState` points to the array of state variables. `pState` is of length `(numTaps/L)+blockSize-1` words where `blockSize` is the number of input samples processed by each call to `riscv_fir_interpolate_f32()`.

Parameters

- `[inout] S`: points to an instance of the floating-point FIR interpolator structure
- `[in] L`: upsample factor
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficient buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process per call

riscv_status **riscv_fir_interpolate_init_q15**(riscv_fir_interpolate_instance_q15 **S*, uint8_t *L*, uint16_t *numTaps*, **const** q15_t **pCoeffs*, q15_t **pState*, uint32_t *blockSize*)

Initialization function for the Q15 FIR interpolator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : filter length `numTaps` is not a multiple of the interpolation factor `L`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: The length of the filter `numTaps` must be a multiple of the interpolation factor `L`.

`pState` points to the array of state variables. `pState` is of length `(numTaps/L)+blockSize-1` words where `blockSize` is the number of input samples processed by each call to `riscv_fir_interpolate_q15()`.

Parameters

- `[inout] S`: points to an instance of the Q15 FIR interpolator structure
- `[in] L`: upsample factor
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficient buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process per call

riscv_status **riscv_fir_interpolate_init_q31**(riscv_fir_interpolate_instance_q31 **S*, uint8_t *L*, uint16_t *numTaps*, **const** q31_t **pCoeffs*, q31_t **pState*, uint32_t *blockSize*)

Initialization function for the Q31 FIR interpolator.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : filter length `numTaps` is not a multiple of the interpolation factor `L`

Details `pCoeffs` points to the array of filter coefficients stored in time reversed order: The length of the filter `numTaps` must be a multiple of the interpolation factor `L`.

`pState` points to the array of state variables. `pState` is of length `(numTaps/L)+blockSize-1` words where `blockSize` is the number of input samples processed by each call to `riscv_fir_interpolate_q31()`.

Parameters

- `[inout] S`: points to an instance of the Q31 FIR interpolator structure
- `[in] L`: upsample factor
- `[in] numTaps`: number of filter coefficients in the filter
- `[in] pCoeffs`: points to the filter coefficient buffer
- `[in] pState`: points to the state buffer
- `[in] blockSize`: number of input samples to process per call

void **riscv_fir_interpolate_q15**(**const** riscv_fir_interpolate_instance_q15 **S*, **const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Processing function for the Q15 FIR interpolator.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. Both coefficients and state variables are represented in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. There is no risk of internal overflow with this approach and the full precision of intermediate multiplications is preserved. After all additions have been performed, the accumulator is truncated to 34.15 format by discarding low 15 bits. Lastly, the accumulator is saturated to yield a result in 1.15 format.

Parameters

- `[in] S`: points to an instance of the Q15 FIR interpolator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

void **riscv_fir_interpolate_q31**(**const** riscv_fir_interpolate_instance_q31 **S*, **const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Processing function for the Q31 FIR interpolator.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. Thus, if the accumulator result overflows it wraps around rather than clip. In order to avoid overflows completely the input signal must be scaled down by `1/(numTaps/L)`. since `numTaps/L` additions occur per output sample. After all multiply-accumulates are performed, the 2.62 accumulator is truncated to 1.32 format and then saturated to 1.31 format.

Parameters

- `[in] S`: points to an instance of the Q31 FIR interpolator structure
- `[in] pSrc`: points to the block of input data
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process

group **groupFilters**

3.3.9 Interpolation Functions

Bilinear Interpolation

float32_t **riscv_bilinear_interp_f32**(**const** riscv_bilinear_interp_instance_f32 **S*, float32_t *X*, float32_t *Y*)

q31_t **riscv_bilinear_interp_q31**(riscv_bilinear_interp_instance_q31 **S*, q31_t *X*, q31_t *Y*)

q15_t **riscv_bilinear_interp_q15**(riscv_bilinear_interp_instance_q15 **S*, q31_t *X*, q31_t *Y*)

q7_t **riscv_bilinear_interp_q7**(riscv_bilinear_interp_instance_q7 **S*, q31_t *X*, q31_t *Y*)

float16_t **riscv_bilinear_interp_f16**(**const** riscv_bilinear_interp_instance_f16 **S*, float16_t *X*, float16_t *Y*)

group **BilinearInterpolate**

Bilinear interpolation is an extension of linear interpolation applied to a two dimensional grid. The underlying function `f(x, y)` is sampled on a regular grid and the interpolation process determines values between the grid points. Bilinear interpolation is equivalent to two step linear interpolation, first in the x-dimension and then in the y-dimension. Bilinear interpolation is often used in image processing to rescale images. The NMSIS DSP library provides bilinear interpolation functions for Q7, Q15, Q31, and floating-point data types.

Algorithm Bilinear interpolation is an extension of linear interpolation applied to a two dimensional grid. The underlying function `f(x, y)` is sampled on a regular grid and the interpolation process determines values between the grid points. Bilinear interpolation is equivalent to two step linear interpolation, first in the x-dimension and then in the y-dimension. Bilinear interpolation is often used in image processing to rescale images. The NMSIS DSP library provides bilinear interpolation functions for Q7, Q15, Q31, and floating-point data types.

The instance structure used by the bilinear interpolation functions describes a two dimensional data table. For floating-point, the instance structure is defined as:

where `numRows` specifies the number of rows in the table; `numCols` specifies the number of columns in the table; and `pData` points to an array of size `numRows*numCols` values. The data table `pTable` is organized in row order and the supplied data values fall on integer indexes. That is, table element (x,y) is located at `pTable[x + y*numCols]` where x and y are integers.

Let `(x, y)` specify the desired interpolation point. Then define:

The interpolated output point is computed as: Note that the coordinates (x, y) contain integer and fractional components. The integer components specify which portion of the table to use while the fractional components control the interpolation processor.

if (x,y) are outside of the table boundary, Bilinear interpolation returns zero output.

Algorithm end of LinearInterpolate group

The instance structure used by the bilinear interpolation functions describes a two dimensional data table. For floating-point, the instance structure is defined as:

where `numRows` specifies the number of rows in the table; `numCols` specifies the number of columns in the table; and `pData` points to an array of size `numRows*numCols` values. The data table `pTable` is organized in row order and the supplied data values fall on integer indexes. That is, table element (x,y) is located at `pTable[x + y*numCols]` where x and y are integers.

Let `(x, y)` specify the desired interpolation point. Then define:

The interpolated output point is computed as: Note that the coordinates (x, y) contain integer and fractional components. The integer components specify which portion of the table to use while the fractional components control the interpolation processor.

if (x,y) are outside of the table boundary, Bilinear interpolation returns zero output.

Functions

float32_t **riscv_bilinear_interp_f32**(**const** riscv_bilinear_interp_instance_f32 **S*, float32_t *X*, float32_t *Y*)

Floating-point bilinear interpolation.

Return out interpolated value.

Parameters

- `[inout] S`: points to an instance of the interpolation structure.
- `[in] X`: interpolation coordinate.
- `[in] Y`: interpolation coordinate.

q31_t **riscv_bilinear_interp_q31**(riscv_bilinear_interp_instance_q31 **S*, q31_t *X*, q31_t *Y*)

Q31 bilinear interpolation.

Return out interpolated value.

Parameters

- `[inout] S`: points to an instance of the interpolation structure.
- `[in] X`: interpolation coordinate in 12.20 format.
- `[in] Y`: interpolation coordinate in 12.20 format.

q15_t **riscv_bilinear_interp_q15**(riscv_bilinear_interp_instance_q15 **S*, q31_t *X*, q31_t *Y*)

Q15 bilinear interpolation.

Return out interpolated value.

Parameters

- `[inout] S`: points to an instance of the interpolation structure.
- `[in] X`: interpolation coordinate in 12.20 format.
- `[in] Y`: interpolation coordinate in 12.20 format.

q7_t **riscv_bilinear_interp_q7**(riscv_bilinear_interp_instance_q7 **S*, q31_t *X*, q31_t *Y*)

Q7 bilinear interpolation.

Return out interpolated value.

Parameters

- `[inout] S`: points to an instance of the interpolation structure.
- `[in] X`: interpolation coordinate in 12.20 format.
- `[in] Y`: interpolation coordinate in 12.20 format.

float16_t **riscv_bilinear_interp_f16** (**const** riscv_bilinear_interp_instance_f16 **S*, float16_t *X*, float16_t *Y*)

Floating-point bilinear interpolation.

Return out interpolated value.

Parameters

- `[inout] S`: points to an instance of the interpolation structure.
- `[in] X`: interpolation coordinate.
- `[in] Y`: interpolation coordinate.

Linear Interpolation

float32_t **riscv_linear_interp_f32** (riscv_linear_interp_instance_f32 **S*, float32_t *x*)

q31_t **riscv_linear_interp_q31** (q31_t **pYData*, q31_t *x*, uint32_t *nValues*)

q15_t **riscv_linear_interp_q15** (q15_t **pYData*, q31_t *x*, uint32_t *nValues*)

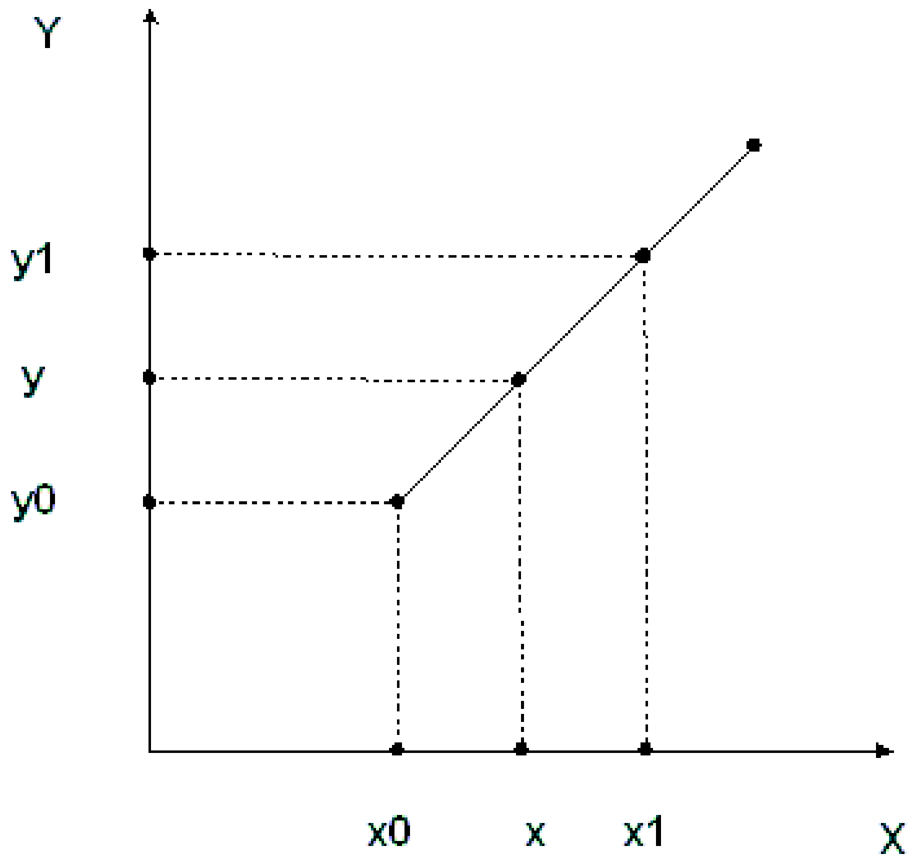
q7_t **riscv_linear_interp_q7** (q7_t **pYData*, q31_t *x*, uint32_t *nValues*)

float16_t **riscv_linear_interp_f16** (riscv_linear_interp_instance_f16 **S*, float16_t *x*)

group **LinearInterpolate**

Linear interpolation is a method of curve fitting using linear polynomials. Linear interpolation works by effectively drawing a straight line between two neighboring samples and returning the appropriate point along that line

end of SplineInterpolate group



A Linear Interpolate function calculates an output value(y), for the input(x) using linear interpolation of the input values x0, x1(nearest input values) and the output values y0 and y1(nearest output values)

Algorithm:

This set of functions implements Linear interpolation process for Q7, Q15, Q31, and floating-point data types. The functions operate on a single sample of data and each call to the function returns a single processed value. `S` points to an instance of the Linear Interpolate function data structure. `x` is the input sample value. The functions returns the output value.

if x is outside of the table boundary, Linear interpolation returns first value of the table if x is below input range and returns last value of table if x is above range.

Functions

float32_t **riscv_linear_interp_f32** (riscv_linear_interp_instance_f32 **S*, float32_t *x*)

Process function for the floating-point Linear Interpolation Function.

Return y processed output sample.

Parameters

- `[inout] S`: is an instance of the floating-point Linear Interpolation structure
- `[in] x`: input sample to process

q31_t **riscv_linear_interp_q31** (q31_t **pYData*, q31_t *x*, uint32_t *nValues*)
Process function for the Q31 Linear Interpolation Function.

Return y processed output sample.

Input sample `x` is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

Parameters

- `[in] pYData`: pointer to Q31 Linear Interpolation table
- `[in] x`: input sample to process
- `[in] nValues`: number of table values

q15_t **riscv_linear_interp_q15** (q15_t **pYData*, q31_t *x*, uint32_t *nValues*)
Process function for the Q15 Linear Interpolation Function.

Return y processed output sample.

Input sample `x` is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

Parameters

- `[in] pYData`: pointer to Q15 Linear Interpolation table
- `[in] x`: input sample to process
- `[in] nValues`: number of table values

q7_t **riscv_linear_interp_q7** (q7_t **pYData*, q31_t *x*, uint32_t *nValues*)
Process function for the Q7 Linear Interpolation Function.

Return y processed output sample.

Input sample `x` is in 12.20 format which contains 12 bits for table index and 20 bits for fractional part. This function can support maximum of table size 2^12.

Parameters

- `[in] pYData`: pointer to Q7 Linear Interpolation table
- `[in] x`: input sample to process
- `[in] nValues`: number of table values

float16_t **riscv_linear_interp_f16** (riscv_linear_interp_instance_f16 **S*, float16_t *x*)
Process function for the floating-point Linear Interpolation Function.

Return y processed output sample.

Parameters

- `[inout] S`: is an instance of the floating-point Linear Interpolation structure
- `[in] x`: input sample to process

Cubic Spline Interpolation

void **riscv_spline_f32** (riscv_spline_instance_f32 **S*, **const** float32_t **xq*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_spline_init_f32** (riscv_spline_instance_f32 **S*, riscv_spline_type *type*, **const** float32_t **x*, **const** float32_t **y*, uint32_t *n*, float32_t **coeffs*, float32_t **tempBuffer*)

group **SplineInterpolate**

Spline interpolation is a method of interpolation where the interpolant is a piecewise-defined polynomial called "spline".

Given a function f defined on the interval [a,b], a set of n nodes x(i) where a=x(1)<x(2)<. . . <x(n)=b and a set of n values y(i) = f(x(i)), a cubic spline interpolant S(x) is defined as:

Introduction

where

Having defined h(i) = x(i+1) - x(i)

Algorithm

It is possible to write the previous conditions in matrix form (Ax=B). In order to solve the system two boundary conidtions are needed.

- Natural spline: S1"(x1)=2*c(1)=0 ; Sn"(xn)=2*c(n)=0 In matrix form:

- Parabolic runout spline: S1"(x1)=2*c(1)=S2"(x2)=2*c(2) ; Sn-1"(xn-1)=2*c(n-1)=Sn"(xn)=2*c(n) In matrix form:

A is a tridiagonal matrix (a band matrix of bandwidth 3) of size N=n+1. The factorization algorithms (A=LU) can be simplified considerably because a large number of zeros appear in regular patterns. The Crout method has been used: 1) Solve LZ=B

2) Solve UX=Z

c(i) for i=1, . . . , n-1 are needed to compute the n-1 polynomials. b(i) and d(i) are computed as:

- b(i) = [y(i+1)-y(i)]/h(i)-h(i)*[c(i+1)+2*c(i)]/3
- d(i) = [c(i+1)-c(i)]/[3*h(i)] Moreover, a(i)=y(i).

It is possible to compute the interpolated vector for x values outside the input range (xq<x(1); xq>x(n)). The coefficients used to compute the y values for xq<x(1) are going to be the ones used for the first interval, while for xq>x(n) the coefficients used for the last interval.

Behaviour outside the given intervals

The initialization function takes as input two arrays that the user has to allocate: `coeffs` will contain the b, c, and d coefficients for the (n-1) intervals (n is the number of known points), hence its size must be 3*(n-1); `tempBuffer` is temporally used for internal computations and its size is n+n-1.

Initialization function

The x input array must be strictly sorted in ascending order and it must not contain twice the same value (x(i)<x(i+1)).

Functions

void **riscv_spline_f32**(riscv_spline_instance_f32 **S*, **const** float32_t **xq*, float32_t **pDst*, uint32_t *blockSize*)

Processing function for the floating-point cubic spline interpolation.

Parameters

- `[in] S`: points to an instance of the floating-point spline structure.
- `[in] xq`: points to the x values ot the interpolated data points.
- `[out] pDst`: points to the block of output data.
- `[in] blockSize`: number of samples of output data.
- `[in] S`: points to an instance of the floating-point spline structure.
- `[in] xq`: points to the x values of the interpolated data points.
- `[out] pDst`: points to the block of output data.
- `[in] blockSize`: number of samples of output data.

void **riscv_spline_init_f32**(riscv_spline_instance_f32 **S*, riscv_spline_type *type*, **const** float32_t **x*, **const** float32_t **y*, uint32_t *n*, float32_t **coeffs*, float32_t **tempBuffer*)

Initialization function for the floating-point cubic spline interpolation.

Parameters

- `[inout] S`: points to an instance of the floating-point spline structure.
- `[in] type`: type of cubic spline interpolation (boundary conditions)
- `[in] x`: points to the x values of the known data points.
- `[in] y`: points to the y values of the known data points.
- `[in] n`: number of known data points.
- `[in] coeffs`: coefficients array for b, c, and d
- `[in] tempBuffer`: buffer array for internal computations

group **groupInterpolation**

These functions perform 1- and 2-dimensional interpolation of data. Linear interpolation is used for 1-dimensional data and bilinear interpolation is used for 2-dimensional data.

3.3.10 Matrix Functions

Matrix Addition

riscv_status **riscv_mat_add_f16**(**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_add_f32**(**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_add_q15**(**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*)

riscv_status **riscv_mat_add_q31**(**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*) *group* **MatrixAdd**

Adds two matrices. The functions check to make sure that `pSrcA`, `pSrcB`, and `pDst` have the same number of rows and columns.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

Functions

riscv_status **riscv_mat_add_f16**(**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix addition.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to first input matrix structure
- `[in] pSrcB`: points to second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_add_f32**(**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix addition.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to first input matrix structure
- `[in] pSrcB`: points to second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_add_q15**(**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*)

Q15 matrix addition.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful

- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcA`: points to first input matrix structure
- `[in] pSrcB`: points to second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_add_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix addition.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcA`: points to first input matrix structure
- `[in] pSrcB`: points to second input matrix structure
- `[out] pDst`: points to output matrix structure

Cholesky and LDLT decompositions

riscv_status **riscv_mat_cholesky_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_cholesky_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_cholesky_f64** (**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

riscv_status **riscv_mat_ldlt_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pl*, riscv_matrix_instance_f32 **pd*, uint16_t **pp*)

riscv_status **riscv_mat_ldlt_f64** (**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pl*, riscv_matrix_instance_f64 **pd*, uint16_t **pp*)

group **MatrixChol**

Computes the Cholesky or LDL^t decomposition of a matrix.

If the input matrix does not have a decomposition, then the algorithm terminates and returns error status RISCV_MATH_DECOMPOSITION_FAILURE.

Functions

riscv_status **riscv_mat_cholesky_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

Floating-point Cholesky decomposition of positive-definite matrix.

Floating-point Cholesky decomposition of Symmetric Positive Definite Matrix.

Return The function returns RISCV_MATH_SIZE_MISMATCH, if the dimensions do not match.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_DECOMPOSITION_FAILURE : Input matrix cannot be decomposed

If the matrix is ill conditioned or only semi-definite, then it is better using the LDL^t decomposition. The decomposition of A is returning a lower triangular matrix U such that A = U U^t

Parameters

- `[in] pSrc`: points to the instance of the input floating-point matrix structure.
- `[out] pDst`: points to the instance of the output floating-point matrix structure.

riscv_status **`riscv_mat_cholesky_f32`**(**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

Floating-point Cholesky decomposition of positive-definite matrix.

Floating-point Cholesky decomposition of Symmetric Positive Definite Matrix.

Return The function returns RISCV_MATH_SIZE_MISMATCH, if the dimensions do not match.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_DECOMPOSITION_FAILURE : Input matrix cannot be decomposed

If the matrix is ill conditioned or only semi-definite, then it is better using the LDL^t decomposition. The decomposition of A is returning a lower triangular matrix U such that A = U U^t

Parameters

- `[in] pSrc`: points to the instance of the input floating-point matrix structure.
- `[out] pDst`: points to the instance of the output floating-point matrix structure.

riscv_status **`riscv_mat_cholesky_f64`**(**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

Floating-point Cholesky decomposition of positive-definite matrix.

Floating-point Cholesky decomposition of Symmetric Positive Definite Matrix.

Return The function returns RISCV_MATH_SIZE_MISMATCH, if the dimensions do not match.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_DECOMPOSITION_FAILURE : Input matrix cannot be decomposed

If the matrix is ill conditioned or only semi-definite, then it is better using the LDL^t decomposition. The decomposition of A is returning a lower triangular matrix U such that A = U U^t

Parameters

- `[in] pSrc`: points to the instance of the input floating-point matrix structure.
- `[out] pDst`: points to the instance of the output floating-point matrix structure.

riscv_status **riscv_mat_ldlt_f32**(**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pl*, riscv_matrix_instance_f32 **pd*, uint16_t **pp*)

Floating-point LDL^t decomposition of positive semi-definite matrix.

Floating-point LDL decomposition of Symmetric Positive Semi-Definite Matrix.

Return The function returns RISCV_MATH_SIZE_MISMATCH, if the dimensions do not match.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_DECOMPOSITION_FAILURE : Input matrix cannot be decomposed

Computes the LDL^t decomposition of a matrix A such that P A P^t = L D L^t.

Parameters

- `[in] pSrc`: points to the instance of the input floating-point matrix structure.
- `[out] pl`: points to the instance of the output floating-point triangular matrix structure.
- `[out] pd`: points to the instance of the output floating-point diagonal matrix structure.
- `[out] pp`: points to the instance of the output floating-point permutation vector.

riscv_status **riscv_mat_ldlt_f64**(**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pl*, riscv_matrix_instance_f64 **pd*, uint16_t **pp*)

Floating-point LDL^t decomposition of positive semi-definite matrix.

Floating-point LDL decomposition of Symmetric Positive Semi-Definite Matrix.

Return The function returns RISCV_MATH_SIZE_MISMATCH, if the dimensions do not match.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_DECOMPOSITION_FAILURE : Input matrix cannot be decomposed

Computes the LDL^t decomposition of a matrix A such that P A P^t = L D L^t.

Parameters

- `[in] pSrc`: points to the instance of the input floating-point matrix structure.
- `[out] pl`: points to the instance of the output floating-point triangular matrix structure.
- `[out] pd`: points to the instance of the output floating-point diagonal matrix structure.
- `[out] pp`: points to the instance of the output floating-point permutation vector.

Complex Matrix Multiplication

riscv_status **riscv_mat_cmplx_mult_f16** (**const** riscv_matrix_instance_f16 *pSrcA*, **const** riscv_matrix_instance_f16 *pSrcB*, riscv_matrix_instance_f16 *pDst*)

riscv_status **riscv_mat_cmplx_mult_f32** (**const** riscv_matrix_instance_f32 *pSrcA*, **const** riscv_matrix_instance_f32 *pSrcB*, riscv_matrix_instance_f32 *pDst*)

riscv_status **riscv_mat_cmplx_mult_q15** (**const** riscv_matrix_instance_q15 *pSrcA*, **const** riscv_matrix_instance_q15 *pSrcB*, riscv_matrix_instance_q15 *pDst*, q15_t *pScratch*)

riscv_status **riscv_mat_cmplx_mult_q31** (**const** riscv_matrix_instance_q31 *pSrcA*, **const** riscv_matrix_instance_q31 *pSrcB*, riscv_matrix_instance_q31 *pDst*)

group **CmplxMatrixMult**

Complex Matrix multiplication is only defined if the number of columns of the first matrix equals the number of rows of the second matrix. Multiplying an `M x N` matrix with an `N x P` matrix results in an `M x P` matrix.

When matrix size checking is enabled, the functions check:

- that the inner dimensions of `pSrcA` and `pSrcB` are equal;
- that the size of the output matrix equals the outer dimensions of `pSrcA` and `pSrcB`.

Functions

riscv_status **riscv_mat_cmplx_mult_f16** (**const** riscv_matrix_instance_f16 *pSrcA*, **const** riscv_matrix_instance_f16 *pSrcB*, riscv_matrix_instance_f16 *pDst*)

Floating-point Complex matrix multiplication.

Floating-point, complex, matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to first input complex matrix structure
- `[in] pSrcB`: points to second input complex matrix structure
- `[out] pDst`: points to output complex matrix structure

riscv_status **riscv_mat_cmplx_mult_f32** (**const** riscv_matrix_instance_f32 *pSrcA*, **const** riscv_matrix_instance_f32 *pSrcB*, riscv_matrix_instance_f32 *pDst*)

Floating-point Complex matrix multiplication.

Floating-point, complex, matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to first input complex matrix structure
- `[in] pSrcB`: points to second input complex matrix structure
- `[out] pDst`: points to output complex matrix structure

riscv_status **riscv_mat_cmplx_mult_q15**(**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*, q15_t **pScratch*)

Q15 Complex matrix multiplication.

Q15, complex, matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Conditions for optimum performance Input, output and state buffers should be aligned by 32-bit

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The inputs to the multiplications are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Parameters

- `[in] pSrcA`: points to first input complex matrix structure
- `[in] pSrcB`: points to second input complex matrix structure
- `[out] pDst`: points to output complex matrix structure
- `[in] pScratch`: points to an array for storing intermediate results

riscv_status **riscv_mat_cmplx_mult_q31**(**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

Q31 Complex matrix multiplication.

Q31, complex, matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. The input is thus scaled down by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are performed internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

Parameters

- `[in] pSrcA`: points to first input complex matrix structure

- `[in] pSrcB`: points to second input complex matrix structure
- `[out] pDst`: points to output complex matrix structure

Complex Matrix Transpose

riscv_status **riscv_mat_cmplx_trans_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_cmplx_trans_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_cmplx_trans_q15** (**const** riscv_matrix_instance_q15 **pSrc*, riscv_matrix_instance_q15 **pDst*)

riscv_status **riscv_mat_cmplx_trans_q31** (**const** riscv_matrix_instance_q31 **pSrc*, riscv_matrix_instance_q31 **pDst*)

group **MatrixComplexTrans**

Tranposes a complex matrix.

Transposing an `M x N` matrix flips it around the center diagonal and results in an `N x M` matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Functions

riscv_status **riscv_mat_cmplx_trans_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix transpose.

Floating-point complex matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_cmplx_trans_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix transpose.

Floating-point complex matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_cmplx_trans_q15** (**const** riscv_matrix_instance_q15 **pSrc*, riscv_matrix_instance_q15 **pDst*)

Q15 complex matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_cmplx_trans_q31** (**const** riscv_matrix_instance_q31 **pSrc*, riscv_matrix_instance_q31 **pDst*)

Q31 complex matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

Matrix Initialization

void **riscv_mat_init_f16** (riscv_matrix_instance_f16 **S*, uint16_t *nRows*, uint16_t *nColumns*, float16_t **pData*)

void **riscv_mat_init_f32** (riscv_matrix_instance_f32 **S*, uint16_t *nRows*, uint16_t *nColumns*, float32_t **pData*)

void **riscv_mat_init_f64** (riscv_matrix_instance_f64 **S*, uint16_t *nRows*, uint16_t *nColumns*, float32_t **pData*)

void **riscv_mat_init_q15** (riscv_matrix_instance_q15 **S*, uint16_t *nRows*, uint16_t *nColumns*, q15_t **pData*)

void **riscv_mat_init_q31** (riscv_matrix_instance_q31 **S*, uint16_t *nRows*, uint16_t *nColumns*, q31_t **pData*) *group* **MatrixInit**

Initializes the underlying matrix data structure. The functions set the `numRows`, `numCols`, and `pData` fields of the matrix data structure.

Functions

void **riscv_mat_init_f16**(riscv_matrix_instance_f16 **S*, uint16_t *nRows*, uint16_t *nColumns*, float16_t **pData*)

Floating-point matrix initialization.

Return none

Parameters

- `[inout] S`: points to an instance of the floating-point matrix structure
- `[in] nRows`: number of rows in the matrix
- `[in] nColumns`: number of columns in the matrix
- `[in] pData`: points to the matrix data array

void **riscv_mat_init_f32**(riscv_matrix_instance_f32 **S*, uint16_t *nRows*, uint16_t *nColumns*, float32_t **pData*)

Floating-point matrix initialization.

Return none

Parameters

- `[inout] S`: points to an instance of the floating-point matrix structure
- `[in] nRows`: number of rows in the matrix
- `[in] nColumns`: number of columns in the matrix
- `[in] pData`: points to the matrix data array

void **riscv_mat_init_f64**(riscv_matrix_instance_f64 **S*, uint16_t *nRows*, uint16_t *nColumns*, float32_t **pData*)

Floating-point matrix initialization.

Return none

Parameters

- `[inout] S`: points to an instance of the floating-point matrix structure
- `[in] nRows`: number of rows in the matrix
- `[in] nColumns`: number of columns in the matrix
- `[in] pData`: points to the matrix data array

void **riscv_mat_init_q15**(riscv_matrix_instance_q15 **S*, uint16_t *nRows*, uint16_t *nColumns*, q15_t **pData*)

Q15 matrix initialization.

Return none

Parameters

- `[inout] S`: points to an instance of the floating-point matrix structure
- `[in] nRows`: number of rows in the matrix
- `[in] nColumns`: number of columns in the matrix
- `[in] pData`: points to the matrix data array

void **riscv_mat_init_q31**(riscv_matrix_instance_q31 **S*, uint16_t *nRows*, uint16_t *nColumns*, q31_t **pData*)

Q31 matrix initialization.

Return none

Parameters

- `[inout] S`: points to an instance of the Q31 matrix structure
- `[in] nRows`: number of rows in the matrix
- `[in] nColumns`: number of columns in the matrix
- `[in] pData`: points to the matrix data array

Matrix Inverse

riscv_status **riscv_mat_inverse_f16**(**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_inverse_f32**(**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_inverse_f64**(**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

riscv_status **riscv_mat_solve_lower_triangular_f16**(**const** riscv_matrix_instance_f16 **lt*, **const** riscv_matrix_instance_f16 **a*, riscv_matrix_instance_f16 **dst*)

riscv_status **riscv_mat_solve_lower_triangular_f32**(**const** riscv_matrix_instance_f32 **lt*, **const** riscv_matrix_instance_f32 **a*, riscv_matrix_instance_f32 **dst*)

riscv_status **riscv_mat_solve_lower_triangular_f64**(**const** riscv_matrix_instance_f64 **lt*, **const** riscv_matrix_instance_f64 **a*, riscv_matrix_instance_f64 **dst*)

riscv_status **riscv_mat_solve_upper_triangular_f16**(**const** riscv_matrix_instance_f16 **ut*, **const** riscv_matrix_instance_f16 **a*, riscv_matrix_instance_f16 **dst*)

riscv_status **riscv_mat_solve_upper_triangular_f32**(**const** riscv_matrix_instance_f32 **ut*, **const** riscv_matrix_instance_f32 **a*, riscv_matrix_instance_f32 **dst*)

riscv_status **riscv_mat_solve_upper_triangular_f64**(**const** riscv_matrix_instance_f64 **ut*, **const** riscv_matrix_instance_f64 **a*, riscv_matrix_instance_f64 **dst*)

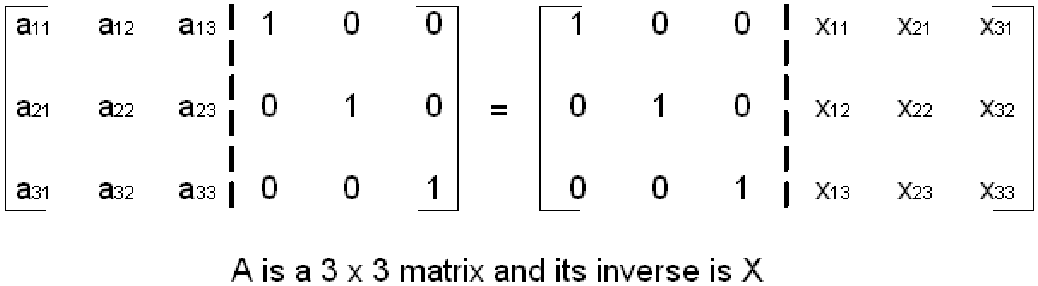
group **MatrixInv**

Computes the inverse of a matrix.

The inverse is defined only if the input matrix is square and non-singular (the determinant is non-zero). The function checks that the input and output matrices are square and of the same size.

Matrix inversion is numerically sensitive and the NMSIS DSP library only supports matrix inversion of floating-point matrices.

Algorithm The Gauss-Jordan method is used to find the inverse. The algorithm performs a sequence of elementary row-operations until it reduces the input matrix to an identity matrix. Applying the same sequence of elementary row-operations to an identity matrix yields the inverse matrix. If the input matrix is singular, then the algorithm terminates and returns error status `RISCV_MATH_SINGULAR`.



Functions

riscv_status **riscv_mat_inverse_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix inverse.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_SINGULAR : Input matrix is found to be singular (non-invertible)

Parameters

- `[in] pSrc`: points to input matrix structure. The source matrix is modified by the function.
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_inverse_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix inverse.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_SINGULAR : Input matrix is found to be singular (non-invertible)

Parameters

- `[in] pSrc`: points to input matrix structure. The source matrix is modified by the function.
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_inverse_f64** (**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

Floating-point (64 bit) matrix inverse.

Floating-point matrix inverse.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed
- RISCV_MATH_SINGULAR : Input matrix is found to be singular (non-invertible)

Parameters

- `[in] pSrc`: points to input matrix structure. The source matrix is modified by the function.
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_solve_lower_triangular_f16**(**const** riscv_matrix_instance_f16 **lt*, **const** riscv_matrix_instance_f16 **a*, riscv_matrix_instance_f16 **dst*)

Solve LT . X = A where LT is a lower triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] lt`: The lower triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of LT . X = A

riscv_status **riscv_mat_solve_lower_triangular_f32**(**const** riscv_matrix_instance_f32 **lt*, **const** riscv_matrix_instance_f32 **a*, riscv_matrix_instance_f32 **dst*)

Solve LT . X = A where LT is a lower triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] lt`: The lower triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of LT . X = A

riscv_status **riscv_mat_solve_lower_triangular_f64**(**const** riscv_matrix_instance_f64 **lt*, **const** riscv_matrix_instance_f64 **a*, riscv_matrix_instance_f64 **dst*)

Solve LT . X = A where LT is a lower triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] lt`: The lower triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of LT . X = A

riscv_status **riscv_mat_solve_upper_triangular_f16** (**const** riscv_matrix_instance_f16 **ut*, **const** riscv_matrix_instance_f16 **a*, riscv_matrix_instance_f16 **dst*)

Solve UT . X = A where UT is an upper triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] ut`: The upper triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of UT . X = A

riscv_status **riscv_mat_solve_upper_triangular_f32** (**const** riscv_matrix_instance_f32 **ut*, **const** riscv_matrix_instance_f32 **a*, riscv_matrix_instance_f32 **dst*)

Solve UT . X = A where UT is an upper triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] ut`: The upper triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of UT . X = A

riscv_status **riscv_mat_solve_upper_triangular_f64** (**const** riscv_matrix_instance_f64 **ut*, **const** riscv_matrix_instance_f64 **a*, riscv_matrix_instance_f64 **dst*)

Solve UT . X = A where UT is an upper triangular matrix.

Return The function returns RISCV_MATH_SINGULAR, if the system can't be solved.

Parameters

- `[in] ut`: The upper triangular matrix
- `[in] a`: The matrix a
- `[out] dst`: The solution X of UT . X = A

Matrix Multiplication

riscv_status **riscv_mat_mult_f16** (**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_mult_f32** (**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_mult_f64** (**const** riscv_matrix_instance_f64 **pSrcA*, **const** riscv_matrix_instance_f64 **pSrcB*, riscv_matrix_instance_f64 **pDst*) riscv_status **riscv_mat_mult_fast_q15** (**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*, q15_t **pState*)

riscv_status **riscv_mat_mult_fast_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

riscv_status **riscv_mat_mult_q15** (**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*, q15_t **pState*)

riscv_status **riscv_mat_mult_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

riscv_status **riscv_mat_mult_q7** (**const** riscv_matrix_instance_q7 **pSrcA*, **const** riscv_matrix_instance_q7 **pSrcB*, riscv_matrix_instance_q7 **pDst*, q7_t **pState*)

group **MatrixMult**

Multiplies two matrices.

Matrix multiplication is only defined if the number of columns of the first matrix equals the number of rows of the second matrix. Multiplying an `M x N` matrix with an `N x P` matrix results in an `M x P` matrix. When matrix size checking is enabled, the functions check: (1) that the inner dimensions of `pSrcA` and `pSrcB` are equal; and (2) that the size of the output matrix equals the outer dimensions of `pSrcA` and `pSrcB`.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} & a_{11} \times b_{12} + a_{12} \times b_{22} + a_{13} \times b_{32} & a_{11} \times b_{13} + a_{12} \times b_{23} + a_{13} \times b_{33} \\ a_{21} \times b_{11} + a_{22} \times b_{21} + a_{23} \times b_{31} & a_{21} \times b_{12} + a_{22} \times b_{22} + a_{23} \times b_{32} & a_{21} \times b_{13} + a_{22} \times b_{23} + a_{23} \times b_{33} \\ a_{31} \times b_{11} + a_{32} \times b_{21} + a_{33} \times b_{31} & a_{31} \times b_{12} + a_{32} \times b_{22} + a_{33} \times b_{32} & a_{31} \times b_{13} + a_{32} \times b_{23} + a_{33} \times b_{33} \end{bmatrix}$$

Functions

riscv_status **riscv_mat_mult_f16** (**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix multiplication.

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] *pSrcA`: points to the first input matrix structure
- `[in] *pSrcB`: points to the second input matrix structure
- `[out] *pDst`: points to output matrix structure

riscv_status **riscv_mat_mult_f32** (**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix multiplication.

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] *pSrcA`: points to the first input matrix structure
- `[in] *pSrcB`: points to the second input matrix structure
- `[out] *pDst`: points to output matrix structure

riscv_status **riscv_mat_mult_f64** (**const** riscv_matrix_instance_f64 **pSrcA*, **const** riscv_matrix_instance_f64 **pSrcB*, riscv_matrix_instance_f64 **pDst*)

Floating-point matrix multiplication.

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] *pSrcA`: points to the first input matrix structure
- `[in] *pSrcB`: points to the second input matrix structure
- `[out] *pDst`: points to output matrix structure

riscv_status **riscv_mat_mult_fast_q15** (**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*, q15_t **pState*)

Q15 matrix multiplication (fast variant).

Q15 matrix multiplication (fast variant) for RISC-V Core with DSP enabled.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The difference between the function riscv_mat_mult_q15() and this fast variant is that the fast variant use a 32-bit rather than a 64-bit accumulator. The result of each 1.15 x 1.15 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.15 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 16 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down one of the input matrices by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are computed internally for each output element.

Remark Refer to riscv_mat_mult_q15() for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure
- `[in] pState`: points to the array for storing intermediate results

riscv_status **riscv_mat_mult_fast_q31**(**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix multiplication (fast variant).

Q31 matrix multiplication (fast variant) for RISC-V Core with DSP enabled.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The difference between the function riscv_mat_mult_q31() and this fast variant is that the fast variant use a 32-bit rather than a 64-bit accumulator. The result of each 1.31 x 1.31 multiplication is truncated to 2.30 format. These intermediate results are accumulated in a 32-bit register in 2.30 format. Finally, the accumulator is saturated and converted to a 1.31 result.

The fast version has the same overflow behavior as the standard version but provides less precision since it discards the low 32 bits of each multiplication result. In order to avoid overflows completely the input signals must be scaled down. Scale down one of the input matrices by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are computed internally for each output element.

Remark Refer to riscv_mat_mult_q31() for a slower implementation of this function which uses 64-bit accumulation to provide higher precision.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_mult_q15**(**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*, q15_t **pState*)

Q15 matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The inputs to the multiplications are in 1.15 format and multiplications yield a 2.30 result. The 2.30 intermediate results are accumulated in a 64-bit accumulator in 34.30 format. This approach provides 33 guard bits and there is no risk of overflow. The 34.30 result is then truncated to 34.15 format by discarding the low 15 bits and then saturated to 1.15 format.

Refer to riscv_mat_mult_fast_q15() for a faster but less precise version of this function.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure
- `[in] pState`: points to the array for storing intermediate results (Unused)

riscv_status **riscv_mat_mult_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix multiplication.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The accumulator has a 2.62 format and maintains full precision of the intermediate multiplication results but provides only a single guard bit. There is no saturation on intermediate additions. Thus, if the accumulator overflows it wraps around and distorts the result. The input signals should be scaled down to avoid intermediate overflows. The input is thus scaled down by log2(numColsA) bits to avoid overflows, as a total of numColsA additions are performed internally. The 2.62 accumulator is right shifted by 31 bits and saturated to 1.31 format to yield the final result.

Remark Refer to riscv_mat_mult_fast_q31() for a faster but less precise implementation of this function.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_mult_q7** (**const** riscv_matrix_instance_q7 **pSrcA*, **const** riscv_matrix_instance_q7 **pSrcB*, riscv_matrix_instance_q7 **pDst*, q7_t **pState*)

Q7 matrix multiplication.

Scaling and Overflow Behavior:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] *pSrcA`: points to the first input matrix structure
- `[in] *pSrcB`: points to the second input matrix structure
- `[out] *pDst`: points to output matrix structure
- `[in] *pState`: points to the array for storing intermediate results (Unused in some versions)

The function is implemented using a 32-bit internal accumulator saturated to 1.7 format.

Matrix Scale

riscv_status **riscv_mat_scale_f16** (**const** riscv_matrix_instance_f16 **pSrc*, float16_t *scale*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_scale_f32** (**const** riscv_matrix_instance_f32 **pSrc*, float32_t *scale*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_scale_q15** (**const** riscv_matrix_instance_q15 **pSrc*, q15_t *scaleFract*, int32_t *shift*, riscv_matrix_instance_q15 **pDst*) riscv_status **riscv_mat_scale_q31** (**const** riscv_matrix_instance_q31 **pSrc*, q31_t *scaleFract*, int32_t *shift*, riscv_matrix_instance_q31 **pDst*)

group **MatrixScale**

Multiplies a matrix by a scalar. This is accomplished by multiplying each element in the matrix by the scalar. For example: The function checks to make sure that the input and output matrices are of the same size.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times K = \begin{bmatrix} a_{11} \times K & a_{12} \times K & a_{13} \times K \\ a_{21} \times K & a_{22} \times K & a_{23} \times K \\ a_{31} \times K & a_{32} \times K & a_{33} \times K \end{bmatrix}$$

In the fixed-point Q15 and Q31 functions, `scale` is represented by a fractional multiplication `scaleFract` and an arithmetic shift `shift`. The shift allows the gain of the scaling operation to exceed 1.0. The overall scale factor applied to the fixed-point data is

Functions

riscv_status **riscv_mat_scale_f16** (**const** riscv_matrix_instance_f16 **pSrc*, float16_t *scale*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix scaling.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[in] scale`: scale factor to be applied
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_scale_f32** (**const** riscv_matrix_instance_f32 **pSrc*, float32_t *scale*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix scaling.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[in] scale`: scale factor to be applied
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_scale_q15** (**const** riscv_matrix_instance_q15 **pSrc*, q15_t *scaleFract*, int32_t *shift*, riscv_matrix_instance_q15 **pDst*)

Q15 matrix scaling.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The input data `*pSrc` and `scaleFract` are in 1.15 format. These are multiplied to yield a 2.30 intermediate result and this is shifted with saturation to 1.15 format.

Parameters

- `[in] pSrc`: points to input matrix
- `[in] scaleFract`: fractional portion of the scale factor
- `[in] shift`: number of bits to shift the result by
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_scale_q31** (**const** riscv_matrix_instance_q31 **pSrc*, q31_t *scaleFract*, int32_t *shift*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix scaling.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The input data `*pSrc` and `scaleFract` are in 1.31 format. These are multiplied to yield a 2.62 intermediate result which is shifted with saturation to 1.31 format.

Parameters

- `[in] pSrc`: points to input matrix
- `[in] scaleFract`: fractional portion of the scale factor
- `[in] shift`: number of bits to shift the result by
- `[out] pDst`: points to output matrix structure

Matrix Subtraction

riscv_status **riscv_mat_sub_f16** (**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_sub_f32** (**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_sub_f64** (**const** riscv_matrix_instance_f64 **pSrcA*, **const** riscv_matrix_instance_f64 **pSrcB*, riscv_matrix_instance_f64 **pDst*)

riscv_status **riscv_mat_sub_q15** (**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*)

riscv_status **riscv_mat_sub_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

group **MatrixSub**

Subtract two matrices. The functions check to make sure that `pSrcA`, pSrcB, and pDst have the same number of rows and columns.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} & a_{13}-b_{13} \\ a_{21}-b_{21} & a_{22}-b_{22} & a_{23}-b_{23} \\ a_{31}-b_{31} & a_{32}-b_{32} & a_{33}-b_{33} \end{bmatrix}$$

Functions

riscv_status **riscv_mat_sub_f16**(**const** riscv_matrix_instance_f16 **pSrcA*, **const** riscv_matrix_instance_f16 **pSrcB*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix subtraction.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_sub_f32**(**const** riscv_matrix_instance_f32 **pSrcA*, **const** riscv_matrix_instance_f32 **pSrcB*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix subtraction.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_sub_f64**(**const** riscv_matrix_instance_f64 **pSrcA*, **const** riscv_matrix_instance_f64 **pSrcB*, riscv_matrix_instance_f64 **pDst*)

Floating-point matrix subtraction.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_sub_q15** (**const** riscv_matrix_instance_q15 **pSrcA*, **const** riscv_matrix_instance_q15 **pSrcB*, riscv_matrix_instance_q15 **pDst*)

Q15 matrix subtraction.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

riscv_status **riscv_mat_sub_q31** (**const** riscv_matrix_instance_q31 **pSrcA*, **const** riscv_matrix_instance_q31 **pSrcB*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix subtraction.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range [0x80000000 0x7FFFFFFF] are saturated.

Parameters

- `[in] pSrcA`: points to the first input matrix structure
- `[in] pSrcB`: points to the second input matrix structure
- `[out] pDst`: points to output matrix structure

Matrix Transpose

riscv_status **riscv_mat_trans_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

riscv_status **riscv_mat_trans_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

riscv_status **riscv_mat_trans_f64** (**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

riscv_status **riscv_mat_trans_q15** (**const** riscv_matrix_instance_q15 **pSrc*, riscv_matrix_instance_q15 **pDst*) riscv_status **riscv_mat_trans_q31** (**const** riscv_matrix_instance_q31 **pSrc*, riscv_matrix_instance_q31 **pDst*)

riscv_status **riscv_mat_trans_q7** (**const** riscv_matrix_instance_q7 **pSrc*, riscv_matrix_instance_q7 **pDst*)

group **MatrixTrans**

Tranposes a matrix.

Transposing an `M x N` matrix flips it around the center diagonal and results in an `N x M` matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Functions

riscv_status **riscv_mat_trans_f16** (**const** riscv_matrix_instance_f16 **pSrc*, riscv_matrix_instance_f16 **pDst*)

Floating-point matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_trans_f32** (**const** riscv_matrix_instance_f32 **pSrc*, riscv_matrix_instance_f32 **pDst*)

Floating-point matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_trans_f64** (**const** riscv_matrix_instance_f64 **pSrc*, riscv_matrix_instance_f64 **pDst*)

Floating-point matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_trans_q15**(**const** riscv_matrix_instance_q15 **pSrc*, riscv_matrix_instance_q15 **pDst*)

Q15 matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_trans_q31**(**const** riscv_matrix_instance_q31 **pSrc*, riscv_matrix_instance_q31 **pDst*)

Q31 matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

riscv_status **riscv_mat_trans_q7**(**const** riscv_matrix_instance_q7 **pSrc*, riscv_matrix_instance_q7 **pDst*)

Q7 matrix transpose.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_SIZE_MISMATCH : Matrix size check failed

Parameters

- `[in] pSrc`: points to input matrix
- `[out] pDst`: points to output matrix

Matrix Vector Multiplication

void **riscv_mat_vec_mult_f16**(**const** riscv_matrix_instance_f16 **pSrcMat*, **const** float16_t **pVec*, float16_t **pDst*)

void **riscv_mat_vec_mult_f32**(**const** riscv_matrix_instance_f32 **pSrcMat*, **const** float32_t **pVec*, float32_t **pDst*)

void **riscv_mat_vec_mult_q15**(**const** riscv_matrix_instance_q15 **pSrcMat*, **const** q15_t **pVec*, q15_t **pDst*)

void **riscv_mat_vec_mult_q31**(**const** riscv_matrix_instance_q31 **pSrcMat*, **const** q31_t **pVec*, q31_t **pDst*)

void **riscv_mat_vec_mult_q7**(**const** riscv_matrix_instance_q7 **pSrcMat*, **const** q7_t **pVec*, q7_t **pDst*)

group **MatrixVectMult**

Multiplies a matrix and a vector.

Functions

void **riscv_mat_vec_mult_f16**(**const** riscv_matrix_instance_f16 **pSrcMat*, **const** float16_t **pVec*, float16_t **pDst*)

Floating-point matrix and vector multiplication.

Parameters

- `[in] *pSrcMat`: points to the input matrix structure
- `[in] *pVec`: points to input vector
- `[out] *pDst`: points to output vector

void **riscv_mat_vec_mult_f32**(**const** riscv_matrix_instance_f32 **pSrcMat*, **const** float32_t **pVec*, float32_t **pDst*)

Floating-point matrix and vector multiplication.

Parameters

- `[in] *pSrcMat`: points to the input matrix structure
- `[in] *pVec`: points to input vector
- `[out] *pDst`: points to output vector

void **riscv_mat_vec_mult_q15**(**const** riscv_matrix_instance_q15 **pSrcMat*, **const** q15_t **pVec*, q15_t **pDst*)

Q15 matrix and vector multiplication.

Parameters

- `[in] *pSrcMat`: points to the input matrix structure
- `[in] *pVec`: points to input vector
- `[out] *pDst`: points to output vector

void **riscv_mat_vec_mult_q31**(**const** riscv_matrix_instance_q31 **pSrcMat*, **const** q31_t **pVec*, q31_t **pDst*)

Q31 matrix and vector multiplication.

Parameters

- `[in] *pSrcMat`: points to the input matrix structure
- `[in] *pVec`: points to the input vector
- `[out] *pDst`: points to the output vector

void **riscv_mat_vec_mult_q7**(**const** riscv_matrix_instance_q7 **pSrcMat*, **const** q7_t **pVec*, q7_t **pDst*)

Q7 matrix and vector multiplication.

Parameters

- `[in] *pSrcMat`: points to the input matrix structure
- `[in] *pVec`: points to the input vector
- `[out] *pDst`: points to the output vector

group **groupMatrix**

This set of functions provides basic matrix math operations. The functions operate on matrix data structures. For example, the type definition for the floating-point matrix structure is shown below: There are similar definitions for Q15 and Q31 data types.

The structure specifies the size of the matrix and then points to an array of data. The array is of size `numRows X numCols` and the values are arranged in row order. That is, the matrix element (i, j) is stored at:

Init Functions There is an associated initialization function for each type of matrix data structure. The initialization function sets the values of the internal structure fields. Refer to riscv_mat_init_f32(), riscv_mat_init_q31() and riscv_mat_init_q15() for floating-point, Q31 and Q15 types, respectively.

Use of the initialization function is optional. However, if initialization function is used then the instance structure cannot be placed into a const data section. To place the instance structure in a const data section, manually initialize the data structure. For example: where `nRows` specifies the number of rows, `nColumns` specifies the number of columns, and `pData` points to the data array.

Size Checking By default all of the matrix functions perform size checking on the input and output matrices. For example, the matrix addition function verifies that the two input matrices and the output matrix all have the same number of rows and columns. If the size check fails the functions return: Otherwise the functions return There is some overhead associated with this matrix size checking. The matrix size checking is enabled via the #define within the library project settings. By default this macro is defined and size checking is enabled. By changing the project settings and undefining this macro size checking is eliminated and the functions run a bit faster. With size checking disabled the functions always return `RISCV_MATH_SUCCESS`.

3.3.11 Quaternion Math Functions

Quaternion conversions

Quaternion to Rotation

void **riscv_quaternion2rotation_f32**(**const** float32_t **pInputQuaternions*, float32_t **pOutputRotations*, uint32_t *nbQuaternions*)

group **QuatRot**

Conversions from quaternion to rotation.

Functions

void **riscv_quaternion2rotation_f32**(**const** float32_t **pInputQuaternions*, float32_t **pOutputRotations*, uint32_t *nbQuaternions*)

Conversion of quaternion to equivalent rotation matrix.

The quaternion a + ib + jc + kd is converted into rotation matrix: Rotation matrix is saved in row order : R00 R01 R02 R10 R11 R12 R20 R21 R22

Return none.

Format of rotation matrix

Parameters

- `[in] pInputQuaternions`: points to an array of normalized quaternions
- `[out] pOutputRotations`: points to an array of 3x3 rotations (in row order)
- `[in] nbQuaternions`: number of quaternions in the array

Rotation to Quaternion

void **riscv_rotation2quaternion_f32**(**const** float32_t **pInputRotations*, float32_t **pOutputQuaternions*, uint32_t *nbQuaternions*)

group **RotQuat**

Conversions from rotation to quaternion.

Functions

void **riscv_rotation2quaternion_f32**(**const** float32_t **pInputRotations*, float32_t **pOutputQuaternions*, uint32_t *nbQuaternions*)

Conversion of a rotation matrix to an equivalent quaternion.

Conversion of a rotation matrix to equivalent quaternion.

q and -q are representing the same rotation. This ambiguity must be taken into account when using the output of this function.

Return none.

Parameters

- `[in] pInputRotations`: points to an array 3x3 rotation matrix (in row order)
- `[out] pOutputQuaternions`: points to an array quaternions
- `[in] nbQuaternions`: number of quaternions in the array

group **QuatConv**

Conversions between quaternion and rotation representations.

Quaternion Conjugate

void **riscv_quaternion_conjugate_f32**(**const** float32_t **pInputQuaternions*, float32_t **pConjugateQuaternions*, uint32_t *nbQuaternions*)

group **QuatConjugate**

Compute the conjugate of a quaternion.

Functions

void **riscv_quaternion_conjugate_f32** (**const** float32_t **pInputQuaternions*, float32_t **pConjugateQuaternions*, uint32_t *nbQuaternions*)

Floating-point quaternion conjugates.

Return none

Parameters

- `[in] pInputQuaternions`: points to the input vector of quaternions
- `[out] pConjugateQuaternions`: points to the output vector of conjugate quaternions
- `[in] nbQuaternions`: number of quaternions in each vector

Quaternion Inverse

void **riscv_quaternion_inverse_f32** (**const** float32_t **pInputQuaternions*, float32_t **pInverseQuaternions*, uint32_t *nbQuaternions*)

group **QuatInverse**

Compute the inverse of a quaternion.

Functions

void **riscv_quaternion_inverse_f32** (**const** float32_t **pInputQuaternions*, float32_t **pInverseQuaternions*, uint32_t *nbQuaternions*)

Floating-point quaternion inverse.

Return none

Parameters

- `[in] pInputQuaternions`: points to the input vector of quaternions
- `[out] pInverseQuaternions`: points to the output vector of inverse quaternions
- `[in] nbQuaternions`: number of quaternions in each vector

Quaternion Norm

void **riscv_quaternion_norm_f32** (**const** float32_t **pInputQuaternions*, float32_t **pNorms*, uint32_t *nbQuaternions*)

group **QuatNorm**

Compute the norm of a quaternion.

Functions

void **riscv_quaternion_norm_f32** (**const** float32_t **pInputQuaternions*, float32_t **pNorms*, uint32_t *nbQuaternions*)

Floating-point quaternion Norm.

Return none

Parameters

- `[in] pInputQuaternions`: points to the input vector of quaternions
- `[out] pNorms`: points to the output vector of norms
- `[in] nbQuaternions`: number of quaternions in the input vector

Quaternion normalization

void **riscv_quaternion_normalize_f32** (**const** float32_t **pInputQuaternions*, float32_t **pNormalizedQuaternions*, uint32_t *nbQuaternions*)

group **QuatNormalized**

Compute a normalized quaternion.

Functions

void **riscv_quaternion_normalize_f32** (**const** float32_t **pInputQuaternions*, float32_t **pNormalizedQuaternions*, uint32_t *nbQuaternions*)

Floating-point normalization of quaternions.

Return none

Parameters

- `[in] pInputQuaternions`: points to the input vector of quaternions
- `[out] pNormalizedQuaternions`: points to the output vector of normalized quaternions
- `[in] nbQuaternions`: number of quaternions in each vector

Quaternion Product

Elementwise Quaternion Product

void **riscv_quaternion_product_f32** (**const** float32_t **qa*, **const** float32_t **qb*, float32_t **qr*, uint32_t *nbQuaternions*)

group **QuatProdVect**

Compute the elementwise product of quaternions.

Functions

void **riscv_quaternion_product_f32** (**const** float32_t **qa*, **const** float32_t **qb*, float32_t **qr*, uint32_t *nbQuaternions*)

Floating-point elementwise product two quaternions.

Return none

Parameters

- `[in] qa`: first array of quaternions
- `[in] qb`: second array of quaternions
- `[out] qr`: elementwise product of quaternions
- `[in] nbQuaternions`: number of quaternions in the array

Quaternion Product

void **riscv_quaternion_product_single_f32** (**const** float32_t *qa, **const** float32_t *qb, float32_t *qr)

group **QuatProdSingle**

Compute the product of two quaternions.

Functions

void **riscv_quaternion_product_single_f32** (**const** float32_t *qa, **const** float32_t *qb, float32_t *qr)

Floating-point product of two quaternions.

Return none

Parameters

- `[in] qa`: first quaternion
- `[in] qb`: second quaternion
- `[out] qr`: product of two quaternions

group **QuatProd**

Compute the product of quaternions.

group **groupQuaternionMath**

Functions to operates on quaternions and convert between a rotation and quaternion representation.

3.3.12 Statistics Functions

Absolute Maximum

void **riscv_absmax_f16** (**const** float16_t *pSrc, uint32_t blockSize, float16_t *pResult, uint32_t *pIndex)

void **riscv_absmax_f32** (**const** float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t *pIndex)

void **riscv_absmax_q15** (**const** q15_t *pSrc, uint32_t blockSize, q15_t *pResult, uint32_t *pIndex)

void **riscv_absmax_q31** (**const** q31_t *pSrc, uint32_t blockSize, q31_t *pResult, uint32_t *pIndex)

void **riscv_absmax_q7** (**const** q7_t *pSrc, uint32_t blockSize, q7_t *pResult, uint32_t *pIndex)

group **AbsMax**

Computes the maximum value of absolute values of an array of data. The function returns both the maximum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_absmax_f16** (**const** float16_t *pSrc, uint32_t blockSize, float16_t *pResult, uint32_t *pIndex)

Maximum value of absolute values of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_absmax_f32**(**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

Maximum value of absolute values of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_absmax_q15**(**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)

Maximum value of absolute values of a Q15 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_absmax_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)

Maximum value of absolute values of a Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_absmax_q7**(**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)

Maximum value of absolute values of a Q7 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector

- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

Absolute Minimum

void **riscv_absmin_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)

void **riscv_absmin_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

void **riscv_absmin_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)

void **riscv_absmin_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)

void **riscv_absmin_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)

group **AbsMin**

Computes the minimum value of absolute values of an array of data. The function returns both the minimum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_absmin_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)

Minimum value of absolute values of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_absmin_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

Minimum value of absolute values of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_absmin_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)

Minimum value of absolute values of a Q15 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_absmin_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)
Minimum value of absolute values of a Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_absmin_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)
Minimum value of absolute values of a Q7 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

Entropy

float16_t **riscv_entropy_f16** (**const** float16_t **pSrcA*, uint32_t *blockSize*)

float32_t **riscv_entropy_f32** (**const** float32_t **pSrcA*, uint32_t *blockSize*)

float64_t **riscv_entropy_f64** (**const** float64_t **pSrcA*, uint32_t *blockSize*)

group **Entropy**
Computes the entropy of a distribution

Functions

float16_t **riscv_entropy_f16** (**const** float16_t **pSrcA*, uint32_t *blockSize*)
Entropy.

Return Entropy -Sum(p ln p)

Parameters

- `[in] pSrcA`: Array of input values.
- `[in] blockSize`: Number of samples in the input array.

float32_t **riscv_entropy_f32**(**const** float32_t **pSrcA*, uint32_t *blockSize*)
Entropy.

Return Entropy -Sum(p ln p)

Parameters

- `[in] pSrcA`: Array of input values.
- `[in] blockSize`: Number of samples in the input array.

float64_t **riscv_entropy_f64**(**const** float64_t **pSrcA*, uint32_t *blockSize*)
Entropy.

Return Entropy -Sum(p ln p)

Parameters

- `[in] pSrcA`: Array of input values.
- `[in] blockSize`: Number of samples in the input array.

Kullback-Leibler divergence

float16_t **riscv_kullback_leibler_f16**(**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*)

float32_t **riscv_kullback_leibler_f32**(**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*)

float64_t **riscv_kullback_leibler_f64**(**const** float64_t **pSrcA*, **const** float64_t **pSrcB*, uint32_t *blockSize*)

group **Kullback-Leibler**
Computes the Kullback-Leibler divergence between two distributions

Functions

float16_t **riscv_kullback_leibler_f16**(**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*)
Kullback-Leibler.

Distribution A may contain 0 with Neon version. Result will be right but some exception flags will be set.

Distribution B must not contain 0 probability.

Return Kullback-Leibler divergence D(A || B)

Parameters

- `[in] *pSrcA`: points to an array of input values for probaility distribution A.
- `[in] *pSrcB`: points to an array of input values for probaility distribution B.
- `[in] blockSize`: number of samples in the input array.

float32_t **riscv_kullback_leibler_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*)

Kullback-Leibler.

Distribution A may contain 0 with Neon version. Result will be right but some exception flags will be set.

Distribution B must not contain 0 probability.

Return Kullback-Leibler divergence D(A || B)

Parameters

- `[in] *pSrcA`: points to an array of input values for probaility distribution A.
- `[in] *pSrcB`: points to an array of input values for probaility distribution B.
- `[in] blockSize`: number of samples in the input array.

float64_t **riscv_kullback_leibler_f64** (**const** float64_t **pSrcA*, **const** float64_t **pSrcB*, uint32_t *blockSize*)

Kullback-Leibler.

Return Kullback-Leibler divergence D(A || B)

Parameters

- `[in] *pSrcA`: points to an array of input values for probaility distribution A.
- `[in] *pSrcB`: points to an array of input values for probaility distribution B.
- `[in] blockSize`: number of samples in the input array.

LogSumExp

float16_t **riscv_logsumexp_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*, float16_t **pTmpBuffer*)

float32_t **riscv_logsumexp_dot_prod_f32** (**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*, float32_t **pTmpBuffer*)

float16_t **riscv_logsumexp_f16** (**const** float16_t **in*, uint32_t *blockSize*)

float32_t **riscv_logsumexp_f32** (**const** float32_t **in*, uint32_t *blockSize*)

group **LogSumExp**

LogSumExp optimizations to compute sum of probabilities with Gaussian distributions

Functions

float16_t **riscv_logsumexp_dot_prod_f16** (**const** float16_t **pSrcA*, **const** float16_t **pSrcB*, uint32_t *blockSize*, float16_t **pTmpBuffer*)

Dot product with log arithmetic.

Vectors are containing the log of the samples

Return The log of the dot product.

Parameters

- `[in] *pSrcA`: points to the first input vector
- `[in] *pSrcB`: points to the second input vector

- `[in] blockSize`: number of samples in each vector
- `[in] *pTmpBuffer`: temporary buffer of length blockSize

float32_t **riscv_logsumexp_dot_prod_f32**(**const** float32_t **pSrcA*, **const** float32_t **pSrcB*, uint32_t *blockSize*, float32_t **pTmpBuffer*)

Dot product with log arithmetic.

Vectors are containing the log of the samples

Return The log of the dot product.

Parameters

- `[in] *pSrcA`: points to the first input vector
- `[in] *pSrcB`: points to the second input vector
- `[in] blockSize`: number of samples in each vector
- `[in] *pTmpBuffer`: temporary buffer of length blockSize

float16_t **riscv_logsumexp_f16**(**const** float16_t **in*, uint32_t *blockSize*)

Computation of the LogSumExp.

In probabilistic computations, the dynamic of the probability values can be very wide because they come from gaussian functions. To avoid underflow and overflow issues, the values are represented by their log. In this representation, multiplying the original exp values is easy : their logs are added. But adding the original exp values is requiring some special handling and it is the goal of the LogSumExp function.

If the values are x1…xn, the function is computing:

ln(exp(x1) + … + exp(xn)) and the computation is done in such a way that rounding issues are minimised.

The max xm of the values is extracted and the function is computing: xm + ln(exp(x1 - xm) + … + exp(xn - xm))

Return LogSumExp

Parameters

- `[in] *in`: Pointer to an array of input values.
- `[in] blockSize`: Number of samples in the input array.

float32_t **riscv_logsumexp_f32**(**const** float32_t **in*, uint32_t *blockSize*)

Computation of the LogSumExp.

In probabilistic computations, the dynamic of the probability values can be very wide because they come from gaussian functions. To avoid underflow and overflow issues, the values are represented by their log. In this representation, multiplying the original exp values is easy : their logs are added. But adding the original exp values is requiring some special handling and it is the goal of the LogSumExp function.

If the values are x1…xn, the function is computing:

ln(exp(x1) + … + exp(xn)) and the computation is done in such a way that rounding issues are minimised.

The max xm of the values is extracted and the function is computing: xm + ln(exp(x1 - xm) + … + exp(xn - xm))

Return LogSumExp

Parameters

- `[in] *in`: Pointer to an array of input values.
- `[in] blockSize`: Number of samples in the input array.

Maximum

void **riscv_max_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)

void **riscv_max_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

void **riscv_max_no_idx_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_max_no_idx_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_max_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)

void **riscv_max_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)

void **riscv_max_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)

group **Max**

Computes the maximum value of an array of data. The function returns both the maximum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_max_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)

Maximum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_max_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

Maximum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_max_no_idx_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

Maximum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here

void **riscv_max_no_idx_f32**(**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)
Maximum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here

void **riscv_max_q15**(**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)
Maximum value of a Q15 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_max_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)
Maximum value of a Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

void **riscv_max_q7**(**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)
Maximum value of a Q7 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: maximum value returned here
- `[out] pIndex`: index of maximum value returned here

Mean

void **riscv_mean_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_mean_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_mean_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

void **riscv_mean_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)

void **riscv_mean_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*)

group **mean**

Calculates the mean of the input vector. Mean is defined as the average of the elements in the vector. The underlying algorithm is used:

There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_mean_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

Mean value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector.
- `[in] blockSize`: number of samples in input vector.
- `[out] pResult`: mean value returned here.

void **riscv_mean_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

Mean value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector.
- `[in] blockSize`: number of samples in input vector.
- `[out] pResult`: mean value returned here.

void **riscv_mean_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

Mean value of a Q15 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. The input is represented in 1.15 format and is accumulated in a 32-bit accumulator in 17.15 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is truncated to yield a result of 1.15 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: mean value returned here

void **riscv_mean_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)
Mean value of a Q31 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.31 format and is accumulated in a 64-bit accumulator in 33.31 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is truncated to yield a result of 1.31 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: mean value returned here

void **riscv_mean_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*)
Mean value of a Q7 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. The input is represented in 1.7 format and is accumulated in a 32-bit accumulator in 25.7 format. There is no risk of internal overflow with this approach, and the full precision of intermediate result is preserved. Finally, the accumulator is truncated to yield a result of 1.7 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: mean value returned here

Minimum

void **riscv_min_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)

void **riscv_min_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)

void **riscv_min_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)

void **riscv_min_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)

void **riscv_min_q7** (**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)

group **Min**
Computes the minimum value of an array of data. The function returns both the minimum value and its position within the array. There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_min_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*, uint32_t **pIndex*)
Minimum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_min_f32**(**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*, uint32_t **pIndex*)
Minimum value of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_min_q15**(**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*, uint32_t **pIndex*)
Minimum value of a Q15 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_min_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*, uint32_t **pIndex*)
Minimum value of a Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

void **riscv_min_q7**(**const** q7_t **pSrc*, uint32_t *blockSize*, q7_t **pResult*, uint32_t **pIndex*)
Minimum value of a Q7 vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector

- `[out] pResult`: minimum value returned here
- `[out] pIndex`: index of minimum value returned here

Power

void **riscv_power_f16** (**const** float16_t *pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_power_f32** (**const** float32_t *pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_power_q15** (**const** q15_t *pSrc*, uint32_t *blockSize*, q63_t **pResult*)

void **riscv_power_q31** (**const** q31_t *pSrc*, uint32_t *blockSize*, q63_t **pResult*)

void **riscv_power_q7** (**const** q7_t *pSrc*, uint32_t *blockSize*, q31_t **pResult*)

group **power**

Calculates the sum of the squares of the elements in the input vector. The underlying algorithm is used:

There are separate functions for floating point, Q31, Q15, and Q7 data types.

Since the result is not divided by the length, those functions are in fact computing something which is more an energy than a power.

Functions

void **riscv_power_f16** (**const** float16_t *pSrc*, uint32_t *blockSize*, float16_t **pResult*)

Sum of the squares of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: sum of the squares value returned here

void **riscv_power_f32** (**const** float32_t *pSrc*, uint32_t *blockSize*, float32_t **pResult*)

Sum of the squares of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: sum of the squares value returned here

void **riscv_power_q15** (**const** q15_t *pSrc*, uint32_t *blockSize*, q63_t **pResult*)

Sum of the squares of the elements of a Q15 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 34.30 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: sum of the squares value returned here

void **riscv_power_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q63_t **pResult*)

Sum of the squares of the elements of a Q31 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.31 format. Intermediate multiplication yields a 2.62 format, and this result is truncated to 2.48 format by discarding the lower 14 bits. The 2.48 result is then added without saturation to a 64-bit accumulator in 16.48 format. With 15 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 16.48 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: sum of the squares value returned here

void **riscv_power_q7**(**const** q7_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)

Sum of the squares of the elements of a Q7 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 32-bit internal accumulator. The input is represented in 1.7 format. Intermediate multiplication yields a 2.14 format, and this result is added without saturation to an accumulator in 18.14 format. With 17 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the return result is in 18.14 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: sum of the squares value returned here

Root mean square (RMS)

void **riscv_rms_f16**(**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_rms_f32**(**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_rms_q15**(**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

void **riscv_rms_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*) *group* **RMS**

Calculates the Root Mean Square of the elements in the input vector. The underlying algorithm is used:

There are separate functions for floating point, Q31, and Q15 data types.

Functions

void **riscv_rms_f16**(**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

Root Mean Square of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: root mean square value returned here

void **riscv_rms_f32**(**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

Root Mean Square of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: root mean square value returned here

void **riscv_rms_q15**(**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

Root Mean Square of the elements of a Q15 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: root mean square value returned here

void **riscv_rms_q31**(**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)

Root Mean Square of the elements of a Q31 vector.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, and intermediate multiplication yields a 2.62 format. The accumulator maintains full precision of the intermediate multiplication results, but provides only a single guard bit. There is no saturation on intermediate additions. If the accumulator overflows, it wraps around and distorts the result. In order to avoid overflows completely, the input signal must be scaled down by log2(blockSize) bits, as a total of blockSize additions are performed internally. Finally, the 2.62 accumulator is right shifted by 31 bits to yield a 1.31 format value.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: root mean square value returned here

Standard deviation

void **riscv_std_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_std_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_std_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

void **riscv_std_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)

group **STD**

Calculates the standard deviation of the elements in the input vector.

The float implementation is relying on riscv_var_f32 which is using a two-pass algorithm to avoid problem of numerical instabilities and cancellation errors.

Fixed point versions are using the standard textbook algorithm since the fixed point numerical behavior is different from the float one.

Algorithm for fixed point versions is summarized below:

There are separate functions for floating point, Q31, and Q15 data types.

Functions

void **riscv_std_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

Standard deviation of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: standard deviation value returned here

void **riscv_std_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

Standard deviation of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector

- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: standard deviation value returned here

void **riscv_std_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)
Standard deviation of the elements of a Q15 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: standard deviation value returned here

void **riscv_std_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)
Standard deviation of the elements of a Q31 vector.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, which is then downshifted by 8 bits which yields 1.23, and intermediate multiplication yields a 2.46 format. The accumulator maintains full precision of the intermediate multiplication results, but provides only a 16 guard bits. There is no saturation on intermediate additions. If the accumulator overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(blockSize)-8 bits, as a total of blockSize additions are performed internally. After division, internal variables should be Q18.46 Finally, the 18.46 accumulator is right shifted by 15 bits to yield a 1.31 format value.

Parameters

- `[in] pSrc`: points to the input vector.
- `[in] blockSize`: number of samples in input vector.
- `[out] pResult`: standard deviation value returned here.

Variance

void **riscv_var_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)

void **riscv_var_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)

void **riscv_var_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)

void **riscv_var_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)

group **variance**
Calculates the variance of the elements in the input vector. The underlying algorithm used is the direct method sometimes referred to as the two-pass method:

There are separate functions for floating point, Q31, and Q15 data types.

Functions

void **riscv_var_f16** (**const** float16_t **pSrc*, uint32_t *blockSize*, float16_t **pResult*)
Variance of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: variance value returned here

void **riscv_var_f32** (**const** float32_t **pSrc*, uint32_t *blockSize*, float32_t **pResult*)
Variance of the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: variance value returned here

void **riscv_var_q15** (**const** q15_t **pSrc*, uint32_t *blockSize*, q15_t **pResult*)
Variance of the elements of a Q15 vector.

Return none

Scaling and Overflow Behavior The function is implemented using a 64-bit internal accumulator. The input is represented in 1.15 format. Intermediate multiplication yields a 2.30 format, and this result is added without saturation to a 64-bit accumulator in 34.30 format. With 33 guard bits in the accumulator, there is no risk of overflow, and the full precision of the intermediate multiplication is preserved. Finally, the 34.30 result is truncated to 34.15 format by discarding the lower 15 bits, and then saturated to yield a result in 1.15 format.

Parameters

- `[in] pSrc`: points to the input vector
- `[in] blockSize`: number of samples in input vector
- `[out] pResult`: variance value returned here

void **riscv_var_q31** (**const** q31_t **pSrc*, uint32_t *blockSize*, q31_t **pResult*)
Variance of the elements of a Q31 vector.

Return none

Scaling and Overflow Behavior The function is implemented using an internal 64-bit accumulator. The input is represented in 1.31 format, which is then downshifted by 8 bits which yields 1.23, and intermediate multiplication yields a 2.46 format. The accumulator maintains full precision of the intermediate multiplication results, and as a consequence has only 16 guard bits. There is no saturation on intermediate additions. If the accumulator overflows it wraps around and distorts the result. In order to avoid overflows completely the input signal must be scaled down by log2(blockSize)-8 bits, as a total of blockSize additions are performed internally. After division, internal variables should be Q18.46 Finally, the 18.46 accumulator is right shifted by 15 bits to yield a 1.31 format value.

Parameters

- [in] pSrc: points to the input vector
- [in] blockSize: number of samples in input vector
- [out] pResult: variance value returned here

group **groupStats**

3.3.13 Support Functions

Barycenter

void **riscv_barycenter_f16** (**const** float16_t **in*, **const** float16_t **weights*, float16_t **out*, uint32_t *nbVectors*, uint32_t *vecDim*)

void **riscv_barycenter_f32** (**const** float32_t **in*, **const** float32_t **weights*, float32_t **out*, uint32_t *nbVectors*, uint32_t *vecDim*)

group **barycenter**

Barycenter of weighted vectors

Functions

void **riscv_barycenter_f16** (**const** float16_t **in*, **const** float16_t **weights*, float16_t **out*, uint32_t *nbVectors*, uint32_t *vecDim*)

Barycenter.

Return None

Parameters

- [in] *in: List of vectors
- [in] *weights: Weights of the vectors
- [out] *out: Barycenter
- [in] nbVectors: Number of vectors
- [in] vecDim: Dimension of space (vector dimension)

void **riscv_barycenter_f32** (**const** float32_t **in*, **const** float32_t **weights*, float32_t **out*, uint32_t *nbVectors*, uint32_t *vecDim*)

Barycenter.

Return None

Parameters

- [in] *in: List of vectors
- [in] *weights: Weights of the vectors
- [out] *out: Barycenter
- [in] nbVectors: Number of vectors
- [in] vecDim: Dimension of space (vector dimension)

Vector sorting algorithms

void **riscv_merge_sort_f32** (**const** riscv_merge_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_merge_sort_init_f32** (riscv_merge_sort_instance_f32 **S*, riscv_sort_dir *dir*, float32_t **buffer*)

void **riscv_sort_f32** (**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_sort_init_f32** (riscv_sort_instance_f32 **S*, riscv_sort_alg *alg*, riscv_sort_dir *dir*)

group **Sorting**

Sort the elements of a vector

There are separate functions for floating-point, Q31, Q15, and Q7 data types.

Functions

void **riscv_bitonic_sort_f32** (**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_bubble_sort_f32** (**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The bubble sort algorithm is a simple comparison algorithm that reads the elements of a vector from the beginning to the end, compares the adjacent ones and swaps them if they are in the wrong order. The procedure is repeated until there is nothing left to swap. Bubble sort is fast for input vectors that are nearly sorted.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_heap_sort_f32** (**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The heap sort algorithm is a comparison algorithm that divides the input array into a sorted and an unsorted region, and shrinks the unsorted region by extracting the largest element and moving it to the sorted region. A heap data structure is used to find the maximum.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed.

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_insertion_sort_f32**(**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The insertion sort is a simple sorting algorithm that reads all the element of the input array and removes one element at a time, finds the location it belongs in the final sorted list, and inserts it there.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed.

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_merge_sort_f32**(**const** riscv_merge_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The merge sort algorithm is a comparison algorithm that divide the input array in sublists and merge them to produce longer sorted sublists until there is only one list remaining.

A work array is always needed. It must be allocated by the user linked to the instance at initialization time.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_merge_sort_init_f32**(riscv_merge_sort_instance_f32 **S*, riscv_sort_dir *dir*, float32_t **buffer*)

Parameters

- `[inout] S`: points to an instance of the sorting structure.
- `[in] dir`: Sorting order.
- `[in] buffer`: Working buffer.

void **riscv_quick_sort_f32**(**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The quick sort algorithm is a comparison algorithm that divides the input array into two smaller sub-arrays and recursively sort them. An element of the array (the pivot) is chosen, all the elements with values smaller than the pivot are moved before the pivot, while all elements with values greater than the pivot are moved after it (partition).

In this implementation the Hoare partition scheme has been used [Hoare, C. A. R. (1 January 1962). "Quicksort". The Computer Journal. 5 (1): 10…16.] The first element has always been chosen as the pivot. The partition algorithm guarantees that the returned pivot is never placed outside the vector, since it is returned only when the pointers crossed each other. In this way it isn't possible to obtain empty partitions and infinite recursion is avoided.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed.

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[inout] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data.
- `[in] blockSize`: number of samples to process.

void **riscv_selection_sort_f32**(**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Algorithm The Selection sort algorithm is a comparison algorithm that divides the input array into a sorted and an unsorted sublist (initially the sorted sublist is empty and the unsorted sublist is the input array), looks for the smallest (or biggest) element in the unsorted sublist, swapping it with the leftmost one, and moving the sublists boundary one element to the right.

It's an in-place algorithm. In order to obtain an out-of-place function, a memcpy of the source vector is performed.

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data
- `[in] blockSize`: number of samples to process.

void **riscv_sort_f32**(**const** riscv_sort_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Generic sorting function.

Parameters

- `[in] S`: points to an instance of the sorting structure.
- `[in] pSrc`: points to the block of input data.
- `[out] pDst`: points to the block of output data.
- `[in] blockSize`: number of samples to process.

void **riscv_sort_init_f32**(riscv_sort_instance_f32 **S*, riscv_sort_alg *alg*, riscv_sort_dir *dir*)

Parameters

- `[inout] S`: points to an instance of the sorting structure.
- `[in] alg`: Selected algorithm.
- `[in] dir`: Sorting order.

Vector Copy

void **riscv_copy_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_copy_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_copy_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_copy_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_copy_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **copy**

Copies sample by sample from source vector to destination vector.

There are separate functions for floating point, Q31, Q15, and Q7 data types.

Functions

void **riscv_copy_f16** (**const** float16_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Copies the elements of a f16 vector.

Copies the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_copy_f32** (**const** float32_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Copies the elements of a floating-point vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_copy_q15** (**const** q15_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Copies the elements of a Q15 vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_copy_q31** (**const** q31_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Copies the elements of a Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_copy_q7** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Copies the elements of a Q7 vector.

Return none

Parameters

- `[in] pSrc`: points to input vector
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

Convert 16-bit floating point value

void **riscv_f16_to_float** (**const** float16_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_f16_to_q15** (**const** float16_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

group **f16_to_x**

Functions

void **riscv_f16_to_float** (**const** float16_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)
Converts the elements of the f16 vector to f32 vector.

Converts the elements of the floating-point vector to Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the f16 input vector
- `[out] pDst`: points to the f32 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_f16_to_q15** (**const** float16_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)
Converts the elements of the f16 vector to Q15 vector.

Converts the elements of the floating-point vector to Q31 vector.

Return none

Details The equation used for the conversion process is:

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Note In order to apply rounding in scalar version, the library should be rebuilt with the ROUNDING macro defined in the preprocessor section of project options.

Parameters

- `[in] pSrc`: points to the f16 input vector
- `[out] pDst`: points to the Q15 output vector
- `[in] blockSize`: number of samples in each vector

Vector Fill

void **riscv_fill_f16** (float16_t *value*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_fill_f32** (float32_t *value*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_fill_q15** (q15_t *value*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_fill_q31** (q31_t *value*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_fill_q7** (q7_t *value*, q7_t **pDst*, uint32_t *blockSize*)

group **Fill**

Fills the destination vector with a constant value.

There are separate functions for floating point, Q31, Q15, and Q7 data types.

Functions

void **riscv_fill_f16** (float16_t *value*, float16_t **pDst*, uint32_t *blockSize*)

Fills a constant value into a f16 vector.

Fills a constant value into a floating-point vector.

Return none

Parameters

- `[in] value`: input value to be filled
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_fill_f32** (float32_t *value*, float32_t **pDst*, uint32_t *blockSize*)

Fills a constant value into a floating-point vector.

Return none

Parameters

- `[in] value`: input value to be filled
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_fill_q15** (q15_t *value*, q15_t **pDst*, uint32_t *blockSize*)

Fills a constant value into a Q15 vector.

Return none

Parameters

- `[in] value`: input value to be filled
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_fill_q31** (q31_t *value*, q31_t **pDst*, uint32_t *blockSize*)

Fills a constant value into a Q31 vector.

Return none

Parameters

- `[in] value`: input value to be filled
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_fill_q7** (q7_t *value*, q7_t **pDst*, uint32_t *blockSize*)

Fills a constant value into a Q7 vector.

Return none

Parameters

- `[in] value`: input value to be filled
- `[out] pDst`: points to output vector
- `[in] blockSize`: number of samples in each vector

Convert 32-bit floating point value

void **riscv_float_to_f16** (**const** float32_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_float_to_q15** (**const** float32_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_float_to_q31** (**const** float32_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_float_to_q7** (**const** float32_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **float_to_x**

Functions

void **riscv_float_to_f16** (**const** float32_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Converts the elements of the floating-point vector to f16 vector.

Converts the elements of the floating-point vector to Q31 vector.

Return none

Parameters

- `[in] pSrc`: points to the f32 input vector
- `[out] pDst`: points to the f16 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_float_to_q15**(**const** float32_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)
Converts the elements of the floating-point vector to Q15 vector.

Return none

Details The equation used for the conversion process is:

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] are saturated.

Note In order to apply rounding, the library should be rebuilt with the ROUNDING macro defined in the preprocessor section of project options.

Parameters

- `[in] pSrc`: points to the floating-point input vector
- `[out] pDst`: points to the Q15 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_float_to_q31**(**const** float32_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)
Converts the elements of the floating-point vector to Q31 vector.

Return none

Details The equation used for the conversion process is:

Scaling and Overflow Behavior The function uses saturating arithmetic. Results outside of the allowable Q31 range[0x80000000 0x7FFFFFFF] are saturated.

Note In order to apply rounding, the library should be rebuilt with the ROUNDING macro defined in the preprocessor section of project options.

Parameters

- `[in] pSrc`: points to the floating-point input vector
- `[out] pDst`: points to the Q31 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_float_to_q7**(**const** float32_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Converts the elements of the floating-point vector to Q7 vector.

Return none.

Description:

The equation used for the conversion process is:

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

Note In order to apply rounding, the library should be rebuilt with the ROUNDING macro defined in the preprocessor section of project options.

Parameters

- `[in] *pSrc`: points to the floating-point input vector
- `[out] *pDst`: points to the Q7 output vector

- `[in] blockSize`: length of the input vector

Convert 16-bit Integer value

void **riscv_q15_to_f16** (**const** q15_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

void **riscv_q15_to_float** (**const** q15_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_q15_to_q31** (**const** q15_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

void **riscv_q15_to_q7** (**const** q15_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **q15_to_x**

Functions

void **riscv_q15_to_f16** (**const** q15_t **pSrc*, float16_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q15 vector to f16 vector.

Converts the elements of the floating-point vector to Q31 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q15 input vector
- `[out] pDst`: points to the f16 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_q15_to_float** (**const** q15_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q15 vector to floating-point vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q15 input vector
- `[out] pDst`: points to the floating-point output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_q15_to_q31** (**const** q15_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q15 vector to Q31 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q15 input vector
- `[out] pDst`: points to the Q31 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_q15_to_q7** (**const** q15_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q15 vector to Q7 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q15 input vector
- `[out] pDst`: points to the Q7 output vector
- `[in] blockSize`: number of samples in each vector

Convert 32-bit Integer value

void **riscv_q31_to_float** (**const** q31_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_q31_to_q15** (**const** q31_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_q31_to_q7** (**const** q31_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **q31_to_x**

Functions

void **riscv_q31_to_float** (**const** q31_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q31 vector to floating-point vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q31 input vector
- `[out] pDst`: points to the floating-point output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_q31_to_q15** (**const** q31_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q31 vector to Q15 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- `[in] pSrc`: points to the Q31 input vector
- `[out] pDst`: points to the Q15 output vector
- `[in] blockSize`: number of samples in each vector

void **riscv_q31_to_q7** (**const** q31_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q31 vector to Q7 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- [in] pSrc: points to the Q31 input vector
- [out] pDst: points to the Q7 output vector
- [in] blockSize: number of samples in each vector

Convert 8-bit Integer value

void **riscv_q7_to_float** (**const** q7_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)

void **riscv_q7_to_q15** (**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_q7_to_q31** (**const** q7_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)

group **q7_to_x**

Functions

void **riscv_q7_to_float** (**const** q7_t **pSrc*, float32_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q7 vector to floating-point vector.

Return none

Details The equation used for the conversion process is:

Parameters

- [in] pSrc: points to the Q7 input vector
- [out] pDst: points to the floating-point output vector
- [in] blockSize: number of samples in each vector

void **riscv_q7_to_q15** (**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q7 vector to Q15 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- [in] pSrc: points to the Q7 input vector
- [out] pDst: points to the Q15 output vector
- [in] blockSize: number of samples in each vector

void **riscv_q7_to_q31** (**const** q7_t **pSrc*, q31_t **pDst*, uint32_t *blockSize*)
Converts the elements of the Q7 vector to Q31 vector.

Return none

Details The equation used for the conversion process is:

Parameters

- [in] pSrc: points to the Q7 input vector

- `[out] pDst`: points to the Q31 output vector
- `[in] blockSize`: number of samples in each vector

Weighted Sum

float16_t **riscv_weighted_sum_f16**(**const** float16_t **in*, **const** float16_t **weigths*, uint32_t *blockSize*)

float32_t **riscv_weighted_sum_f32**(**const** float32_t **in*, **const** float32_t **weigths*, uint32_t *blockSize*)

group **weightedsum**

Weighted sum of values

Functions

float16_t **riscv_weighted_sum_f16**(**const** float16_t **in*, **const** float16_t **weigths*, uint32_t *blockSize*)

Weighted sum.

Return Weighted sum

Parameters

- `[in] *in`: Array of input values.
- `[in] *weigths`: Weights
- `[in] blockSize`: Number of samples in the input array.

float32_t **riscv_weighted_sum_f32**(**const** float32_t **in*, **const** float32_t **weigths*, uint32_t *blockSize*)

Weighted sum.

Return Weighted sum

Parameters

- `[in] *in`: Array of input values.
- `[in] *weigths`: Weights
- `[in] blockSize`: Number of samples in the input array.

group **groupSupport**

3.3.14 SVM Functions

Linear SVM

void **riscv_svm_linear_init_f16**(riscv_svm_linear_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*)

void **riscv_svm_linear_init_f32**(riscv_svm_linear_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*) void **riscv_svm_linear_predict_f16** (**const** riscv_svm_linear_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

void **riscv_svm_linear_predict_f32** (**const** riscv_svm_linear_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

group **linearsvm**

Linear SVM classifier

Functions

void **riscv_svm_linear_init_f16** (riscv_svm_linear_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*)

SVM linear instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: Parameters for the SVM function
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID

void **riscv_svm_linear_init_f32** (riscv_svm_linear_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*)

SVM linear instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: Parameters for the SVM function
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID

void **riscv_svm_linear_predict_f16**(**const** riscv_svm_linear_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

SVM linear prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the linear SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

void **riscv_svm_linear_predict_f32**(**const** riscv_svm_linear_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

SVM linear prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the linear SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

Polynomial SVM

void **riscv_svm_polynomial_init_f16**(riscv_svm_polynomial_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, int32_t *degree*, float16_t *coef0*, float16_t *gamma*)

void **riscv_svm_polynomial_init_f32**(riscv_svm_polynomial_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, int32_t *degree*, float32_t *coef0*, float32_t *gamma*)

void **riscv_svm_polynomial_predict_f16**(**const** riscv_svm_polynomial_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

void **riscv_svm_polynomial_predict_f32**(**const** riscv_svm_polynomial_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

group **polysvm**

Polynomial SVM classifier

Functions

void **riscv_svm_polynomial_init_f16**(riscv_svm_polynomial_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, int32_t *degree*, float16_t *coef0*, float16_t *gamma*)

SVM polynomial instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the polynomial SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] degree`: Polynomial degree
- `[in] coef0`: coeff0 (scikit-learn terminology)
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_polynomial_init_f32** (riscv_svm_polynomial_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, int32_t *degree*, float32_t *coef0*, float32_t *gamma*)

SVM polynomial instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the polynomial SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] degree`: Polynomial degree
- `[in] coef0`: coeff0 (scikit-learn terminology)
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_polynomial_predict_f16** (**const** riscv_svm_polynomial_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

SVM polynomial prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the polynomial SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

void **riscv_svm_polynomial_predict_f32** (**const** riscv_svm_polynomial_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

SVM polynomial prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the polynomial SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

RBF SVM

void **riscv_svm_rbf_init_f16** (riscv_svm_rbf_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, float16_t *gamma*)

void **riscv_svm_rbf_init_f32** (riscv_svm_rbf_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, float32_t *gamma*)

void **riscv_svm_rbf_predict_f16** (**const** riscv_svm_rbf_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

void **riscv_svm_rbf_predict_f32** (**const** riscv_svm_rbf_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

group **rbfsvm**

RBF SVM classifier

Functions

void **riscv_svm_rbf_init_f16** (riscv_svm_rbf_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, float16_t *gamma*)

SVM radial basis function instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the polynomial SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept

- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_rbf_init_f32**(riscv_svm_rbf_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, float32_t *gamma*)

SVM radial basis function instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the polynomial SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_rbf_predict_f16**(**const** riscv_svm_rbf_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

SVM rbf prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the rbf SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: decision value

void **riscv_svm_rbf_predict_f32**(**const** riscv_svm_rbf_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

SVM rbf prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the rbf SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: decision value

Sigmoid SVM

void **riscv_svm_sigmoid_init_f16** (riscv_svm_sigmoid_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, float16_t *coef0*, float16_t *gamma*)

void **riscv_svm_sigmoid_init_f32** (riscv_svm_sigmoid_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, float32_t *coef0*, float32_t *gamma*)

void **riscv_svm_sigmoid_predict_f16** (**const** riscv_svm_sigmoid_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

void **riscv_svm_sigmoid_predict_f32** (**const** riscv_svm_sigmoid_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

group **sigmoidsvm**

Sigmoid SVM classifier

Functions

void **riscv_svm_sigmoid_init_f16** (riscv_svm_sigmoid_instance_f16 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float16_t *intercept*, **const** float16_t **dualCoefficients*, **const** float16_t **supportVectors*, **const** int32_t **classes*, float16_t *coef0*, float16_t *gamma*)

SVM sigmoid instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the rbf SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] coef0`: coeff0 (scikit-learn terminology)
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_sigmoid_init_f32** (riscv_svm_sigmoid_instance_f32 **S*, uint32_t *nbOfSupportVectors*, uint32_t *vectorDimension*, float32_t *intercept*, **const** float32_t **dualCoefficients*, **const** float32_t **supportVectors*, **const** int32_t **classes*, float32_t *coef0*, float32_t *gamma*)

SVM sigmoid instance init function.

Classes are integer used as output of the function (instead of having -1,1 as class values).

Return none.

Parameters

- `[in] S`: points to an instance of the rbf SVM structure.
- `[in] nbOfSupportVectors`: Number of support vectors
- `[in] vectorDimension`: Dimension of vector space
- `[in] intercept`: Intercept
- `[in] dualCoefficients`: Array of dual coefficients
- `[in] supportVectors`: Array of support vectors
- `[in] classes`: Array of 2 classes ID
- `[in] coef0`: coeff0 (scikit-learn terminology)
- `[in] gamma`: gamma (scikit-learn terminology)

void **riscv_svm_sigmoid_predict_f16**(**const** riscv_svm_sigmoid_instance_f16 **S*, **const** float16_t **in*, int32_t **pResult*)

SVM sigmoid prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the rbf SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

void **riscv_svm_sigmoid_predict_f32**(**const** riscv_svm_sigmoid_instance_f32 **S*, **const** float32_t **in*, int32_t **pResult*)

SVM sigmoid prediction.

Return none.

Parameters

- `[in] S`: Pointer to an instance of the rbf SVM structure.
- `[in] in`: Pointer to input vector
- `[out] pResult`: Decision value

group **groupSVM**

This set of functions is implementing SVM classification on 2 classes. The training must be done from scikit-learn. The parameters can be easily generated from the scikit-learn object. Some examples are given in DSP/Testing/PatternGeneration/SVM.py

If more than 2 classes are needed, the functions in this folder will have to be used, as building blocks, to do multi-class classification.

No multi-class classification is provided in this SVM folder.

3.3.15 Transform Functions

Complex FFT Functions

Complex FFT Tables

const uint16_t **riscvBitRevTable**[1024]

const uint64_t **twiddleCoefF64_16**[32]

const uint64_t **twiddleCoefF64_32**[64]

const uint64_t **twiddleCoefF64_64**[128]

const uint64_t **twiddleCoefF64_128**[256]

const uint64_t **twiddleCoefF64_256**[512]

const uint64_t **twiddleCoefF64_512**[1024]

const uint64_t **twiddleCoefF64_1024**[2048]

const uint64_t **twiddleCoefF64_2048**[4096]

const uint64_t **twiddleCoefF64_4096**[8192]

const float32_t **twiddleCoef_16**[32]

const float32_t **twiddleCoef_32**[64]

const float32_t **twiddleCoef_64**[128]

const float32_t **twiddleCoef_128**[256]

const float32_t **twiddleCoef_256**[512]

const float32_t **twiddleCoef_512**[1024]

const float32_t **twiddleCoef_1024**[2048]

const float32_t **twiddleCoef_2048**[4096]

const float32_t **twiddleCoef_4096**[8192]

const q31_t **twiddleCoef_16_q31**[24]

const q31_t **twiddleCoef_32_q31**[48]

const q31_t **twiddleCoef_64_q31**[96]

const q31_t **twiddleCoef_128_q31**[192]

const q31_t **twiddleCoef_256_q31**[384]

const q31_t **twiddleCoef_512_q31**[768]

const q31_t **twiddleCoef_1024_q31**[1536]

const q31_t **twiddleCoef_2048_q31**[3072]

const q31_t **twiddleCoef_4096_q31**[6144]

const q15_t **twiddleCoef_16_q15**[24]

const q15_t **twiddleCoef_32_q15**[48]

const q15_t **twiddleCoef_64_q15**[96]

const q15_t **twiddleCoef_128_q15**[192]

const q15_t **twiddleCoef_256_q15**[384]

const q15_t **twiddleCoef_512_q15**[768]

const q15_t **twiddleCoef_1024_q15**[1536]

const q15_t **twiddleCoef_2048_q15**[3072]

const q15_t **twiddleCoef_4096_q15**[6144]

const float16_t **twiddleCoefF16_16**[32]

const float16_t **twiddleCoefF16_32**[64]

const float16_t **twiddleCoefF16_64**[128]

const float16_t **twiddleCoefF16_128**[256]

const float16_t **twiddleCoefF16_256**[512]

const float16_t **twiddleCoefF16_512**[1024]

const float16_t **twiddleCoefF16_1024**[2048]

const float16_t **twiddleCoefF16_2048**[4096]

const float16_t **twiddleCoefF16_4096**[8192]

const float16_t **twiddleCoefF16_rfft_32**[32]

const float16_t **twiddleCoefF16_rfft_64**[64]

const float16_t **twiddleCoefF16_rfft_128**[128]

const float16_t **twiddleCoefF16_rfft_256**[256]

const float16_t **twiddleCoefF16_rfft_512**[512]

const float16_t **twiddleCoefF16_rfft_1024**[1024]

const float16_t **twiddleCoefF16_rfft_2048**[2048]

const float16_t **twiddleCoefF16_rfft_4096**[4096]

group **CFFT_CIFFT**

Variables

const uint16_t **riscvBitRevTable**[1024]

Table for bit reversal process.

Pseudo code for Generation of Bit reversal Table is

where N = 4096, logN2 = 12

N is the maximum FFT Size supported

const uint64_t **twiddleCoefF64_16**[32]

Double Precision Floating-point Twiddle factors Table Generation.

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 16, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_32**[64]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 32, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_64**[128]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 64, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_128**[256]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 128, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_256**[512]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 256, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_512**[1024]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 512, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_1024**[2048]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 1024, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_2048**[4096]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 2048, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const uint64_t **twiddleCoefF64_4096**[8192]

Example code for Double Precision Floating-point Twiddle factors Generation:

where N = 4096, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_16**[32]

Example code for Floating-point Twiddle factors Generation:

where N = 16, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_32**[64]

Example code for Floating-point Twiddle factors Generation:

where N = 32, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_64**[128]

Example code for Floating-point Twiddle factors Generation:

where N = 64, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_128**[256]

Example code for Floating-point Twiddle factors Generation:

where N = 128, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_256**[512]

Example code for Floating-point Twiddle factors Generation:

where N = 256, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_512**[1024]

Example code for Floating-point Twiddle factors Generation:

where N = 512, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_1024**[2048]

Example code for Floating-point Twiddle factors Generation:

where N = 1024, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_2048**[4096]

Example code for Floating-point Twiddle factors Generation:

where N = 2048, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float32_t **twiddleCoef_4096**[8192]

Example code for Floating-point Twiddle factors Generation:

where N = 4096, PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const q31_t **twiddleCoef_16_q31**[24]
Q31 Twiddle factors Table.

Example code for Q31 Twiddle factors Generation::

where N = 16, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_32_q31**[48]

Example code for Q31 Twiddle factors Generation::

where N = 32, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_64_q31**[96]

Example code for Q31 Twiddle factors Generation::

where N = 64, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_128_q31**[192]

Example code for Q31 Twiddle factors Generation::

where N = 128, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_256_q31**[384]

Example code for Q31 Twiddle factors Generation::

where N = 256, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_512_q31**[768]

Example code for Q31 Twiddle factors Generation::

where N = 512, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_1024_q31**[1536]

Example code for Q31 Twiddle factors Generation::

where N = 1024, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_2048_q31**[3072]

Example code for Q31 Twiddle factors Generation::

where N = 2048, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q31_t **twiddleCoef_4096_q31**[6144]

Example code for Q31 Twiddle factors Generation::

where N = 4096, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to Q31(Fixed point 1.31): round(twiddleCoefQ31(i) * pow(2, 31))

const q15_t **twiddleCoef_16_q15**[24]

q15 Twiddle factors Table

Example code for q15 Twiddle factors Generation::

where N = 16, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_32_q15**[48]

Example code for q15 Twiddle factors Generation::

where N = 32, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_64_q15**[96]

Example code for q15 Twiddle factors Generation::

where N = 64, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_128_q15**[192]

Example code for q15 Twiddle factors Generation::

where N = 128, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_256_q15**[384]

Example code for q15 Twiddle factors Generation::

where N = 256, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_512_q15**[768]

Example code for q15 Twiddle factors Generation::

where N = 512, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_1024_q15**[1536]

Example code for q15 Twiddle factors Generation::

where N = 1024, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_2048_q15**[3072]

Example code for q15 Twiddle factors Generation::

where N = 2048, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const q15_t **twiddleCoef_4096_q15**[6144]

Example code for q15 Twiddle factors Generation::

where N = 4096, PI = 3.14159265358979

Cos and Sin values are interleaved fashion

Convert Floating point to q15(Fixed point 1.15): round(twiddleCoefq15(i) * pow(2, 15))

const float16_t **twiddleCoefF16_16**[32]

Floating-point Twiddle factors Table Generation.

Example code for Floating-point Twiddle factors Generation:

where N = 16 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_32**[64]

Example code for Floating-point Twiddle factors Generation:

where N = 32 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_64**[128]

Example code for Floating-point Twiddle factors Generation:

where N = 64 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_128**[256]

Example code for Floating-point Twiddle factors Generation:

where N = 128 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_256**[512]

Example code for Floating-point Twiddle factors Generation:

where N = 256 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_512**[1024]

Example code for Floating-point Twiddle factors Generation:

where N = 512 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_1024**[2048]

Example code for Floating-point Twiddle factors Generation:

where N = 1024 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_2048**[4096]

Example code for Floating-point Twiddle factors Generation:

where N = 2048 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_4096**[8192]

Example code for Floating-point Twiddle factors Generation:

where N = 4096 and PI = 3.14159265358979

Cos and Sin values are in interleaved fashion

const float16_t **twiddleCoefF16_rfft_32**[32]

Example code for Floating-point RFFT Twiddle factors Generation:

Real and Imag values are in interleaved fashion

const float16_t **twiddleCoefF16_rfft_64**[64]

const float16_t **twiddleCoefF16_rfft_128**[128]

const float16_t **twiddleCoefF16_rfft_256**[256]

const float16_t **twiddleCoefF16_rfft_512**[512]

const float16_t **twiddleCoefF16_rfft_1024**[1024]

const float16_t **twiddleCoefF16_rfft_2048**[2048]

const float16_t **twiddleCoefF16_rfft_4096**[4096]

void **riscv_cfft_f16** (**const** riscv_cfft_instance_f16 **S*, float16_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_f32** (**const** riscv_cfft_instance_f32 **S*, float32_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_f64** (**const** riscv_cfft_instance_f64 **S*, float64_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_init_f16** (riscv_cfft_instance_f16 **S*, uint16_t *fftLen*)

riscv_status **riscv_cfft_init_f32** (riscv_cfft_instance_f32 **S*, uint16_t *fftLen*)

riscv_status **riscv_cfft_init_f64** (riscv_cfft_instance_f64 **S*, uint16_t *fftLen*)

riscv_status **riscv_cfft_init_q15** (riscv_cfft_instance_q15 **S*, uint16_t *fftLen*)

riscv_status **riscv_cfft_init_q31** (riscv_cfft_instance_q31 **S*, uint16_t *fftLen*)

void **riscv_cfft_q15** (**const** riscv_cfft_instance_q15 **S*, q15_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_q31** (**const** riscv_cfft_instance_q31 **S*, q31_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_radix2_f16** (**const** riscv_cfft_radix2_instance_f16 **S*, float16_t **pSrc*)

void **riscv_cfft_radix2_f32** (**const** riscv_cfft_radix2_instance_f32 **S*, float32_t **pSrc*)

riscv_status **riscv_cfft_radix2_init_f16** (riscv_cfft_radix2_instance_f16 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix2_init_f32** (riscv_cfft_radix2_instance_f32 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix2_init_q15** (riscv_cfft_radix2_instance_q15 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix2_init_q31** (riscv_cfft_radix2_instance_q31 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_radix2_q15** (**const** riscv_cfft_radix2_instance_q15 **S*, q15_t **pSrc*)

void **riscv_cfft_radix2_q31** (**const** riscv_cfft_radix2_instance_q31 **S*, q31_t **pSrc*)

void **riscv_cfft_radix4by2_f16** (float16_t **pSrc*, uint32_t *fftLen*, **const** float16_t **pCoef*)

void **riscv_cfft_radix4_f16** (**const** riscv_cfft_radix4_instance_f16 **S*, float16_t **pSrc*)

void **riscv_cfft_radix4_f32** (**const** riscv_cfft_radix4_instance_f32 **S*, float32_t **pSrc*)

riscv_status **riscv_cfft_radix4_init_f16** (riscv_cfft_radix4_instance_f16 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix4_init_f32** (riscv_cfft_radix4_instance_f32 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix4_init_q15** (riscv_cfft_radix4_instance_q15 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

riscv_status **riscv_cfft_radix4_init_q31** (riscv_cfft_radix4_instance_q31 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

void **riscv_cfft_radix4_q15** (**const** riscv_cfft_radix4_instance_q15 **S*, q15_t **pSrc*)

void **riscv_cfft_radix4_q31** (**const** riscv_cfft_radix4_instance_q31 **S*, q31_t **pSrc*) *group* **ComplexFFT**

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT can be orders of magnitude faster than the DFT, especially for long lengths. The algorithms described in this section operate on complex data. A separate set of functions is devoted to handling of real sequences.

There are separate algorithms for handling floating-point, Q15, and Q31 data types. The algorithms available for each data type are described next.

The FFT functions operate in-place. That is, the array holding the input data will also be used to hold the corresponding result. The input data is complex and contains `2*fftLen` interleaved values as shown below. The FFT result will be contained in the same array and the frequency domain values will have the same interleaving.

Floating-point The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-8 stages are performed along with a single radix-2 or radix-4 stage, as needed. The algorithm supports lengths of [16, 32, 64, …, 4096] and each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may grow by a factor of `fftLen` when computing the forward transform. The inverse transform includes a scale of `1/fftLen` as part of the calculation and this matches the textbook definition of the inverse FFT.

For the MVE version, the new riscv_cfft_init_f32 initialization function is **mandatory**. **Compilation flags are available to include only the required tables for the needed FFTs.** Other FFT versions can continue to be initialized as explained below.

For not MVE versions, pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in `riscv_const_structs.h`. Include this header in your function and then pass one of the constant structures as an argument to riscv_cfft_f32. For example:

`riscv_cfft_f32(riscv_cfft_sR_f32_len64, pSrc, 1, 1)`

computes a 64-point inverse complex FFT including bit reversal. The data structures are treated as constant data and not modified during the calculation. The same data structure can be reused for multiple transforms including mixing forward and inverse transforms.

Earlier releases of the library provided separate radix-2 and radix-4 algorithms that operated on floating-point data. These functions are still provided but are deprecated. The older functions are slower and less general than the new functions.

An example of initialization of the constants for the riscv_cfft_f32 function follows:

```
const static riscv_cfft_instance_f32 *S;
...
  switch (length) {
    case 16:
      S = &riscv_cfft_sR_f32_len16;
      break;
    case 32:
      S = &riscv_cfft_sR_f32_len32;
      break;
    case 64:
      S = &riscv_cfft_sR_f32_len64;
      break;
    case 128:
      S = &riscv_cfft_sR_f32_len128;
      break;
    case 256:
      S = &riscv_cfft_sR_f32_len256;
      break;
```

(continues on next page)

(continued from previous page)

```
    case 512:
      S = &riscv_cfft_sR_f32_len512;
      break;
    case 1024:
      S = &riscv_cfft_sR_f32_len1024;
      break;
    case 2048:
      S = &riscv_cfft_sR_f32_len2048;
      break;
    case 4096:
      S = &riscv_cfft_sR_f32_len4096;
      break;
  }
```

The new riscv_cfft_init_f32 can also be used.

Q15 and Q31 The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-4 stages are performed along with a single radix-2 stage, as needed. The algorithm supports lengths of [16, 32, 64, ..., 4096] and each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may grow by a factor of `fftLen` when computing the forward transform. The inverse transform includes a scale of `1/fftLen` as part of the calculation and this matches the textbook definition of the inverse FFT.

Pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in `riscv_const_structs.h`. Include this header in your function and then pass one of the constant structures as an argument to riscv_cfft_q31. For example:

`riscv_cfft_q31(riscv_cfft_sR_q31_len64, pSrc, 1, 1)`

computes a 64-point inverse complex FFT including bit reversal. The data structures are treated as constant data and not modified during the calculation. The same data structure can be reused for multiple transforms including mixing forward and inverse transforms.

Earlier releases of the library provided separate radix-2 and radix-4 algorithms that operated on floating-point data. These functions are still provided but are deprecated. The older functions are slower and less general than the new functions.

An example of initialization of the constants for the riscv_cfft_q31 function follows:

```
const static riscv_cfft_instance_q31 *S;
...
  switch (length) {
    case 16:
      S = &riscv_cfft_sR_q31_len16;
      break;
    case 32:
      S = &riscv_cfft_sR_q31_len32;
      break;
    case 64:
      S = &riscv_cfft_sR_q31_len64;
      break;
    case 128:
      S = &riscv_cfft_sR_q31_len128;
      break;
    case 256:
      S = &riscv_cfft_sR_q31_len256;
      break;
```

(continues on next page)

(continued from previous page)

```
    case 512:
      S = &riscv_cfft_sR_q31_len512;
      break;
    case 1024:
      S = &riscv_cfft_sR_q31_len1024;
      break;
    case 2048:
      S = &riscv_cfft_sR_q31_len2048;
      break;
    case 4096:
      S = &riscv_cfft_sR_q31_len4096;
      break;
  }
```

Functions

void **riscv_cfft_f16** (**const** riscv_cfft_instance_f16 **S*, float16_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Processing function for the floating-point complex FFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point CFFT structure
- `[inout] p1`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_cfft_f32** (**const** riscv_cfft_instance_f32 **S*, float32_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Processing function for the floating-point complex FFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point CFFT structure
- `[inout] p1`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output

– value = 1: enables bit reversal of output

void **riscv_cfft_f64** (**const** riscv_cfft_instance_f64 **S*, float64_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Processing function for the Double Precision floating-point complex FFT.

Return none

Parameters

- `[in] S`: points to an instance of the Double Precision floating-point CFFT structure
- `[inout] p1`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place
- `[in] ifftFlag`: flag that selects transform direction
  – value = 0: forward transform
  – value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  – value = 0: disables bit reversal of output
  – value = 1: enables bit reversal of output

riscv_status **riscv_cfft_init_f16** (riscv_cfft_instance_f16 **S*, uint16_t *fftLen*)

Initialization function for the cfft f16 function.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Use of this function is mandatory only for the MVE version of the FFT. Other versions can still initialize directly the data structure using variables declared in riscv_const_structs.h

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT structure
- `[in] fftLen`: fft length (number of complex samples)

riscv_status **riscv_cfft_init_f32** (riscv_cfft_instance_f32 **S*, uint16_t *fftLen*)

Initialization function for the cfft f32 function.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Use of this function is mandatory only for the MVE version of the FFT. Other versions can still initialize directly the data structure using variables declared in riscv_const_structs.h

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT structure
- `[in] fftLen`: fft length (number of complex samples)

riscv_status **riscv_cfft_init_f64** (riscv_cfft_instance_f64 **S*, uint16_t *fftLen*)

Initialization function for the cfft f64 function.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Use of this function is mandatory only for the MVE version of the FFT. Other versions can still initialize directly the data structure using variables declared in riscv_const_structs.h

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT structure
- `[in] fftLen`: fft length (number of complex samples)

riscv_status **`riscv_cfft_init_q15`** (riscv_cfft_instance_q15 **S*, uint16_t *fftLen*)

Initialization function for the cfft q15 function.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Use of this function is mandatory only for the MVE version of the FFT. Other versions can still initialize directly the data structure using variables declared in riscv_const_structs.h

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT structure
- `[in] fftLen`: fft length (number of complex samples)

riscv_status **`riscv_cfft_init_q31`** (riscv_cfft_instance_q31 **S*, uint16_t *fftLen*)

Initialization function for the cfft q31 function.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Use of this function is mandatory only for the MVE version of the FFT. Other versions can still initialize directly the data structure using variables declared in riscv_const_structs.h

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT structure
- `[in] fftLen`: fft length (number of complex samples)

void **`riscv_cfft_q15`** (**`const`** riscv_cfft_instance_q15 **S*, q15_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Processing function for Q15 complex FFT.

Return none

Parameters

- `[in] S`: points to an instance of Q15 CFFT structure
- `[inout] p1`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform

  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_cfft_q31** (**const** riscv_cfft_instance_q31 **S*, q31_t **p1*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Processing function for the Q31 complex FFT.

Return none

Parameters

- `[in] S`: points to an instance of the fixed-point CFFT structure
- `[inout] p1`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_cfft_radix2_f16** (**const** riscv_cfft_radix2_instance_f16 **S*, float16_t **pSrc*)

Radix-2 CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Radix-2 CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

void **riscv_cfft_radix2_f32** (**const** riscv_cfft_radix2_instance_f32 **S*, float32_t **pSrc*)

Radix-2 CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Radix-2 CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

riscv_status **riscv_cfft_radix2_init_f16** (riscv_cfft_radix2_instance_f16 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the floating-point CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful

- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT/CIFFT structure
- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **`riscv_cfft_radix2_init_f32`** (riscv_cfft_radix2_instance_f32 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the floating-point CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT/CIFFT structure
- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output

– value = 1: enables bit reversal of output

riscv_status **riscv_cfft_radix2_init_q15** (riscv_cfft_radix2_instance_q15 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the Q15 CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the Q15 CFFT/CIFFT structure.
- `[in] fftLen`: length of the FFT.
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **riscv_cfft_radix2_init_q31** (riscv_cfft_radix2_instance_q31 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the Q31 CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the Q31 CFFT/CIFFT structure

- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_cfft_radix2_q15**(**const** riscv_cfft_radix2_instance_q15 **S*, q15_t **pSrc*)
Processing function for the fixed-point CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the fixed-point CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

void **riscv_cfft_radix2_q31**(**const** riscv_cfft_radix2_instance_q31 **S*, q31_t **pSrc*)
Processing function for the fixed-point CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the fixed-point CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

void **riscv_cfft_radix4by2_f16**(float16_t **pSrc*, uint32_t *fftLen*, **const** float16_t **pCoef*)

void **riscv_cfft_radix4_f16**(**const** riscv_cfft_radix4_instance_f16 **S*, float16_t **pSrc*)
Processing function for the floating-point Radix-4 CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Radix-4 CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

void **riscv_cfft_radix4_f32**(**const** riscv_cfft_radix4_instance_f32 **S*, float32_t **pSrc*)
Processing function for the floating-point Radix-4 CFFT/CIFFT.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point Radix-4 CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

riscv_status **riscv_cfft_radix4_init_f16** (riscv_cfft_radix4_instance_f16 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the floating-point CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT/CIFFT structure
- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **riscv_cfft_radix4_init_f32** (riscv_cfft_radix4_instance_f32 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the floating-point CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the floating-point CFFT/CIFFT structure
- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction

- value = 0: forward transform
- value = 1: inverse transform

- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **`riscv_cfft_radix4_init_q15`** (riscv_cfft_radix4_instance_q15 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the Q15 CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the Q15 CFFT/CIFFT structure
- `[in] fftLen`: length of the FFT
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **`riscv_cfft_radix4_init_q31`** (riscv_cfft_radix4_instance_q31 **S*, uint16_t *fftLen*, uint8_t *ifftFlag*, uint8_t *bitReverseFlag*)

Initialization function for the Q31 CFFT/CIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Details The parameter `ifftFlag` controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT otherwise CFFT is calculated

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

The parameter `fftLen` Specifies length of CFFT/CIFFT process. Supported FFT Lengths are 16, 64, 256, 1024.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an instance of the Q31 CFFT/CIFFT structure.
- `[in] fftLen`: length of the FFT.
- `[in] ifftFlag`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_cfft_radix4_q15**(**const** riscv_cfft_radix4_instance_q15 **S*, q15_t **pSrc*)

Processing function for the Q15 CFFT/CIFFT.

Return none

Input and output formats: Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different FFT sizes. The input and output formats for different FFT sizes and number of bits to upscale are mentioned in the tables below for CFFT and CIFFT:

CFFT Size	Input format	Output Format	Number of bits to upscale
16	1.15	5.11	4
64	1.15	7.9	6
256	1.15	9.7	8
1024	1.15	11.5	10

CIFFT Size
16
64
256
1024

Parameters

- `[in] S`: points to an instance of the Q15 CFFT/CIFFT structure.
- `[inout] pSrc`: points to the complex data buffer. Processing occurs in-place.

void **riscv_cfft_radix4_q31**(**const** riscv_cfft_radix4_instance_q31 **S*, q31_t **pSrc*)

Processing function for the Q31 CFFT/CIFFT.

Return none

Input and output formats: Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different FFT sizes. The input and output formats for different FFT sizes and number of bits to upscale are mentioned in the tables below for CFFT and CIFFT:

CFFT Size	Input format	Output Format	Number of bits to upscale
16	1.31	5.27	4
64	1.31	7.25	6
256	1.31	9.23	8
1024	1.31	11.21	10

CIFFT Size
16
64
256
1024

Parameters

- `[in] S`: points to an instance of the Q31 CFFT/CIFFT structure
- `[inout] pSrc`: points to the complex data buffer of size `2*fftLen`. Processing occurs in-place

DCT Type IV Functions

DCT Type IV Tables

const float32_t **Weights_128**[256]

const float32_t **cos_factors_128**[128]

const float32_t **Weights_512**[1024]

const float32_t **cos_factors_512**[512]

const float32_t **Weights_2048**[4096]

const float32_t **cos_factors_2048**[2048]

const float32_t **Weights_8192**[16384]

const float32_t **cos_factors_8192**[8192]

const q31_t **WeightsQ31_128**[256]

const q31_t **cos_factorsQ31_128**[128]

const q31_t **WeightsQ31_512**[1024]

const q31_t **cos_factorsQ31_512**[512]

const q31_t **WeightsQ31_2048**[4096]

const q31_t **cos_factorsQ31_2048**[2048]

const q31_t **WeightsQ31_8192**[16384]

const q31_t **cos_factorsQ31_8192**[8192]

group **DCT4_IDCT4_Table**

end of RealFFT_Table group

Variables

const float32_t **Weights_128**[256]

Weights Table.

Weights tables are generated using the formula :

C command to generate the table

where `N` is the Number of weights to be calculated and `c` is `pi/(2*N)`

In the tables below the real and imaginary values are placed alternatively, hence the array length is `2*N`.

cosFactor tables are generated using the formula :

C command to generate the table

where `N` is the number of factors to generate and `c` is `pi/(2*N)`

const float32_t **cos_factors_128**[128]

const float32_t **Weights_512**[1024]

const float32_t **cos_factors_512**[512]

const float32_t **Weights_2048**[4096]

const float32_t **cos_factors_2048**[2048]

const float32_t **Weights_8192**[16384]

const float32_t **cos_factors_8192**[8192]

const q31_t **WeightsQ31_128**[256]

Weights tables are generated using the formula :

C command to generate the table

where `N` is the Number of weights to be calculated and `c` is `pi/(2*N)`

Convert the output to q31 format by multiplying with 2^31 and saturated if required.

In the tables below the real and imaginary values are placed alternatively, hence the array length is `2*N`.

cosFactor tables are generated using the formula :

C command to generate the table

where `N` is the number of factors to generate and `c` is `pi/(2*N)`

Then converted to q31 format by multiplying with 2^31 and saturated if required.

const q31_t **cos_factorsQ31_128**[128]

const q31_t **WeightsQ31_512**[1024]

const q31_t **cos_factorsQ31_512**[512]

const q31_t **WeightsQ31_2048**[4096]

const q31_t **cos_factorsQ31_2048**[2048]

const q31_t **WeightsQ31_8192**[16384]

const q31_t **cos_factorsQ31_8192**[8192]

void **riscv_dct4_f32** (**const** riscv_dct4_instance_f32 **S*, float32_t **pState*, float32_t **pInlineBuffer*)

riscv_status **riscv_dct4_init_f32** (riscv_dct4_instance_f32 **S*, riscv_rfft_instance_f32 **S_RFFT*, riscv_cfft_radix4_instance_f32 **S_CFFT*, uint16_t *N*, uint16_t *Nby2*, float32_t *normalize*)

riscv_status **riscv_dct4_init_q15** (riscv_dct4_instance_q15 **S*, riscv_rfft_instance_q15 **S_RFFT*, riscv_cfft_radix4_instance_q15 **S_CFFT*, uint16_t *N*, uint16_t *Nby2*, q15_t *normalize*) riscv_status **riscv_dct4_init_q31** (riscv_dct4_instance_q31 *S, riscv_rfft_instance_q31 *S_RFFT, riscv_cfft_radix4_instance_q31 *S_CFFT, uint16_t N, uint16_t Nby2, q31_t normalize)

void **riscv_dct4_q15** (**const** riscv_dct4_instance_q15 *S, q15_t *pState, q15_t *pInlineBuffer)

void **riscv_dct4_q31** (**const** riscv_dct4_instance_q31 *S, q31_t *pState, q31_t *pInlineBuffer)

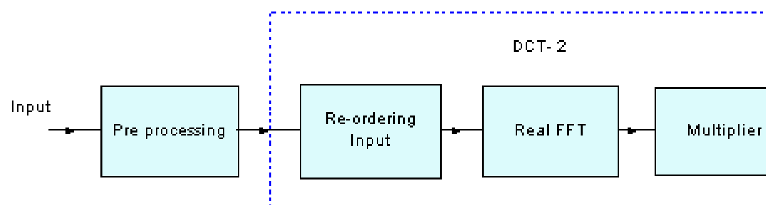
group **DCT4_IDCT4**

Representation of signals by minimum number of values is important for storage and transmission. The possibility of large discontinuity between the beginning and end of a period of a signal in DFT can be avoided by extending the signal so that it is even-symmetric. Discrete Cosine Transform (DCT) is constructed such that its energy is heavily concentrated in the lower part of the spectrum and is very widely used in signal and image coding applications. The family of DCTs (DCT type- 1,2,3,4) is the outcome of different combinations of homogeneous boundary conditions. DCT has an excellent energy-packing capability, hence has many applications and in data compression in particular.

DCT is essentially the Discrete Fourier Transform(DFT) of an even-extended real signal. Reordering of the input data makes the computation of DCT just a problem of computing the DFT of a real signal with a few additional operations. This approach provides regular, simple, and very efficient DCT algorithms for practical hardware and software implementations.

DCT type-II can be implemented using Fast fourier transform (FFT) internally, as the transform is applied on real values, Real FFT can be used. DCT4 is implemented using DCT2 as their implementations are similar except with some added pre-processing and post-processing. DCT2 implementation can be described in the following steps:

- Re-ordering input
- Calculating Real FFT
- Multiplication of weights and Real FFT output and getting real part from the product.



This process is explained by the block diagram below:

Algorithm The N-point type-IV DCT is defined as a real, linear transformation by the formula: where `k = 0, 1, 2, ..., N-1`

$$X_c(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos\left[\left(n+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)\frac{\pi}{N}\right]$$

Its inverse is defined as follows: where `n = 0, 1, 2, ..., N-1`

$$x(n) = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X_c(k) \cos\left[\left(n+\frac{1}{2}\right)\left(k+\frac{1}{2}\right)\frac{\pi}{N}\right]$$

The DCT4 matrices become involutory (i.e. they are self-inverse) by multiplying with an overall scale factor of sqrt(2/N). The symmetry of the transform matrix indicates that the fast algorithms for the forward and inverse transform computation are identical. Note that the implementation of Inverse DCT4 and DCT4 is same, hence same process function can be used for both.

Lengths supported by the transform: As DCT4 internally uses Real FFT, it supports all the lengths 128, 512, 2048 and 8192. The library provides separate functions for Q15, Q31, and floating-point data types.

Instance Structure The instances for Real FFT and FFT, cosine values table and twiddle factor table are stored in an instance data structure. A separate instance structure must be defined for each transform. There are separate instance structure declarations for each of the 3 supported data types.

Initialization Functions There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Initializes Real FFT as its process function is used internally in DCT4, by calling riscv_rfft_init_f32().

Use of the initialization function is optional. However, if the initialization function is used, then the instance structure cannot be placed into a const data section. To place an instance structure into a const data section, the instance structure must be manually initialized. Manually initialize the instance structure as follows: where `N` is the length of the DCT4; `Nby2` is half of the length of the DCT4; `normalize` is normalizing factor used and is equal to `sqrt(2/N)`; `pTwiddle` points to the twiddle factor table; `pCosFactor` points to the cosFactor table; `pRfft` points to the real FFT instance; `pCfft` points to the complex FFT instance; The CFFT and RFFT structures also needs to be initialized, refer to riscv_cfft_radix4_f32() and riscv_rfft_f32() respectively for details regarding static initialization.

Fixed-Point Behavior Care must be taken when using the fixed-point versions of the DCT4 transform functions. In particular, the overflow and saturation behavior of the accumulator used in each function must be considered. Refer to the function specific documentation below for usage guidelines.

Functions

void **riscv_dct4_f32**(**const** riscv_dct4_instance_f32 **S*, float32_t **pState*, float32_t **pInlineBuffer*)

Processing function for the floating-point DCT4/IDCT4.

Return none

Parameters

- `[in] S`: points to an instance of the floating-point DCT4/IDCT4 structure
- `[in] pState`: points to state buffer
- `[inout] pInlineBuffer`: points to the in-place input and output buffer

riscv_status **riscv_dct4_init_f32**(riscv_dct4_instance_f32 **S*, riscv_rfft_instance_f32 **S_RFFT*, riscv_cfft_radix4_instance_f32 **S_CFFT*, uint16_t *N*, uint16_t *Nby2*, float32_t *normalize*)

Initialization function for the floating-point DCT4/IDCT4.

DCT Size	Normalizing factor value
2048	0.03125
512	0.0625
128	0.125

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `N` is not a supported transform length

Normalizing factor The normalizing factor is `sqrt(2/N)`, which depends on the size of transform `N`. Floating-point normalizing factors are mentioned in the table below for different DCT sizes:

Parameters

- `[inout] S`: points to an instance of floating-point DCT4/IDCT4 structure
- `[in] S_RFFT`: points to an instance of floating-point RFFT/RIFFT structure
- `[in] S_CFFT`: points to an instance of floating-point CFFT/CIFFT structure
- `[in] N`: length of the DCT4
- `[in] Nby2`: half of the length of the DCT4
- `[in] normalize`: normalizing factor.

riscv_status **riscv_dct4_init_q15**(riscv_dct4_instance_q15 **S*, riscv_rfft_instance_q15 **S_RFFT*, riscv_cfft_radix4_instance_q15 **S_CFFT*, uint16_t *N*, uint16_t *Nby2*, q15_t *normalize*)

Initialization function for the Q15 DCT4/IDCT4.

DCT Size	Normalizing factor value (hexadecimal)
2048	0x400
512	0x800
128	0x1000

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `N` is not a supported transform length

Normalizing factor The normalizing factor is `sqrt(2/N)`, which depends on the size of transform `N`. Normalizing factors in 1.15 format are mentioned in the table below for different DCT sizes:

Parameters

- `[inout] S`: points to an instance of Q15 DCT4/IDCT4 structure
- `[in] S_RFFT`: points to an instance of Q15 RFFT/RIFFT structure
- `[in] S_CFFT`: points to an instance of Q15 CFFT/CIFFT structure
- `[in] N`: length of the DCT4
- `[in] Nby2`: half of the length of the DCT4
- `[in] normalize`: normalizing factor

riscv_status **riscv_dct4_init_q31** (riscv_dct4_instance_q31 **S*, riscv_rfft_instance_q31 **S_RFFT*, riscv_cfft_radix4_instance_q31 **S_CFFT*, uint16_t *N*, uint16_t *Nby2*, q31_t *normalize*)

Initialization function for the Q31 DCT4/IDCT4.

DCT Size	Normalizing factor value (hexadecimal)
2048	0x4000000
512	0x8000000
128	0x10000000

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `N` is not a supported transform length

Normalizing factor: The normalizing factor is `sqrt(2/N)`, which depends on the size of transform `N`. Normalizing factors in 1.31 format are mentioned in the table below for different DCT sizes:

Parameters

- `[inout] S`: points to an instance of Q31 DCT4/IDCT4 structure.
- `[in] S_RFFT`: points to an instance of Q31 RFFT/RIFFT structure
- `[in] S_CFFT`: points to an instance of Q31 CFFT/CIFFT structure
- `[in] N`: length of the DCT4.
- `[in] Nby2`: half of the length of the DCT4.
- `[in] normalize`: normalizing factor.

void **riscv_dct4_q15** (**const** riscv_dct4_instance_q15 **S*, q15_t **pState*, q15_t **pInlineBuffer*)

Processing function for the Q15 DCT4/IDCT4.

DCT Size	Input format	Output format	Number of bits to upscale
2048	1.15	11.5	10
512	1.15	9.7	8
128	1.15	7.9	6

Return none

Input an output formats Internally inputs are downscaled in the RFFT process function to avoid overflows. Number of bits downscaled, depends on the size of the transform. The input and output formats for different DCT sizes and number of bits to upscale are mentioned in the table below:

Parameters

- `[in] S`: points to an instance of the Q15 DCT4 structure.
- `[in] pState`: points to state buffer.

- `[inout] pInlineBuffer`: points to the in-place input and output buffer.

void **riscv_dct4_q31** (**const** riscv_dct4_instance_q31 **S*, q31_t **pState*, q31_t **pInlineBuffer*)

Processing function for the Q31 DCT4/IDCT4.

DCT Size	Input format	Output format	Number of bits to upscale
2048	2.30	12.20	11
512	2.30	10.22	9
128	2.30	8.24	7

Return none

Input an output formats Input samples need to be downscaled by 1 bit to avoid saturations in the Q31 DCT process, as the conversion from DCT2 to DCT4 involves one subtraction. Internally inputs are downscaled in the RFFT process function to avoid overflows. Number of bits downscaled, depends on the size of the transform. The input and output formats for different DCT sizes and number of bits to upscale are mentioned in the table below:

Parameters

- `[in] S`: points to an instance of the Q31 DCT4 structure.
- `[in] pState`: points to state buffer.
- `[inout] pInlineBuffer`: points to the in-place input and output buffer.

Real FFT Functions

Real FFT Tables

const float32_t **realCoefA**[8192]

const float32_t **realCoefB**[8192]

const q31_t **realCoefAQ31**[8192]

const q31_t **realCoefBQ31**[8192]

const q15_t __ALIGNED(4)

group **RealFFT_Table**

Functions

const q15_t __ALIGNED(4)

Weights Table.

Q15 table for reciprocal.

end of DCT4_IDCT4_Table group

Generation fixed-point realCoefAQ15 array in Q15 format:

n = 4096

Convert to fixed point Q15 format round(pATable[i] * pow(2, 15))

Generation of real_CoefB array:

n = 4096

Convert to fixed point Q15 format round(pBTable[i] * pow(2, 15))

Weights tables are generated using the formula :

C command to generate the table

where `N` is the Number of weights to be calculated and `c` is `pi/(2*N)`

Converted the output to q15 format by multiplying with 2^31 and saturated if required.

In the tables below the real and imaginary values are placed alternatively, hence the array length is `2*N`.

cosFactor tables are generated using the formula :

C command to generate the table

where `N` is the number of factors to generate and `c` is `pi/(2*N)`

Then converted to q15 format by multiplying with 2^31 and saturated if required.

Variables

const float32_t **realCoefA**[8192]

Generation of realCoefA array:

n = 4096

const float32_t **realCoefB**[8192]

Generation of realCoefB array:

n = 4096

const q31_t **realCoefAQ31**[8192]

Generation fixed-point realCoefAQ31 array in Q31 format:

n = 4096

Convert to fixed point Q31 format round(pATable[i] * pow(2, 31))

const q31_t **realCoefBQ31**[8192]

Generation of realCoefBQ31 array:

n = 4096

Convert to fixed point Q31 format round(pBTable[i] * pow(2, 31))

void **riscv_rfft_f32** (**const** riscv_rfft_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*)

void **riscv_rfft_fast_f16** (**const** riscv_rfft_fast_instance_f16 **S*, float16_t **p*, float16_t **pOut*, uint8_t *ifftFlag*)

void **riscv_rfft_fast_f32** (**const** riscv_rfft_fast_instance_f32 **S*, float32_t **p*, float32_t **pOut*, uint8_t *ifftFlag*)

void **riscv_rfft_fast_f64** (riscv_rfft_fast_instance_f64 **S*, float64_t **p*, float64_t **pOut*, uint8_t *ifftFlag*)

riscv_status **riscv_rfft_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*, uint16_t *fftLen*)

riscv_status **riscv_rfft_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*, uint16_t *fftLen*)

static riscv_status **riscv_rfft_32_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_64_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_128_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_256_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_512_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_1024_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_2048_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

static riscv_status **riscv_rfft_4096_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)

riscv_status **riscv_rfft_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*, uint16_t *fftLen*)

riscv_status **riscv_rfft_init_f32** (riscv_rfft_instance_f32 **S*, riscv_cfft_radix4_instance_f32 **S_CFFT*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

riscv_status **riscv_rfft_init_q15** (riscv_rfft_instance_q15 **S*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

riscv_status **riscv_rfft_init_q31** (riscv_rfft_instance_q31 **S*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

void **riscv_rfft_q15** (**const** riscv_rfft_instance_q15 **S*, q15_t **pSrc*, q15_t **pDst*)

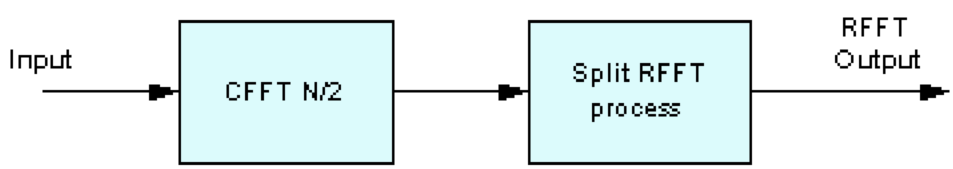
void **riscv_rfft_q31** (**const** riscv_rfft_instance_q31 **S*, q31_t **pSrc*, q31_t **pDst*)

group **RealFFT**

The NMSIS DSP library includes specialized algorithms for computing the FFT of real data sequences. The FFT is defined over complex data but in many applications the input is real. Real FFT algorithms take advantage of the symmetry properties of the FFT and have a speed advantage over complex algorithms of the same length.

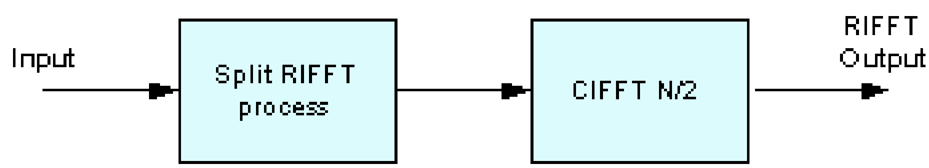
The Fast RFFT algorithm relays on the mixed radix CFFT that save processor usage.

The real length N forward FFT of a sequence is computed using the steps shown below.



The real sequence is initially treated as if it were complex to perform a CFFT. Later, a processing stage reshapes the data to obtain half of the frequency spectrum in complex format. Except the first complex number that contains the two real numbers X[0] and X[N/2] all the data is complex. In other words, the first complex sample contains two real values packed.

The input for the inverse RFFT should keep the same format as the output of the forward RFFT. A first processing stage pre-process the data to later perform an inverse CFFT.



The algorithms for floating-point, Q15, and Q31 data are slightly different and we describe each algorithm in turn.

Floating-point The main functions are riscv_rfft_fast_f16() and riscv_rfft_fast_init_f16().

The FFT of a real N-point sequence has even symmetry in the frequency domain. The second half of the data equals the conjugate of the first half flipped in frequency. Looking at the data, we see that we can uniquely represent the FFT using only N/2 complex numbers. These are packed into the output array in alternating real and imaginary components:

X = { real[0], imag[0], real[1], imag[1], real[2], imag[2] … real[(N/2)-1], imag[(N/2)-1 }

It happens that the first complex number (real[0], imag[0]) is actually all real. real[0] represents the DC offset, and imag[0] should be 0. (real[1], imag[1]) is the fundamental frequency, (real[2], imag[2]) is the first harmonic and so on.

The real FFT functions pack the frequency domain data in this fashion. The forward transform outputs the data in this form and the inverse transform expects input data in this form. The function always performs the needed bitreversal so that the input and output data is always in normal order. The functions support lengths of [32, 64, 128, …, 4096] samples.

Q15 and Q31 The real algorithms are defined in a similar manner and utilize N/2 complex transforms behind the scenes.

The complex transforms used internally include scaling to prevent fixed-point overflows. The overall scaling equals 1/(fftLen/2). Due to the use of complex transform internally, the source buffer is modified by the rfft.

A separate instance structure must be defined for each transform used but twiddle factor and bit reversal tables can be reused.

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Initializes twiddle factor table and bit reversal table pointers.
- Initializes the internal complex FFT data structure.

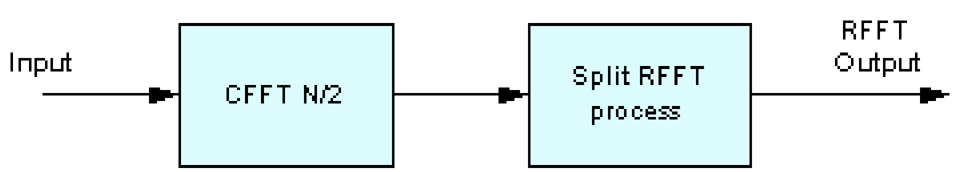
Use of the initialization function is optional **except for MVE versions where it is mandatory**. If you don't use the initialization functions, then the structures should be initialized with code similar to the one below: where `fftLenReal` is the length of the real transform; `fftLenBy2` length of the internal complex transform (fftLenReal/2). `ifftFlagR` Selects forward (=0) or inverse (=1) transform. `bitReverseFlagR` Selects bit reversed output (=0) or normal order output (=1). `twidCoefRModifier` stride modifier for the twiddle factor table. The value is based on the FFT length; `pTwiddleAReal`points to the A array of twiddle coefficients; `pTwiddleBReal`points to the B array of twiddle coefficients; `pCfft` points to the CFFT Instance structure. The CFFT structure must also be initialized.

Note that with MVE versions you can't initialize instance structures directly and **must use the initialization function**.

The NMSIS DSP library includes specialized algorithms for computing the FFT of real data sequences. The FFT is defined over complex data but in many applications the input is real. Real FFT algorithms take advantage of the symmetry properties of the FFT and have a speed advantage over complex algorithms of the same length.

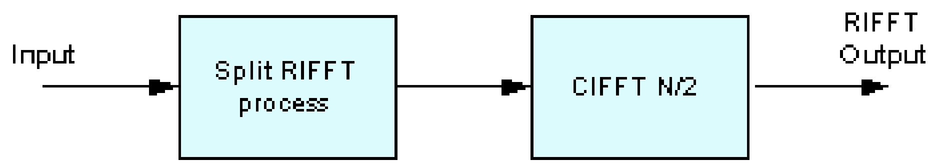
The Fast RFFT algorithm relays on the mixed radix CFFT that save processor usage.

The real length N forward FFT of a sequence is computed using the steps shown below.



The real sequence is initially treated as if it were complex to perform a CFFT. Later, a processing stage reshapes the data to obtain half of the frequency spectrum in complex format. Except the first complex number that contains the two real numbers X[0] and X[N/2] all the data is complex. In other words, the first complex sample contains two real values packed.

The input for the inverse RFFT should keep the same format as the output of the forward RFFT. A first processing stage pre-process the data to later perform an inverse CFFT.



The algorithms for floating-point, Q15, and Q31 data are slightly different and we describe each algorithm in turn.

Floating-point The main functions are riscv_rfft_fast_f32() and riscv_rfft_fast_init_f32(). The older functions riscv_rfft_f32() and riscv_rfft_init_f32() have been deprecated but are still documented.

The FFT of a real N-point sequence has even symmetry in the frequency domain. The second half of the data equals the conjugate of the first half flipped in frequency. Looking at the data, we see that we can uniquely represent the FFT using only N/2 complex numbers. These are packed into the output array in alternating real and imaginary components:

X = { real[0], imag[0], real[1], imag[1], real[2], imag[2] … real[(N/2)-1], imag[(N/2)-1 }

It happens that the first complex number (real[0], imag[0]) is actually all real. real[0] represents the DC offset, and imag[0] should be 0. (real[1], imag[1]) is the fundamental frequency, (real[2], imag[2]) is the first harmonic and so on.

The real FFT functions pack the frequency domain data in this fashion. The forward transform outputs the data in this form and the inverse transform expects input data in this form. The function always performs the needed bitreversal so that the input and output data is always in normal order. The functions support lengths of [32, 64, 128, …, 4096] samples.

Q15 and Q31 The real algorithms are defined in a similar manner and utilize N/2 complex transforms behind the scenes.

The complex transforms used internally include scaling to prevent fixed-point overflows. The overall scaling equals 1/(fftLen/2). Due to the use of complex transform internally, the source buffer is modified by the rfft.

A separate instance structure must be defined for each transform used but twiddle factor and bit reversal tables can be reused.

There is also an associated initialization function for each data type. The initialization function performs the following operations:

- Sets the values of the internal structure fields.
- Initializes twiddle factor table and bit reversal table pointers.
- Initializes the internal complex FFT data structure.

Use of the initialization function is optional **except for MVE versions where it is mandatory**. If you don't use the initialization functions, then the structures should be initialized with code similar to the one below: where `fftLenReal` is the length of the real transform; `fftLenBy2` length of the internal complex transform (fftLenReal/2). `ifftFlagR` Selects forward (=0) or inverse (=1) transform. `bitReverseFlagR` Selects bit reversed output (=0) or normal order output (=1). `twidCoefRModifier` stride modifier for the twiddle factor table. The value is based on the FFT length; `pTwiddleAReal`points to the A array of twiddle coefficients; `pTwiddleBReal`points to the B array of twiddle coefficients; `pCfft` points to the CFFT Instance structure. The CFFT structure must also be initialized.

Note that with MVE versions you can't initialize instance structures directly and **must use the initialization function**.

Functions

void **riscv_rfft_f32**(**const** riscv_rfft_instance_f32 **S*, float32_t **pSrc*, float32_t **pDst*)

Processing function for the floating-point RFFT/RIFFT. Source buffer is modified by this function.

Return none

For the RIFFT, the source buffer must at least have length fftLenReal + 2. The last two elements must be equal to what would be generated by the RFFT: (pSrc[0] - pSrc[1]) and 0.0f

Parameters

- `[in] S`: points to an instance of the floating-point RFFT/RIFFT structure
- `[in] pSrc`: points to the input buffer
- `[out] pDst`: points to the output buffer

void **riscv_rfft_fast_f16**(**const** riscv_rfft_fast_instance_f16 **S*, float16_t **p*, float16_t **pOut*, uint8_t *ifftFlag*)

Processing function for the floating-point real FFT.

Return none

Parameters

- `[in] S`: points to an riscv_rfft_fast_instance_f16 structure
- `[in] p`: points to input buffer (Source buffer is modified by this function.)
- `[in] pOut`: points to output buffer
- `[in] ifftFlag`:
  - value = 0: RFFT
  - value = 1: RIFFT

void **riscv_rfft_fast_f32**(**const** riscv_rfft_fast_instance_f32 **S*, float32_t **p*, float32_t **pOut*, uint8_t *ifftFlag*)

Processing function for the floating-point real FFT.

Return none

Parameters

- `[in] S`: points to an riscv_rfft_fast_instance_f32 structure
- `[in] p`: points to input buffer (Source buffer is modified by this function.)
- `[in] pOut`: points to output buffer
- `[in] ifftFlag`:
  - value = 0: RFFT
  - value = 1: RIFFT

void **riscv_rfft_fast_f64** (riscv_rfft_fast_instance_f64 **S*, float64_t **p*, float64_t **pOut*, uint8_t *ifftFlag*)

Processing function for the Double Precision floating-point real FFT.

Return none

Parameters

- `[in] S`: points to an riscv_rfft_fast_instance_f64 structure
- `[in] p`: points to input buffer (Source buffer is modified by this function.)
- `[in] pOut`: points to output buffer
- `[in] ifftFlag`:
  - value = 0: RFFT
  - value = 1: RIFFT

static riscv_status **riscv_rfft_32_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)

Initialization function for the 32pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_64_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)

Initialization function for the 64pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_128_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)

Initialization function for the 128pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_256_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)

Initialization function for the 256pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_512_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)
Initialization function for the 512pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_1024_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)
Initialization function for the 1024pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_2048_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)
Initialization function for the 2048pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

static riscv_status **riscv_rfft_4096_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*)
Initialization function for the 4096pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure

riscv_status **riscv_rfft_fast_init_f16** (riscv_rfft_fast_instance_f16 **S*, uint16_t *fftLen*)
Initialization function for the floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Description The parameter `fftLen` specifies the length of RFFT/CIFFT process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f16 structure
- `[in] fftLen`: length of the Real Sequence

static riscv_status **riscv_rfft_32_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 32pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_64_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 64pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_128_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 128pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_256_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 256pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_512_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 512pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_1024_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 1024pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_2048_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 2048pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

static riscv_status **riscv_rfft_4096_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*)
Initialization function for the 4096pt floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure

riscv_status **riscv_rfft_fast_init_f32** (riscv_rfft_fast_instance_f32 **S*, uint16_t *fftLen*)
Initialization function for the floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful

- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Description The parameter `fftLen` specifies the length of RFFT/CIFFT process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f32 structure
- `[in] fftLen`: length of the Real Sequence

static riscv_status **riscv_rfft_32_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 32pt double precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_64_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 64pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_128_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 128pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_256_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 256pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_512_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 512pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_1024_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 1024pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_2048_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 2048pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

static riscv_status **riscv_rfft_4096_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*)
Initialization function for the 4096pt Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : an error is detected

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure

riscv_status **riscv_rfft_fast_init_f64** (riscv_rfft_fast_instance_f64 **S*, uint16_t *fftLen*)
Initialization function for the Double Precision floating-point real FFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLen` is not a supported length

Description The parameter `fftLen` specifies the length of RFFT/CIFFT process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096.

This Function also initializes Twiddle factor table pointer and Bit reversal table pointer.

Parameters

- `[inout] S`: points to an riscv_rfft_fast_instance_f64 structure
- `[in] fftLen`: length of the Real Sequence

riscv_status **riscv_rfft_init_f32** (riscv_rfft_instance_f32 **S*, riscv_cfft_radix4_instance_f32 **S_CFFT*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

Initialization function for the floating-point RFFT/RIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLenReal` is not a supported length

Description The parameter `fftLenReal`specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 128, 512, 2048.

The parameter `ifftFlagR` controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

Parameters

- `[inout] S`: points to an instance of the floating-point RFFT/RIFFT structure
- `[inout] S_CFFT`: points to an instance of the floating-point CFFT/CIFFT structure
- `[in] fftLenReal`: length of the FFT.
- `[in] ifftFlagR`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **riscv_rfft_init_q15** (riscv_rfft_instance_q15 **S*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

Initialization function for the Q15 RFFT/RIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLenReal` is not a supported length

Details The parameter `fftLenReal` specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

The parameter `ifftFlagR` controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

Parameters

- `[inout] S`: points to an instance of the Q15 RFFT/RIFFT structure
- `[in] fftLenReal`: length of the FFT
- `[in] ifftFlagR`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

riscv_status **riscv_rfft_init_q31** (riscv_rfft_instance_q31 **S*, uint32_t *fftLenReal*, uint32_t *ifftFlagR*, uint32_t *bitReverseFlag*)

Initialization function for the Q31 RFFT/RIFFT.

Return execution status

- RISCV_MATH_SUCCESS : Operation successful
- RISCV_MATH_ARGUMENT_ERROR : `fftLenReal` is not a supported length

Details The parameter `fftLenReal` specifies length of RFFT/RIFFT Process. Supported FFT Lengths are 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

The parameter `ifftFlagR` controls whether a forward or inverse transform is computed. Set(=1) ifftFlagR to calculate RIFFT, otherwise RFFT is calculated.

The parameter `bitReverseFlag` controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise output is in bit reversed order.

This function also initializes Twiddle factor table.

Parameters

- `[inout] S`: points to an instance of the Q31 RFFT/RIFFT structure
- `[in] fftLenReal`: length of the FFT
- `[in] ifftFlagR`: flag that selects transform direction
  - value = 0: forward transform
  - value = 1: inverse transform
- `[in] bitReverseFlag`: flag that enables / disables bit reversal of output
  - value = 0: disables bit reversal of output
  - value = 1: enables bit reversal of output

void **riscv_rfft_q15** (**const** riscv_rfft_instance_q15 **S*, q15_t **pSrc*, q15_t **pDst*)

Processing function for the Q15 RFFT/RIFFT.

Return none

Input an output formats Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different RFFT sizes. The input and output formats for different RFFT sizes and number of bits to upscale are mentioned in the tables below for RFFT and RIFFT:

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.15	5.11	4
64	1.15	6.10	5
128	1.15	7.9	6
256	1.15	8.8	7
512	1.15	9.7	8
1024	1.15	10.6	9
2048	1.15	11.5	10
4096	1.15	12.4	11
8192	1.15	13.3	12

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.15	5.11	0
64	1.15	6.10	0
128	1.15	7.9	0
256	1.15	8.8	0
512	1.15	9.7	0
1024	1.15	10.6	0
2048	1.15	11.5	0
4096	1.15	12.4	0
8192	1.15	13.3	0

If the input buffer is of length N, the output buffer must have length 2*N. The input buffer is modified by this function.

For the RIFFT, the source buffer must at least have length fftLenReal + 2. The last two elements must be equal to what would be generated by the RFFT: (pSrc[0] - pSrc[1]) >> 1 and 0

Parameters

- `[in] S`: points to an instance of the Q15 RFFT/RIFFT structure
- `[in] pSrc`: points to input buffer (Source buffer is modified by this function.)
- `[out] pDst`: points to output buffer

void **riscv_rfft_q31**(**const** riscv_rfft_instance_q31 **S*, q31_t **pSrc*, q31_t **pDst*)

Processing function for the Q31 RFFT/RIFFT.

Return none

Input an output formats Internally input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence the output format is different for different RFFT sizes. The input and output formats for different RFFT sizes and number of bits to upscale are mentioned in the tables below for RFFT and RIFFT:

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.31	5.27	4
64	1.31	6.26	5
128	1.31	7.25	6
256	1.31	8.24	7
512	1.31	9.23	8
1024	1.31	10.22	9
2048	1.31	11.21	10
4096	1.31	21.20	11
8192	1.31	13.19	12

RFFT Size	Input Format	Output Format	Number of bits to upscale
32	1.31	5.27	0
64	1.31	6.26	0
128	1.31	7.25	0
256	1.31	8.24	0
512	1.31	9.23	0
1024	1.31	10.22	0
2048	1.31	11.21	0
4096	1.31	12.20	0
8192	1.31	13.19	0

If the input buffer is of length N, the output buffer must have length 2*N. The input buffer is modified by this function.

For the RIFFT, the source buffer must at least have length fftLenReal + 2. The last two elements must be equal to what would be generated by the RFFT: (pSrc[0] - pSrc[1]) >> 1 and 0

Parameters

- `[in] S`: points to an instance of the Q31 RFFT/RIFFT structure
- `[in] pSrc`: points to input buffer (Source buffer is modified by this function)
- `[out] pDst`: points to output buffer

group **`groupTransforms`**

3.4 Changelog

3.4.1 V1.0.2

This is release `1.0.2` version of NMSIS-DSP library.

- Sync up to CMSIS DSP library 1.9.0
- Adding initial support for RISC-V vector extension support
- **Caution**: `riscv_math.h` is separated into several header files. Extra `PrivateInclude` folder is included as header folder.

3.4.2 V1.0.1

This is release `V1.0.1` version of NMSIS-DSP library.

- Both Nuclei RISC-V 32 and 64 bit cores are supported now.
- Libraries are optimized for RISC-V 32 and 64 bit DSP instructions.
- The NN examples are now using Nuclei SDK as running environment.

3.4.3 V1.0.0

This is the first version of NMSIS-DSP library.

We adapt the CMSIS-DSP v1.6.0 library to use RISCV DSP instructions, all the API names now are renamed from `arm_xxx` to `riscv_xxx`.

CHAPTER

FOUR

NMSIS NN

4.1 Overview

4.1.1 Introduction

This user manual describes the NMSIS NN software library, a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Nuclei N/NX Class Processors cores.

The library is divided into a number of functions each covering a specific category:

- Neural Network Convolution Functions
- Neural Network Activation Functions
- Fully-connected Layer Functions
- Neural Network Pooling Functions
- Softmax Functions
- Neural Network Support Functions

The library has separate functions for operating on different weight and activation data types including 8-bit integers (q7_t) and 16-bit integers (q15_t). The descrition of the kernels are included in the function description.

The implementation details are also described in this paper CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs[14] .

4.1.2 Block Diagram

4.1.3 Examples

The library ships with a number of examples which demonstrate how to use the library functions.

- *Convolutional Neural Network Example* (page 735)
- *Gated Recurrent Unit Example* (page 736)

[14] https://arxiv.org/abs/1801.06601

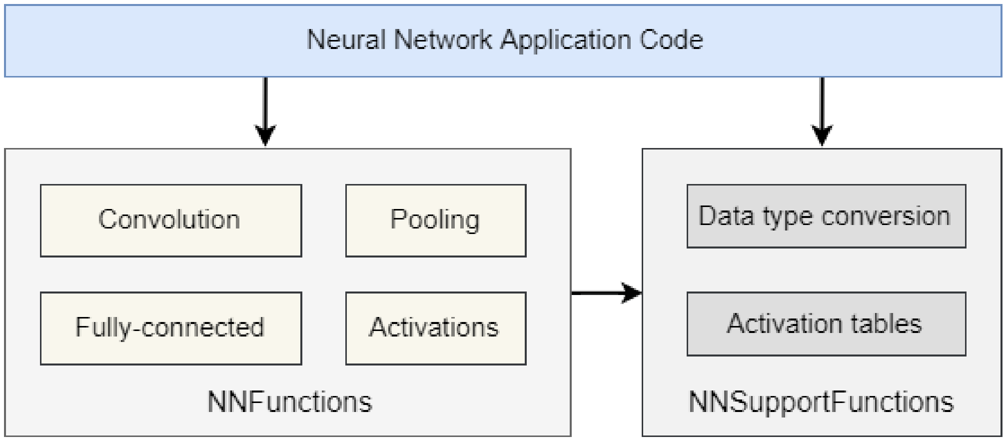


Fig. 1: NMSIS NN Block Diagram

4.1.4 Pre-processor Macros

Each library project have differant pre-processor macros.

This library is only built for little endian targets.

RISCV_MATH_DSP: Define macro RISCV_MATH_DSP, If the silicon supports DSP instructions.

RISCV_NN_TRUNCATE: Define macro RISCV_NN_TRUNCATE to use floor instead of round-to-the-nearest-int for the computation.

4.2 Using NMSIS-NN

Here we will describe how to run the nmsis nn examples in Nuclei Spike.

4.2.1 Preparation

- Nuclei Modified Spike - `xl_spike`
- Nuclei SDK modified for `xl_spike` branch `dev_xlspike`
- Nuclei RISCV GNU Toolchain
- CMake >= 3.5

4.2.2 Tool Setup

1. Export **PATH** correctly for *xl_spike* and *riscv-nuclei-elf-gcc*

```
export PATH=/path/to/xl_spike/bin:/path/to/riscv-nuclei-elf-gcc/bin/:$PATH
```

4.2.3 Build NMSIS NN Library

1. Download or clone NMSIS source code into **NMSIS** directory.
2. cd to *NMSIS/NMSIS/* directory

3. Build NMSIS NN library using `make gen_nn_lib`
4. Strip debug informations using `make strip_nn_lib` to make the generated library smaller
5. The nn library will be generated into `./Library/NN/GCC` folder
6. The nn libraries will be look like this:

```
$ ll Library/NN/GCC/
total 3000
-rw-r--r-- 1 hqfang nucleisys 128482 Jul 14 14:51 libnmsis_nn_rv32imac.a
-rw-r--r-- 1 hqfang nucleisys 281834 Jul 14 14:51 libnmsis_nn_rv32imacp.a
-rw-r--r-- 1 hqfang nucleisys 128402 Jul 14 14:51 libnmsis_nn_rv32imafc.a
-rw-r--r-- 1 hqfang nucleisys 282750 Jul 14 14:51 libnmsis_nn_rv32imafcp.a
-rw-r--r-- 1 hqfang nucleisys 128650 Jul 14 14:51 libnmsis_nn_rv32imafdc.a
-rw-r--r-- 1 hqfang nucleisys 282978 Jul 14 14:51 libnmsis_nn_rv32imafdcp.a
-rw-r--r-- 1 hqfang nucleisys 183918 Jul 14 14:51 libnmsis_nn_rv64imac.a
-rw-r--r-- 1 hqfang nucleisys 418598 Jul 14 14:51 libnmsis_nn_rv64imacp.a
-rw-r--r-- 1 hqfang nucleisys 184206 Jul 14 14:51 libnmsis_nn_rv64imafc.a
-rw-r--r-- 1 hqfang nucleisys 418070 Jul 14 14:51 libnmsis_nn_rv64imafcp.a
-rw-r--r-- 1 hqfang nucleisys 184454 Jul 14 14:51 libnmsis_nn_rv64imafdc.a
-rw-r--r-- 1 hqfang nucleisys 419774 Jul 14 14:51 libnmsis_nn_rv64imafdcp.a
```

7. library name with extra `p` is build with RISCV DSP enabled.
   - `libnmsis_nn_rv32imac.a`: Build for **RISCV_ARCH=rv32imac** without DSP enabled.
   - `libnmsis_nn_rv32imacp.a`: Build for **RISCV_ARCH=rv32imac** with DSP enabled.

Note:

- You can also directly build both DSP and NN library using `make gen`
- You can strip the generated DSP and NN library using `make strip`

4.2.4 How to run

1. Set environment variables `NUCLEI_SDK_ROOT` and `NUCLEI_SDK_NMSIS`, and set Nuclei SDK SoC to *xl-spike*

```
export NUCLEI_SDK_ROOT=/path/to/nuclei_sdk
export NUCLEI_SDK_NMSIS=/path/to/NMSIS/NMSIS
export SOC=xlspike
```

2. Let us take `./cifar10/` for example
2. `cd ./cifar10/`
3. Run with RISCV DSP enabled NMSIS-NN library for CORE `n307`

```
# Clean project
make DSP_ENABLE=ON CORE=n307 clean
# Build project
make DSP_ENABLE=ON CORE=n307 all
# Run application using xl_spike
make DSP_ENABLE=ON CORE=n307 run
```

4. Run with RISCV DSP disabled NMSIS-NN library for CORE `n307`

```
make DSP_ENABLE=OFF CORE=n307 clean
make DSP_ENABLE=OFF CORE=n307 all
make DSP_ENABLE=OFF CORE=n307 run
```

Note:

- You can easily run this example in your hardware, if you have enough memory to run it, just modify the SOC to the one your are using in step 1.

4.3 NMSIS NN API

If you want to access doxygen generated NMSIS NN API, please click NMSIS NN API Doxygen Documentation.

4.3.1 Neural Network Functions

Activation Functions

void **riscv_nn_activations_direct_q15** (q15_t **data*, uint16_t *size*, uint16_t *int_width*, riscv_nn_activation_type *type*)

void **riscv_nn_activations_direct_q7** (q7_t **data*, uint16_t *size*, uint16_t *int_width*, riscv_nn_activation_type *type*)

void **riscv_relu6_s8** (q7_t **data*, uint16_t *size*)

void **riscv_relu_q15** (q15_t **data*, uint16_t *size*)

void **riscv_relu_q7** (q7_t **data*, uint16_t *size*)

group **Acti**

Perform activation layers, including ReLU (Rectified Linear Unit), sigmoid and tanh

Functions

void **riscv_nn_activations_direct_q15** (q15_t **data*, uint16_t *size*, uint16_t *int_width*, riscv_nn_activation_type *type*)

neural network activation function using direct table look-up

Q15 neural network activation function using direct table look-up.

Note Refer header file for details.

void **riscv_nn_activations_direct_q7** (q7_t **data*, uint16_t *size*, uint16_t *int_width*, riscv_nn_activation_type *type*)

Q7 neural network activation function using direct table look-up.

This is the direct table look-up approach.

Parameters

- `[inout] data`: pointer to input
- `[in] size`: number of elements
- `[in] int_width`: bit-width of the integer part, assume to be smaller than 3

- `[in] type`: type of activation functions

Assume here the integer part of the fixed-point is <= 3. More than 3 just not making much sense, makes no difference with saturation followed by any of these activation functions.

void **riscv_relu6_s8** (q7_t **data*, uint16_t *size*)
s8 ReLU6 function

Parameters

- `[inout] data`: pointer to input
- `[in] size`: number of elements

void **riscv_relu_q15** (q15_t **data*, uint16_t *size*)
Q15 RELU function.

Optimized relu with QSUB instructions.

Parameters

- `[inout] data`: pointer to input
- `[in] size`: number of elements

void **riscv_relu_q7** (q7_t **data*, uint16_t *size*)
Q7 RELU function.

Optimized relu with QSUB instructions.

Parameters

- `[inout] data`: pointer to input
- `[in] size`: number of elements

Basic math functions

riscv_status **riscv_elementwise_add_s8** (**const** int8_t **input_1_vect*, **const** int8_t **input_2_vect*, **const** int32_t *input_1_offset*, **const** int32_t *input_1_mult*, **const** int32_t *input_1_shift*, **const** int32_t *input_2_offset*, **const** int32_t *input_2_mult*, **const** int32_t *input_2_shift*, **const** int32_t *left_shift*, int8_t **output*, **const** int32_t *out_offset*, **const** int32_t *out_mult*, **const** int32_t *out_shift*, **const** int32_t *out_activation_min*, **const** int32_t *out_activation_max*, **const** uint32_t *block_size*)

riscv_status **riscv_elementwise_mul_s8** (**const** int8_t **input_1_vect*, **const** int8_t **input_2_vect*, **const** int32_t *input_1_offset*, **const** int32_t *input_2_offset*, int8_t **output*, **const** int32_t *out_offset*, **const** int32_t *out_mult*, **const** int32_t *out_shift*, **const** int32_t *out_activation_min*, **const** int32_t *out_activation_max*, **const** uint32_t *block_size*)

group **BasicMath**
Element wise add and multiplication functions.

Functions

riscv_status **riscv_elementwise_add_s8**(**const** int8_t **input_1_vect*, **const** int8_t **input_2_vect*, **const** int32_t *input_1_offset*, **const** int32_t *input_1_mult*, **const** int32_t *input_1_shift*, **const** int32_t *input_2_offset*, **const** int32_t *input_2_mult*, **const** int32_t *input_2_shift*, **const** int32_t *left_shift*, int8_t **output*, **const** int32_t *out_offset*, **const** int32_t *out_mult*, **const** int32_t *out_shift*, **const** int32_t *out_activation_min*, **const** int32_t *out_activation_max*, **const** uint32_t *block_size*)

s8 element wise add of two vectors

Return The function returns RISCV_MATH_SUCCESS

Parameters

- `[in] input_1_vect`: pointer to input vector 1
- `[in] input_2_vect`: pointer to input vector 2
- `[in] input_1_offset`: offset for input 1. Range: Range: -127 to 128
- `[in] input_1_mult`: multiplier for input 1
- `[in] input_1_shift`: shift for input 1
- `[in] input_2_offset`: offset for input 2. Range: Range: -127 to 128
- `[in] input_2_mult`: multiplier for input 2
- `[in] input_2_shift`: shift for input 2
- `[in] left_shift`: input left shift
- `[inout] output`: pointer to output vector
- `[in] out_offset`: output offset
- `[in] out_mult`: output multiplier
- `[in] out_shift`: output shift
- `[in] out_activation_min`: minimum value to clamp output to
- `[in] out_activation_max`: maximum value to clamp output to
- `[in] block_size`: number of samples

riscv_status **riscv_elementwise_mul_s8**(**const** int8_t **input_1_vect*, **const** int8_t **input_2_vect*, **const** int32_t *input_1_offset*, **const** int32_t *input_2_offset*, int8_t **output*, **const** int32_t *out_offset*, **const** int32_t *out_mult*, **const** int32_t *out_shift*, **const** int32_t *out_activation_min*, **const** int32_t *out_activation_max*, **const** uint32_t *block_size*)

s8 element wise multiplication of two vectors

s8 element wise multiplication

Note Refer header file for details.

Concatenation Functions

void **riscv_concatenation_s8_w** (**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint32_t *offset_w*)

void **riscv_concatenation_s8_x** (**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_x*, **const** uint32_t *offset_x*)

void **riscv_concatenation_s8_y** (**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_y*, **const** uint32_t *offset_y*)

void **riscv_concatenation_s8_z** (**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_z*, **const** uint32_t *offset_z*)

group **Concatenation**

Functions

void **riscv_concatenation_s8_w** (**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint32_t *offset_w*)

int8/uint8 concatenation function to be used for concatenating N-tensors along the W axis (Batch size) This function should be called for each input tensor to concatenate. The argument offset_w will be used to store the input tensor in the correct position in the output tensor

i.e. offset_w = 0 for(i = 0 i < num_input_tensors; ++i) { riscv_concatenation_s8_w(&input[i], …, &output, …, …, offset_w) offset_w += input_w[i] }

This function assumes that the output tensor has:

1. The same width of the input tensor
2. The same height of the input tensor
3. The same number o channels of the input tensor

Unless specified otherwise, arguments are mandatory.

Note This function, data layout independent, can be used to concatenate either int8 or uint8 tensors because it does not involve any arithmetic operation

Parameters

- `[in] input`: Pointer to input tensor
- `[in] input_x`: Width of input tensor
- `[in] input_y`: Height of input tensor
- `[in] input_z`: Channels in input tensor
- `[in] input_w`: Batch size in input tensor
- `[out] output`: Pointer to output tensor
- `[in] offset_w`: The offset on the W axis to start concatenating the input tensor It is user responsibility to provide the correct value

void **riscv_concatenation_s8_x**(**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_x*, **const** uint32_t *offset_x*)

int8/uint8 concatenation function to be used for concatenating N-tensors along the X axis This function should be called for each input tensor to concatenate. The argument offset_x will be used to store the input tensor in the correct position in the output tensor

i.e. offset_x = 0 for(i = 0 i < num_input_tensors; ++i) { riscv_concatenation_s8_x(&input[i], . . . , &output, . . . , . . . , offset_x) offset_x += input_x[i] }

This function assumes that the output tensor has:

1. The same height of the input tensor
2. The same number of channels of the input tensor
3. The same batch size of the input tensor

Unless specified otherwise, arguments are mandatory.

Input constraints offset_x is less than output_x

Note This function, data layout independent, can be used to concatenate either int8 or uint8 tensors because it does not involve any arithmetic operation

Parameters

- `[in] input`: Pointer to input tensor
- `[in] input_x`: Width of input tensor
- `[in] input_y`: Height of input tensor
- `[in] input_z`: Channels in input tensor
- `[in] input_w`: Batch size in input tensor
- `[out] output`: Pointer to output tensor
- `[in] output_x`: Width of output tensor
- `[in] offset_x`: The offset (in number of elements) on the X axis to start concatenating the input tensor It is user responsibility to provide the correct value

void **riscv_concatenation_s8_y**(**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_y*, **const** uint32_t *offset_y*)

int8/uint8 concatenation function to be used for concatenating N-tensors along the Y axis This function should be called for each input tensor to concatenate. The argument offset_y will be used to store the input tensor in the correct position in the output tensor

i.e. offset_y = 0 for(i = 0 i < num_input_tensors; ++i) { riscv_concatenation_s8_y(&input[i], . . . , &output, . . . , . . . , offset_y) offset_y += input_y[i] }

This function assumes that the output tensor has:

1. The same width of the input tensor
2. The same number of channels of the input tensor
3. The same batch size of the input tensor

Unless specified otherwise, arguments are mandatory.

Input constraints offset_y is less than output_y

Note This function, data layout independent, can be used to concatenate either int8 or uint8 tensors because it does not involve any arithmetic operation

Parameters

- `[in] input`: Pointer to input tensor
- `[in] input_x`: Width of input tensor
- `[in] input_y`: Height of input tensor
- `[in] input_z`: Channels in input tensor
- `[in] input_w`: Batch size in input tensor
- `[out] output`: Pointer to output tensor
- `[in] output_y`: Height of output tensor
- `[in] offset_y`: The offset on the Y axis to start concatenating the input tensor It is user responsibility to provide the correct value

void **riscv_concatenation_s8_z**(**const** int8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_z*, **const** uint16_t *input_w*, int8_t **output*, **const** uint16_t *output_z*, **const** uint32_t *offset_z*)

int8/uint8 concatenation function to be used for concatenating N-tensors along the Z axis This function should be called for each input tensor to concatenate. The argument offset_z will be used to store the input tensor in the correct position in the output tensor

i.e. offset_z = 0 for(i = 0 i < num_input_tensors; ++i) { riscv_concatenation_s8_z(&input[i], . . . , &output, . . . , . . . , offset_z) offset_z += input_z[i] }

This function assumes that the output tensor has:

1. The same width of the input tensor
2. The same height of the input tensor
3. The same batch size of the input tensor

Unless specified otherwise, arguments are mandatory.

Input constraints offset_z is less than output_z

Note This function, data layout independent, can be used to concatenate either int8 or uint8 tensors because it does not involve any arithmetic operation

Parameters

- `[in] input`: Pointer to input tensor
- `[in] input_x`: Width of input tensor
- `[in] input_y`: Height of input tensor
- `[in] input_z`: Channels in input tensor
- `[in] input_w`: Batch size in input tensor
- `[out] output`: Pointer to output tensor
- `[in] output_z`: Channels in output tensor
- `[in] offset_z`: The offset on the Z axis to start concatenating the input tensor It is user responsibility to provide the correct value

Convolution Functions

riscv_status **riscv_convolve_1_x_n_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_convolve_1_x_n_s8_get_buffer_size**(**const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*)

riscv_status **riscv_convolve_1x1_HWC_q7_fast_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_1x1_s8_fast**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_convolve_1x1_s8_fast_get_buffer_size**(**const** nmsis_nn_dims **input_dims*)

riscv_status **riscv_convolve_HWC_q15_basic**(**const** q15_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q15_fast**(**const** q15_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q15_fast_nonsquare**(**const** q15_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q7_basic**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q7_basic_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q7_fast**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q7_fast_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_HWC_q7_RGB**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

riscv_status **riscv_convolve_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_convolve_s8_get_buffer_size**(**const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*)

riscv_status **riscv_convolve_wrapper_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_convolve_wrapper_s8_get_buffer_size**(**const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*)

riscv_status **riscv_depthwise_conv_3x3_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*) static void **depthwise_conv_s8_mult_4** (const int8_t **input*, const int32_t *input_x*, const int32_t *input_y*, const int32_t *input_ch*, const int8_t **kernel*, const int32_t *output_ch*, const int32_t *ch_mult*, const int32_t *kernel_x*, const int32_t *kernel_y*, const int32_t *pad_x*, const int32_t *pad_y*, const int32_t *stride_x*, const int32_t *stride_y*, const int32_t **bias*, int8_t **output*, const int32_t **output_shift*, const int32_t **output_mult*, const int32_t *output_x*, const int32_t *output_y*, const int32_t *output_offset*, const int32_t *input_offset*, const int32_t *output_activation_min*, const int32_t *output_activation_max*)

static void **depthwise_conv_s8_generic** (const q7_t **input*, const uint16_t *input_batches*, const uint16_t *input_x*, const uint16_t *input_y*, const uint16_t *input_ch*, const q7_t **kernel*, const uint16_t *output_ch*, const uint16_t *ch_mult*, const uint16_t *kernel_x*, const uint16_t *kernel_y*, const uint16_t *pad_x*, const uint16_t *pad_y*, const uint16_t *stride_x*, const uint16_t *stride_y*, const int32_t **bias*, q7_t **output*, const int32_t **output_shift*, const int32_t **output_mult*, const uint16_t *output_x*, const uint16_t *output_y*, const int32_t *output_offset*, const int32_t *input_offset*, const int32_t *output_activation_min*, const int32_t *output_activation_max*)

riscv_status **riscv_depthwise_conv_s8** (const nmsis_nn_context **ctx*, const nmsis_nn_dw_conv_params **dw_conv_params*, const nmsis_nn_per_channel_quant_params **quant_params*, const nmsis_nn_dims **input_dims*, const q7_t **input_data*, const nmsis_nn_dims **filter_dims*, const q7_t **filter_data*, const nmsis_nn_dims **bias_dims*, const int32_t **bias_data*, const nmsis_nn_dims **output_dims*, q7_t **output_data*)

riscv_status **riscv_depthwise_conv_s8_opt** (const nmsis_nn_context **ctx*, const nmsis_nn_dw_conv_params **dw_conv_params*, const nmsis_nn_per_channel_quant_params **quant_params*, const nmsis_nn_dims **input_dims*, const q7_t **input_data*, const nmsis_nn_dims **filter_dims*, const q7_t **filter_data*, const nmsis_nn_dims **bias_dims*, const int32_t **bias_data*, const nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_depthwise_conv_s8_opt_get_buffer_size** (const nmsis_nn_dims **input_dims*, const nmsis_nn_dims **filter_dims*)

static void **depthwise_conv_u8_mult_4** (**const** uint8_t **input*, **const** int32_t *input_x*, **const** int32_t *input_y*, **const** int32_t *input_ch*, **const** uint8_t **kernel*, **const** int32_t *output_ch*, **const** int32_t *ch_mult*, **const** int32_t *kernel_x*, **const** int32_t *kernel_y*, **const** int32_t *pad_x*, **const** int32_t *pad_y*, **const** int32_t *stride_x*, **const** int32_t *stride_y*, **const** int32_t **bias*, uint8_t **output*, **const** int32_t *output_shift*, **const** int32_t *output_mult*, **const** int32_t *output_x*, **const** int32_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*)

static void **depthwise_conv_u8_generic** (**const** uint8_t **input*, **const** int32_t *input_x*, **const** int32_t *input_y*, **const** int32_t *input_ch*, **const** uint8_t **kernel*, **const** int32_t *output_ch*, **const** int32_t *ch_mult*, **const** int32_t *kernel_x*, **const** int32_t *kernel_y*, **const** int32_t *pad_x*, **const** int32_t *pad_y*, **const** int32_t *stride_x*, **const** int32_t *stride_y*, **const** int32_t **bias*, uint8_t **output*, **const** int32_t *output_shift*, **const** int32_t *output_mult*, **const** int32_t *output_x*, **const** int32_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*)

riscv_status **riscv_depthwise_conv_u8_basic_ver1** (**const** uint8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_ch*, **const** uint8_t **kernel*, **const** uint16_t *kernel_x*, **const** uint16_t *kernel_y*, **const** int16_t *ch_mult*, **const** int16_t *pad_x*, **const** int16_t *pad_y*, **const** int16_t *stride_x*, **const** int16_t *stride_y*, **const** int16_t *dilation_x*, **const** int16_t *dilation_y*, **const** int32_t **bias*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_offset*, uint8_t **output*, **const** uint16_t *output_x*, **const** uint16_t *output_y*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*, **const** int32_t *output_shift*, **const** int32_t *output_mult*)

riscv_status **riscv_depthwise_conv_wrapper_s8** (**const** nmsis_nn_context **ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*) int32_t **riscv_depthwise_conv_wrapper_s8_get_buffer_size**(**const** nmsis_nn_dw_conv_params *dw_conv_params*, **const** nmsis_nn_dims *input_dims*, **const** nmsis_nn_dims *filter_dims*, **const** nmsis_nn_dims *output_dims*)

riscv_status **riscv_depthwise_separable_conv_HWC_q7**(**const** q7_t *Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t *wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t *bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t *Im_out*, **const** uint16_t *dim_im_out*, q15_t *bufferA*, q7_t *bufferB*)

riscv_status **riscv_depthwise_separable_conv_HWC_q7_nonsquare**(**const** q7_t *Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t *wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t *bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t *Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t *bufferA*, q7_t *bufferB*)

group **NNConv**

Collection of convolution, depthwise convolution functions and their variants.

The convolution is implemented in 2 steps: im2col and GEMM

im2col is a process of converting each patch of image data into a column. After im2col, the convolution is computed as matrix-matrix multiplication.

To reduce the memory footprint, the im2col is performed partially. Each iteration, only a few column (i.e., patches) are generated and computed with GEMM kernels similar to NMSIS-DSP riscv_mat_mult functions.

Functions

riscv_status **riscv_convolve_1_x_n_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

1xn convolution

- Supported framework : TensorFlow Lite Micro
- The following constrains on the arguments apply
  1. input_dims->n equals 1
  2. ouput_dims->w is a multiple of 4
  3. Explicit constraints(since it is for 1xN convolution) -## input_dims->h equals 1 -## output_dims->h equals 1 -## filter_dims->h equals 1

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` if argument constraints fail. or, `RISCV_MATH_SUCCESS` on successful completion.

Parameters

- `[inout] ctx`: Function context that contains the additional buffer if required by the function. riscv_convolve_1_x_n_s8_get_buffer_size will return the buffer_size if required
- `[in] conv_params`: Convolution parameters (e.g. strides, dilations, pads,...). Range of conv_params->input_offset : [-127, 128] Range of conv_params->output_offset : [-128, 127]
- `[in] quant_params`: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [C_OUT, 1, WK, C_IN] where WK is the horizontal spatial filter dimension
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT]
- `[in] bias_data`: Optional bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [N, H, W, C_OUT]
- `[out] output_data`: Output data pointer. Data type: int8

int32_t **riscv_convolve_1_x_n_s8_get_buffer_size**(**const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*)

Get the required additional buffer size for 1xn convolution.

Return The function returns required buffer size(bytes)

Parameters

- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]

- `[in] filter_dims`: Filter tensor dimensions. Format: [C_OUT, 1, WK, C_IN] where WK is the horizontal spatial filter dimension

riscv_status **riscv_convolve_1x1_HWC_q7_fast_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

Fast Q7 version of 1x1 convolution (non-sqaure shape)

This function is optimized for convolution with 1x1 kernel size (i.e., dim_kernel_x=1 and dim_kernel_y=1). It can be used for the second half of MobileNets [1] after depthwise separable convolution.

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in_x`: input tensor dimention x
- `[in] dim_im_in_y`: input tensor dimention y
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel_x`: filter kernel size x
- `[in] dim_kernel_y`: filter kernel size y
- `[in] padding_x`: padding size x
- `[in] padding_y`: padding size y
- `[in] stride_x`: convolution stride x
- `[in] stride_y`: convolution stride y
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor

- [in] dim_im_out_x: output tensor dimension x
- [in] dim_im_out_y: output tensor dimension y
- [inout] bufferA: pointer to buffer space for input
- [inout] bufferB: pointer to buffer space for output

This function is the version with full list of optimization tricks, but with some constraints: ch_im_in is multiple of 4 ch_im_out is multiple of 2

[1] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications https://arxiv.org/abs/1704.04861

riscv_status **riscv_convolve_1x1_s8_fast**(**const** nmsis_nn_context *ctx, **const** nmsis_nn_conv_params *conv_params, **const** nmsis_nn_per_channel_quant_params *quant_params, **const** nmsis_nn_dims *input_dims, **const** q7_t *input_data, **const** nmsis_nn_dims *filter_dims, **const** q7_t *filter_data, **const** nmsis_nn_dims *bias_dims, **const** int32_t *bias_data, **const** nmsis_nn_dims *output_dims, q7_t *output_data)

Fast s8 version for 1x1 convolution (non-square shape)

- Supported framework : TensorFlow Lite Micro
- The following constrains on the arguments apply
  1. input_dims->c is a multiple of 4
  2. conv_params->padding.w = conv_params->padding.h = 0
  3. conv_params->stride.w = conv_params->stride.h = 1

Return The function returns either RISCV_MATH_SIZE_MISMATCH if argument constraints fail. or, RISCV_MATH_SUCCESS on successful completion.

Parameters

- [inout] ctx: Function context that contains the additional buffer if required by the function. riscv_convolve_1x1_s8_fast_get_buffer_size will return the buffer_size if required
- [in] conv_params: Convolution parameters (e.g. strides, dilations, pads,…). Range of conv_params->input_offset : [-127, 128] Range of conv_params->output_offset : [-128, 127]
- [in] quant_params: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- [in] input_dims: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]
- [in] input_data: Input (activation) data pointer. Data type: int8
- [in] filter_dims: Filter tensor dimensions. Format: [C_OUT, 1, 1, C_IN]
- [in] filter_data: Filter data pointer. Data type: int8
- [in] bias_dims: Bias tensor dimensions. Format: [C_OUT]
- [in] bias_data: Optional bias data pointer. Data type: int32
- [in] output_dims: Output tensor dimensions. Format: [N, H, W, C_OUT]
- [out] output_data: Output data pointer. Data type: int8

int32_t **riscv_convolve_1x1_s8_fast_get_buffer_size**(**const** nmsis_nn_dims **input_dims*)

Get the required buffer size for riscv_convolve_1x1_s8_fast.

Return The function returns the required buffer size in bytes

Parameters

- `[in] input_dims`: Input (activation) dimensions

riscv_status **riscv_convolve_HWC_q15_basic**(**const** q15_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Basic Q15 convolution function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

This basic version is designed to work for any input tensor and weight dimension.

riscv_status **riscv_convolve_HWC_q15_fast** (**const** q15_t *Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Fast Q15 convolution function.

Buffer size:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

Input dimension constraints:

ch_im_in is multiple of 2

ch_im_out is multiple of 2

riscv_status **riscv_convolve_HWC_q15_fast_nonsquare**(**const** q15_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q15_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q15_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q15_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

Fast Q15 convolution function (non-sqaure shape)

Buffer size:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in_x`: input tensor dimention x
- `[in] dim_im_in_y`: input tensor dimention y
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel_x`: filter kernel size x
- `[in] dim_kernel_y`: filter kernel size y
- `[in] padding_x`: padding size x
- `[in] padding_y`: padding size y
- `[in] stride_x`: convolution stride x
- `[in] stride_y`: convolution stride y
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out_x`: output tensor dimension x
- `[in] dim_im_out_y`: output tensor dimension y
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

Input dimension constraints:

ch_im_in is multiple of 2

ch_im_out is multiple of 2

riscv_status **riscv_convolve_HWC_q7_basic**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Basic Q7 convolution function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

This basic version is designed to work for any input tensor and weight dimension.

riscv_status **riscv_convolve_HWC_q7_basic_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

Basic Q7 convolution function (non-sqaure shape)

Basic Q7 convolution function (non-square shape)

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in_x`: input tensor dimention x
- `[in] dim_im_in_y`: input tensor dimention y
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel_x`: filter kernel size x
- `[in] dim_kernel_y`: filter kernel size y
- `[in] padding_x`: padding size x
- `[in] padding_y`: padding size y
- `[in] stride_x`: convolution stride x
- `[in] stride_y`: convolution stride y
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out_x`: output tensor dimension x
- `[in] dim_im_out_y`: output tensor dimension y
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

riscv_status **riscv_convolve_HWC_q7_fast**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Fast Q7 convolution function.

Buffer size:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

Input dimension constraints:

ch_im_in is multiple of 4 (because of the SIMD32 read and swap)

ch_im_out is multiple of 2 (bacause 2x2 mat_mult kernel)

The im2col converts the Q7 tensor input into Q15 column, which is stored in bufferA. There is reordering happenning during this im2col process with riscv_q7_to_q15_reordered_no_shift. For every four elements, the second and third elements are swapped.

The computation kernel riscv_nn_mat_mult_kernel_q7_q15_reordered does the GEMM computation with the reordered columns.

To speed-up the determination of the padding condition, we split the computation into 3x3 parts, i.e., {top, mid, bottom} X {left, mid, right}. This reduces the total number of boundary condition checks and improves the data copying performance.

riscv_status **riscv_convolve_HWC_q7_fast_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

Fast Q7 convolution function (non-sqaure shape)

This function is the version with full list of optimization tricks, but with some constraints: ch_im_in is multiple of 4 ch_im_out is multiple of 2

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in_x`: input tensor dimention x
- `[in] dim_im_in_y`: input tensor dimention y
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel_x`: filter kernel size x
- `[in] dim_kernel_y`: filter kernel size y
- `[in] padding_x`: padding size x
- `[in] padding_y`: padding size y
- `[in] stride_x`: convolution stride x
- `[in] stride_y`: convolution stride y
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out_x`: output tensor dimension x
- `[in] dim_im_out_y`: output tensor dimension y
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

riscv_status **riscv_convolve_HWC_q7_RGB**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Q7 convolution function for RGB image.

Q7 version of convolution for RGB image.

Buffer size:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

Input dimension constraints:

ch_im_in equals 3

This kernel is written exclusively for convolution with ch_im_in equals 3. This applies on the first layer of CNNs which has input image with RGB format.

riscv_status **riscv_convolve_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Basic s8 convolution function.

1. Supported framework: TensorFlow Lite micro
2. q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.
3. Additional memory is required for optimization. Refer to argument 'ctx' for details.

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[inout] ctx`: Function context that contains the additional buffer if required by the function. riscv_convolve_s8_get_buffer_size will return the buffer_size if required
- `[in] conv_params`: Convolution parameters (e.g. strides, dilations, pads,…). Range of conv_params->input_offset : [-127, 128] Range of conv_params->output_offset : [-128, 127]
- `[in] quant_params`: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [C_OUT, HK, WK, C_IN] where HK and WK are the spatial filter dimensions
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT]
- `[in] bias_data`: Optional bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [N, H, W, C_OUT]
- `[out] output_data`: Output data pointer. Data type: int8

int32_t **riscv_convolve_s8_get_buffer_size**(**const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*)

Get the required buffer size for s8 convolution function.

Return The function returns required buffer size(bytes)

Parameters

- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]
- `[in] filter_dims`: Filter tensor dimensions. Format: [C_OUT, HK, WK, C_IN] where HK and WK are the spatial filter dimensions

riscv_status **riscv_convolve_wrapper_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

s8 convolution layer wrapper function with the main purpose to call the optimal kernel available in nmsis-nn to perform the convolution.

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` if argument constraints fail. or, `RISCV_MATH_SUCCESS` on successful completion.

Parameters

- `[inout] ctx`: Function context that contains the additional buffer if required by the function. riscv_convolve_wrapper_s8_get_buffer_size will return the buffer_size if required
- `[in] conv_params`: Convolution parameters (e.g. strides, dilations, pads,…). Range of conv_params->input_offset : [-127, 128] Range of conv_params->output_offset : [-128, 127]
- `[in] quant_params`: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN]
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [C_OUT, HK, WK, C_IN] where HK and WK are the spatial filter dimensions
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT]
- `[in] bias_data`: Bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [N, H, W, C_OUT]
- `[out] output_data`: Output data pointer. Data type: int8

int32_t **riscv_convolve_wrapper_s8_get_buffer_size**(**const** nmsis_nn_conv_params **conv_params*, **const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*)

Get the required buffer size for riscv_convolve_wrapper_s8.

Return The function returns required buffer size(bytes)

Parameters

- `[in] conv_params`: Convolution parameters (e.g. strides, dilations, pads,…). Range of conv_params->input_offset : [-127, 128] Range of conv_params->output_offset : [-128, 127]
- `[in] input_dims`: Input (activation) dimensions. Format: [N, H, W, C_IN]
- `[in] filter_dims`: Filter dimensions. Format: [C_OUT, HK, WK, C_IN] where HK and WK are the spatial filter dimensions
- `[in] output_dims`: Output tensor dimensions. Format: [N, H, W, C_OUT]

riscv_status **riscv_depthwise_conv_3x3_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Optimized s8 depthwise convolution function for 3x3 kernel size with some constraints on the input arguments(documented below). Refer riscv_depthwise_conv_s8() for function argument details.

- Supported framework : TensorFlow Lite Micro
- The following constrains on the arguments apply
  1. Number of input channel equals number of output channels
  2. Filter height and width equals 3
  3. Padding along x is either 0 or 1.

Return The function returns one of the following `RISCV_MATH_SIZE_MISMATCH` - Unsupported dimension of tensors `RISCV_MATH_ARGUMENT_ERROR` - Unsupported pad size along the x axis `RISCV_MATH_SUCCESS` - Successful operation

static void **depthwise_conv_s8_mult_4**(**const** int8_t **input*, **const** int32_t *input_x*, **const** int32_t *input_y*, **const** int32_t *input_ch*, **const** int8_t **kernel*, **const** int32_t *output_ch*, **const** int32_t *ch_mult*, **const** int32_t *kernel_x*, **const** int32_t *kernel_y*, **const** int32_t *pad_x*, **const** int32_t *pad_y*, **const** int32_t *stride_x*, **const** int32_t *stride_y*, **const** int32_t **bias*, int8_t **output*, **const** int32_t **output_shift*, **const** int32_t **output_mult*, **const** int32_t *output_x*, **const** int32_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*)

static void **depthwise_conv_s8_generic**(**const** q7_t **input*, **const** uint16_t *input_batches*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_ch*, **const** q7_t **kernel*, **const** uint16_t *output_ch*, **const** uint16_t *ch_mult*, **const** uint16_t *kernel_x*, **const** uint16_t *kernel_y*, **const** uint16_t *pad_x*, **const** uint16_t *pad_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** int32_t **bias*, q7_t **output*, **const** int32_t **output_shift*, **const** int32_t **output_mult*, **const** uint16_t *output_x*, **const** uint16_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*)

riscv_status **riscv_depthwise_conv_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Basic s8 depthwise convolution function that doesn't have any constraints on the input dimensions.

- Supported framework: TensorFlow Lite
- q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[inout] ctx`: Function context (e.g. temporary buffer). Check the function definition file to see if an additional buffer is required. Optional function {API}_get_buffer_size() provides the buffer size if an additional buffer is required. exists if additional memory is.
- `[in] dw_conv_params`: Depthwise convolution parameters (e.g. strides, dilations, pads,…) dw_conv_params->dilation is not used. Range of dw_conv_params->input_offset : [-127, 128] Range of dw_conv_params->input_offset : [-128, 127]
- `[in] quant_params`: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [1, H, W, C_IN] Batch argument N is not used.
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [1, H, W, C_OUT]
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT]
- `[in] bias_data`: Bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [1, H, W, C_OUT]
- `[inout] output_data`: Output data pointer. Data type: int8

riscv_status **riscv_depthwise_conv_s8_opt**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Optimized s8 depthwise convolution function with constraint that in_channel equals out_channel. Refer riscv_depthwise_conv_s8() for function argument details.

- Supported framework: TensorFlow Lite
- The following constrains on the arguments apply
  1. Number of input channel equals number of output channels or ch_mult equals 1
- q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.
- Reccomended when number of channels is 4 or greater.

Return The function returns one of the following `RISCV_MATH_SIZE_MISMATCH` - input channel != output channel or ch_mult != 1 `RISCV_MATH_SUCCESS` - Successful operation

Note If number of channels is not a multiple of 4, upto 3 elements outside the boundary will be read out for the following if MVE optimizations(Arm Helium Technology) are used.

- Output shift
- Output multiplier
- Output bias

- kernel

int32_t **riscv_depthwise_conv_s8_opt_get_buffer_size**(**const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*)

Get the required buffer size for optimized s8 depthwise convolution function with constraint that in_channel equals out_channel.

Return The function returns required buffer size in bytes

Parameters

- `[in] input_dims`: Input (activation) tensor dimensions. Format: [1, H, W, C_IN] Batch argument N is not used.
- `[in] filter_dims`: Filter tensor dimensions. Format: [1, H, W, C_OUT]

static void **depthwise_conv_u8_mult_4**(**const** uint8_t **input*, **const** int32_t *input_x*, **const** int32_t *input_y*, **const** int32_t *input_ch*, **const** uint8_t **kernel*, **const** int32_t *output_ch*, **const** int32_t *ch_mult*, **const** int32_t *kernel_x*, **const** int32_t *kernel_y*, **const** int32_t *pad_x*, **const** int32_t *pad_y*, **const** int32_t *stride_x*, **const** int32_t *stride_y*, **const** int32_t **bias*, uint8_t **output*, **const** int32_t *output_shift*, **const** int32_t *output_mult*, **const** int32_t *output_x*, **const** int32_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*)

static void **depthwise_conv_u8_generic**(**const** uint8_t **input*, **const** int32_t *input_x*, **const** int32_t *input_y*, **const** int32_t *input_ch*, **const** uint8_t **kernel*, **const** int32_t *output_ch*, **const** int32_t *ch_mult*, **const** int32_t *kernel_x*, **const** int32_t *kernel_y*, **const** int32_t *pad_x*, **const** int32_t *pad_y*, **const** int32_t *stride_x*, **const** int32_t *stride_y*, **const** int32_t **bias*, uint8_t **output*, **const** int32_t *output_shift*, **const** int32_t *output_mult*, **const** int32_t *output_x*, **const** int32_t *output_y*, **const** int32_t *output_offset*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*) riscv_status **riscv_depthwise_conv_u8_basic_ver1**(**const** uint8_t **input*, **const** uint16_t *input_x*, **const** uint16_t *input_y*, **const** uint16_t *input_ch*, **const** uint8_t **kernel*, **const** uint16_t *kernel_x*, **const** uint16_t *kernel_y*, **const** int16_t *ch_mult*, **const** int16_t *pad_x*, **const** int16_t *pad_y*, **const** int16_t *stride_x*, **const** int16_t *stride_y*, **const** int16_t *dilation_x*, **const** int16_t *dilation_y*, **const** int32_t **bias*, **const** int32_t *input_offset*, **const** int32_t *filter_offset*, **const** int32_t *output_offset*, uint8_t **output*, **const** uint16_t *output_x*, **const** uint16_t *output_y*, **const** int32_t *output_activation_min*, **const** int32_t *output_activation_max*, **const** int32_t *output_shift*, **const** int32_t *output_mult*)

uint8 depthwise convolution function with asymmetric quantization

uint8 depthwise convolution function with asymmetric quantization Unless specified otherwise, arguments are mandatory.

Return The function returns one of the following `RISCV_MATH_SIZE_MISMATCH` - Not supported dimension of tensors `RISCV_MATH_SUCCESS` - Successful operation `RISCV_MATH_ARGUMENT_ERROR` - Implementation not available

Parameters

- `[in] input`: Pointer to input tensor
- `[in] input_x`: Width of input tensor
- `[in] input_y`: Height of input tensor
- `[in] input_ch`: Channels in input tensor
- `[in] kernel`: Pointer to kernel weights
- `[in] kernel_x`: Width of kernel
- `[in] kernel_y`: Height of kernel
- `[in] ch_mult`: Number of channel multiplier
- `[in] pad_x`: Padding sizes x
- `[in] pad_y`: Padding sizes y
- `[in] stride_x`: Convolution stride along the width
- `[in] stride_y`: Convolution stride along the height
- `[in] dilation_x`: Dilation along width. Not used and intended for future enhancement.
- `[in] dilation_y`: Dilation along height. Not used and intended for future enhancement.
- `[in] bias`: Pointer to optional bias values. If no bias is available, NULL is expected
- `[in] input_offset`: Input tensor zero offset
- `[in] filter_offset`: Kernel tensor zero offset

- `[in] output_offset`: Output tensor zero offset
- `[inout] output`: Pointer to output tensor
- `[in] output_x`: Width of output tensor
- `[in] output_y`: Height of output tensor
- `[in] output_activation_min`: Minimum value to clamp the output to. Range : {0, 255}
- `[in] output_activation_max`: Minimum value to clamp the output to. Range : {0, 255}
- `[in] output_shift`: Amount of right-shift for output
- `[in] output_mult`: Output multiplier for requantization

riscv_status **riscv_depthwise_conv_wrapper_s8**(**const** nmsis_nn_context *ctx*, **const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_per_channel_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Wrapper function to pick the right optimized s8 depthwise convolution function.

- Supported framework: TensorFlow Lite
- Picks one of the the following functions
  1. riscv_depthwise_conv_s8()
  2. riscv_depthwise_conv_3x3_s8() - RISC-V CPUs with DSP extension only
  3. riscv_depthwise_conv_s8_opt()
- q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.
- Check details of riscv_depthwise_conv_s8_opt() for potential data that can be accessed outside of the boundary.

Return The function returns `RISCV_MATH_SUCCESS` - Successful completion.

Parameters

- `[inout] ctx`: Function context (e.g. temporary buffer). Check the function definition file to see if an additional buffer is required. Optional function {API}_get_buffer_size() provides the buffer size if required.
- `[in] dw_conv_params`: Depthwise convolution parameters (e.g. strides, dilations, pads,…) dw_conv_params->dilation is not used. Range of dw_conv_params->input_offset : [-127, 128] Range of dw_conv_params->output_offset : [-128, 127]
- `[in] quant_params`: Per-channel quantization info. It contains the multiplier and shift values to be applied to each output channel
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [H, W, C_IN] Batch argument N is not used and assumed to be 1.
- `[in] input_data`: Input (activation) data pointer. Data type: int8

- `[in] filter_dims`: Filter tensor dimensions. Format: [1, H, W, C_OUT]
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT]
- `[in] bias_data`: Bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [1, H, W, C_OUT]
- `[inout] output_data`: Output data pointer. Data type: int8

int32_t **riscv_depthwise_conv_wrapper_s8_get_buffer_size**(**const** nmsis_nn_dw_conv_params **dw_conv_params*, **const** nmsis_nn_dims **input_dims*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*)

Get size of additional buffer required by riscv_depthwise_conv_wrapper_s8()

Return Size of additional memory required for optimizations in bytes.

Parameters

- `[in] dw_conv_params`: Depthwise convolution parameters (e.g. strides, dilations, pads,…) dw_conv_params->dilation is not used. Range of dw_conv_params->input_offset : [-127, 128] Range of dw_conv_params->input_offset : [-128, 127]
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [H, W, C_IN] Batch argument N is not used and assumed to be 1.
- `[in] filter_dims`: Filter tensor dimensions. Format: [1, H, W, C_OUT]
- `[in] output_dims`: Output tensor dimensions. Format: [1, H, W, C_OUT]

riscv_status **riscv_depthwise_separable_conv_HWC_q7**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out*, q15_t **bufferA*, q7_t **bufferB*)

Q7 depthwise separable convolution function.

Buffer size:

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimension
- `[in] ch_im_in`: number of input tensor channels

- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

bufferA size: 2*ch_im_in*dim_kernel*dim_kernel

bufferB size: 0

Input dimension constraints:

ch_im_in equals ch_im_out

Implementation: There are 3 nested loop here: Inner loop: calculate each output value with MAC instruction over an accumulator Mid loop: loop over different output channel Outer loop: loop over different output (x, y) riscv_status **riscv_depthwise_separable_conv_HWC_q7_nonsquare**(**const** q7_t **Im_in*, **const** uint16_t *dim_im_in_x*, **const** uint16_t *dim_im_in_y*, **const** uint16_t *ch_im_in*, **const** q7_t **wt*, **const** uint16_t *ch_im_out*, **const** uint16_t *dim_kernel_x*, **const** uint16_t *dim_kernel_y*, **const** uint16_t *padding_x*, **const** uint16_t *padding_y*, **const** uint16_t *stride_x*, **const** uint16_t *stride_y*, **const** q7_t **bias*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, q7_t **Im_out*, **const** uint16_t *dim_im_out_x*, **const** uint16_t *dim_im_out_y*, q15_t **bufferA*, q7_t **bufferB*)

Q7 depthwise separable convolution function (non-square shape)

This function is the version with full list of optimization tricks, but with some constraints: ch_im_in is equal to ch_im_out

Return The function returns either `RISCV_MATH_SIZE_MISMATCH` or `RISCV_MATH_SUCCESS` based on the outcome of size checking.

Parameters

- `[in] Im_in`: pointer to input tensor
- `[in] dim_im_in_x`: input tensor dimension x
- `[in] dim_im_in_y`: input tensor dimension y
- `[in] ch_im_in`: number of input tensor channels
- `[in] wt`: pointer to kernel weights
- `[in] ch_im_out`: number of filters, i.e., output tensor channels
- `[in] dim_kernel_x`: filter kernel size x
- `[in] dim_kernel_y`: filter kernel size y
- `[in] padding_x`: padding sizes x
- `[in] padding_y`: padding sizes y
- `[in] stride_x`: convolution stride x

- `[in] stride_y`: convolution stride y
- `[in] bias`: pointer to bias
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[inout] Im_out`: pointer to output tensor
- `[in] dim_im_out_x`: output tensor dimension x
- `[in] dim_im_out_y`: output tensor dimension y
- `[inout] bufferA`: pointer to buffer space for input
- `[inout] bufferB`: pointer to buffer space for output

Fully-connected Layer Functions

riscv_status **riscv_fully_connected_mat_q7_vec_q15** (**const** q15_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_mat_q7_vec_q15_opt** (**const** q15_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_q15** (**const** q15_t **pV*, **const** q15_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q15_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_q15_opt** (**const** q15_t **pV*, **const** q15_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q15_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_q7** (**const** q7_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q7_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_q7_opt** (**const** q7_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q7_t **pOut*, q15_t **vec_buffer*)

riscv_status **riscv_fully_connected_s8** (**const** nmsis_nn_context **ctx*, **const** nmsis_nn_fc_params **fc_params*, **const** nmsis_nn_per_tensor_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*) int32_t **riscv_fully_connected_s8_get_buffer_size** (**const** nmsis_nn_dims **filter_dims*)

USE_INTRINSIC

group **FC**

Collection of fully-connected and matrix multiplication functions.

Fully-connected layer is basically a matrix-vector multiplication with bias. The matrix is the weights and the input/output vectors are the activation values. Supported {weight, activation} precisions include {8-bit, 8-bit}, {16-bit, 16-bit}, and {8-bit, 16-bit}.

Here we have two types of kernel functions. The basic function implements the function using regular GEMV approach. The opt functions operates with weights in interleaved formats.

Defines

USE_INTRINSIC

Mixed Q15-Q7 opt fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: 0

Q7_Q15 version of the fully connected layer

Weights are in q7_t and Activations are in q15_t

Limitation: x4 version requires weight reordering to work

Here we use only one pointer to read 4 rows in the weight matrix. So if the original q7_t matrix looks like this:

| a11 | a12 | a13 | a14 | a15 | a16 | a17 |

| a21 | a22 | a23 | a24 | a25 | a26 | a27 |

| a31 | a32 | a33 | a34 | a35 | a36 | a37 |

| a41 | a42 | a43 | a44 | a45 | a46 | a47 |

| a51 | a52 | a53 | a54 | a55 | a56 | a57 |

| a61 | a62 | a63 | a64 | a65 | a66 | a67 |

We operates on multiple-of-4 rows, so the first four rows becomes

| a11 | a21 | a12 | a22 | a31 | a41 | a32 | a42 |

| a13 | a23 | a14 | a24 | a33 | a43 | a34 | a44 |

| a15 | a25 | a16 | a26 | a35 | a45 | a36 | a46 |

The column left over will be in-order. which is: | a17 | a27 | a37 | a47 |

For the left-over rows, we do 1x1 computation, so the data remains as its original order.

So the stored weight matrix looks like this:

| a11 | a21 | a12 | a22 | a31 | a41 |

| a32 | a42 | a13 | a23 | a14 | a24 |

| a33 | a43 | a34 | a44 | a15 | a25 |

| a16 | a26 | a35 | a45 | a36 | a46 |

| a17 | a27 | a37 | a47 | a51 | a52 |

| a53 | a54 | a55 | a56 | a57 | a61 |

| a62 | a63 | a64 | a65 | a66 | a67 |

Functions

riscv_status **riscv_fully_connected_mat_q7_vec_q15**(**const** q15_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

Mixed Q15-Q7 fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: 0

Q7_Q15 version of the fully connected layer

Weights are in q7_t and Activations are in q15_t riscv_status **riscv_fully_connected_mat_q7_vec_q15_opt** (**const** q15_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

Mixed Q15-Q7 opt fully-connected layer function.

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

riscv_status **riscv_fully_connected_q15** (**const** q15_t **pV*, **const** q15_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q15_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

Q15 opt fully-connected layer function.

Q15 basic fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: 0

riscv_status **riscv_fully_connected_q15_opt** (**const** q15_t **pV*, **const** q15_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q15_t **bias*, q15_t **pOut*, q15_t **vec_buffer*)

Q15 opt fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: 0

Here we use only one pointer to read 4 rows in the weight matrix. So if the original matrix looks like this:

| a11 | a12 | a13 |

| a21 | a22 | a23 |

| a31 | a32 | a33 |

| a41 | a42 | a43 |

| a51 | a52 | a53 |

| a61 | a62 | a63 |

We operates on multiple-of-4 rows, so the first four rows becomes

| a11 | a12 | a21 | a22 | a31 | a32 | a41 | a42 |

| a13 | a23 | a33 | a43 |

Remaining rows are kept the same original order.

So the stored weight matrix looks like this:

| a11 | a12 | a21 | a22 | a31 | a32 | a41 | a42 |

| a13 | a23 | a33 | a43 | a51 | a52 | a53 | a61 |

| a62 | a63 |

riscv_status **riscv_fully_connected_q7** (**const** q7_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q7_t **pOut*, q15_t **vec_buffer*)

Q7 basic fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: dim_vec

This basic function is designed to work with regular weight matrix without interleaving.

riscv_status **riscv_fully_connected_q7_opt**(**const** q7_t **pV*, **const** q7_t **pM*, **const** uint16_t *dim_vec*, **const** uint16_t *num_of_rows*, **const** uint16_t *bias_shift*, **const** uint16_t *out_shift*, **const** q7_t **bias*, q7_t **pOut*, q15_t **vec_buffer*)

Q7 opt fully-connected layer function.

Buffer size:

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] pV`: pointer to input vector
- `[in] pM`: pointer to matrix weights
- `[in] dim_vec`: length of the vector
- `[in] num_of_rows`: number of rows in weight matrix
- `[in] bias_shift`: amount of left-shift for bias
- `[in] out_shift`: amount of right-shift for output
- `[in] bias`: pointer to bias
- `[inout] pOut`: pointer to output vector
- `[inout] vec_buffer`: pointer to buffer space for input

vec_buffer size: dim_vec

This opt function is designed to work with interleaved weight matrix. The vector input is assumed in q7_t format, we call riscv_q7_to_q15_no_shift_shuffle function to expand into q15_t format with certain weight re-ordering, refer to the function comments for more details. Here we use only one pointer to read 4 rows in the weight matrix. So if the original q7_t matrix looks like this:

| a11 | a12 | a13 | a14 | a15 | a16 | a17 |

| a21 | a22 | a23 | a24 | a25 | a26 | a27 |

| a31 | a32 | a33 | a34 | a35 | a36 | a37 |

| a41 | a42 | a43 | a44 | a45 | a46 | a47 |

| a51 | a52 | a53 | a54 | a55 | a56 | a57 |

| a61 | a62 | a63 | a64 | a65 | a66 | a67 |

We operates on multiple-of-4 rows, so the first four rows becomes

| a11 | a21 | a13 | a23 | a31 | a41 | a33 | a43 |

| a12 | a22 | a14 | a24 | a32 | a42 | a34 | a44 |

| a15 | a25 | a35 | a45 | a16 | a26 | a36 | a46 |

So within the kernel, we first read the re-ordered vector in as:

| b1 | b3 | and | b2 | b4 |

the four q31_t weights will look like

| a11 | a13 |, | a21 | a23 |, | a31 | a33 |, | a41 | a43 |

| a12 | a14 |, | a22 | a24 |, | a32 | a34 |, | a42 | a44 |

The column left over will be in-order. which is:

| a17 | a27 | a37 | a47 |

For the left-over rows, we do 1x1 computation, so the data remains as its original order.

So the stored weight matrix looks like this:

| a11 | a21 | a13 | a23 | a31 | a41 |

| a33 | a43 | a12 | a22 | a14 | a24 |

| a32 | a42 | a34 | a44 | a15 | a25 |

| a35 | a45 | a16 | a26 | a36 | a46 |

| a17 | a27 | a37 | a47 | a51 | a52 |

| a53 | a54 | a55 | a56 | a57 | a61 |

| a62 | a63 | a64 | a65 | a66 | a67 |

riscv_status **riscv_fully_connected_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_fc_params **fc_params*, **const** nmsis_nn_per_tensor_quant_params **quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** q7_t **filter_data*, **const** nmsis_nn_dims **bias_dims*, **const** int32_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

Basic s8 Fully Connected function.

- Supported framework: TensorFlow Lite
- q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[inout] ctx`: Function context (e.g. temporary buffer). Check the function definition file to see if an additional buffer is required. Optional function {API}_get_buffer_size() provides the buffer size if an additional buffer is required.
- `[in] fc_params`: Fully Connected layer parameters (e.g. strides, dilations, pads,...) Range of fc_params->input_offset : [-127, 128] fc_params->filter_offset : 0 Range of fc_params->output_offset : [-128, 127]
- `[in] quant_params`: Per-tensor quantization info. It contains the multiplier and shift values to be applied to the output tensor.
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [N, H, W, C_IN] Input dimension is taken as Nx(H * W * C_IN)
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Two dimensional filter dimensions. Format: [N, C] N : accumulation depth and equals (H * W * C_IN) from input_dims C : output depth and equals C_OUT in output_dims H & W : Not used
- `[in] filter_data`: Filter data pointer. Data type: int8
- `[in] bias_dims`: Bias tensor dimensions. Format: [C_OUT] N, H, W : Not used
- `[in] bias_data`: Bias data pointer. Data type: int32
- `[in] output_dims`: Output tensor dimensions. Format: [N, C_OUT] N : Batches C_OUT : Output depth H & W : Not used.
- `[inout] output_data`: Output data pointer. Data type: int8

int32_t **riscv_fully_connected_s8_get_buffer_size**(**const** nmsis_nn_dims **filter_dims*)

Get the required buffer size for S8 basic fully-connected and matrix multiplication layer function for TF Lite.

Return The function returns required buffer size in bytes

Parameters

- `[in] filter_dims`: dimension of filter

Pooling Functions

riscv_status **riscv_avgpool_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_pool_params **pool_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

int32_t **riscv_avgpool_s8_get_buffer_size**(**const** int *dim_dst_width*, **const** int *ch_src*)

riscv_status **riscv_max_pool_s8**(**const** nmsis_nn_context **ctx*, **const** nmsis_nn_pool_params **pool_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

void **riscv_maxpool_q7_HWC**(q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** uint16_t *dim_im_out*, q7_t **bufferA*, q7_t **Im_out*) void **riscv_avepool_q7_HWC** (q7_t *Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** uint16_t *dim_im_out*, q7_t **bufferA*, q7_t **Im_out*)

group **Pooling**

Perform pooling functions, including max pooling and average pooling

Functions

riscv_status **riscv_avgpool_s8** (**const** nmsis_nn_context **ctx*, **const** nmsis_nn_pool_params **pool_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

s8 average pooling function.

- Supported Framework: TensorFlow Lite

Return The function returns `RISCV_MATH_SUCCESS` - Successful operation

Parameters

- `[inout] ctx`: Function context (e.g. temporary buffer). Check the function definition file to see if an additional buffer is required. Optional function {API}_get_buffer_size() provides the buffer size if an additional buffer is required.
- `[in] pool_params`: Pooling parameters
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [H, W, C_IN] Argument 'N' is not used.
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [H, W] Argument N and C are not used.
- `[in] output_dims`: Output tensor dimensions. Format: [H, W, C_OUT] Argument N is not used. C_OUT equals C_IN.
- `[inout] output_data`: Output data pointer. Data type: int8

int32_t **riscv_avgpool_s8_get_buffer_size** (**const** int *dim_dst_width*, **const** int *ch_src*)

Get the required buffer size for S8 average pooling function.

Return The function returns required buffer size in bytes

Parameters

- `[in] dim_dst_width`: output tensor dimension
- `[in] ch_src`: number of input tensor channels

riscv_status **riscv_max_pool_s8** (**const** nmsis_nn_context **ctx*, **const** nmsis_nn_pool_params **pool_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **filter_dims*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

s8 max pooling function.

- Supported Framework: TensorFlow Lite

Return The function returns `RISCV_MATH_SUCCESS` - Successful operation

Parameters

- `[inout] ctx`: Function context (e.g. temporary buffer). Check the function definition file to see if an additional buffer is required. Optional function {API}_get_buffer_size() provides the buffer size if an additional buffer is required.
- `[in] pool_params`: Pooling parameters
- `[in] input_dims`: Input (activation) tensor dimensions. Format: [H, W, C_IN] Argument 'N' is not used.
- `[in] input_data`: Input (activation) data pointer. Data type: int8
- `[in] filter_dims`: Filter tensor dimensions. Format: [H, W] Argument N and C are not used.
- `[in] output_dims`: Output tensor dimensions. Format: [H, W, C_OUT] Argument N is not used. C_OUT equals C_IN.
- `[inout] output_data`: Output data pointer. Data type: int8

void **riscv_maxpool_q7_HWC**(q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** uint16_t *dim_im_out*, q7_t **bufferA*, q7_t **Im_out*)

Q7 max pooling function.

The pooling function is implemented as split x-pooling then y-pooling.

Parameters

- `[inout] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels
- `[in] dim_kernel`: filter kernel size
- `[in] padding`: padding sizes
- `[in] stride`: convolution stride
- `[in] dim_im_out`: output tensor dimension
- `[inout] bufferA`: Not used
- `[inout] Im_out`: pointer to output tensor

This pooling function is input-destructive. Input data is undefined after calling this function.

void **riscv_avepool_q7_HWC**(q7_t **Im_in*, **const** uint16_t *dim_im_in*, **const** uint16_t *ch_im_in*, **const** uint16_t *dim_kernel*, **const** uint16_t *padding*, **const** uint16_t *stride*, **const** uint16_t *dim_im_out*, q7_t **bufferA*, q7_t **Im_out*)

Q7 average pooling function.

Buffer size:

Parameters

- `[inout] Im_in`: pointer to input tensor
- `[in] dim_im_in`: input tensor dimention
- `[in] ch_im_in`: number of input tensor channels

- [in] dim_kernel: filter kernel size
- [in] padding: padding sizes
- [in] stride: convolution stride
- [in] dim_im_out: output tensor dimension
- [inout] bufferA: pointer to buffer space for input
- [inout] Im_out: pointer to output tensor

bufferA size: 2*dim_im_out*ch_im_in

The pooling function is implemented as split x-pooling then y-pooling.

This pooling function is input-destructive. Input data is undefined after calling this function.

Reshape Functions

void **riscv_reshape_s8** (**const** int8_t **input*, int8_t **output*, **const** uint32_t *total_size*)

group **Reshape**

Functions

void **riscv_reshape_s8** (**const** int8_t **input*, int8_t **output*, **const** uint32_t *total_size*)

Reshape a s8 vector into another with different shape.

Basic s8 reshape function.

Refer header file for details.

Softmax Functions

void **riscv_softmax_q15** (**const** q15_t **vec_in*, **const** uint16_t *dim_vec*, q15_t **p_out*)

void **riscv_softmax_q7** (**const** q7_t **vec_in*, **const** uint16_t *dim_vec*, q7_t **p_out*)

void **riscv_softmax_s8** (**const** int8_t **input*, **const** int32_t *num_rows*, **const** int32_t *row_size*, **const** int32_t *mult*, **const** int32_t *shift*, **const** int32_t *diff_min*, int8_t **output*)

void **riscv_softmax_u8** (**const** uint8_t **input*, **const** int32_t *num_rows*, **const** int32_t *row_size*, **const** int32_t *mult*, **const** int32_t *shift*, **const** int32_t *diff_min*, uint8_t **output*)

void **riscv_softmax_with_batch_q7** (**const** q7_t **vec_in*, **const** uint16_t *nb_batches*, **const** uint16_t *dim_vec*, q7_t **p_out*)

group **Softmax**

EXP(2) based softmax functions.

Functions

void **riscv_softmax_q15** (**const** q15_t **vec_in*, **const** uint16_t *dim_vec*, q15_t **p_out*)

Q15 softmax function.

Here, instead of typical e based softmax, we use 2-based softmax, i.e.,:

Parameters

- `[in] vec_in`: pointer to input vector
- `[in] dim_vec`: input vector dimention
- `[out] p_out`: pointer to output vector

y_i = 2^(x_i) / sum(2^x_j)

The relative output will be different here. But mathematically, the gradient will be the same with a log(2) scaling factor.

void **riscv_softmax_q7**(**const** q7_t **vec_in*, **const** uint16_t *dim_vec*, q7_t **p_out*)

Q7 softmax function.

Here, instead of typical natural logarithm e based softmax, we use 2-based softmax here, i.e.,:

Parameters

- `[in] vec_in`: pointer to input vector
- `[in] dim_vec`: input vector dimention
- `[out] p_out`: pointer to output vector

y_i = 2^(x_i) / sum(2^x_j)

The relative output will be different here. But mathematically, the gradient will be the same with a log(2) scaling factor.

void **riscv_softmax_s8**(**const** int8_t **input*, **const** int32_t *num_rows*, **const** int32_t *row_size*, **const** int32_t *mult*, **const** int32_t *shift*, **const** int32_t *diff_min*, int8_t **output*)

S8 softmax function.

Note Supported framework: TensorFlow Lite micro (bit-accurate)

Parameters

- `[in] input`: Pointer to the input tensor
- `[in] num_rows`: Number of rows in the input tensor
- `[in] row_size`: Number of elements in each input row
- `[in] mult`: Input quantization multiplier
- `[in] shift`: Input quantization shift within the range [0, 31]
- `[in] diff_min`: Minimum difference with max in row. Used to check if the quantized exponential operation can be performed
- `[out] output`: Pointer to the output tensor

void **riscv_softmax_u8**(**const** uint8_t **input*, **const** int32_t *num_rows*, **const** int32_t *row_size*, **const** int32_t *mult*, **const** int32_t *shift*, **const** int32_t *diff_min*, uint8_t **output*)

U8 softmax function.

Note Supported framework: TensorFlow Lite micro (bit-accurate)

Parameters

- `[in] input`: Pointer to the input tensor
- `[in] num_rows`: Number of rows in the input tensor
- `[in] row_size`: Number of elements in each input row

- `[in] mult`: Input quantization multiplier
- `[in] shift`: Input quantization shift within the range [0, 31]
- `[in] diff_min`: Minimum difference with max in row. Used to check if the quantized exponential operation can be performed
- `[out] output`: Pointer to the output tensor

void **riscv_softmax_with_batch_q7** (**const** q7_t **vec_in*, **const** uint16_t *nb_batches*, **const** uint16_t *dim_vec*, q7_t **p_out*)

Q7 softmax function with batch parameter.

Here, instead of typical natural logarithm e based softmax, we use 2-based softmax here, i.e.,:

Parameters

- `[in] vec_in`: pointer to input vector
- `[in] nb_batches`: number of batches
- `[in] dim_vec`: input vector dimention
- `[out] p_out`: pointer to output vector

y_i = 2^(x_i) / sum(2^x_j)

The relative output will be different here. But mathematically, the gradient will be the same with a log(2) scaling factor.

SVDF Layer Functions

riscv_status **riscv_svdf_s8** (**const** nmsis_nn_context **input_ctx*, **const** nmsis_nn_context **output_ctx*, **const** nmsis_nn_svdf_params **svdf_params*, **const** nmsis_nn_per_tensor_quant_params **input_quant_params*, **const** nmsis_nn_per_tensor_quant_params **output_quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **state_dims*, q15_t **state_data*, **const** nmsis_nn_dims **weights_feature_dims*, **const** q7_t **weights_feature_data*, **const** nmsis_nn_dims **weights_time_dims*, **const** q15_t **weights_time_data*, **const** nmsis_nn_dims **bias_dims*, **const** q31_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

group **SVDF**

Functions

riscv_status **riscv_svdf_s8** (**const** nmsis_nn_context **input_ctx*, **const** nmsis_nn_context **output_ctx*, **const** nmsis_nn_svdf_params **svdf_params*, **const** nmsis_nn_per_tensor_quant_params **input_quant_params*, **const** nmsis_nn_per_tensor_quant_params **output_quant_params*, **const** nmsis_nn_dims **input_dims*, **const** q7_t **input_data*, **const** nmsis_nn_dims **state_dims*, q15_t **state_data*, **const** nmsis_nn_dims **weights_feature_dims*, **const** q7_t **weights_feature_data*, **const** nmsis_nn_dims **weights_time_dims*, **const** q15_t **weights_time_data*, **const** nmsis_nn_dims **bias_dims*, **const** q31_t **bias_data*, **const** nmsis_nn_dims **output_dims*, q7_t **output_data*)

s8 SVDF function

1. Supported framework: TensorFlow Lite micro
2. q7 is used as data type eventhough it is s8 data. It is done so to be consistent with existing APIs.

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] input_ctx`: Temporary scratch buffer
- `[in] output_ctx`: Temporary output scratch buffer
- `[in] svdf_params`: SVDF Parameters Range of svdf_params->input_offset : [-128, 127] Range of svdf_params->output_offset : [-128, 127]
- `[in] input_quant_params`: Input quantization parameters
- `[in] output_quant_params`: Output quantization parameters
- `[in] input_dims`: Input tensor dimensions
- `[in] input_data`: Pointer to input tensor
- `[in] state_dims`: State tensor dimensions
- `[in] state_data`: Pointer to state tensor
- `[in] weights_feature_dims`: Weights (feature) tensor dimensions
- `[in] weights_feature_data`: Pointer to the weights (feature) tensor
- `[in] weights_time_dims`: Weights (time) tensor dimensions
- `[in] weights_time_data`: Pointer to the weights (time) tensor
- `[in] bias_dims`: Bias tensor dimensions
- `[in] bias_data`: Pointer to bias tensor
- `[in] output_dims`: Output tensor dimensions
- `[out] output_data`: Pointer to the output tensor

group **groupNN**

A collection of functions to perform basic operations for neural network layers. Functions with a _s8 suffix support TensorFlow Lite framework.

4.3.2 Neural Network Data Conversion Functions

void **riscv_q7_to_q15_no_shift**(**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_q7_to_q15_reordered_no_shift**(**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

void **riscv_q7_to_q15_reordered_with_offset**(**const** q7_t **src*, q15_t **dst*, uint32_t *block_size*, q15_t *offset*)

void **riscv_q7_to_q15_with_offset**(**const** q7_t **src*, q15_t **dst*, uint32_t *block_size*, q15_t *offset*)

void **riscv_q7_to_q7_no_shift**(**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

void **riscv_q7_to_q7_reordered_no_shift**(**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

group **nndata_convert**

Perform data type conversion in-between neural network operations

Functions

void **riscv_q7_to_q15_no_shift** (**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q7 vector to Q15 vector without left-shift.

Converts the elements of the q7 vector to q15 vector without left-shift.

The equation used for the conversion process is:

Description:

Parameters

- `[in] *pSrc`: points to the Q7 input vector
- `[out] *pDst`: points to the Q15 output vector
- `[in] blockSize`: length of the input vector

void **riscv_q7_to_q15_reordered_no_shift** (**const** q7_t **pSrc*, q15_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q7 vector to reordered Q15 vector without left-shift.

Converts the elements of the q7 vector to reordered q15 vector without left-shift.

This function does the q7 to q15 expansion with re-ordering

Parameters

- `[in] *pSrc`: points to the Q7 input vector
- `[out] *pDst`: points to the Q15 output vector
- `[in] blockSize`: length of the input vector

is converted into:

This looks strange but is natural considering how sign-extension is done at assembly level.

The expansion of other other oprand will follow the same rule so that the end results are the same.

The tail (i.e., last (N % 4) elements) will still be in original order.

void **riscv_q7_to_q15_reordered_with_offset** (**const** q7_t **src*, q15_t **dst*, uint32_t *block_size*, q15_t *offset*)

Converts the elements of the Q7 vector to a reordered Q15 vector with an added offset.

Converts the elements of the q7 vector to reordered q15 vector with an added offset.

Note Refer header file for details.

void **riscv_q7_to_q15_with_offset** (**const** q7_t **src*, q15_t **dst*, uint32_t *block_size*, q15_t *offset*)

Converts the elements from a q7 vector to a q15 vector with an added offset.

The equation used for the conversion process is:

Description:

Parameters

- `[in] src`: pointer to the q7 input vector
- `[out] dst`: pointer to the q15 output vector
- `[in] block_size`: length of the input vector
- `[in] offset`: q7 offset to be added to each input vector element.

void **riscv_q7_to_q7_no_shift** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q7 vector to Q7 vector without left-shift.

The equation used for the conversion process is:

Return none.

Description:

Parameters

- `[in] *pSrc`: points to the Q7 input vector
- `[out] *pDst`: points to the Q7 output vector
- `[in] blockSize`: length of the input vector

void **riscv_q7_to_q7_reordered_no_shift** (**const** q7_t **pSrc*, q7_t **pDst*, uint32_t *blockSize*)

Converts the elements of the Q7 vector to reordered Q7 vector without left-shift.

This function does the q7 to q7 expansion with re-ordering

Return none.

Parameters

- `[in] *pSrc`: points to the Q7 input vector
- `[out] *pDst`: points to the Q7 output vector
- `[in] blockSize`: length of the input vector

is converted into:

This looks strange but is natural considering how sign-extension is done at assembly level.

The expansion of other other oprand will follow the same rule so that the end results are the same.

The tail (i.e., last (N % 4) elements) will still be in original order.

4.3.3 Basic Math Functions for Neural Network Computation

void **riscv_nn_accumulate_q7_to_q15** (q15_t **dst*, **const** q7_t **src*, uint32_t *block_size*)

void **riscv_nn_accumulate_q7_to_q7** (q7_t **dst*, **const** q7_t **src*, uint32_t *block_size*)

void **riscv_nn_add_q7** (**const** q7_t **input*, q31_t **output*, uint32_t *block_size*)

q7_t ***riscv_nn_depthwise_conv_nt_t_padded_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, **const** int32_t *lhs_offset*, **const** uint16_t *num_ch*, **const** int32_t **out_shift*, **const** int32_t **out_mult*, **const** int32_t *out_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*, **const** uint16_t *row_x_col*, **const** int32_t ***const** *output_bias*, q7_t **out*)

q7_t ***riscv_nn_depthwise_conv_nt_t_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, **const** int32_t *lhs_offset*, **const** uint16_t *num_ch*, **const** int32_t **out_shift*, **const** int32_t **out_mult*, **const** int32_t *out_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*, **const** uint16_t *row_x_col*, **const** int32_t ***const** *output_bias*, q7_t **out*) riscv_status **riscv_nn_mat_mul_core_1x_s8** (int32_t *row_elements*, **const** int8_t **row_base*, **const** int8_t **col_base*, int32_t ***const** *sum_col*, int32_t ***const** *output*)

riscv_status **riscv_nn_mat_mul_core_4x_s8** (**const** int32_t *row_elements*, **const** int32_t *offset*, **const** int8_t **row_base*, **const** int8_t **col_base*, int32_t ***const** *sum_col*, int32_t ***const** *output*)

riscv_status **riscv_nn_mat_mult_nt_t_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, **const** q31_t **bias*, q7_t **dst*, **const** int32_t **dst_multipliers*, **const** int32_t **dst_shifts*, **const** int32_t *lhs_rows*, **const** int32_t *rhs_rows*, **const** int32_t *rhs_cols*, **const** int32_t *lhs_offset*, **const** int32_t *dst_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

void **riscv_nn_mult_q15** (q15_t **pSrcA*, q15_t **pSrcB*, q15_t **pDst*, **const** uint16_t *out_shift*, uint32_t *blockSize*)

void **riscv_nn_mult_q7** (q7_t **pSrcA*, q7_t **pSrcB*, q7_t **pDst*, **const** uint16_t *out_shift*, uint32_t *blockSize*)

riscv_status **riscv_nn_vec_mat_mult_t_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, **const** q31_t **bias*, q7_t **dst*, **const** int32_t *lhs_offset*, **const** int32_t *rhs_offset*, **const** int32_t *dst_offset*, **const** int32_t *dst_multiplier*, **const** int32_t *dst_shift*, **const** int32_t *rhs_cols*, **const** int32_t *rhs_rows*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

riscv_status **riscv_nn_vec_mat_mult_t_svdf_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, q15_t **dst*, **const** int32_t *lhs_offset*, **const** int32_t *rhs_offset*, **const** int32_t *scatter_offset*, **const** int32_t *dst_multiplier*, **const** int32_t *dst_shift*, **const** int32_t *rhs_cols*, **const** int32_t *rhs_rows*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

group **NNBasicMath**

Basic Math Functions for Neural Network Computation

Functions

void **riscv_nn_accumulate_q7_to_q15** (q15_t **dst*, **const** q7_t **src*, uint32_t *block_size*)

Converts the elements from a q7 vector and accumulate to a q15 vector.

The equation used for the conversion process is:

Description:

Parameters

- `[in] *src`: points to the q7 input vector
- `[out] *dst`: points to the q15 output vector
- `[in] block_size`: length of the input vector

void **riscv_nn_accumulate_q7_to_q7** (q7_t **dst*, **const** q7_t **src*, uint32_t *block_size*)

Converts the elements from a q7 vector and accumulate to a q7 vector.

The equation used for the conversion process is:

Description:

Parameters

- `[in] *src`: points to the q7 input vector
- `[out] *dst`: points to the q7 output vector
- `[in] block_size`: length of the input vector

void **riscv_nn_add_q7**(**const** q7_t **input*, q31_t **output*, uint32_t *block_size*)

Non-saturating addition of elements of a q7 vector.

2^24 samples can be added without saturating the result.

Description:

Parameters

- `[in] *input`: Pointer to the q7 input vector
- `[out] *output`: Pointer to the q31 output variable.
- `[in] block_size`: length of the input vector

The equation used for the conversion process is:

q7_t ***riscv_nn_depthwise_conv_nt_t_padded_s8**(**const** q7_t **lhs*, **const** q7_t **rhs*, **const** int32_t *lhs_offset*, **const** uint16_t *num_ch*, **const** int32_t **out_shift*, **const** int32_t **out_mult*, **const** int32_t *out_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*, **const** uint16_t *row_x_col*, **const** int32_t ***const** *output_bias*, q7_t **out*)

Depthwise convolution of transposed rhs matrix with 4 lhs matrices. To be used in padded cases where the padding is -lhs_offset(Range: int8). Dimensions are the same for lhs and rhs.

Return The function returns one of the two

- Updated output pointer if an implementation is available
- NULL if no implementation is available.

Note If number of channels is not a multiple of 4, upto 3 elements outside the boundary will be read out for the following.

- Output shift
- Output multiplier
- Output bias
- rhs

Parameters

- `[in] lhs`: Input left-hand side matrix
- `[in] rhs`: Input right-hand side matrix (transposed)
- `[in] lhs_offset`: LHS matrix offset(input offset). Range: -127 to 128
- `[in] num_ch`: Number of channels in LHS/RHS
- `[in] out_shift`: Per channel output shift. Length of vector is equal to number of channels

- `[in] out_mult`: Per channel output multiplier. Length of vector is equal to number of channels
- `[in] out_offset`: Offset to be added to the output values. Range: -127 to 128
- `[in] activation_min`: Minimum value to clamp the output to. Range: int8
- `[in] activation_max`: Maximum value to clamp the output to. Range: int8
- `[in] row_x_col`: (row_dimension * col_dimension) of LHS/RHS matrix
- `[in] output_bias`: Per channel output bias. Length of vector is equal to number of channels
- `[in] out`: Output pointer

q7_t ***riscv_nn_depthwise_conv_nt_t_s8**(**const** q7_t **lhs*, **const** q7_t **rhs*, **const** int32_t *lhs_offset*, **const** uint16_t *num_ch*, **const** int32_t **out_shift*, **const** int32_t **out_mult*, **const** int32_t *out_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*, **const** uint16_t *row_x_col*, **const** int32_t ***const** *output_bias*, q7_t **out*)

Depthwise convolution of transposed rhs matrix with 4 lhs matrices. To be used in non-padded cases. Dimensions are the same for lhs and rhs.

Return The function returns one of the two

- Updated output pointer if an implementation is available
- NULL if no implementation is available.

Note If number of channels is not a multiple of 4, upto 3 elements outside the boundary will be read out for the following.

- Output shift
- Output multiplier
- Output bias
- rhs

Parameters

- `[in] lhs`: Input left-hand side matrix
- `[in] rhs`: Input right-hand side matrix (transposed)
- `[in] lhs_offset`: LHS matrix offset(input offset). Range: -127 to 128
- `[in] num_ch`: Number of channels in LHS/RHS
- `[in] out_shift`: Per channel output shift. Length of vector is equal to number of channels.
- `[in] out_mult`: Per channel output multiplier. Length of vector is equal to number of channels.
- `[in] out_offset`: Offset to be added to the output values. Range: -127 to 128
- `[in] activation_min`: Minimum value to clamp the output to. Range: int8
- `[in] activation_max`: Maximum value to clamp the output to. Range: int8
- `[in] row_x_col`: (row_dimension * col_dimension) of LHS/RHS matrix
- `[in] output_bias`: Per channel output bias. Length of vector is equal to number of channels.

- [in] out: Output pointer

riscv_status **riscv_nn_mat_mul_core_1x_s8** (int32_t *row_elements*, **const** int8_t **row_base*, **const** int8_t **col_base*, int32_t ***const** *sum_col*, int32_t ***const** *output*)

General Matrix-multiplication without requantization for one row & one column.

Pseudo-code *output = 0 sum_col = 0 for (i = 0; i < row_elements; i++) *output += row_base[i] * col_base[i] sum_col += col_base[i]

Return The function returns the multiply-accumulated result of the row by column.

Parameters

- [in] row_elements: number of row elements
- [in] row_base: pointer to row operand
- [in] col_base: pointer to col operand
- [out] sum_col: pointer to store sum of column elements
- [out] output: pointer to store result of multiply-accumulate

riscv_status **riscv_nn_mat_mul_core_4x_s8** (**const** int32_t *row_elements*, **const** int32_t *offset*, **const** int8_t **row_base*, **const** int8_t **col_base*, int32_t ***const** *sum_col*, int32_t ***const** *output*)

General Matrix-multiplication without requantization for four rows and one column.

Pseudo-code output[0] = 0 .. output[3] = 0 sum_col = 0 for (i = 0; i < row_elements; i++) output[0] += row_base[i] * col_base[i] .. output[3] += row_base[i + (row_elements * 3)] * col_base[i] sum_col += col_base[i]

Return The function returns the multiply-accumulated result of the row by column

Parameters

- [in] row_elements: number of row elements
- [in] offset: offset between rows. Can be the same as row_elements. For e.g, in a 1x1 conv scenario with stride as 1.
- [in] row_base: pointer to row operand
- [in] col_base: pointer to col operand
- [out] sum_col: pointer to store sum of column elements
- [out] output: pointer to store result(4 int32's) of multiply-accumulate

riscv_status **riscv_nn_mat_mult_nt_t_s8** (**const** q7_t **lhs*, **const** q7_t **rhs*, **const** q31_t **bias*, q7_t **dst*, **const** int32_t **dst_multipliers*, **const** int32_t **dst_shifts*, **const** int32_t *lhs_rows*, **const** int32_t *rhs_rows*, **const** int32_t *rhs_cols*, **const** int32_t *lhs_offset*, **const** int32_t *dst_offset*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

General Matrix-multiplication function with per-channel requantization. This function assumes:

- LHS input matrix NOT transposed (nt)
- RHS input matrix transposed (t)

Note This operation also performs the broadcast bias addition before the requantization

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] lhs`: Pointer to the LHS input matrix
- `[in] rhs`: Pointer to the RHS input matrix
- `[in] bias`: Pointer to the bias vector. The length of this vector is equal to the number of output columns (or RHS input rows)
- `[out] dst`: Pointer to the output matrix with "m" rows and "n" columns
- `[in] dst_multipliers`: Pointer to the multipliers vector needed for the per-channel requantization. The length of this vector is equal to the number of output columns (or RHS input rows)
- `[in] dst_shifts`: Pointer to the shifts vector needed for the per-channel requantization. The length of this vector is equal to the number of output columns (or RHS input rows)
- `[in] lhs_rows`: Number of LHS input rows
- `[in] rhs_rows`: Number of RHS input rows
- `[in] rhs_cols`: Number of LHS/RHS input columns
- `[in] lhs_offset`: Offset to be applied to the LHS input value
- `[in] dst_offset`: Offset to be applied the output result
- `[in] activation_min`: Minimum value to clamp down the output. Range : int8
- `[in] activation_max`: Maximum value to clamp up the output. Range : int8

void **riscv_nn_mult_q15** (q15_t **pSrcA*, q15_t **pSrcB*, q15_t **pDst*, **const** uint16_t *out_shift*, uint32_t *blockSize*)

Q7 vector multiplication with variable output shifts.

q7 vector multiplication with variable output shifts

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q15 range [0x8000 0x7FFF] will be saturated.

Parameters

- `[in] *pSrcA`: pointer to the first input vector
- `[in] *pSrcB`: pointer to the second input vector
- `[out] *pDst`: pointer to the output vector
- `[in] out_shift`: amount of right-shift for output
- `[in] blockSize`: number of samples in each vector

void **riscv_nn_mult_q7** (q7_t **pSrcA*, q7_t **pSrcB*, q7_t **pDst*, **const** uint16_t *out_shift*, uint32_t *blockSize*)

Q7 vector multiplication with variable output shifts.

q7 vector multiplication with variable output shifts

Scaling and Overflow Behavior:

The function uses saturating arithmetic. Results outside of the allowable Q7 range [0x80 0x7F] will be saturated.

Parameters

- `[in] *pSrcA`: pointer to the first input vector
- `[in] *pSrcB`: pointer to the second input vector
- `[out] *pDst`: pointer to the output vector
- `[in] out_shift`: amount of right-shift for output
- `[in] blockSize`: number of samples in each vector

riscv_status **riscv_nn_vec_mat_mult_t_s8**(**const** q7_t **lhs*, **const** q7_t **rhs*, **const** q31_t **bias*, q7_t **dst*, **const** int32_t *lhs_offset*, **const** int32_t *rhs_offset*, **const** int32_t *dst_offset*, **const** int32_t *dst_multiplier*, **const** int32_t *dst_shift*, **const** int32_t *rhs_cols*, **const** int32_t *rhs_rows*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

s8 Vector by Matrix (transposed) multiplication

Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] lhs`: Input left-hand side vector
- `[in] rhs`: Input right-hand side matrix (transposed)
- `[in] bias`: Input bias
- `[out] dst`: Output vector
- `[in] lhs_offset`: Offset to be added to the input values of the left-hand side vector. Range: -127 to 128
- `[in] rhs_offset`: Not used
- `[in] dst_offset`: Offset to be added to the output values. Range: -127 to 128
- `[in] dst_multiplier`: Output multiplier
- `[in] dst_shift`: Output shift
- `[in] rhs_cols`: Number of columns in the right-hand side input matrix
- `[in] rhs_rows`: Number of rows in the right-hand side input matrix
- `[in] activation_min`: Minimum value to clamp the output to. Range: int8
- `[in] activation_max`: Maximum value to clamp the output to. Range: int8

riscv_status **riscv_nn_vec_mat_mult_t_svdf_s8**(**const** q7_t **lhs*, **const** q7_t **rhs*, q15_t **dst*, **const** int32_t *lhs_offset*, **const** int32_t *rhs_offset*, **const** int32_t *scatter_offset*, **const** int32_t *dst_multiplier*, **const** int32_t *dst_shift*, **const** int32_t *rhs_cols*, **const** int32_t *rhs_rows*, **const** int32_t *activation_min*, **const** int32_t *activation_max*)

s8 Vector by Matrix (transposed) multiplication with s16 output

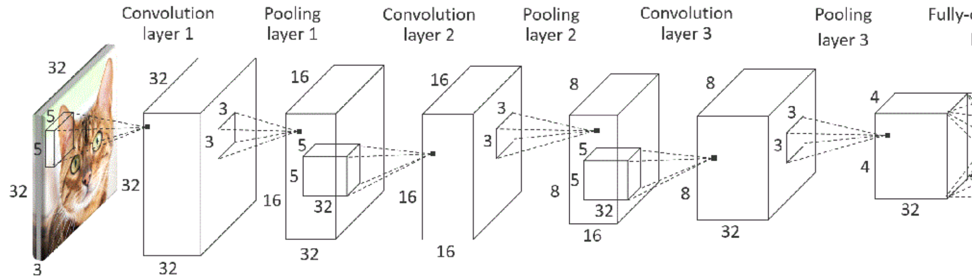
Return The function returns `RISCV_MATH_SUCCESS`

Parameters

- `[in] lhs`: Input left-hand side vector
- `[in] rhs`: Input right-hand side matrix (transposed)
- `[out] dst`: Output vector
- `[in] lhs_offset`: Offset to be added to the input values of the left-hand side vector. Range: -127 to 128
- `[in] rhs_offset`: Not used
- `[in] scatter_offset`: Address offset for dst. First output is stored at 'dst', the second at 'dst + scatter_offset' and so on.
- `[in] dst_multiplier`: Output multiplier
- `[in] dst_shift`: Output shift
- `[in] rhs_cols`: Number of columns in the right-hand side input matrix
- `[in] rhs_rows`: Number of rows in the right-hand side input matrix
- `[in] activation_min`: Minimum value to clamp the output to. Range: int16
- `[in] activation_max`: Maximum value to clamp the output to. Range: int16

4.3.4 Convolutional Neural Network Example

group **CNNExample**



Refer riscv_nnexamples_cifar10.cpp

Description:

Demonstrates a convolutional neural network (CNN) example with the use of convolution, ReLU activation, pooling and fully-connected functions.

Model definition:

The CNN used in this example is based on CIFAR-10 example from Caffe [1]. The neural network consists of 3 convolution layers interspersed by ReLU activation and max pooling layers, followed by a fully-connected layer at the end. The input to the network is a 32x32 pixel color image, which will be classified into one of the 10 output classes. This example model implementation needs 32.3 KB to store weights, 40 KB for activations and 3.1 KB for storing the `im2col` data.

Variables Description:

- `conv1_wt`, `conv2_wt`, `conv3_wt` are convolution layer weight matrices
- `conv1_bias`, `conv2_bias`, `conv3_bias` are convolution layer bias arrays

- `ip1_wt`, ip1_bias point to fully-connected layer weights and biases
- `input_data` points to the input image data
- `output_data` points to the classification output
- `col_buffer` is a buffer to store the `im2col` output
- `scratch_buffer` is used to store the activation data (intermediate layer outputs)

NMSIS DSP Software Library Functions Used:

- riscv_convolve_HWC_q7_RGB()
- riscv_convolve_HWC_q7_fast()
- riscv_relu_q7()
- riscv_maxpool_q7_HWC()
- riscv_avepool_q7_HWC()
- riscv_fully_connected_q7_opt()
- riscv_fully_connected_q7()

[1] https://github.com/BVLC/caffe

4.3.5 Gated Recurrent Unit Example

group **`GRUExample`**

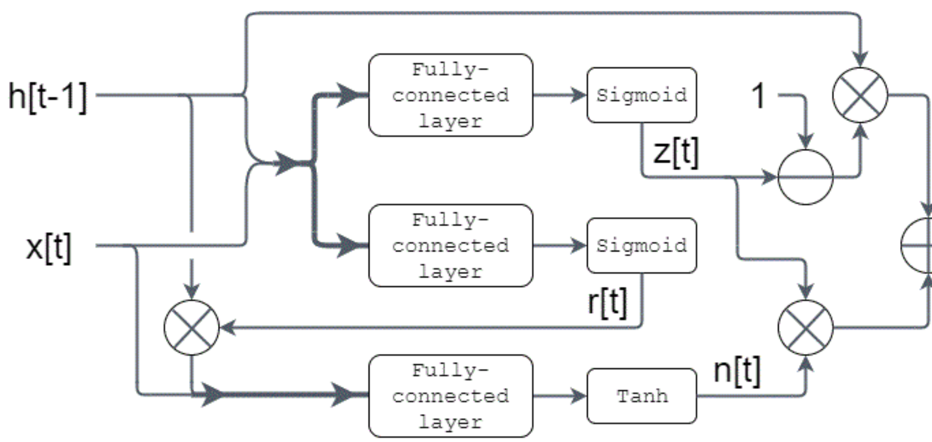
Refer riscv_nnexamples_gru.cpp

Description:

Demonstrates a gated recurrent unit (GRU) example with the use of fully-connected, Tanh/Sigmoid activation functions.

Model definition:

GRU is a type of recurrent neural network (RNN). It contains two sigmoid gates and one hidden state.



The computation can be summarized as:

Variables Description:

- `update_gate_weights`, `reset_gate_weights`, `hidden_state_weights` are weights corresponding to update gate (W_z), reset gate (W_r), and hidden state (W_n).
- `update_gate_bias`, `reset_gate_bias`, `hidden_state_bias` are layer bias arrays
- `test_input1`, `test_input2`, `test_history` are the inputs and initial history

The buffer is allocated as:

| reset | input | history | update | hidden_state |

In this way, the concatination is automatically done since (reset, input) and (input, history) are physically concatinated in memory.

The ordering of the weight matrix should be adjusted accordingly.

NMSIS DSP Software Library Functions Used:

- riscv_fully_connected_mat_q7_vec_q15_opt()
- riscv_nn_activations_direct_q15()
- riscv_mult_q15()
- riscv_offset_q15()
- riscv_sub_q15()
- riscv_copy_q15()

4.4 Changelog

4.4.1 V1.0.2

This is release `1.0.2` version of NMSIS-NN library.

- Sync up to CMSIS NN library 3.0.0
- Initial support for RISC-V vector extension support

4.4.2 V1.0.1

This is release `V1.0.1` version of NMSIS-DSP library.

- Both Nuclei RISC-V 32 and 64 bit cores are supported now.
- Libraries are optimized for RISC-V 32 and 64 bit DSP instructions.
- The DSP examples are now using Nuclei SDK as running environment.

4.4.3 V1.0.0

This is the first version of NMSIS-NN library.

We adapt the CMSIS-NN v1.0.0 library to use RISCV DSP instructions, all the API names now are renamed from `arm_xxx` to `riscv_xxx`.

CHAPTER

FIVE

CHANGELOG

5.1 V1.0.2-RC1

This is the release candidate version `V1.0.2-RC1` release of Nuclei MCU Software Interface Standard(NMSIS).

The following changes has been made since `V1.0.1`.

- **Device Templates**
  - DOWNLOAD_MODE_xxx macros are removed from riscv_encoding.h, it is now defined as enum in `<Device.h>`, and can be customized by soc vendor.
  - startup code now don't rely on DOWNLOAD_MODE macro, instead it now rely on a new macro called VECTOR_TABLE_REMAPPED, when VECTOR_TABLE_REMAPPED is defined, it means the vector table's lma != vma, such as vector table need to be copied from flash to ilm when boot up
  - Add **BIT**, **BITS**, **REG**, **ADDR** related macros in `<Device.h>`
- **NMSIS-Core**
  - Nuclei Cache CCM operation APIs are now introduced in core_feature_cache.h
- **NMSIS-DSP/NN**
  - Merged the official CMSIS 5.8.0 release, CMSIS-DSP 1.9.0, CMSIS-NN 3.0.0
  - RISC-V Vector extension and P-extension support for DSP/NN libraries are added

5.2 V1.0.1

This is the offical `V1.0.1` release of Nuclei MCU Software Interface Standard(NMSIS).

The following changes has been maded since `V1.0.1-RC1`.

- **Device Templates**
  - I/D Cache enable assemble code in startup_<Device>.S are removed now
  - Cache control updates in System_<Device>.c
    - I-Cache will be enabled if __ICACHE_PRESENT = 1 defined in <Device.h>
    - D-Cache will be enabled if __DCACHE_PRESENT = 1 defined in <Device.h>

5.3 V1.0.1-RC1

This is release candidate version V1.0.1-RC1 of NMSIS.

- **NMSIS-Core**
  - Add RISC-V DSP 64bit intrinsic functions in core_feature_dsp.h
  - Add more CSR definitions in riscv_encoding.h
  - Update arm compatiable functions for RISC-V dsp instruction cases in core_compatiable.h
- **NMSIS-DSP**
  - Optimize RISC-V 32bit DSP library implementation
  - Add support for Nuclei RISC-V 64bit DSP SIMD instruction for DSP library
  - Add test cases used for DSP library testing, mainly for internal usage
  - Change the examples and tests to use Nuclei SDK as running environment
- **NMSIS-NN**
  - Add support for Nuclei RISC-V 64bit DSP SIMD instruction for NN library
  - Change the examples and tests to use Nuclei SDK as running environment
- **Device Templates**
  - Add DDR DOWNLOAD_MODE in device templates
  - Modifications to startup_<Device>.S files
    * _premain_init is added to replace _init
    * _postmain_fini is added to replace _fini
  - If you have implemented your init or de-init functions through _init or _fini, please use _premain_init and _postmain_fini functions defined system_<Device>.c now

5.4 V1.0.0-beta1

Main changes in release **V1.0.0-beta1**.

- **NMSIS-Core**
  - Fix SysTick_Reload implementation
  - Update ECLIC_Register_IRQ implementation to allow handler == NULL
  - Fix MTH offset from 0x8 to 0xB, this will affect function of ECLIC_GetMth and ECLIC_SetMth
  - Fix wrong macro check in cache function
  - Add missing SOC_INT_MAX enum definition in Device template
  - In System_<Device>.c, ECLIC NLBits set to __ECLIC_INTCTLBITS, which means all the bits are for level, no bits for priority

5.5 V1.0.0-beta

Main changes in release **V1.0.0-beta**.

- **NMSIS-Core**
  - Fix error typedef of `CSR_MCAUSE_Type`
  - Change `CSR_MCACHE_CTL_DE` to future value `0x00010000`
  - Fix names in CSR naming, `CSR_SCRATCHCSW` -> `CSR_MSCRATCHCSW`, and `CSR_SCRATCHCSWL` -> `CSR_MSCRATCHCSWL`
  - Add macros in `riscv_encoding.h`: `MSTATUS_FS_INITIAL`, `MSTATUS_FS_CLEAN`, `MSTATUS_FS_DIRTY`
- **Documentation**
  - Fix an typo in *core_template_intexc.rst*
  - Add cross references of Nuclei ISA Spec
  - Update appendix
  - Refines tables and figures

5.6 V1.0.0-alpha.1

API changes has been maded to system timer.

- Start from Nuclei N core version 1.4, MSTOP register is renamed to MTIMECTL to provide more features
- Changes made to NMSIS/Core/core_feature_timer.h
  - MSTOP register name changed to MTIMECTL due to core spec changes
  - SysTimer_SetMstopValue renamed to SysTimer_SetControlValue
  - SysTimer_GetMstopValue renamed to SysTimer_GetControlValue
  - Add SysTimer_Start and SysTimer_Stop to start or stop system timer counter
  - SysTick_Reload function is introduced to reload system timer
  - Macro names started with SysTimer_xxx are changed, please check in the code.
- Removed unused lines of code in DSP and NN library source code which has unused macros which will not work for RISCV cores.
- Fix some documentation issues, mainly typos and invalid cross references.

5.7 V1.0.0-alpha

This is the `V1.0.0-alpha` release of Nuclei MCU Software Interface Standard(NMSIS).

In this release, we have release three main compoments:

- **NMSIS-Core**: Standardized API for the Nuclei processor core and peripherals.
- **NMSIS-DSP**: DSP library collection optimized for the Nuclei Processors which has RISC-V SIMD instruction set.

- **NMSIS-NN**: Efficient neural network library developed to maximize the performance and minimize the memory footprint Nuclei Processors which has RISC-V SIMD instruction set.

We also released totally new Nuclei-SDK[15] which is an SDK implementation based on the **NMSIS-Core** for Nuclei N/NX evaluation cores running on HummingBird Evaluation Kit.

[15] https://github.com/Nuclei-Software/nuclei-sdk

CHAPTER SIX

GLOSSARY

API (Application Program Interface) A defined set of routines and protocols for building application software.

DSP (Digital Signal Processing) is the use of digital processing, such as by computers or more specialized digital signal processors, to perform a wide variety of signal processing operations.

ISR (Interrupt Service Routine) Also known as an interrupt handler, an ISR is a callback function whose execution is triggered by a hardware interrupt (or software interrupt instructions) and is used to handle high-priority conditions that require interrupting the current code executing on the processor.

NN (Neural Network) is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes.

XIP (eXecute In Place) a method of executing programs directly from long term storage rather than copying it into RAM, saving writable memory for dynamic data and not the static program code.

CHAPTER

SEVEN

APPENDIX

- **Nuclei Tools and Documents**: https://nucleisys.com/download.php
- **Nuclei riscv-openocd Repo**: https://github.com/riscv-mcu/riscv-openocd
- **Nuclei riscv-binutils-gdb**: https://github.com/riscv-mcu/riscv-binutils-gdb
- **Nuclei riscv-gnu-toolchain**: https://github.com/riscv-mcu/riscv-gnu-toolchain
- **Nuclei riscv-newlib**: https://github.com/riscv-mcu/riscv-newlib
- **Nuclei riscv-gcc**: https://github.com/riscv-mcu/riscv-gcc
- **Nuclei SDK**: https://github.com/Nuclei-Software/nuclei-sdk
- **NMSIS**: https://doc.nucleisys.com/nmsis/
- **Nuclei Bumblebee Core Document**: https://github.com/nucleisys/Bumblebee_Core_Doc
- **Nuclei RISC-V IP Products**: https://www.nucleisys.com/product.php
- **RISC-V MCU Community Website**: https://www.riscv-mcu.com/
- **Nuclei Spec**: https://doc.nucleisys.com/nuclei_spec

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

D

E

F

I

M

N

P

R

S

T

U

V

W

X