Black Iron CPU Software Development Guide

Xuantie

November 05, 2024

Hangzhou C-SKY MicroSystems Co., LTD

Address: Room 201, 2F, Building 5, No. 699 Wangshang Road, Hangzhou, Zhejiang, China

Website: www.xrvm.cn

Revision History

Version	Description	date
1.1	Fixed some vdsp intrinsic interface description errors. Added	2020.02.11
1.2	RISC-V series CPU programming chapter. Fixed some writing errors.	2020.05.07
1.3	Add vector floating point instruction description	2020.05.11
1.4	Modify the naming rules of T-Head extension instructions and add c906	2020.06.20
1.5	Add libcc-rt description section1.6	2020.08.05
	Add abi type lp64dv Add e907 related	2020.08.09
1.7	CPU Add 900 series P extension related CPU Revised some	2020.12.26
1.8	typos in the document Added RISC- V architecture embedded assembly constraint table Moved "Embedded Assembly" from Chapter 3 to Chapter 7, and	2021.04.22
	added notes1.9 Added	2021.05.26
2.0	c920 Added CPU r910 and r920 to support Xuantie 900 series V2.0 version tool Added Xuantie 800	2021.07.20
	series dsp section2.1 Added CPU c908 and c908v	2021.11.13
2.2	The -mcpu option is added to the XuanTie 900 series. Newlib is added to support reentrancy. Supports the XuanTie 900 series V2.6 version tool	2020.05.26
3.0	Support for the XuanTie LLVM compiler	2023.05.24
3.1	Added CPU c910v2, c920v2 and c908i. Added a section on nested interrupt function attributes. Added a section on bf16 type description. Added a section on dynamic linker name description. Fixed the problem of missing zfh in march of c906fd, c910 and r910. Fixed the problem of missing xtheadvdot in march	2023.10.08
3.2	of c908v. Added CPU c920v2.c908v. Added c907 series CPU. Updated marches corresponding to c910v2 series and c908 series. Deleted ABI lp64dv and lp64v. Added a section on KO file size optimization.	2024.03.18
3.3	Fix the problem of missing zcb in C907 arch	2024.07.13
3.4	Added CPU c910v3, c920v3, c910v3-cp, c920v3-cp and r908 series Added general coprocessor extension C intrinsic interface Added RISC-V Vector usage instructions	2024.11.1

Corresponding tool version

	Version Number
Category: Hexagon 800 Series	V3.10
GNU Tools Hexagon 900 Series	V3.0.0
GNU Tools Hexagon LLVM Tools	V2.0.0

Software Development Guide

Chapter 1 Toolchain Introduction

Traditional toolchain definitions usually include compilers, assemblers, linkers, etc. All these components work together to implement the translation process from C/C++ source code to executable files, as shown in Figure 1.1. The compiler processes the input source file, including lexical analysis, syntax analysis, semantic checking, and assembly code generation. When the input source file does not conform to the C/C++ language standard or the GNU extended syntax specification, the compiler will generate corresponding diagnostic information to prompt the program developer that the corresponding source code file has syntax or semantic errors. Due to the design principle of separation of modern computer software libraries, a linker is usually required to link multiple target files and several static libraries and dynamic shared libraries into a complete executable file. These three parts together form a tool set for program developers to use.

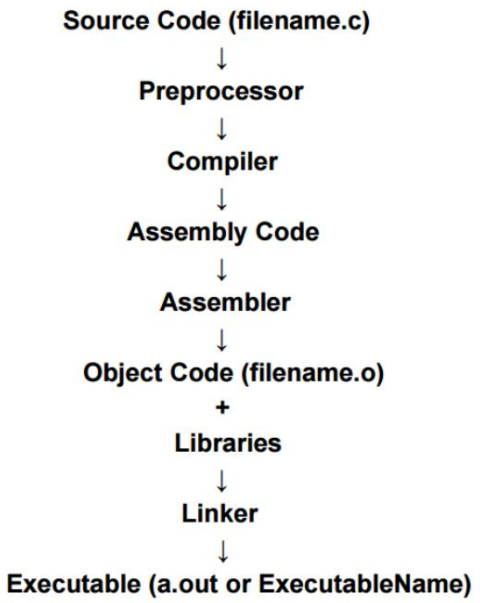

Figure 1.1: The compiler's processing flow for the input source file

Currently, Xuantie has two tool chains, one based on the GNU tool chain and the other based on the LLVM compilation framework.

The contents described or given examples in the documentation are applicable to both toolchains. For details on the specific parts, see the following sections:

- Instructions for using the *GNU* toolchain
- *LLVM* toolchain usage instructions

Chapter 2 GNU Toolchain Instructions

Currently Xuantie has two sets of GNU tool chains, one for CSKY series CPUs and the other for RISC-V series CPUs.

This chapter contains the following parts:

2.1 Toolchain component versions, names, and language support

The current release version of the CSKY series toolchain is based on GCC 6.3, and the RISC-V series toolchain is based on GCC 10.2.

The prefixes are shown in Table 2.1 , the names of the main components of the tool chain are shown in Table 2.2 , their support for C language versions is shown in Table 2.3 , and their support for C++ language versions is shown in Table 2.

The support is shown in Table 2.4 , and the language standard can be selected by option -std=[language standard name].

Table 2.1: Toolchain names

CPU Series	Target System	Platform	C Library	Toolchain Name Prefix	Old Version Name Prefix
XuanTie 600 series	elf	minilibc	csky-elf-		none
	linux		glibc	csky-linux-gnu-	csky-linux-
			uclibc	csky-linux-uclibc-	csky-linux-
Black Iron 800 Series	elf		minilibc	csky-elfabiv2-	csky-abiv2-elf-none
			newlib	csky-noneabiv2-	
	linux		glibc	csky-linux-gnuabiv2-	csky-abiv2-linux-csky-
			uclibc	linux-uclibcabiv2-	abiv2-linux-
Black Iron 900 Series	elf		newlib	riscv64-unknown-elf-	none
	linux		glibc	riscv64-unknown-linux -	None

Remark:

1. The new version of the tool is compatible with the old toolchain name. This document uses the new toolchain name as an example.
2. Unless otherwise specified, the examples in this manual are applicable to the full range of Hematite tools and use one of the tools as an example.

Table 2.2: Component names

Components	name
C compiler [prefix] -gcc	
C++ compiler [prefix] -g++	
Assembler [prefix] -as	
Linker [prefix] -ld	

Table 2.3: C language standard support

C language standard	CSKY	RISC-V
c89/c90	yes	yes
gnu89/gnu90	yes	yes
c99	yes	yes
gnu99	yes	yes
c11	yes	yes
gnu11	default	yes
gnu17	no	default

Table 2.4: C++ language standard support

C++ language standard	CSKY	RISC-V
c++98	yes	yes
gnu++98	yes	yes
c++03	yes	yes
gnu++03	yes	yes
C++11	yes	yes
gnu++11	yes	yes
c++14	yes	yes
gnu++14	default	default
c++17	no	yes
gnu++17	no	yes

2.2 General Option Description

In order to avoid redundant description of all command line options of the toolchain, this section will introduce some of the command line options that are most frequently used by ordinary application developers.

For a detailed description of the command line options, see the GCC command line online documentation.

2.2.1 Compiler Commands

1. Options for controlling output

-E

Only C/C++ preprocessing is performed, and subsequent syntax analysis, code generation, etc. are not performed.

-fsyntax-only

Usually used to control the compiler to only perform semantic checking and not perform subsequent processes. This option is usually used to test whether the input source file conforms to the C/C++ language standard without generating any files.

-c

Only let the compiler generate the target file (usually a file with the suffix .o) without performing the subsequent linking process.

-S

Instructs the compiler to generate assembly code only.

-v

Check the compiler version and print out the compiler compilation commands, assembler commands and linker commands. It will also print out the search directory for header files.

-###

This option must be placed after the gcc command. Its function is similar to the -v option. The difference from -v is that this option will not perform the actual compilation, assembly, and linking process, but only print the execution command. Developers can use this option to obtain the compiler's compilation command to facilitate debugging of the source program. Note that usually debugging gcc does not enter the compiler process. Gcc is a driver control program that generates different commands to call the compiler (cc1 for C language programs and cc1plus for C++ programs), assembler (as) and linker (ld) to achieve the desired purpose.

--version

Displays the version number of GCC used.

2. Language Standards

-std

This option is used to control the language standards that the compiler can support. The default language standards used by the current Xuantie compiler are gnu99 (for C language) and gnu++14 (for C++ language). Of course, developers can use this option to force the compiler to use a specific standard.

3. Debug format support

-g

Controls the compiler to insert debugging information (default is dwarf format) supported by the local system into the target code or assembly code for use by the gdb debugger. This option is usually used in the program development stage to assist developers in debugging and testing. It is recommended to turn off this option when releasing the version.

-ggdb

Controls the compiler to generate debugging information in the format suitable for the gdb debugger.

-gdwarf

Controls the compiler to generate debugging information conforming to the dwarf debugging format.

-gcoff

Controls the compiler's generation of coff-format debugging information.

-gxcoff

Controls the compiler to generate debugging information in the GNU extended coff format.

4. Control the format of diagnostic information

When the compiler parses the input source code file, if the input source file contains syntax or semantic errors, the compiler will generate several diagnostic messages to prompt the developer the source file location and diagnostic category (fatal, error, warning) where the diagnostic information occurs, and give detailed diagnostic description information.

-fmessage-length=n

The option controls the width of the output diagnostic description text to n characters.

-fdiagnostics-show-location=once

Controls whether the compiler should output source code locations only once (when subsequent diagnostic locations are in the same file, same column).

-fdiagnostics-show-location=every-line

Instructs the compiler to output the complete location information (filename: line number: column number) for each diagnostic location.

-fno-diagnostics-color instructs

the compiler not to colorize diagnostic information.

-fno-diagnostics-show-option

Instructs the compiler not to generate diagnostic information.

-fno-diagnostics-show-caret

Instructs the compiler not to generate ^ symbols to indicate where they are generated.

5. Optimization options

This type of option is usually used to control the compiler's optimization strategy, telling the compiler whether it should perform certain optimizations to increase the running speed of the program, such as loop unrolling, or whether it should avoid certain optimization strategies to reduce the size of the target file, so that the target program can run more efficiently in embedded devices with small memory capacity.

-O0

The default optimization level for GCC/G++ does not perform any optimizations to reduce compilation time, and usually generates debugging information.

-O -O1

These two options have the same meaning and are usually used to control the compiler to perform some high-yield optimizations, thereby reducing code size and execution time.

-Og

Optimize the debugging experience and provide reasonable optimization levels while maintaining fast compilation and a good debugging experience.

-O2

This option will enable optimization options that require more compilation time but can significantly increase program execution speed. Note that this option usually increases the size of the object code.

-O3

This option turns on all options in O2, and also turns on several other optimizations to improve the performance of the target code.

-Os

This optimization option is usually used to tell the compiler to reduce the size of the target code as much as possible while maintaining performance. It will remove some optimization strategies that will increase the size of the target code from all options enabled by O2.

-Ofast

This option is usually used to tell the compiler to generate the fastest running object code possible, regardless of the object code size. It will turn on all the options in O3.

2.2.2 Assembler Commands

For most developers, there are few opportunities to use the assembler directly. Therefore, this section will introduce the more commonly used options.

For more information, developers can consult the development manual of CSKY or RISC-V assembler.

-g --gen-debug

Tells the assembler to generate debuggable information for the object code.

-o

Specify the output target file name, the default is a.out

2.2.2.1 CSKY Assembler Specific Options

-march=architecture

Generate target code for the CPU of the specified architecture, such as: *-march=ck803* will tell the compiler to generate instructions supported by the ck803 series CPU, but will not generate instructions for the ck810 series CPU.

-mcpu=CPU model

Specifies to generate optimized code for the selected CPU *features, such as: -mcpu=ck803er1* will enable dspv2 instructions.

-m{no-}jump

If the target address of jbf, jbt, and jbr instructions exceeds the instruction offset range, the instructions will be converted into jmpi instructions. This is disabled by default.

2.2.2.2 RISC-V assembler specific options

-march=architecture

Generate target code for the CPU of the specified architecture, such as: *-march=rv32imac* will tell the compiler to generate target code for the 32-bit RISC-V series CPU that includes the basic integer instruction set ('I' instruction set), atomic operation instruction set ('A' instruction set), and compressed instruction set ('C' instruction set).

2.2.3 Examples

This section takes CSKY as an example to describe the usage and function of each option mentioned above. The methods described are also applicable to RISC-V. This section uses two files, bar.h and bar.c. The bar.h header file declares a function with the signature *int sum(int num)* , and implements and calls the function in bar.c. The contents of the bar.h file are as follows

```
#ifndef BAR_H
#define BAR_H

/**
  * Perform the accumulation , Calculate the sum from 1 to len , If the input len is less than or equal to 0,
  operation * Return 0.
  */
int sum(int len);
#endif
```

The bar.c file code content is as follows

```
#include "bar.h"

int a = 30;
```

(Continued on next page)

(Continued from previous page)

```
int b;

int main()
{
    b = sum(a);
    return 0;
}

int sum(int len)
{
    int res = 0;
    for (int i = 1; i <= len; i++)
        res += i;
    return res;
}
```

The usual compilation command is as follows

```
csky-elfabiv2-gcc bar.c -o bar
```

After the above command is executed, an executable file named bar will be generated in the directory where the source file is located. The file can be simulated and executed using the qemu-system-cskyv2 simulator (the simulator is used to simulate the execution of elf format files), and the corresponding output information will be displayed on the standard output device (usually a terminal or command line window).

In some development scenarios, such as debugging a source code file, the file usually includes many include header files, and the developer does not know The corresponding header file search directory. In this case, the developer needs to know how to generate preprocessing files to avoid looking for unknown header file search directories.

To achieve the above purpose, developers need to manually pass the -E option to the compiler to tell the gcc compiler that it only needs to perform preprocessing. When the gcc compiler generates a file with the same name as the source file but with the suffix .i (for C language files, if the source file is C++, the suffix is .ii), the command is as follows

```
csky-elfabiv2-gcc bar.c -E
```

The output preprocessed file is named bar.i and its content is as follows. Obviously, the compiler will extract the corresponding function declaration in the header file to the location where the function is used. This implements an important principle in the C/C++ language - variables or functions must be declared before they are used.

```
int sum(int len);

int a = 30;
int b;

int main()
{
    b = sum(a);
    return 0;
}
```

(Continued on next page)

(Continued from previous page)

```
int sum(int len)
{
    if (len <= 0) return 0;
    int res = 0;
    for (int i = 1; i <= len; i++)
        res += i;
    return res;
}
```

In other scenarios, such as checking whether the generated assembly code is correct, in addition to using the csky-elfabiv2-objdump tool to disassemble the generated target code, another more direct way to deal with it is to pass the -S option to the compiler. At this time, the gcc compiler will only run to the assembly code generation stage, and will not continue to call the assembler and linker to generate executable code. The specific commands are as follows.

```
csky-elfabiv2-gcc bar.c -S
```

The contents of the generated assembly file bar.s are as follows

```
# To save space , Omit the main function code and remove irrelevant debugging code
sum:
        subi sp, sp, 4
        st.w l4, (sp, 0)
        mov l4, sp
        subi sp, sp, 12
        subi a3, l4, 12
        st.w a0, (a3, 0)
        subi a3, l4, 12
        ld.w a3, (a3, 0)
        jbhz a3, .L4
        movi a3, 0
        jbr .L5
.L4:
        subi a3, l4, 4
        movi a2, 0
        st.w a2, (a3, 0)
        subi a3, l4, 8
        movi a2, 0
        st.w a2, (a3, 0)
        jbr .L6
.L7:
        subi a3, l4, 4
        subi a1, l4, 4
        subi a2, l4, 8
        ld.w a1, (a1, 0)
        ld.w a2, (a2, 0)
```

(Continued on next page)

(Continued from previous page)

```
        addu a2, a2, a1
        st.w a2, (a3, 0)
        subi a3, l4, 8
        subi a2, l4, 8
        ld.w a2, (a2, 0)
        addi a2, a2, 1
        st.w a2, (a3, 0)
.L6:
        subi a2, l4, 8
        subi a3, l4, 12
        ld.w a2, (a2, 0)
        ld.w a3, (a3, 0)
        cmplt a2, a3
        jbt .L7
        subi a3, l4, 4
        ld.w a3, (a3, 0)
.L5:
        mov a0, a3
        mov sp, l4
        ld.w l4, (sp, 0)
        addi sp, sp, 4
        rts
```

In order to improve program performance, developers usually turn on the -On(n >= 1) option to achieve better program performance. Taking the above assembly as an example, when

When -O2 is turned on, you can observe the following assembly code and find that the overall size of the program is significantly reduced, and the instructions are more concise, as shown in the figure:

```
# To save space , Omit the main function code and remove irrelevant debugging code
sum:
        jblsz a0, .L10
        movi a3, 0
        mov a2, a3
.L9:
        addu a2, a2, a3
        addi a3, a3, 1
        cmpne a0, a3
        jbt .L9
        mov a0, a2
        rts

.L10:
        movi a2, 0
        mov a0, a2
        rts
```

Similarly, when -Os is turned on, you will find that the number of instructions in the generated assembly file is greatly reduced compared to when optimization is not turned on.

Similarly, other options can be tested using this method to observe the compiler's output.

2.3 Passing options to the assembler and linker

By default, GCC includes the process of pre-compilation, compilation, assembly, and linking when it is executed. It will automatically call the preprocessor, assembler, and linker. In some cases, Developers need to pass options to each component. The transfer method is described in Table 3.4 :

Table 2.5: Delivery options for each component

GCC option effects	
-Wp,[parameter]	pass parameters to the preprocessor
-Wa,[parameter]	pass parameters to the assembler
-Wl,[parameter]	pass parameters to the linker
-L [path]	Add path for linker to search for libraries

2.4 Toolchain Errors and Warnings

In order to better improve the development efficiency of program developers and improve the stability of program runtime, most tool chains provide a good program checking mechanism to generate diagnostic information at the location of illegal code snippets with potential security or performance risks, and print out diagnostic information that is easy for program developers to read and understand, including error information (error), warning information (warning) and some modification suggestions (note).

This chapter will focus on the classification and format of the diagnostic information output by the CSKY series and RISC-V series tool chains, and use several examples to illustrate how to The diagnostic information is used to repair the code snippet (this chapter uses the CSKY series compiler as an example, but the rules are generally applicable to the RISC-V series compiler).

This chapter contains the following parts:

- The format of compiler error and warning messages
- Compiler diagnostic information options
- Use *pragma* preprocessing commands to control error and warning messages
- Other tools options for controlling error and warning messages

2.4.1 Format of Compiler Error and Warning Messages

Compiler error and warning messages are the most commonly used diagnostic information categories that most developers come into contact with on a daily basis. This type of information is usually generated by the compiler's front end (lexical analysis, syntax analysis, and semantic checking) to inform developers that the compiled source program contains numbers, identifiers, language structures, and illegal operations on certain types of variables that do not conform to the C language standard (or GNU extended syntax).

2.4.1.1 Error message format

Take the following code snippet (file name is bar.c) as an example:

```
int a;
int x = a;
```

Use the following shell *command to compile. The -fsyntax-only* option is used to tell the compiler to only perform front-end actions, without performing back-end optimization and code generation.

```
csky-elfabiv2-gcc bar.c -fsyntax-only
```

The error message generated by the above command is:

```
bar.c:2:9: error: initializer element is not constant

int b = a;
        ^
```

The error message format will be displayed as follows:

```
bar.c:2:9: error: initializer element is not constant
   |       |  |       |                                    |
   |       |  |       |                                    |_____________________Diagnostic Notes
   |       |  |       |__________________________________________Indicates the diagnostics of the error category
   |       |  |__________________________________________The column number where the diagnostic information is located
   |       |_____________________________________________The line number where the diagnostic information is located
   |_____________________________________________Source file name where the diagnostic occurred
```

2.4.1.2 Warning diagnostic format

```
/* Test function. */

foo(int *ptr)

{

    ptr = ptr + 2;

    return *ptr;

}
```

Use the following shell *command to compile. The -fsyntax-only* option is used to tell the compiler to only perform front-end actions, without performing back-end optimization and code generation.

```
csky-elfabiv2-gcc bar.c -fsyntax-only
```

The error message generated by the above command is:

```
foo.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]

foo(int *ptr)

^~~
```

Warning diagnostic information format will be displayed as follows:

```
foo.c:1:1: warning: return type defaults to 'int' [-Wimplicit-int]
   |       |  |       |                                    |
   |       |  |       |                                    |_____________________Diagnostic Notes
   |       |  |       |_____________________________________________ indicates that the diagnosis is a warning message
```

(Continued on next page)

(Continued from previous page)

```
|       | |_____The warning occurs in the first column of the source file|
|       _____The warning occurs in the first line of the source file|
_____The warning occurs in the source file foo.c
```

2.4.1.3 Other diagnostic information formats

In addition to reporting errors and warnings, compilers usually provide some error repair suggestions in some cases, telling program developers how to fix the error. The following code snippet is used here to illustrate this situation. In the following code, the malloc function is called in the main function to allocate 10 bytes of space on the heap, and then assigned a value of 1. Note that there is no include<stdlib.h> in the code.

```
int main()
{
    int* ptr = (int*)malloc(10);
    *ptr = 1;
    return 0;
}
```

Compiling with csky-elfabiv2-gcc yields the following diagnostic information

```
implicit.c: In function 'main':
implicit.c:3:20: warning: implicit declaration of function 'malloc' [-Wimplicit-
 ÿfunction-declaration]
    int* ptr = (int*)malloc(10);
                     ^~~~~~
implicit.c:3:20: warning: incompatible implicit declaration of built-in function
 ÿ'malloc'
implicit.c:3:20: note: include '<stdlib.h>' or provide a declaration of 'malloc'
```

From the above diagnostic information, we can see that in addition to the warning information, there is also a note information, which usually suggests how to use include '<stdlib.h>' to fix the problem.

2.4.2 Compiler Diagnostic Information Options

Usually when developing embedded systems, in order to minimize the risk of potential vulnerabilities in the program, it is necessary to ensure that there are no warnings or errors when compiling the source program. However, a considerable number of developers will ignore the warning messages generated by the compiler, and more importantly, when the compiler generates warning messages, the compilation process will continue as usual. This will cover up potential risks in the future. To avoid this situation, the compiler provides a series of options to meet this type of demand, the most important of which is -Werror, which will convert all warning diagnostic information into error information display, thereby avoiding the hiding of the error.

At the same time, the compiler also provides some options to control the compiler to generate warning information, prompting program developers to pay attention to the areas, in this case it is more appropriate to use the -Wall option, this option will turn on all warning messages, rather than the default off (whether the default state outputs warning information depends on the compiler version). However, sometimes developers only need to convert specific warning messages into error messages, and the compiler provides a -Wxxx (xxx is the name of the warning option, such as the implicit-function-declaration mentioned in the previous section) option for each warning option to turn on specific warning messages.

Some legacy code will display some warning messages as error messages in order to be compatible with different GCC versions. To avoid newer GCC versions from compiling If it fails, you can use -Wnoxxx (xxx is the name of the warning option) to turn off this type of warning message, such as: -Wnoimplicit-function-declaration.

2.4.3 Using pragma preprocessing commands to control errors and warnings

In addition to controlling the compiler's diagnostic information output through command line options, the compiler also supports more flexible control of the display of diagnostic information in source files through #pragma preprocessing options. In addition, you can also use #pragma to output custom error or warning information, or selectively turn off all or specified diagnostic options.

2.4.3.1 #pragma GCC error

This option is used by program developers to set custom error diagnostic information in source files, such as the following code snippet (the file name is pragma-error.c):

```
#pragma GCC error "This is an error issured by pragma"
```

Compile using csky-elfabiv2-gcc command, the error message displayed is:

```
pragam-error.c:1:20: Error: This is an error issured by pragma
#pragma GCC error "This is an error issured by pragma"
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

From the above results, we can see that #pragma GCC error allows the compiler to flexibly output specific, customized error messages.

2.4.3.2 #pragma GCC warning

Similar to #pragma GCC error, the warning option is used to tell the compiler to generate specific, customized warning messages, and the display format is basically the same.

The following code snippet is used as an example to illustrate this situation.

```
#pragma GCC warning "This is a warning issued by pragma"
```

Compile using csky-elfabiv2-gcc command, the warning message displayed is:

```
pragam-warning.c:1:20: Error: This is an error issured by pragma #pragma GCC warning "This is a
warning issured by pragma"
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

2.4.3.3 #pragma message

This option is only used to output a compiler diagnostic message, not a warning or error message.

```
#pragma message "message produced by pragma message directive"
```

Compile using csky-elfabiv2-gcc command, the warning message displayed is:

```
pragam-message.c:1:9: PS: #pragma message: message produced by pragma message #pragma message "message produced by
pragma message"
        ^~~~~~
```

2.4.3.4 #pragma GCC diagnostics

The three #pragma subclasses mentioned above are usually used to control the compiler to output custom diagnostic information, but this option is used to tell the compiler to ignore, display, or display a warning option as an error when encountering this command when compiling a source file, as shown in the following example.

```
#pragma GCC diagnostic ignored "-Wimplicit-int"
bar() // "Return type defaults to 'int'" should appear here
{
    #pragma GCC diagnostic warning "-Wimplicit-function-declaration" // Turn on -Wimplicit-
 ÿfunction-declaration options
    int *ptr = (int*)malloc(sizeof(int)); // Display warning
    *ptr = 1;
    #pragma GCC diagnostic error "-Wimplicit-function-declaration" ÿ function-declaration is displayed as an            // Set -Wimplicit-
 error
    memset(ptr, 0, sizeof(int)); // Display warning as an error
    return 0;
}
```

The following diagnostic information appears when using csky-elfabiv2-gcc to compile. By observing the following information, we can clearly see the role played by the above preprocessing instructions.

```
implicit.c: In function 'bar': implicit.c:5:20:
warning: implicit declaration of function 'malloc' [-Wimplicit-function-declaration]
      int *ptr = (int*)malloc(sizeof(int)); // Display warning
                                  ^~~~~~
implicit.c:5:20: warning: implicit declaration incompatible with built-in function 'malloc'
implicit.c:5:20: note: include '<stdlib.h>' or provide a declaration of 'malloc' implicit.c:8:3: error: implicit declaration of function 'memset' [-
Werror=implicit-function-declaration] memset(ptr, 0, sizeof(int)); // show warning as error

      ^~~~~~
implicit.c:8:3: warning: implicit declaration incompatible with built-in function 'memset'
implicit.c:8:3: note: include '<string.h>' or provide a declaration of 'memset'
```

2.4.4 Other Tools Options for Controlling Error and Warning Messages

2.4.4.1 Assembler options for controlling diagnostic information

In some cases, when using the assembler alone, developers also need to control the output behavior of outputting diagnostic information, such as not outputting Warning messages or display warning messages as errors.

1. -W

Hide warning message

2. –warn

Do not hide warning messages

3. –fatal-warnings

Show warnings as errors

4. -Z

Generate target files even if there are errors

Chapter 3 LLVM Tool Chain Usage Instructions

The XuanTie LLVM compiler is a high-performance, high-reliability toolchain developed based on the open source LLVM 15. It performs deep co-optimization of software and hardware based on the XuanTie processor, while optimizing performance and code density for a variety of common fields. Currently, the XuanTie LLVM compiler focuses on the support of C and C++ programming languages, supports the full range of CSKY and RISCV processors, and supports both Linux and Windows platforms.

General information about LLVM can be found at https://llvm.org/docs/UserGuides.html

For general instructions on using CLANG, refer to https://releases.llvm.org/15.0.0/tools/clang/docs/UsersManual.html

This chapter will introduce some special or common instructions, mainly including the following parts:

- Toolchain component names and language support
- General Option Description
- Pass options to the assembler and linker
- compatibility

3.1 Toolchain component names and language support

The names of the main components of the toolchain are shown in Table 3.1 , their support for C language versions is shown in Table 3.2 , and their support for C++ language versions is shown in Table 3.3 . The language standard can be selected by option -std=[language standard name].

Table 3.1: Component names

Component Name	
C compiler clang	
C++ compiler clang++	assembler
llvm-mc linker [prefix]	-ld

Note: The linker currently used by the LLVM toolchain is the linker in the GNU suite. For specific names, see Toolchain component versions, names, and language support.

Table 3.2: C language standard support

C language standard	support
c89/c90	yes
gnu89/gnu90	yes
c99	yes
gnu99	yes
c11	yes
gnu11	yes
gnu17	default

Table 3.3: C++ language standard support

C++ language standard	support
c++98	yes
gnu++98	yes
c++03	yes
gnu++03	yes
C++11	yes
gnu++11	yes
c++14	yes
gnu++14	default
c++17	yes
gnu++17	yes

3.2 General Option Description

This section will introduce some of the most frequently used command line options for ordinary application developers. For detailed command line options, please refer to CLANG Command Line Online documentation.

1. Optimization options

This type of option is usually used to control the compiler's optimization strategy, telling the compiler whether it should perform certain optimizations to increase the program's running speed or Reduce the size of the object code.

-O0

The default optimization level does not perform any optimization to reduce compilation time and does not cause inaccurate debugging information, thus facilitating debugging.

-O -O1

These two options have the same meaning and are usually used to control the compiler to perform some high-yield optimizations while reducing code size and Execution time purpose.

-Og

Optimize the debugging experience and provide reasonable optimization levels while maintaining fast compilation and a good debugging experience.

-O2

This option will enable optimization options that require more compilation time but can significantly increase program execution speed. Note that this option usually increases the size of the object code.

-O3

This option turns on all options in O2, and also turns on several other optimizations to improve the performance of the target code.

-Os

This optimization option is usually used to tell the compiler to reduce the size of the target code as much as possible while maintaining performance. It will remove some optimization strategies that will increase the size of the target code from all options enabled by O2.

-Oz

On the basis of Os, enable more options to optimize code size.

-Ofast

Enables additional aggressive optimizations on top of O3 that may violate some strict language standards.

2. Options for controlling output

-E

Only C/C++ preprocessing is performed, and subsequent syntax analysis, code generation, etc. are not performed.

-fsyntax-only

Usually used to control the compiler to only perform semantic checking and not perform subsequent processes. This option is usually used to test whether the input source file conforms to the C/C++ language standard without generating any files.

-c

Only let the compiler generate the target file (usually a file with the suffix .o) without performing the subsequent linking process.

-S

Instructs the compiler to generate assembly code only.

-###

This option has a similar function to the -v option. The difference is that this option does not perform the actual compilation, assembly, and linking process, but only prints the execution commands.

3. Information viewing options

-v

Check the compiler version and print out the compiler compilation commands, assembler commands and linker commands. It will also print out the search directory for header files.

--version

Displays the version number.

3.3 Passing options to the assembler and linker

By default, the compiler includes the process of pre-compilation, compilation, assembly, and linking. It will automatically call the preprocessor, assembler, and linker. In some cases, the developer needs to pass options to each component. The transfer method is described in Table 3.4 :

Table 3.4: Delivery options for each component

CLANG Options	effect
-Wp,/-Xpreprocessor [parameters] pass parameters to the preprocessor	
-Wa,/-Xassembler [parameters] pass parameters to the assembler	
-Wl,/-Xlinker [parameters] pass parameters to the linker	
-L [path] Add path for linker to search for libraries	

3.4 Compatibility

Currently, Clang is compatible with most of the features of GCC. For common compatibility and porting issues, please refer to the official documentation. https://clang.llvm.org/compatibility.html.

Chapter 4 XuanTie 800 Series CPU Programming

Xuantie 800 series CPU is a processor developed based on CSKY architecture. This chapter mainly introduces the C and C++ programming process, which involves CSKY Architecture-related special usage, such as cpu selection, instruction set selection, assembly programming, vdsp and dsp instruction intrinsic interfaces, and minilibc.

This chapter contains the following parts:

- How to add options for processors
- Instruction set introduction
- How to use hard floating point instructions
- Assembly language programming
- *vdsp*
- *dsp*
- *minilibc*

4.1 How to add options for processors

Currently, our CSKY supports many different CPU models. You can view all CPU models supported by the current tool through GCC or CLANG options.

Execute the following command on the command line:

```
csky-elfabiv2-gcc --target-help
clang -print-supported-cpus
```

The command execution result shows all CPU models supported by the compiler. They follow a set of basic naming rules, such as:

```
CK810 CEFHMTV R2
  |    |    |
  |    |    |
  |    |    |_____ R:Revision 2: The second version of the CPU
  |    |_____ Enhanced instruction set AZ
  |_____ CPU microarchitecture model
```

The meaning of the enhanced instruction set symbols is shown in Table 4.1 :

Table 4.1: Meaning of enhanced instruction set symbols

Enhanced instruction set symbol	full name	illustrate
C	Crypto	enhance encryption enhancement
E	EDSP	DSP Enhancements
F	FPU	floating point
H	Shield	Physical resistance to attack
M	Memory enhance	
T	TEE	Trusted Execution Environment
V	VDSP	Vector DSP

Generally, when we compile some projects, we specify the corresponding CPU for compilation through the compilation option -mcpu:

```
csky-elfabiv2-gcc -mcpu=ck810f helloworld.c
```

In some cases, it can also be used by adding some csky cpu feature switch options, such as using hardware floating point operation function:

```
csky-elfabiv2-gcc -mcpu=ck810f -mfloat-abi=hard helloworld.c
```

4.2 Introduction to instruction set

The previous chapter has introduced the architecture of CSKY. The basic instruction set corresponding to each architecture can be found in the CSKY CPU Instruction Implementation Reference Manual.

In addition to the basic instruction set, CSKY has two sets of dsp instruction sets, two sets of vdsp instruction sets, and three sets of floating-point instruction sets. Please refer to the following chapters for details.

4.2.1 DSP instruction set

There are two sets of DSP instruction sets, namely: DSP 1.0 and DSP 2.0. The corresponding relationship between instruction sets and CPUs is shown in Table 4.2 :

Table 4.2: Correspondence between DSP instruction set and CPU

CPU model	dsp instruction set version
ck803 series CPU with 'e' label	dsp 1.0
ck803r1 and above (including r1) with 'e' label CPU	dsp 2.0
ck804 series CPU	DSP 2.0
ck807 series CPU	DSP 1.0
ck810 series CPU	DSP 1.0

4.2.2 vdsp instruction set

There are two sets of vdsp instruction sets, namely: vdspv1 and vdspv2. The corresponding relationship between the instruction set and the CPU is shown in Table 4.3 :

Table 4.3: Correspondence between vdsp instruction set and cpu

CPU Model	vdsp instruction set version
ck805 series CPU vdspv2 ck810 series CPU with 'v' label vdspv1	
ck860 series CPU with 'v' label vdspv2	

4.2.3 Floating-point instruction set

There are three sets of floating-point instruction sets: fpuv1, fpuv2 and fpuv3. The correspondence between instruction sets and CPUs is shown in Table 4.4 :

Table 4.4: Correspondence between floating-point instruction set and CPU

CPU model	Floating point instruction set version
ck610 series CPUs with 'f' label fpuv1 ck803 series CPUs with 'f'	
label fpuv2 single precision floating point ck804 series CPUs with 'f' label fpuv2 single	
precision floating point ck805 series CPUs with 'f' label fpuv2 single precision floating	
point ck807 series CPUs with 'f' label fpuv2 ck810 series CPUs with 'f' label fpuv2	
ck860 series CPUs with 'f' label fpuv3	

Note: If you need the compiler to compile hardware floating-point instructions, in addition to adding the correct CPU model, you also need to add additional options. For details, see How to use hard floating-point instructions

---

4.2.4 CPU version and basic instruction set

Different CPU versions have different basic instruction sets, as shown in Table 4.5 :

Table 4.5: Relationship between CPU version and basic instruction set

CPU Model Basic Instruction Set Description	
ck803r1 Added mul.u32 mul.s32 mula.u32 mula.s32 mula.32.l mulall.s16.s to the ck803r2 Added bnezad to the ck803r1 basic instruction set ck803r3 Added divul divsl	
to the ck803r2 basic instruction set	

4.3 How to use hard floating-point instructions

Currently, some of our *CPUs* support hardware floating-point units. How can we make the gcc compiler generate code with hardware floating-point instructions?

Features to control the compiler to generate code containing hardware floating-point instructions:

```
csky-elfabiv2-gcc -mcpu=ck810f -mfloat-abi=hard helloworld.c
```

Currently we support multiple floating point ABI rules, controlled **by the -mfloat-abi** compilation option:

soft: Use software floating-point operations

hard: Use hardware floating-point operations

softfp: Same as hard option, but the parameters and return values do not use floating point registers.

Note: For floating point control options, the compiler is compatible with the old version. -msoft-float is equivalent to -mfloat-abi=soft, and -mhard-float is equivalent to -mfloat-abi=hard.

4.4 Assembly Language Programming

Some developers need to write assembly files by hand and compile them into object files, generally using the following basic commands:

```
csky-elfabiv2-gcc -c [input assembly file name] -o [output target file name]
```

This chapter contains the following parts:

- Assembly instruction format
- Preprocess assembly files
- Assembly directives
- Register alias

4.4.1 Assembly instruction format

The format of assembly instructions is divided into two parts: instruction name and operand name, separated by a space, as follows:

```
>                    Instruction name operand1, operand2, …
```

The types of operands are shown in Table 4.6 :

Table 4.6: Operand types in assembly instructions

Operand type	Writing format	Example
General register	general register name, see register alias for details	abs r1
v1 Floating point	fr0-fr31	fabss fr1
register v2 Floating	vr0-vr15	fabss vr0-vr15
point register v2 Vector register Same as above, the floating point module and vector module in abiv2 use the same set of registers vabs.8 vr1		
Memory address with immediate offset (rx, offset)		ld.w r1, (r2, 4)
Memory address with register index (rx, ry « n)		ldr.w r1, (r3, r2 « 1)
Address reference symbol name control register		bsr functionname
cr<z, sel> (sel group, z register) General register sequence		mtcr r1, cr<0, 0>
	rx-ry, rz…	push r4-r11,r15

Note: In general, the destination operand is written before the source register, except for the st.[bhw], str.[bhw], and mtcr instructions.

4.4.2 Preprocessing assembly files

When an assembly file contains some C language macro instructions (such as #define, #include, #if, etc.) and comments, it must be preprocessed.

GCC determines whether the assembly file needs preprocessing based on its suffix:

- When the suffix is (.S), it means that the file contains macro instructions that need to be preprocessed
- When the suffix is (.s), it means that the file only contains assembly instructions and does not need to be preprocessed

For example, an assembly file (test.S) containing macro instructions is as follows:

```
#define P 2                    /* Same syntax as C language , Macro definition*/
movi t0,P
```

By adding the -E option to gcc, you can get the preprocessed assembly file. The command and generated file are as follows:

```
csky-elfabiv2-gcc -E test.S -o test.s
```

File test.s:

```
movi t0, 2
```

Note: Do not confuse #include, #if, etc. with .include, .if, etc. #include, #if, etc. are C language macro instructions that need to be processed by the preprocessor, while .include, .if etc. are assembly instructions and only need to be processed by the assembler.

4.4.3 Assembly Directives

In addition to assembly instructions, the assembly source program also contains pseudo instructions, which have no corresponding instructions in the CPU instruction set. Assembly pseudo instructions can be expanded into a

There are three main reasons for using pseudo-instructions:

1. Since the offset distance of the target address of the jump instruction relative to the instruction itself is uncertain, the assembler needs to decide which jump instruction to use;
2. Make the writing of some instructions more concise;
3. The assembly instructions of C-SKY V2.0 are compatible with the assembly instructions of C-SKY V1.0.

The assembly pseudo instructions are shown in Table 4.7 :

Table 4.7: Assembly directives

Directives	Extended instructions		CPU
clrc	cmpne r0,r0	Description Clear the	all
cmplei rd,n	cmplti rd, n+1	C bit and compare the signed immediate value Use less than compatible less than or equal to	all
cmpls rd,rs	cmphs rs, rd	Immediate unsigned comparison Use greater than or equal to be compatible with less than or equal to	all
cmpgt rd,rs	cmplt rs, rd	Signed comparison of immediate values Use less than compatible with greater than or equal to	all
jbsr label	abiv1: bsr label or jsri label abiv2: bsr label	Jump to subroutine	all
jbr label	abiv1: br label or jmpi label abiv2: br label	Unconditional jump	all
jbf label	abiv1: bf label or bt 1f jmpi label 1:… abiv2: bf label (16/32 bits) or bt 1f (16 bits) br/jmpi label (32 bits) 1:…	Jump if C bit is 0	all

Continued on next page

Table 4.7 – Continued from previous page

Directives	Extended instructions		CPU
jbt label	abiv1: bt label or bf 1f jmpi label 1:… abiv2: bt label (16/32 bits) or bf 1f (16 bits) br/jmpi label (32 bits) 1:…	Description: Jump when C bit is 1	all
rts	jmp r15	Return the negated number	all
neg rd	abiv1: rsubi rd,0 abiv2: not rd, rd addi rd, 1	from the subroutine	all
rotlc rd,1	addc rd,rd	Addition with carry all	
rotri rd,imm	rotli rd,32-imm	Immediate number circularly shifts all left	
setc	cmphs r0,r0	Set C bit all	
tstle rd	cmplti rd,1	Test register value is non-positive all	
tstlt rd	btsti rd,31	Test register value is negative all	
tstne rd	cmplnei rd,0	Test register value is non-zero all	
bgeni rz,imm	movi rz,immpow immpow is 2 raised to the power of imm	Set the immth bit of the register Set to 1, other positions to 0	V2.0
ldq r4-r7,(rx)	ldm r4-r7,(rx)	r4=(rx,0),r5=(rx,4), r6=(rx,8),r7=(rx,12)	V2.0
stq r4-r7,(rx)	stm r4-r7,(rx)	(rx,0)=r4,(rx,4)=r5, (rx,8)=r6,(rx,12)=r7	V2.0
mov rz,rx	mov rz,rx or lsli rz,rx,0	rz=rx If rz and rx are both r0~r15, then mov If rz or rx is r16~r31, it is lsli	V2.0
movf rz,rx	incf rz,rx,0	If the C bit is 0, rz=rx If	V2.0
movt rz,rx	inct rz,rx,0	the C bit is 1, rz=rx	V2.0
not rz,rx	nor rz,rx,rx	bitwise negation	V2.0
rsub rz,rx,ry	subu rz,ry,rx	rz=ry-rx	V2.0
rsubi rx,imm16	movi r1,imm16 subu rx,r1,rx	rz=imm16-rx	V2.0
sextb rz,rx	sext rz,rx,7,0	Take the first byte of rx and Sign extension to rz	V2.0

Continued on next page

Table 4.7 – Continued from previous page

Directives	Extended instructions		CPU
sexth rz,rx	sext rz,rx,15,0	Description Take the first word of rx, and Sign extension to rz	V2.0
zextb rz,rx	zext rz,rx,7,0	Take the first byte of rx and Unsign-extend rz	V2.0
zexth rz,rx	zext rz,rx,15,0	Take the first word of rx and Unsign-extend rz	V2.0
lrw rz,imm32	movih rz,imm32_hi16 ori rzÿrz,imm32_lo16	Load 32-bit immediate value into register Device	V2.0
jbez rx,label	bez rx,label or bnez rx,1f br/jmpi label (32 bits) 1:…	If rx is zero, jump to the subroutine sequence	v2.0
jbnez rx,label	bnez rx,label or bez rx,1f br/jmpi label (32 bits) 1:…	If rx is not equal to zero, jump to the sub program	v2.0
jbhz rx,label	bhz rx,label or blsz rx,1f br/jmpi label (32 bits) 1:…	If rx is greater than zero, jump to the subroutine sequence	v2.0
jblsz rx,label	blsz rx,label or bhz rx,1f br/jmpi label (32 bits) 1:…	If rx is less than or equal to zero, jump to Subroutines	v2.0
jblz rx,label	blz rx,label or bhsz rx,1f br/jmpi label (32 bits) 1:…	If rx is less than zero, jump to the subroutine sequence	v2.0
jbhsz rx,label	bhsz rx,label or blz rx,1f br/jmpi label (32 bits) 1:…	If rx is greater than or equal to zero, jump to Subroutines	v2.0

4.4.4 Register Aliases

Many general registers (r0-r31) support aliases, which help improve the readability and compatibility of assembly code in certain situations, as shown in Table 4.8 As shown in Table 4.9 :

Table 4.8: CSKY ABI V1 register aliases

V1 Register Name	Alias	Description
r2-r3		a0-a1 parameter passing/return value passing
r4-r7		a2-a5 parameter transfer
r8-r13		l0-l5 stores local variables (need to be saved and restored at the beginning and end of the function when used)
r14		l10/gb stores local variables/storages the base address of the GOT table when compiled with the PIC option
r15		lr stores the return address
r16-r19		l6-l9 store local variables (need to be saved and restored at the beginning and end of the function when used)
r20-r25		t0-t5 stores temporary data (no need to save and restore at the beginning and end of the function when used)
r31	tls	TLS Register

Table 4.9: CSKY ABI V2 register aliases

V2 Register Name	Alias	Description
r0-r1		a0-a1 parameter passing/return value passing
r2-r3		a2-a3 parameter transfer
r4-r11		l0-l7 stores local variables (need to be saved and restored at the beginning and end of the function when used)
r12-r13		t0-t1 stores temporary data (no need to save and restore at the beginning and end of the function when used)
r14		sp storage stack pointer
r15		lr stores the return address
r16-r17		l8-l9 stores local variables (need to be saved and restored at the beginning and end of the function when used)
r18-r25		t2-t9 stores temporary data (no need to save and restore at the beginning and end of the function when used)
r28		rgb/rdb stores the data section base address/stores the GOT table base address when compiled with the PIC option
r29		rtb stores the base address of the text section
r30		svbr stores the base address of the handler
r31	tls	TLS Register

Note: The parameter register can also be used as a temporary register when the parameter is not used. It does not need to be saved and restored at the beginning and end of the function.

4.5 vdsp

Currently, the CSKY architecture supports two versions of VDSP instruction sets, namely vdspv1 and vdspv2. vdspv1 can be configured with 64-bit and 128-bit The compiler controls the bit width of the generated target code through the option -mvdsp-width=<size> (default 128); vdspv2 bit width is 128 bits. ck810 uses vdspv1, ck860 and ck805 use vdspv2.

The compiler determines whether the generated target program supports vdsp instructions based on the CPU options. The CPUs that support vdsp for ck810 are:

ck810v, ck810fv, ck810tv, ck810ftv (i.e. 810 includes v CPU).

ck860 The CPUs that support vdsp are:

ck860v ck860fv (860 with v CPU)

All ck805 CPUs support vdsp.

The compiler generates vector instructions in the following situations:

- Vector operation expressions
- Loop optimization
- Use intrinsic functions

The first two are for basic scenarios, while the third is for scenarios that require deep optimization. For detailed instructions, please refer to the following sections in this chapter. part:

- Vector data type
- Rules for passing parameters and return values of vector types
- Vector operation expressions
- Loop optimization to generate vector instructions (currently only supported in the *GNU* toolchain)
- *Intrinsic* function interface naming rules
- *Vdspv2* intrinsic interface

4.5.1 Vector Data Types

Vector data types are usually built on top of common data types. For example, the vector data type int8x8_t represents an 8-bit integer data type. A type consisting of 8 elements has a total bit width of 64 bits. The naming convention is as follows:

> [element type][element width]x[number of elements]_t

The element type is int, uint or float. When using, you need to reference the header file csky_vdsp.h. The vector data types supported by vdspv1 and vdspv2 are shown in the table. 4.10 and Table 4.11 :

Table 4.10: Vector data types for vdspv1

vdspv1 64-bit	128-bit	
int	int8x8_t	int8x16_t
	int16x4_t	int16x8_t
	int32x2_t	int32x4_t
uint	uint8x8_t	uint8x16_t
	uint16x4_t	uint16x8_t
	uint32x2_t	uint32x4_t

Table 4.11: Vector data types for vdspv2

vdspv2 128-bit	
int	int8x16_t
	int16x8_t
	int32x4_t
	int64x2_t
uint	uint8x16_t
	uint16x8_t
	uint32x4_t
	uint64x2_t
float	float32x4_t
	float64x2_t

4.5.2 Rules for passing vector type parameters and return values

By default or when option -mfloat-abi=soft/softfp is enabled, vector type arguments and return values are still passed using ordinary registers.

When the option -mfloat-abi=hard is enabled, vector type parameters and return values are no longer passed using ordinary registers. Vector type parameters are passed through registers vr0-vr3

When the vector type parameters exceed 4, the remaining parameters will be passed through the stack. The return value of the vector type is passed through register vr0.

4.5.3 Vector Operation Expressions

The compiler supports vector operation expressions, which consist of vector type variables and operators. The compiler generates corresponding vector instructions based on these expressions.

4.5.3.1 Definition of vector type variables

There are two ways to define vector type variables:

- The first method is the same as array definition, such as:

```
#include<csky_vdsp.h>

int32x4_t a = {1,2,3,4};
```

- The second method is to define an array first, and then convert the array address into a vector pointer type, such as:

```
#include<csky_vdsp.h>

int a[ ] = {1,2,3,4};
int32x4_t *ap = (int32x4_t *)a;
```

4.5.3.2 Operators

C language uses operators to represent arithmetic operations, and the same is true for variables of vector type.

Currently, the operators supported by vector expressions are as follows:

- Addition: +
- Subtraction:-
- Multiplication: *
- Comparison operators: >, <, !=, >=, <=, ==
- Logical operators: &, |, ^
- Shift operators: », «

Here is a simple example:

```
#include<csky_vdsp.h>

int32x4_t a = {1,2,3,4};
int32x4_t b = {5,6,7,8};
int32x4_t c = {2,4,6,8};

int32x4_t vfunc ()
{
    return a        * b + c;
}
```

4.5.4 Loop optimization to generate vector instructions (currently only supported in the GNU toolchain)

The compiler supports optimizing some loops into vector instructions. The compiler will try to optimize the loop into vector instructions when the following conditions are met:

- Current CPUs support vector instructions
- The optimization level is -O1 or above, and option -ftree-loop-vectorize is added

  (This option is enabled by default with -O3)

For example, the following loop:

```
 void svfun1 (int *a,int *b,int *c)
{
    for (int i = 0;i < 4;i++)
        c[i] = a[i] + b[i]; /*scalar operation*/
}
```

If the current CPU supports 128-bit vector addition instructions, after loop optimization is enabled, the optimized code above will be as shown in the following pseudo code:

```
#include <csky_vdsp.h>

int32x4_t svfun2 (int32x4_t va,int32x4_t vb)
{
        int32x4_t vc = va + vb; /* vector operation */
        return vc;
}
```

4.5.5 Intrinsic function interface naming rules

The function name of the intrinsic interface is basically consistent with the instruction name. If the instruction name contains ".", it will be replaced by "_" in the function name. For example, the intrinsic interface function name corresponding to the instruction vmfvr.u32 is vmfvr_u32. The parameter and return value types of the function are determined by the data type of the instruction operand. For example, the instruction vmfvr.u32 rz,vr[index], its function is to transfer the index-th element in the vector register to the ordinary register rz, so the declaration of the function vmfvr_u32 is as follows:

```
uint32_t vmfvr_u32 (uint32x4_t __a, const int32_t __b);
```

The first parameter is of type uint32x4_t, the second parameter is of type int32_t, and the return value is of type uint32_t.

4.5.6 Intrinsic interfaces of vdspv2

Currently, the following CPUs support the compilation of vdspv2 instructions:

- ck860v ck860fv (860 with v CPU)
- All CPUs of ck805

The vdspv2 instructions can be divided into the following parts:

- Integer addition, subtraction, and comparison instructions
- Integer multiplication instructions
- Integer reciprocal, reciprocal square root, e- exponential fast calculation and approximation instructions
- Integer shift instructions
- Integer move (MOV), element operation, bit operation instructions
- Integer immediate value generation instructions
- *LOAD/ STORE* instructions
- Floating point addition and subtraction comparison instructions
- Floating point multiplication instructions
- Floating point reciprocal, reciprocal square root, e exponential fast calculation and approximation instructions
- Floating point conversion instructions

4.5.6.1 Integer addition, subtraction and comparison instructions

vadd.t && vsub.t

- uint8x16_t vadd_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vadd_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vadd_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vadd_u64 (uint64x2_t, uint32x4_t)
- int8x16_t vadd_s8 (int8x16_t, int8x16_t)
- int16x8_t vadd_s16 (int16x8_t, int16x8_t)
- int32x4_t vadd_s32 (int32x4_t, int32x4_t)
- int64x2_t vadd_s64 (int64x2_t, int32x4_t)

```
>>> Function Description: Vector addition
        Assuming parameters Vx, Vy, return value Vz
        Vz(i)=Vx(i)+Vy(i);                    i=0:(number-1)
```

- uint8x16_t vsub_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vsub_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vsub_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vsub_u64 (uint64x2_t, uint32x4_t)
- int8x16_t vsub_s8 (int8x16_t, int8x16_t)
- int16x8_t vsub_s16 (int16x8_t, int16x8_t)
- int32x4_t vsub_s32 (int32x4_t, int32x4_t)
- int64x2_t vsub_s64 (int64x2_t, int32x4_t)

```
>>> Function Description: Vector Subtraction
        Assuming parameters Vx, Vy, return value Vz
        Vz(i)=Vx(i)-Vy(i);                    i=0:(number-1)
```

vadd.te && vsub.te

- uint16x16_t vadd_u8_e (uint8x16_t, uint8x16_t)
- uint32x8_t vadd_u16_e (uint16x8_t, uint16x8_t)
- uint64x4_t vadd_u32_e (uint32x4_t, uint32x4_t)

```
>>> Function description: Vector unsigned extension first
        extends the parameters to zero, then vector addition
        assumes parameters Vx, Vy, and returns value Vz
        Vz(i)=extend(Vx(i))+extend(Vy(i)) i=0:number-1
```

- int16x16_t vadd_s8_e (int8x16_t, int8x16_t)

- int32x8_t vadd_s16_e (int16x8_t, int16x8_t)
- int64x4_t vadd_s32_e (int32x4_t, int32x4_t)

>>> Function Description: Vector signed extension addition

First, the parameters are sign-extended. Second, vector addition

assumes parameters Vx, Vy, and returns value Vz.

Vz(i)=extend(Vx(i))+extend(Vy(i)) i=0:number-1

- uint16x16_t vsub_u8_e (uint8x16_t, uint8x16_t)
- uint32x8_t vsub_u16_e (uint16x8_t, uint16x8_t)
- uint64x4_t vsub_u32_e (uint32x4_t, uint32x4_t)

>>> Function description: Vector unsigned extension subtraction first

extends the parameters to zero, then vector subtraction

assumes parameters Vx, Vy, and returns value Vz

Vz(i)=extend(Vx(i))-extend(Vy(i)) i=0:number-1

- int16x16_t vsub_s8_e (int8x16_t, int8x16_t)
- int32x8_t vsub_s16_e (int16x8_t, int16x8_t)
- int64x4_t vsub_s32_e (int32x4_t, int32x4_t)

>>> Function description: Vector signed extension addition First,

the parameters are signed extended Second, vector subtraction

Assume parameters Vx, Vy, return value Vz

Vz(i)=extend(Vx(i))-extend(Vy(i)) i=0:number-1

vadd.th && vsub.th

- uint16x8_t vadd_u16_h (uint16x8_t, uint16x8_t)
- uint32x4_t vadd_u32_h (uint32x4_t, uint32x4_t)
- uint64x2_t vadd_u64_h (uint64x2_t, uint64x4_t)
- int16x8_t vadd_s16_h (int16x8_t, int16x8_t)
- int32x4_t vadd_s32_h (int32x4_t, int32x4_t)
- int64x2_t vadd_s64_h (int64x2_t, int64x4_t)

>>> Function description: Vector high-order

addition The result of addition takes the high half of the elements and puts them into the low

half of the return value vector in order Assume that Vx

and Vy are two parameters, and Vz is the return value tmp(i)=(Vx(i)+Vy(i))[element_size-1:element_size/2]; i=0:(number-1)

Vz(i)={Tmp(2i+1), Tmp(2i)}; i=0:number/2-1

- uint16x8_t vsub_u16_h (uint16x8_t, uint16x8_t)
- uint32x4_t vsub_u32_h (uint32x4_t, uint32x4_t)
- uint64x2_t vsub_u64_h (uint64x2_t, uint64x4_t)

- int16x8_t vsub_s16_h (int16x8_t, int16x8_t)
- int32x4_t vsub_s32_h (int32x4_t, int32x4_t)
- int64x2_t vsub_s64_h (int64x2_t, int64x4_t)

>>> Function description: The result of vector
high-order addition and subtraction is to take the high half of the elements and put them into the
low half of the return value vector in order. Assume that
Vx and Vy are two parameters and Vz is the return value. tmp(i)=(Vx(i)-Vy(i))[element_size-1:element_size/2]; i=0:(number-1)
Vz(i)={Tmp(2i+1), Tmp(2i)}; i=0:number/2-1

vadd.ts && vsub.ts

- uint8x16_t vadd_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vadd_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vadd_u32_s (uint32x4_t, uint32x4_t)
- uint64x2_t vadd_u64_s (uint64x2_t, uint64x4_t)
- int8x16_t vadd_s8_s (int8x16_t, int8x16_t)
- int16x8_t vadd_s16_s (int16x8_t, int16x8_t)
- int32x4_t vadd_s32_s (int32x4_t, int32x4_t)
- int64x2_t vadd_s64_s (int64x2_t, int64x4_t)

>>> Function Description: Vector Saturation Addition
Assume Vx and Vy are two parameters, Vz is the return value, and U/S indicates whether
it is signed or not (T==S); (select according to the element U/S type)
Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;
Min=signed ? -2^(element_size-1) : 0;
If Vx(i)+Vy(i)>Max Vz(i)=Max;
Else if Vx(i)+Vy(i)<Min Vz(i)=Min;
Else Vz(i)= Vx(i)+Vy(i);
End i=0:(number-1)

- uint8x16_t vsub_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vsub_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vsub_u32_s (uint32x4_t, uint32x4_t)
- uint64x2_t vsub_u64_s (uint64x2_t, uint64x4_t)
- int8x16_t vsub_s8_s (int8x16_t, int8x16_t)
- int16x8_t vsub_s16_s (int16x8_t, int16x8_t)
- int32x4_t vsub_s32_s (int32x4_t, int32x4_t)
- int64x2_t vsub_s64_s (int64x2_t, int64x4_t)

>>> Function Description: Vector saturation subtraction

Assume Vx and Vy are two parameters, Vz is the return value, and U/S indicates whether it is signed or not (T==S); (select according to the element U/S type)

Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;

Min=signed ? -2^(element_size-1) : 0;

If Vx(i)-Vy(i)>Max Vz(i)=Max;

Else if Vx(i)-Vy(i)<Min Vz(i)=Min;

Else Vz(i)= Vx(i)-Vy(i);

End i=0:(number-1)

vadd.t.rh && vsub.t.rh

- int16x8_t vadd_s16_rh (int16x8_t, int16x8_t)
- int32x4_t vadd_s32_rh (int32x4_t, int32x4_t)
- int64x2_t vadd_s64_rh (int64x2_t, int64x2_t)
- uint16x8_t vadd_u16_rh (uint16x8_t, uint16x8_t)
- uint32x4_t vadd_u32_rh (uint32x4_t, uint32x4_t)
- uint64x2_t add_u64_rh (uint64x2_t, uint64x2_t)

>>> Function description: The addition result is rounded to take the high half

Assume Vx, Vy are two parameters, Vz is the return value round=1<<(element_size/2-1);

Tmp(i)=(Vx(i)+Vy(i)+round) [element_size-1:element_size/2]; (the addition result is rounded and the high half is taken) i=0:(number-1)

Vz(i)={Tmp(2i+1), Tmp(2i)}; The results are placed in the lower half of the destination register Vz in order (default) i=0:number/2-1

- int16x8_t vsub_s16_rh (int16x8_t, int16x8_t)
- int32x4_t vsub_s32_rh (int32x4_t, int32x4_t)
- int64x2_t vsub_s64_rh (int64x2_t, int64x2_t)
- uint16x8_t vsub_u16_rh (uint16x8_t, uint16x8_t)
- uint32x4_t vsub_u32_rh (uint32x4_t, uint32x4_t)
- uint64x2_t asub_u64_rh (uint64x2_t, uint64x2_t)

>>> Function description: The subtraction result is rounded to the upper half

Assume Vx, Vy are two parameters, Vz is the return value round=1<<(element_size/2-1);

Tmp(i)=(Vx(i)-Vy(i)+round) [element_size-1:element_size/2]; (The addition result is rounded and the high half is taken) i=0:(number-1)

Vz(i)={Tmp(2i+1), Tmp(2i)}; The results are placed in the lower half of the destination register Vz in order (default) i=0:number/2-1

vaddh.t && vsubh.t

- int8x16_t vaddh_s8 (int8x16_t, int8x16_t)
- int16x8_t vaddh_s16 (int16x8_t, int16x8_t)
- int32x4_t vaddh_s32 (int32x4_t, int32x4_t)
- uint8x16_t vaddh_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vaddh_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vaddh_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Additive average

operation assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)=(Vx(i)+Vy(i))>>1; i=0:number-1 For U, right shift is

logical right shift For S, right shift is arithmetic right shift

- int8x16_t vsubh_s8 (int8x16_t, int8x16_t)
- int16x8_t vsubh_s16 (int16x8_t, int16x8_t)
- int32x4_t vsubh_s32 (int32x4_t, int32x4_t)
- uint8x16_t vsubh_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vsubh_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vsubh_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Subtraction average

operation assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)=(Vx(i)-Vy(i))>>1; i=0:number-1 For U, right shift is

logical right shift For S, right shift is arithmetic right shift

vaddh.tr && vsubh.tr

- int8x16_t vaddh_s8_r (int8x16_t, int8x16_t)
- int16x8_t vaddh_s16_r (int16x8_t, int16x8_t)
- int32x4_t vaddh_s32_r (int32x4_t, int32x4_t)
- uint8x16_t vaddh_u8_r (uint8x16_t, uint8x16_t)
- uint16x8_t vaddh_u16_r (uint16x8_t, uint16x8_t)
- uint32x4_t vaddh_u32_r (uint32x4_t, uint32x4_t)

>>> Function description: Addition, averaging and rounding operation

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=(Vx(i)+Vy(i)+1)>>1; i=0:number-1 For U, right shift is

logical right shift , For S, right shift is arithmetic right shift

- int8x16_t vsubh_s8_r (int8x16_t, int8x16_t)
- int16x8_t vsubh_s16_r (int16x8_t, int16x8_t)

- int32x4_t vsubh_s32_r (int32x4_t, int32x4_t)
- uint8x16_t vsubh_u8_r (uint8x16_t, uint8x16_t)
- uint16x8_t vsubh_u16_r (uint16x8_t, uint16x8_t)
- uint32x4_t vsubh_u32_r (uint32x4_t, uint32x4_t)

>>> Function description: Subtraction average and rounding operation

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=(Vx(i)-Vy(i)+1)>>1; i=0:number-1 For U, right shift is logical right shift , For S, right shift is arithmetic right shift

vadd.tx & vsub.tx

- int16x16_t vadd_s8_x (int16x16_t, int8x16_t)
- int32x8_t vadd_s16_x (int32x8_t, int16x8_t)
- int64x4_t vadd_s32_x (int64x4_t, int32x4_t)
- uint16x16_t vadd_u8_x (uint16x16_t, uint8x16_t)
- uint32x8_t vadd_u16_x (uint32x8_t, uint16x8_t)
- uint64x4_t vadd_u32_x (uint64x4_t, uint32x4_t)

>>> Function description: Extended addition

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i) =Vx(i) + extend(Vy(i)); i=0:number-1 extend Zero-extend or sign-extend the value to twice the element width according to U/S

- int16x16_t vsub_s8_x (int16x16_t, int8x16_t)
- int32x8_t vsub_s16_x (int32x8_t, int16x8_t)
- int64x4_t vsub_s32_x (int64x4_t, int32x4_t)
- uint16x16_t vsub_u8_x (uint16x16_t, uint8x16_t)
- uint32x8_t vsub_u16_x (uint32x8_t, uint16x8_t)
- uint64x4_t vsub_u32_x (uint64x4_t, uint32x4_t)

>>> Function description: Extended subtraction

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i) =Vx(i) - extend(Vy(i)); i=0:number-1 extend Zero-extend or sign-extend the value to twice the element width according to U/S

vpadd.t

- int8x16_t vpadd_s8 (int8x16_t, int8x16_t)
- int16x8_t vpadd_s16 (int16x8_t, int16x8_t)
- int32x4_t vpadd_s32 (int32x4_t, int32x4_t)

- int64x2_t vpadd_s64 (int64x2_t, int64x2_t)
- uint8x16_t vpadd_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vpadd_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vpadd_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vpadd_u64 (uint64x2_t, uint64x2_t)

```
>>> Function Description: Vector adjacent element addition
    Assume Vx, Vy are two parameters, Vz is the
    Vz(i)=Vx(2i)+Vx(2i+1);                    return value i=0:(number/2-1)
    Vz(number/2+i)=Vy(2i)+Vy(2i+1);                    i=0:(number/2-1)
```

vpadd.ts

- int8x16_t vpadd_s8_s (int8x16_t, int8x16_t)
- int16x8_t vpadd_s16_s (int16x8_t, int16x8_t)
- int32x4_t vpadd_s32_s (int32x4_t, int32x4_t)
- int64x2_t vpadd_s64_s (int64x2_t, int64x2_t)
- uint8x16_t vpadd_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vpadd_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vpadd_u32_s (uint32x4_t, uint32x4_t)
- uint64x2_t vpadd_u64_s (uint64x2_t, uint64x2_t)

```
>>> Function description: Saturation addition of adjacent elements
    of a vector Assume that Vx and Vy are two parameters, Vz is the return value, and U/
    S is the sign bit signed=(T==S); (select according to the element U/S type)
    Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
    Min=signed? -2^(element_size-1):0;
    If (Vx(2i) + Vx(2i+1))>Max Vz(i)=Max;
    Else if (Vx(2i) + Vx(2i+1))<Min Vz(i)=Min;
    Else Vz(i) = Vx(2i) + Vx(2i+1);
    End i=0:(number/2-1)
    If (Vy(2i) + Vy(2i+1))>Max Vz(number/2+i)=Max;
    Else if (Vy(2i) + Vy(2i+1))<Min Vz(number/2+i)=Min;
    Else Vz(number/2+i) = Vy(2i) + Vy(2i+1);
    End i=0:(number/2-1)
```

vpadd.te

- int16x8_t vpadd_s8_e (int8x16_t)
- int32x4_t vpadd_s16_e (int16x8_t)
- int64x2_t vpadd_s32_e (int32x4_t)

- uint16x8_t vpadd_u8_e (uint8x16_t)
- uint32x4_t vpadd_u16_e (uint16x8_t)
- uint64x2_t vpadd_u32_e (uint32x4_t)

>>> Function description: Vector adjacent element extension addition

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(2i+1:2i) =extend(Vx(2i)) +extend(Vx(2i+1)); i=0:(number/2-1)

extend Zero-extend or sign-extend the value to twice the element width according to U/S

vpadda.te

- int16x8_t vpadda_s8_e (int16x8_t, int8x16_t)
- int32x4_t vpadda_s16_e (int32x4_t, int16x8_t)
- int64x2_t vpadda_s32_e (int64x2_t, int32x4_t)
- uint16x8_t vpadda_u8_e (uint16x8_t, uint8x16_t)
- uint32x4_t vpadda_u16_e (uint32x4_t, uint16x8_t)
- uint64x2_t vpadda_u32_e (uint64x2_t, uint32x4_t)

>>> Function description: Vector adjacent elements expansion

accumulation Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(2i+1:2i) =Vz(2i+1:2i) +extend(Vx(2i))+extend(Vx(2i+1)); i=0:(number/2-ÿ1)

extend Zero-extend or sign-extend the value to twice the element width according to U/S

vsax.ts

- int8x16_t vsax_s8_s (int8x16_t, int8x16_t)
- int16x8_t vsax_s16_s (int16x8_t, int16x8_t)
- int32x4_t vsax_s32_s (int32x4_t, int32x4_t)
- uint8x16_t vsax_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vsax_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vsax_u32_s (uint32x4_t, uint32x4_t)

>>> Function description: Vector offset

subtraction and addition Assume Vx, Vy are two parameters, Vz is the return

value, U/S is the sign bit signed=(T==S); (select according to the element U/S type)

Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;

Min=signed? -2^(element_size-1):0;

If (Vx(2i+1)-Vy(2i))>Max Vz(2i+1)=Max;

Else if (Vx(2i+1)-Vy(2i))<Min Vz(2i+1)=Min;

Else Vz(2i+1)= Vx(2i+1)-Vy(2i);

(Continued on next page)

(Continued from previous page)

```
End i=0:(number/2-1)
If (Vx(2i)+Vy(2i+1))>Max Vz(2i)=Max;
Else if (Vx(2i)+Vy(2i+1))<Min Vz(2i)=Min;
Else Vz(2i)= Vx(2i)+Vy(2i+1);
End i=0:(number/2-1)
```

vasx.ts

- int8x16_t vasx_s8_s (int8x16_t, int8x16_t)
- int16x8_t vasx_s16_s (int16x8_t, int16x8_t)
- int32x4_t vasx_s32_s (int32x4_t, int32x4_t)
- uint8x16_t vasx_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vasx_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vasx_u32_s (uint32x4_t, uint32x4_t)

```
>>> Function description: Vector offset addition and
    subtraction Assume that Vx and Vy are two parameters, Vz is the return value, and U/S
    is the sign bit signed=(T==S); (select according to the element U/S type)
    Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
    Min=signed? -2^(element_size-1):0;
    If (Vx(2i+1) +Vy(2i))>Max Vz(2i+1)=Max;
    Else if (Vx(2i+1) +Vy(2i))<Min Vz(2i+1)=Min;
    Else Vz(2i+1) = Vx(2i+1)+Vy(2i);
    End i=0:(number/2-1)
    If (Vx(2i)-Vy(2i+1))>Max Vz(2i)=Max;
    Else if (Vx(2i)-Vy(2i+1))<Min Vz(2i)=Min;
    Else Vz(2i)= Vx(2i)-Vy(2i+1);
    End i=0:(number/2-1)
```

vsaxh.t

- int8x16_t vsaxh_s8(int8x16_t, int8x16_t)
- int16x8_t vsaxh_s16(int16x8_t, int16x8_t)
- int32x4_t vsaxh_s32(int32x4_t, int32x4_t)
- uint8x16_t vsaxh_u8(uint8x16_t, uint8x16_t)
- uint16x8_t vsaxh_u16(uint16x8_t, uint16x8_t)
- uint32x4_t vsaxh_u32(uint32x4_t, uint32x4_t)

```
>>> Function description: Get the average value after vector misalignment subtraction
    and addition. Assume that Vx and Vy are two parameters, Vz is the return value, and U/S is the sign bit
```

(Continued on next page)

(Continued from previous page)

Vz(2i+1)=(Vx(2i+1)-Vy(2i)) >>1;

Vz(2i)=(Vx(2i)+Vy(2i+1)) >>1; i=0:(number/2-1)

For U, right shift is logical right shift ， For S, right shift is arithmetic right shift

vasxh.t

- int8x16_t vasxh_s8(int8x16_t, int8x16_t)
- int16x8_t vasxh_s16(int16x8_t, int16x8_t)
- int32x4_t vasxh_s32(int32x4_t, int32x4_t)
- uint8x16_t vasxh_u8(uint8x16_t, uint8x16_t)
- uint16x8_t vasxh_u16(uint16x8_t, uint16x8_t)
- uint32x4_t vasxh_u32(uint32x4_t, uint32x4_t)

>>> Function description: Get the average value after vector misalignment addition and subtraction

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(2i+1)=(Vx(2i+1)+Vy(2i))>>1;

Vz(2i)=(Vx(2i)-Vy(2i+1))>>1; i=0:(number/2-1) For S, right shift is arithmetic

For U, right shift is logical right shift ， right shift

vabs.t

- int8x16_t vabs_s8(int8x16_t)
- int16x8_t vabs_s16(int16x8_t)
- int32x4_t vabs_s32(int32x4_t)

>>> Function description: vector element takes absolute value.

Assume Vx is the parameter and Vz is the return value.

Vz(i)=abs(Vx(i)); i=0:number-1

vabs.ts

- int8x16_t vabs_s8_s(int8x16_t)
- int16x8_t vabs_s16_s(int16x8_t)
- int32x4_t vabs_s32_s(int32x4_t)

>>> Function description: vector element saturation absolute

value Assume Vx is the parameter, Vz is the return value, U/S is the sign bit

If Vx(i)==-2^(element_size-1) Vz(i)= 2^(element_size-1)-1;

Else Vz(i)=abs(Vx(i));

End i=0:number-1

vsabs.ts

- int8x16_t vsabs_s8_s(int8x16_t, int8x16_t)
- int16x8_t vsabs_s16_s(int16x8_t, int16x8_t)
- int32x4_t vsabs_s32_s(int32x4_t, int32x4_t)
- uint8x16_t vsabs_u8_s(uint8x16_t, uint8x16_t)
- uint16x8_t vsabs_u16_s(uint16x8_t, uint16x8_t)
- uint32x4_t vsabs_u32_s(uint32x4_t, uint32x4_t)

>>> Function description: vector element subtraction saturated

absolute value Assume Vx, Vy are parameters, Vz is the return value, U/S is the sign bit

U: Max=2^(element_size)-1; Min= - Max;

S: Max=2^(element_size-1)-1, Min= - Max;

If (Vx(i)-Vy(i)) < Min || (Vx(i)-Vy(i)) > Max

Vz(i)= Max;

Else Vz(i)=abs(Vx(i)-Vy(i));

End i=0:number-1

vsabs.te

- int16x16_t vsabs_s8_e(int8x16_t, int8x16_t)
- int32x8_t vsabs_s16_e(int16x8_t, int16x8_t)
- int64x4_t vsabs_s32_e(int32x4_t, int32x4_t)
- uint16x16_t vsabs_u8_e(uint8x16_t, uint8x16_t)
- uint32x8_t vsabs_u16_e(uint16x8_t, uint16x8_t)
- uint64x4_t vsabs_u32_e(uint32x4_t, uint32x4_t)

>>> Function description: Take the absolute value after vector element expansion and subtraction

Assume Vx, Vy are parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=abs(extend(Vx(i))-extend(Vy(i))); i=0:number-1

extend Zero-extend or sign-extend the value to twice the element width according to U/S

vsabsa.t

- int8x16_t vsabsa_s8(int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vsabsa_s16(int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vsabsa_s32(int32x4_t, int32x4_t, int32x4_t)
- uint8x16_t vsabsa_u8(uint8x16_t, uint8x16_t, uint8x16_t)
- uint16x8_t vsabsa_u16(uint16x8_t, uint16x8_t, uint16x8_t)
- uint32x4_t vsabsa_u32(uint32x4_t, uint32x4_t, uint32x4_t)

>>> Function description: Take the absolute value after subtracting vector , Then add
elements. Assume that Vz, Vx, Vy are parameters, Vz is the return value, and U/S is the sign bit.
Vz(i)= Vz(i)+abs(Vx(i)-Vy(i)) ; i=0:number-1

vsabsa.te

- int16x16_t vsabsa_s8_e(int16x16_t, int8x16_t, int8x16_t)
- int32x8_t vsabsa_s16_e(int32x8_t, int16x8_t, int16x8_t)
- int64x4_t vsabsa_s32_e(int64x4_t, int32x4_t, int32x4_t)
- uint16x16_t vsabsa_u8_e(uint16x16_t, uint8x16_t, uint8x16_t)
- uint32x8_t vsabsa_u16_e(uint32x8_t, uint16x8_t, uint16x8_t)
- uint64x4_t vsabsa_u32_e(uint64x4_t, uint32x4_t, uint32x4_t)

>>> Function description: vector element expansion, subtraction, absolute value, and then accumulation.
Assume that Vz, Vx, Vy are parameters, Vz is the return value, and U/S is the sign bit.
Vz(i)= Vz(i)+abs(extend(Vx(i))-extend(Vy(i))); i=0:number-1
extend Zero-extend or sign-extend the value to twice the element width according to U/S

vneg.t

- int8x16_t vneg_s8 (int8x16_t)
- int16x8_t vneg_s16 (int16x8_t)
- int32x4_t vneg_s32 (int32x4_t)

>>> Function description: Negative vector element
Assume Vx is the parameter and Vz is the return value
Vz(i)=-Vx(i) ; i=0:number-1

vneg.ts

- int8x16_t vneg_s8_s (int8x16_t)
- int16x8_t vneg_s16_s (int16x8_t)
- int32x4_t vneg_s32_s (int32x4_t)

>>> Function description: Vector element saturation is negative.
Assume Vx is the parameter and Vz is the return value.
If Vx(i)==-2^(element_size-1) Vz(i)= 2^(element_size-1)-1;
Else Vz(i)=-Vx(i));
End i=0:number-1

vmax.t && vmin.t

- int8x16_t vmax_s8 (int8x16_t, int8x16_t)
- int16x8_t vmax_s16 (int16x8_t, int16x8_t)
- int32x4_t vmax_s32 (int32x4_t, int32x4_t)
- uint8x16_t vmax_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vmax_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vmax_u32 (uint32x4_t, uint32x4_t)

>>> Function description: vector element takes the maximum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max((Vx(i),Vy(i)) ; i=0:number-1 max takes the larger value of the two elements

- int8x16_t vmin_s8 (int8x16_t, int8x16_t)
- int16x8_t vmin_s16 (int16x8_t, int16x8_t)
- int32x4_t vmin_s32 (int32x4_t, int32x4_t)
- uint8x16_t vmin_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vmin_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vmin_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Minimum value of vector elements

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=min((Vx(i),Vy(i)) ; i=0:number-1 min takes the smaller value of the two elements

vpmax.t && vpmin.t

- int8x16_t vpmax_s8 (int8x16_t, int8x16_t)
- int16x8_t vpmax_s16 (int16x8_t, int16x8_t)
- int32x4_t vpmax_s32 (int32x4_t, int32x4_t)
- uint8x16_t vpmax_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vpmax_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vpmax_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Take the maximum value of adjacent elements of a vector

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max(Vx(2i),Vx(2i+1)); i=0:(number/2-1)

Vz(number/2+i)=max(Vy(2i),Vy(2i+1)); max takes the larger value of the two elements i=0:(number/2-1)

- int8x16_t vpmin_s8 (int8x16_t, int8x16_t)

- int16x8_t vpmin_s16 (int16x8_t, int16x8_t)
- int32x4_t vpmin_s32 (int32x4_t, int32x4_t)
- uint8x16_t vpmin_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vpmin_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vpmin_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Take the minimum value of adjacent elements of a vector. Assume that Vx and Vy are two parameters and Vz is the return value.

Vz(i)=min(Vx(2i),Vx(2i+1)); i=0:(number/2-1)

Vz(number/2+i)=min(Vy(2i),Vy(2i+1)); i=0:(number/2-1)

min takes the smaller value of the two elements

vcmp[ne/hs/lt/h/ls]zt

- int8x16_t vcmpnez_s8 (int8x16_t)
- int16x8_t vcmpnez_s16 (int16x8_t)
- int32x4_t vcmpnez_s32 (int32x4_t)
- uint8x16_t vcmpnez_u8 (uint8x16_t)
- uint16x8_t vcmpnez_u16 (uint16x8_t)
- uint32x4_t vcmpnez_u32 (uint32x4_t)

>>> Function description: Vector elements are not equal to 0.

Assume that Vx is the parameter and Vz is the return value.

If Vx(i)!=0 Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmplsz_s8 (int8x16_t)
- int16x8_t vcmplsz_s16 (int16x8_t)
- int32x4_t vcmplsz_s32 (int32x4_t)

>>> Function description: vector elements are less than or equal to 0.

Assume that Vx is the parameter and Vz is the return value.

If Vx(i)ÿ0 Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmpltz_s8 (int8x16_t)
- int16x8_t vcmpltz_s16 (int16x8_t)
- int32x4_t vcmpltz_s32 (int32x4_t)

>>> Function description: vector element is less than 0

Assume Vx is the parameter and Vz is the return value

If Vx(i)<0 Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmmphz_s8 (int8x16_t)
- int16x8_t vcmmphz_s16 (int16x8_t)
- int32x4_t vcmmphz_s32 (int32x4_t)

>>> Function description: vector elements are greater than 0

Assume Vx is the parameter and Vz is the return value

If Vx(i)>0 Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmphsz_s8 (int8x16_t)
- int16x8_t vcmphsz_s16 (int16x8_t)
- int32x4_t vcmphsz_s32 (int32x4_t)

>>> Function description: vector elements are greater than or equal to 0

Assume Vx is the parameter and Vz is the return value

If Vx(i)ÿ0 Vz(i)=11…111;

Else Vz(i)=00…000 ;

i=0:number-1

vcmp[ne/hs/h/lt/ls].t

- int8x16_t vcmplt_s8 (int8x16_t, int8x16_t)
- int16x8_t vcmplt_s16 (int16x8_t, int16x8_t)
- int32x4_t vcmplt_s32 (int32x4_t, int32x4_t)
- uint8x16_t vcmplt_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vcmplt_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vcmplt_u32 (uint32x4_t, uint32x4_t)

>>> Function description: vector element is less

than assuming Vx, Vy are two parameters, Vz is the return value

If Vx(i)<Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmpls_s8 (int8x16_t, int8x16_t)
- int16x8_t vcmpls_s16 (int16x8_t, int16x8_t)

- int32x4_t vcmpls_s32 (int32x4_t, int32x4_t)
- uint8x16_t vcmpls_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vcmpls_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vcmpls_u32 (uint32x4_t, uint32x4_t)

>>> Function Description: Vector element is less than or equal to

Assume Vx, Vy are two parameters, Vz is the return value

If Vx(i)<=Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmphs_s8 (int8x16_t, int8x16_t)
- int16x8_t vcmphs_s16 (int16x8_t, int16x8_t)
- int32x4_t vcmphs_s32 (int32x4_t, int32x4_t)
- uint8x16_t vcmphs_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vcmphs_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vcmphs_u32 (uint32x4_t, uint32x4_t)

>>> Function Description: Vector elements are greater than or equal to

Assume Vx, Vy are two parameters, Vz is the return value

If Vx(i)>=Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmph_s8 (int8x16_t, int8x16_t)
- int16x8_t vcmph_s16 (int16x8_t, int16x8_t)
- int32x4_t vcmph_s32 (int32x4_t, int32x4_t)
- uint8x16_t vcmph_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vcmph_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vcmph_u32 (uint32x4_t, uint32x4_t)

>>> Function description: vector element is greater

than assuming Vx, Vy are two parameters, Vz is the return value

If Vx(i)>Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x16_t vcmpne_s8 (int8x16_t, int8x16_t)
- int16x8_t vcmpne_s16 (int16x8_t, int16x8_t)
- int32x4_t vcmpne_s32 (int32x4_t, int32x4_t)
- uint8x16_t vcmpne_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vcmpne_u16 (uint16x8_t, uint16x8_t)

- uint32x4_t vcmpne_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Vector elements are not equal.

Assume that Vx and Vy are two parameters and Vz is the return value.

If Vx(i)!=Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

vclip.t

- int8x16_t vclip_s8 (int8x16_t, const int)
- int16x8_t vclip_s16 (int16x8_t, const int)
- int32x4_t vclip_s32 (int32x4_t, const int)
- int64x2_t vclip_s64 (int64x2_t, const int)
- uint8x16_t vclip_u8 (uint8x16_t, const int)
- uint16x8_t vclip_u16 (uint16x8_t, const int)
- uint32x4_t vclip_u32 (uint32x4_t, const int)
- uint64x2_t vclip_u64 (uint64x2_t, const int)

>>> Function description: Vector clipping to saturation value

Assume Vx,imm6 are two parameters, Vz is the return value, and U/S is the sign bit

U: Max=2^(imm6)-1, Min=0;

S: Max=2^(imm6-1)-1, Min=-2^(imm6-1); Regardless of T is

U/S, Vx(i) is always considered as a signed number If Vx(i)>Max Vz(i)=Max;

else if Vx(i)<Min Vz(i)=Min;

else Vz(i)=Vx(i);

end i=0:number-1

U:imm6 ranges from 0 to (element_size-1)

S:imm6 range is 1 ~ (element_size)

4.5.6.2 Integer multiplication instructions

vmul.t && vmuli.t

- int8x16_t vmul_s8 (int8x16_t, int8x16_t)
- int16x8_t vmul_s16 (int16x8_t, int16x8_t)
- int32x4_t vmul_s32 (int32x4_t, int32x4_t)
- uint8x16_t vmul_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vmul_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vmul_u32 (uint32x4_t, uint32x4_t)

>>> Function description: Vector element

multiplication assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)=Vx(i)*Vy(i); i=0:number-1

- int8x16_t vmuli_s8 (int8x16_t, int8x16_t, const int)
- int16x8_t vmuli_s16 (int16x8_t, int16x8_t, const int)
- int32x4_t vmuli_s32 (int32x4_t, int32x4_t, const int)
- uint8x16_t vmuli_u8 (uint8x16_t, uint8x16_t, const int)
- uint16x8_t vmuli_u16 (uint16x8_t, uint16x8_t, const int)
- uint32x4_t vmuli_u32 (uint32x4_t, uint32x4_t, const int)

>>> Function Description: Vector element multiplication

Assume Vx, Vy, index are three parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=Vx(i)*Vy(index)); i=0:number-1

The range of index is 0~(128/element_size -1)

vmul.th && vmuli.th

- int8x16_t vmul_s8_h (int8x16_t, int8x16_t)
- int16x8_t vmul_s16_h (int16x8_t, int16x8_t)
- int32x4_t vmul_s32_h (int32x4_t, int32x4_t)
- uint8x16_t vmul_u8_h (uint8x16_t, uint8x16_t)
- uint16x8_t vmul_u16_h (uint16x8_t, uint16x8_t)
- uint32x4_t vmul_u32_h (uint32x4_t, uint32x4_t)

>>> Function description: Vector element multiplication takes the high half.

Assume Vx and Vy are two parameters and Vz is the return value.

Vz(i)=(Vx(i)*Vy(i)) [2*element_size-1: element_size]; i=0:number-1 Take the high part of the multiplication result

- int8x16_t vmuli_s8_h (int8x16_t, int8x16_t, const int)
- int16x8_t vmuli_s16_h (int16x8_t, int16x8_t, const int)
- int32x4_t vmuli_s32_h (int32x4_t, int32x4_t, const int)
- uint8x16_t vmuli_u8_h (uint8x16_t, uint8x16_t, const int)
- uint16x8_t vmuli_u16_h (uint16x8_t, uint16x8_t, const int)
- uint32x4_t vmuli_u32_h (uint32x4_t, uint32x4_t, const int)

>>> Function description: vector element multiplication takes the high half

Assume Vx, Vy, index are three parameters, Vz is the return value

Vz(i)=(Vx(i)*Vy(index)) [2*element_size-1:element_size]; i=0:number-1 The range of the high index of the multiplication result is 0~(128/

element_size -1)

vmul.te && vmuli.te

- int16x16_t vmul_s8_e (int8x16_t, int8x16_t)
- int32x8_t vmul_s16_e (int16x8_t, int16x8_t)
- int64x4_t vmul_s32_e (int32x4_t, int32x4_t)
- uint16x16_t vmul_u8_e (uint8x16_t, uint8x16_t)
- uint32x8_t vmul_u16_e (uint16x8_t, uint16x8_t)
- uint64x4_t vmul_u32_e (uint32x4_t, uint32x4_t)

>>> Function Description: Vector element extension multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=(Vx(i)*Vy(i))[2*element_size-1:0]; i=0:(number-1) The multiplication result takes full precision, that is, twice the element width

- int16x16_t vmuli_s8_e (int8x16_t, int8x16_t, const int)
- int32x8_t vmuli_s16_e (int16x8_t, int16x8_t, const int)
- int64x4_t vmuli_s32_e (int32x4_t, int32x4_t, const int)
- uint16x16_t vmuli_u8_e (uint8x16_t, uint8x16_t, const int)
- uint32x8_t vmuli_u16_e (uint16x8_t, uint16x8_t, const int)
- uint64x4_t vmuli_u32_e (uint32x4_t, uint32x4_t, const int)

>>> Function Description: Vector element extension multiplication

Assume Vx, Vy, index are three parameters, Vz is the return value

Vz(i)=(Vx(i)*Vy(index))[2*element_size-1:0]; i=0:(number-1) The multiplication result takes full precision, that is, twice the element width. The range of index is 0~(128/element_size -1)

vmula.t && vmulai.t

- int8x16_t vmula_s8 (int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vmula_s16 (int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vmula_s32 (int32x4_t, int32x4_t, int32x4_t)
- uint8x16_t vmula_u8 (uint8x16_t, uint8x16_t, uint8x16_t)
- uint16x8_t vmula_u16 (uint16x8_t, uint16x8_t, uint16x8_t)
- uint32x4_t vmula_u32 (uint32x4_t, uint32x4_t, uint32x4_t)

>>> Function description: Vector element multiplication and accumulation Assume that Vz, Vx, Vy are three parameters At the same time Vz is the return value

Vz(i)=Vz(i)+Vx(i)*Vy(index)); the range of index i=0:number-1 is 0~(128/element_size -1)

- int8x16_t vmulai_s8 (int8x16_t, int8x16_t, int8x16_t, const int)

- int16x8_t vmulai_s16 (int16x8_t, int16x8_t, int16x8_t, const int)
- int32x4_t vmulai_s32 (int32x4_t, int32x4_t, int32x4_t, const int)
- uint8x16_t vmulai_u8 (uint8x16_t, uint8x16_t, uint8x16_t, const int)
- uint16x8_t vmulai_u16 (uint16x8_t, uint16x8_t, uint16x8_t, const int)
- uint32x4_t vmulai_u32 (uint32x4_t, uint32x4_t, uint32x4_t, const int)

>>> Function description: vector element multiplication and accumulation

Assume Vz, Vx, Vy, index are 4 parameters ， At the same time Vz is the return value

Vz(i)=Vz(i)+Vx(i)*Vy(index)); i=0:number-1

The range of index is 0~(128/element_size -1)

vmula.te && vmulai.te

- int16x16_t vmula_s8_e (int16x16_t, int8x16_t, int8x16_t)
- int32x8_t vmula_s16_e (int32x8_t, int16x8_t, int16x8_t)
- int64x4_t vmula_s32_e (int64x4_t, int32x4_t, int32x4_t)
- uint16x16_t vmula_u8_e (uint16x16_t, uint8x16_t, uint8x16_t)
- uint32x8_t vmula_u16_e (uint32x8_t, uint16x8_t, uint16x8_t)
- uint64x4_t vmula_u32_e (uint64x4_t, uint32x4_t, uint32x4_t)

>>> Function description: Vector element expansion multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters ， At the same time Vz is the return value

Vz(i)=Vz(i)+(Vx(i)*Vy(i))[2*element_size-1:0]; i=0:(number-1)

The multiplication result takes full precision, that is, twice the element width

- int16x16_t vmulai_s8_e (int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vmulai_s16_e (int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vmulai_s32_e (int64x4_t, int32x4_t, int32x4_t, const int)
- uint16x16_t vmulai_u8_e (uint16x16_t, uint8x16_t, uint8x16_t, const int)
- uint32x8_t vmulai_u16_e (uint32x8_t, uint16x8_t, uint16x8_t, const int)
- uint64x4_t vmulai_u32_e (uint64x4_t, uint32x4_t, uint32x4_t, const int)

>>> Function description: Vector element expansion multiplication and

accumulation Assume that Vz, Vx, Vy, index are 4 parameters， At the same time Vz is the return value

Vz(i)=Vz(i)+(Vx(i)*Vy(index))[2*element_size-1:0]; i=0:(number-1) The multiplication result takes full precision,

that is, twice the element width. The range of index is

0~(128/element_size -1)

vmuls.t && vmulsi.t

- int8x16_t vmuls_s8 (int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vmuls_s16 (int16x8_t, int16x8_t, int16x8_t)

- int32x4_t vmuls_s32 (int32x4_t, int32x4_t, int32x4_t)
- uint8x16_t vmuls_u8 (uint8x16_t, uint8x16_t, uint8x16_t)
- uint16x8_t vmuls_u16 (uint16x8_t, uint16x8_t, uint16x8_t)
- uint32x4_t vmuls_u32 (uint32x4_t, uint32x4_t, uint32x4_t)

>>> Function description: Vector element multiplication and subtraction Assume Vz, Vx, Vy are three parameters and Vz is the return value Vz(i)=Vz(i)-Vx(i)*Vy(i); i=0:number-1

- int8x16_t vmulsi_s8 (int8x16_t __c, int8x16_t __a, int8x16_t __b, const int __index)
- int16x8_t vmulsi_s16 (int16x8_t __c, int16x8_t __a, int16x8_t __b, const int __index)
- int32x4_t vmulsi_s32 (int32x4_t __c, int32x4_t __a, int32x4_t __b, const int __index)
- uint8x16_t vmulsi_u8 (uint8x16_t __c, uint8x16_t __a, uint8x16_t __b, const int __index)
- uint16x8_t vmulsi_u16 (uint16x8_t __c, uint16x8_t __a, uint16x8_t __b, const int __index)
- uint32x4_t vmulsi_u32 (uint32x4_t __c, uint32x4_t __a, uint32x4_t __b, const int __index)

>>> Function description: vector element multiplication and subtraction Assume Vz, Vx, Vy, index are 4 parameters , At the same time Vz is the return value Vz(i)=Vz(i)-Vx(i)*Vy(index)); the range of index i=0:number-1 is 0~(128/element_size -1)

vmuls.te && vmulsi.te

- int16x16_t vmuls_s8_e (int16x16_t, int8x16_t, int8x16_t)
- int32x8_t vmuls_s16_e (int32x8_t, int16x8_t, int16x8_t)
- int64x4_t vmuls_s32_e (int64x4_t, int32x4_t, int32x4_t)
- uint16x16_t vmuls_u8_e (uint16x16_t, uint8x16_t, uint8x16_t)
- uint32x8_t vmuls_u16_e (uint32x8_t, uint16x8_t, uint16x8_t)
- uint64x4_t vmuls_u32_e (uint64x4_t, uint32x4_t, uint32x4_t)

>>> Function description: vector element expansion multiplication and subtraction Assume Vz, Vx, Vy are 3 parameters, Vz is the return value Vz(i)=Vz(i)-(Vx(i)*Vy(i)) [2*element_size-1:0]; i=0:(number-1) The multiplication result takes full precision, that is, twice the element width

- int16x16_t vmulsi_s8_e (int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vmulsi_s16_e (int32x8_t, int16x8_t, int16x8_t, const int)

- int64x4_t vmulsi_s32_e (int64x4_t, int32x4_t, int32x4_t, const int)
- uint16x16_t vmulsi_u8_e (uint16x16_t, uint8x16_t, uint8x16_t, const int)
- uint32x8_t vmulsi_u16_e (uint32x8_t, uint16x8_t, uint16x8_t, const int)
- uint64x4_t vmulsi_u32_e (uint64x4_t, uint32x4_t, uint32x4_t, const int)

>>> Function description: vector element expansion multiplication and subtraction

Assume Vz, Vx, Vy, index are 4 parameters , At the same time Vz is the return value

Vz(2i+1:2i)=(Vz(2i+1:2i)-(Vx(i)*Vy(index))[2*element_size-1:0]; i=0:(number-ÿ1)

The multiplication result takes full precision, that is, twice the element width

The range of index is 0~(128/element_size -1)

vmulaca.t && vmulacai.t

- int32x4_t vmulaca_s8 (int8x16_t, int8x16_t)
- int64x2_t vmulaca_s16 (int16x8_t, int16x8_t)
- uint32x4_t vmulaca_u8 (uint8x16_t, uint8x16_t)
- uint64x2_t vmulaca_u16 (uint16x8_t, uint16x8_t)

>>> Function description: vector chain multiplication and accumulation

Assume Vx and Vy are two parameters, Vz is the return value, and U/S indicates the sign bit.

Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0]; The multiplication result takes full precision, that is, twice the element width i=0:number-1

Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]); extend means extending the accumulated result to the bit width of the destination element according to U/S, which is 4 times the bit width of the source operand element. i=number/4-1

- int32x4_t vmulacai_s8 (int8x16_t, int8x16_t, const int __index)
- int64x2_t vmulacai_s16 (int16x8_t, int16x8_t, const int __index)
- uint32x4_t vmulacai_u8 (uint8x16_t, uint8x16_t, const int __index)
- uint64x2_t vmulacai_u16 (uint16x8_t, uint16x8_t, const int __index)

>>> Function description: vector with index chain multiplication and accumulation

Assume Vx, Vy, index are three parameters, Vz is the return value, and U/S indicates the sign bit.

Tmp(4i)=(Vx(4i)*Vy(4*index))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+1)=(Vx(4i+1)*Vy(4*index+1))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+2)=(Vx(4i+2)*Vy(4*index+2))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+3)=(Vx(4i+3)*Vy(4*index+3))[2*element_size-1:0]; The multiplication result takes full precision, which is twice the element width i=0:number/4-1

Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]); i=number/4-1

extend means to extend the accumulated result to the bit width of the destination element according to U/S, which is 4 times the bit width of the source operand element

vmulacaa.t && vmulacaai.t

- int32x4_t vmulacaa_s8 (int32x4_t, int8x16_t, int8x16_t)
- int64x2_t vmulacaa_s16 (int64x2_t, int16x8_t, int16x8_t)
- uint32x4_t vmulacaa_u8 (uint32x4_t, uint8x16_t, uint8x16_t)
- uint64x2_t vmulacaa_u16 (uint64x2_t, uint16x8_t, uint16x8_t)

>>> Function description: vector chain multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters , At the same time, Vz is the return value, U/S is the sign bit

Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0]; The multiplication i=0:number-1

result takes full precision, that is, twice the element width

Vz(4i+3:4i)= Vz(4i+3:4i)+ extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]);

ÿi=number/4-1

extend means to extend the accumulated result to the bit width of the destination element according to U/S, which is 4 times the bit width of the source operand element

- int32x4_t vmulacaai_s8 (int32x4_t, int8x16_t, int8x16_t, const int)
- int64x2_t vmulacaai_s16 (int64x2_t, int16x8_t, int16x8_t, const int)
- uint32x4_t vmulacaai_u8 (uint32x4_t, uint8x16_t, uint8x16_t, const int)
- uint64x2_t vmulacaai_u16 (uint64x2_t, uint16x8_t, uint16x8_t, const int)

>>> Function description: vector with index chain multiplication and accumulation

Assume Vz, Vx, Vy, index are 4 parameters and Vz is the return value, U/S is the sign bit

Tmp(4i)=(Vx(4i)*Vy(4*index))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+1)=(Vx(4i+1)*Vy(4*index+1))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+2)=(Vx(4i+2)*Vy(4*index+2))[2*element_size-1:0]; i=0:number/4-1

Tmp(4i+3)=(Vx(4i+3)*Vy(4*index+3))[2*element_size-1:0]; i=0:number/4-1

The multiplication result takes full precision, that is, twice the element width

Vz(4i+3:4i)= Vz(4i+3:4i)+ extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]);

ÿi=number/4-1

extend means to extend the accumulated result to the bit width of the destination element according to U/S, which is 4 times the bit width of the source operand element

The range of index is 0 ~ (128/(element_size*4) -1)

vrmul.t.se && vrmuli.t.se

- int16x16_t vrmul_s8_se (int8x16_t, int8x16_t)
- int32x8_t vrmul_s16_se (int16x8_t, int16x8_t)
- int64x4_t vrmul_s32_se (int32x4_t, int32x4_t)

>>> Function Description: Vector extension with saturated fractional multiplication

Assume Vx, Vy are two parameters, Vz is the return value

If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))

Vz(i)= 2^(2*element_size-1)-1;

Else Vz(i)= Vx(i)*Vy(i))*2[2*element_size-1:0];

(The multiplication result takes full precision, which is twice the element width)

End i=0:(number-1)

- int16x16_t vrmuli_s8_se (int8x16_t, int8x16_t, const int)
- int32x8_t vrmuli_s16_se (int16x8_t, int16x8_t, const int)
- int64x4_t vrmuli_s32_se (int32x4_t, int32x4_t, const int)

```
>>> Function description: Vector with index extension with saturated decimal
        multiplication Assume Vx, Vy, index are three parameters, Vz is the return value
        If (Vx(i)== -2^(element_size-1)) && (Vy(index)== -2^(element_size-1))
        Vz(i)= 2^(2*element_size-1)-1;
        Else Vz(i)= Vx(i)*Vy(index))*2[2*element_size-1:0];
         (The multiplication result takes full precision, which is twice the element width)
        End             i=0:(number-1)
        The range of index is 0~(128/element_size -1)
```

vrmulh.ts && vrmulhi.ts

- int8x16_t vrmulh_s8_s (int8x16_t, int8x16_t)
- int16x8_t vrmulh_s16_s (int16x8_t, int16x8_t)
- int32x4_t vrmulh_s32_s (int32x4_t, int32x4_t)

```
>>> Function description: Vector multiplication with saturation and high half decimal
        If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))
        Vz(i)= 2^(element_size-1)-1;
        Else Vz(i)= Vx(i)*Vy(i))*2[2*element_size-1:element_size];
         (The multiplication result is taken as the high bit)
        End             i=0:(number-1)
```

- int8x16_t vrmulhi_s8_s (int8x16_t, int8x16_t, const int)
- int16x8_t vrmulhi_s16_s (int16x8_t, int16x8_t, const int)
- int32x4_t vrmulhi_s32_s (int32x4_t, int32x4_t, const int)

```
>>> Function description: Vector with index and saturation high semi-decimal
        multiplication Assume Vx, Vy, index are 3 parameters, Vz is the return value
        If (Vx(i)== -2^(element_size-1)) && (Vy(index)== -2^(element_size-1))
        Vz(i)= 2^(element_size-1)-1;
        Else Vz(i)= Vx(i)*Vy(index))*2[2*element_size-1:element_size];
         (The multiplication result is taken as the high bit)
        End             i=0:(number-1)
        The range of index is 0~(128/element_size -1)
```

vrmulh.t.rs && vrmulhi.t.rs

- int8x16_t vrmulh_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrmulh_s16_rs(int16x8_t, int16x8_t)

- int32x4_t vrmulh_s32_rs(int32x4_t, int32x4_t)

>>> Function description: Vector multiplication with saturation and high half rounding

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<(element_size-1);

If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))

Vz(i)= 2^(element_size-1)-1;

Else

Vz(i)= (Vx(i)*Vy(i))*2+round)[2*element_size-1:element_size]; (The multiplication result is the high bit)

End i=0:(number-1)

- int8x16_t vrmulhi_s8_rs(int8x16_t, int8x16_t, const int)
- int16x8_t vrmulhi_s16_rs(int16x8_t, int16x8_t, const int)
- int32x4_t vrmulhi_s32_rs(int32x4_t, int32x4_t, const int)

>>> Function description: vector with index and saturation, high half rounding, decimal multiplication

Assume Vx, Vy, index are 3 parameters, Vz is the return value

round=1<<(element_size-1);

If (Vx(i)== -2^(element_size-1)) && (Vy(index)== -2^(element_size-1))

Vz(i)= 2^(element_size-1)-1;

Else Vz(i)= (Vx(i)*Vy(index))*2+round)[2*element_size-1:element_size]; (The multiplication result is taken as the high bit)

End i=0:(number-1)

The range of index is 0~(128/element_size -1)

vrmulha.t.rs && vrmulhai.t.rs

- int8x16_t vrmulha_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrmulha_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrmulha_s32_rs(int32x4_t, int32x4_t)

>>> Function description: Vector with saturation, high half rounding, decimal multiplication and accumulation. Assume that Vx and Vy are two parameters, and Vz is the return value.

round=1<<(element_size-1);

Tmp(i)= (Vz(i)<<element_size)+ Vx(i)*Vy(i)*2+round; i=0:(number-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1

Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (take the high part of the multiplication and accumulation result)

End i=0:(number-1)

(Note: Saturation operation is performed after accumulation)

- int8x16_t vrmulhai_s8_rs(int8x16_t, int8x16_t, const int)
- int16x8_t vrmulhai_s16_rs(int16x8_t, int16x8_t, const int)
- int32x4_t vrmulhai_s32_rs(int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation, high half rounding, decimal multiplication and accumulation. Assume Vx, Vy, index are 3 parameters, Vz is the return value

round=1<<(element_size-1);

Tmp(i)= (Vz(i)<<element_size)+ Vx(i)*Vy(index)*2+round; i=0:(number-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size];

(Take the high part of the multiplication and accumulation result)

End i=0:(number-1)

(Note: Saturation operation is performed after accumulation) The range of index is 0~(128/element_size -1)

vrmulhs.t.rs && vrmulhsi.t.rs

- int8x16_t vrmulhs_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrmulhs_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrmulhs_s32_rs(int32x4_t, int32x4_t)

>>> Function description: vector with saturation, high half rounding, decimal multiplication and subtraction

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<(element_size-1);

Tmp(i)= (Vz(i)<<element_size)-Vx(i)*Vy(i)*2+round; i=0:(number-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication and subtraction result)

End i=0:(number-1)

(Note: Saturation operation is performed after accumulation)

- int8x16_t vrmulhsi_s8_rs(int8x16_t, int8x16_t, const int)
- int16x8_t vrmulhsi_s16_rs(int16x8_t, int16x8_t, const int)
- int32x4_t vrmulhsi_s32_rs(int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation, high half rounding, decimal multiplication and subtraction

Assume Vx, Vy, index are 3 parameters, Vz is the return value

round=1<<(element_size-1);

Tmp(i)= (Vz(i)<<element_size)- Vx(i)*Vy(index)*2+round; i=0:(number-1)

(Continued on next page)

(Continued from previous page)

Tmp(i) retains the full precision of the operation
If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;
Else if Tmp(i)<-2^(2*element_size-1)
Vz(i)= -2^(element_size-1);
Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication and subtraction result)
End i=0:(number-1)
(Note: Saturation operation is performed
after accumulation) The range of index is 0~(128/element_size -1)

vrmulshr.te && vrmulshri.te

- int16x16_t vrmulshr_s8_e(int8x16_t, int8x16_t, const int)
- int32x8_t vrmulshr_s16_e(int16x8_t, int16x8_t, const int)
- int64x4_t vrmulshr_s32_e(int32x4_t, int32x4_t, const int)

>>> Function Description: Vector extension with shifted decimal multiplication
Assume Vx, Vy, imm4 are 3 parameters and Vz is the return value
Vz(i)= (Vx(i)*Vy(i))>>imm4; i=0:(number-1)
The multiplication result retains all precision and then performs arithmetic right shift imm4=0~15

- int16x16_t vrmulshri_s8_e(int8x16_t, const int, const int)
- int32x8_t vrmulshri_s16_e(int16x8_t, const int, const int)
- int64x4_t vrmulshri_s32_e(int32x4_t, const int, const int)

>>> Function description: Vector with index extension and shift decimal
multiplication Assume Vx, imm4, index is 4 is the parameter, Vz is the return value
Vz(i)=(Vx(i)*Vx+1(index))>>imm4; i=0:(number-1) The multiplication result retains
all precision and is arithmetic right shifted imm4=0~15. The range of index
is 0~(128/element_size -1)

vrmulsa.te && vrmulsai.te

- int16x16_t vrmulsa_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrmulsa_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrmulsa_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: Vector expansion with shift fractional multiplication and
accumulation. Assume that Vz, Vx, Vy, and imm are 4 parameters and Vz is the return value.
Vz(i)=Vz(i) + ((Vx(i)*Vy(i))>>imm); i=0:(number-1)
The multiplication result retains full precision and is then arithmetic shifted right , Then add imm=0~15

- int16x16_t vrmulsai_s8_e(int16x16_t, int8x16_t, const int, const int)
- int32x8_t vrmulsai_s16_e(int32x8_t, int16x8_t, const int, const int)

- int64x4_t vrmulsai_s32_e(int64x4_t, int32x4_t, const int, const int)

>>> Function description: vector with index extension and shift decimal multiplication and accumulation

Assume Vz, Vx, imm, index are 4 parameters and,Vz is the return value

Vz(i)=Vz(i) + ((Vx(i)*Vx+1(index))>>imm); i=0:(number-1)

The multiplication result retains full precision and is , Then add imm=0~15 arithmetic shifted right. The index range is 0~(128/element_size -1)

vrmulss.te && vrmulssi.te

- int16x16_t vrmulss_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrmulss_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrmulss_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: Vector extension with shift decimal multiplication and accumulation

Assume Vz, Vx, Vy, imm are 4 parameters , At the same time Vz is the return value

Vz(i)=Vz(i) + ((-Vx(i)*Vy(i))>>imm); i=0:(number-1)

The multiplication result retains all precision, performs arithmetic right shift, and then subtracts imm=0~15

- int16x16_t vrmulssi_s8_e(int16x16_t, int8x16_t, const int, const int)
- int32x8_t vrmulssi_s16_e(int32x8_t, int16x8_t, const int, const int)
- int64x4_t vrmulssi_s32_e(int64x4_t, int32x4_t, const int, const int)

>>> Function description: vector with index extension and shift decimal multiplication and accumulation

Assume Vz, Vx, imm, index are 4 parameters and,Vz is the return value

Vz(i)=Vz(i) + ((-Vx(i)*Vx+1(index))>>imm); i=0:(number-1)

The multiplication result retains full precision and is , Then subtract imm = 0~15 arithmetic shifted right. The index range is 0~(128/element_size -1)

vrmulxaa.t.rs && vrmulxaai.t.rs

- int8x16_t vrmulxaa_s8_rs(int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vrmulxaa_s16_rs(int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vrmulxaa_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function description: vector with saturated complex real and imaginary parts cross-multiply and accumulate, and round off the high half

Assume Vz, Vx, Vy are parameters , At the same time Vz is the return value

round=1<<(element_size-1);

Tmp(2i+1)=Vz(2i+1)<<element_size+Vx(2i)*Vy(2i+1)*2+round; i=0:(number/2-1)

Tmp(2i)=Vz(2i)<<element_size+ Vx(2i)*Vy(2i)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

(Continued on next page)

(Continued from previous page)

Vz(i)= -2^(element_size-1);
Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication and accumulation result)
End i=0:(number-1)
(Note: Saturation operation is performed after accumulation)

- int8x16_t vrmulxaai_s8_rs(int8x16_t, int8x16_t, int8x16_t, const int)
- int16x8_t vrmulxaai_s16_rs(int16x8_t, int16x8_t, int16x8_t, const int)
- int32x4_t vrmulxaai_s32_rs(int32x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation complex number real part imaginary part cross multiplication accumulation accumulation take high half rounding assuming Vz, Vx, Vy, index are 4 parameters and Vz is the return value
round=1<<(element_size-1);
Tmp(2i+1)=Vz(2i+1)<<element_size+Vx(2i)*Vy(2index+1)*2+round; i=0:(number/2-ÿ1)
Tmp(2i)=Vz(2i)<<element_size+ Vx(2i)*Vy(2index)*2+round; i=0:(number/2-1)
Tmp(i) retains the full precision of the operation
If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;
Else if Tmp(i)<-2^(2*element_size-1)
Vz(i)= -2^(element_size-1);
Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication and accumulation result)
End i=0:(number-1)
(Note: Saturation operation is performed after accumulation) The range of index is 0 ~ (128/(element_size*2) -1)

vrmulxas.t.rs && vrmulxas.t.rs

- int8x16_t vrmulxas_s8_rs(int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vrmulxas_s16_rs(int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vrmulxas_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function description: vector with saturated complex real and imaginary parts, cross-multiply, accumulate, subtract, round off, assume Vz, Vx, Vy are three parameters, and Vz is the return value
round=1<<(element_size-1);
Tmp(2i+1)=Vz(2i+1)<<element_size+Vx(2i+1)*Vy(2i)*2+round; i=0:(number/2-1)
Tmp(2i)=Vz(2i)<<element_size-Vx(2i+1)*Vy(2i+1)*2+round; i=0:(number/2-1)
Tmp(i) retains the full precision of the operation
If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;
Else if Tmp(i)<-2^(2*element_size-1)
Vz(i)= -2^(element_size-1);
Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the result of multiplication and accumulation)
End i=0:(number-1)
(Note: Saturation operation is performed after accumulation and subtraction)

- int8x16_t vrmulxasi_s8_rs(int8x16_t, int8x16_t, int8x16_t, const int)

- int16x8_t vrmulxasi_s16_rs(int16x8_t, int16x8_t, int16x8_t, const int)
- int32x4_t vrmulxasi_s32_rs(int32x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation complex number real part imaginary part cross multiplication accumulation accumulation subtraction high half rounding assuming Vz, Vx, Vy, index are 4 parameters and Vz is the return value

round=1<<(element_size-1);

Tmp(2i+1)=Vz(2i+1)<<element_size+Vx(2i+1)*Vy(2index)*2+round; i=0:(number/2-ÿ1)

Tmp(2i)=Vz(2i)<<element_size-Vx(2i+1)*Vy(2index+1)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the result of multiplication and accumulation)

End i=0:(number-1)

(Note: Saturation operation is performed after accumulation and subtraction) The range of index is 0 ~ (128/(element_size*2) -1)

vrmulxss.t.rs && vrmulxssi.t.rs

- int8x16_t vrmulxss_s8_rs(int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vrmulxss_s16_rs(int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vrmulxss_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function description: vector with saturated complex real and imaginary parts cross-multiply and subtract, and round off the high half. Assume Vz, Vx, Vy are three parameters , Vz is the return value.

round=1<<(element_size-1);

Tmp(2i+1)=Vz(2i+1)<<element_size-Vx(2i)*Vy(2i+1)*2+round; i=0:(number/2-1)

Tmp(2i)=Vz(2i)<<element_size-Vx(2i)*Vy(2i)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the result of multiplication and accumulation)

End i=0:(number-1)

(Note: Saturation operation is performed after accumulation and subtraction)

- int8x16_t vrmulxssi_s8_rs(int8x16_t, int8x16_t, int8x16_t, const int)
- int16x8_t vrmulxssi_s16_rs(int16x8_t, int16x8_t, int16x8_t, const int)
- int32x4_t vrmulxssi_s32_rs(int32x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation complex number real part imaginary part cross multiplication cumulative subtraction cumulative subtraction high half rounding assuming Vz, Vx, Vy, index are 4 parameters and Vz is the return value

round=1<<(element_size-1);

(Continued on next page)

(Continued from previous page)

Tmp(2i+1)=Vz(2i+1)<<element_size-Vx(2i)*Vy(2index+1)*2+round; i=0:(number/2-ÿ1)

Tmp(2i)=Vz(2i)<<element_size-Vx(2i)*Vy(2index)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the result of multiplication and accumulation)

End i=0:(number-1)

(Note: Saturation operation is performed after

accumulation and subtraction) The range of index is 0 ~ (128/(element_size*2) -1)

vrmulxsa.t.rs && vrmulxsai.t.rs

- int8x16_t vrmulxsa_s8_rs(int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vrmulxsa_s16_rs(int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vrmulxsa_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function description: vector with saturated complex real and imaginary parts cross-multiply, subtract, add, and round off to the

Assume Vz, Vx, Vy are 3 parameters , highest half. Vz is the return value.

round=1<<(element_size-1);

Tmp(2i+1)=Vz(2i+1)<<element_size-Vx(2i+1)*Vy(2i)*2+round; i=0:(number/2-1)

Tmp(2i)=Vz(2i)<<element_size+Vx(2i+1)*Vy(2i+1)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

- int8x16_t vrmulxsai_s8_rs(int8x16_t, int8x16_t, int8x16_t, const int)
- int16x8_t vrmulxsai_s16_rs(int16x8_t, int16x8_t, int16x8_t, const int)
- int32x4_t vrmulxsai_s32_rs(int32x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: vector with index with saturation complex number real part imaginary part cross multiplication accumulation subtraction accumulation

take high half rounding assuming Vz, Vx, Vy, index are 4 parameters and Vz is the return value

round=1<<(element_size-1);

Tmp(2i+1)=Vz(2i+1)<<element_size-Vx(2i+1)*Vy(2index)*2+round; i=0:(number/2-ÿ1)

Tmp(2i)=Vz(2i)<<element_size+Vx(2i+1)*Vy(2index+1)*2+round; i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the result of multiplication and accumulation)

End i=0:(number-1)

(Note: Saturation operation is performed after

accumulation and subtraction) The range of index is 0 ~ (128/(element_size*2) -1)

vrcmul.t.rs

- int8x16_t vrcmul_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrcmul_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrcmul_s32_rs(int32x4_t, int32x4_t)

>>> Function description: complex number multiplication

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<element_size-1

Tmp(2i+1)=Vx(2i)*Vy(2i+1)*2+Vx(2i+1)*Vy(2i)*2+round;

i=0:(number/2-1)

Tmp(2i)=Vx(2i)*Vy(2i)*2-Vx(2i+1)*Vy(2i+1)*2+round;

i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication/addition/subtraction result)

End i=0:(number-1)

(Note: Saturation operation is performed after addition and subtraction)

vrcmula.te

- int16x16_t vrcmula_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrcmula_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrcmula_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: Complex multiplication right shift expansion

accumulation Assume Vz, Vx, Vy, imm are 4 parameters, At the same time Vz is the return value

Vz(4i+3:4i+2)=Vz(4i+3:4i+2) + ((Vx(2i)*Vy(2i+1))>>imm) + ((Vx(2i+1)*Vy(2i))>>
ÿimm);

i=0:(number/2-1) (imaginary part)

Vz(4i+1:4i)=Vz(4i+1:4i) + ((Vx(2i)*Vy(2i))>>imm) + ((-Vx(2i+1)*Vy(2i+1))>>
ÿimm);

i=0:(number/2-1) (real part) The result of

complex multiplication retains all precision and is then arithmetic shifted right Then add imm=0~15

vrcmulc.t.rs

- int8x16_t vrcmulc_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrcmulc_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrcmulc_s32_rs(int32x4_t, int32x4_t)

>>> Function description: Complex conjugate multiplication conj(x)*y

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<element_size-1

Tmp(2i+1)=Vx(2i)*Vy(2i+1)*2-Vx(2i+1)*Vy(2i)*2+round;

i=0:(number/2-1)

Tmp(2i)=Vx(2i)*Vy(2i)*2+Vx(2i+1)*Vy(2i+1)*2+round;

i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication/addition/subtraction result)

End i=0:(number-1)

(Note: Saturation operation is performed after addition and subtraction)

vrcmulca.te

- int16x16_t vrcmulca_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrcmulca_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrcmulca_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: complex conjugate multiplication right shift expansion

Assume Vz, Vx, Vy, imm are 4 parameters ， accumulation and Vz is the return value

Vz(4i+3:4i+2)=Vz(4i+3:4i+2) + ((Vx(2i)*Vy(2i+1))>>imm4) + ((-Vx(2i+1)*Vy(2i))>

ÿ>imm4);

i=0:(number/2-1) (imaginary part)

Vz(4i+1:4i)=Vz(4i+1:4i) + ((Vx(2i)*Vy(2i))>>imm4) + ((Vx(2i+1)*Vy(2i+1))>>

ÿimm4);

i=0:(number/2-1) (real part) The result of

complex multiplication retains all precision and is then arithmetic shifted right Then add imm=0~15

vrcmuln.t.rs

- int8x16_t vrcmuln_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrcmuln_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrcmuln_s32_rs(int32x4_t, int32x4_t)

>>> Function description: Complex number negative multiplication (-x)*y

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<element_size-1

Tmp(2i+1)= -Vx(2i)*Vy(2i+1)*2-Vx(2i+1)*Vy(2i)*2+round;

i=0:(number/2-1)

Tmp(2i)=-Vx(2i)*Vy(2i)*2+Vx(2i+1)*Vy(2i+1)*2+round;

(Continued on next page)

(Continued from previous page)

i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

Else Vz(i)=Tmp(i)[2*element_size-1:element_size]; (Take the high part of the multiplication/addition/subtraction result)

End i=0:(number-1)

(Note: Saturation operation is performed after addition and subtraction)

vrcmulna.te

- int16x16_t vrcmulna_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrcmulna_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrcmulna_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

>>> Function description: complex number negative multiplication right shift expansion accumulation

Assume Vz, Vx, Vy, imm4 are 4 parameters , At the same time Vz is the return value

Vz(4i+3:4i+2)=Vz(4i+3:4i+2) + ((-Vx(2i)*Vy(2i+1))>>imm4) + ((-

ÿVx(2i+1)*Vy(2i))>>imm4);

i=0:(number/2-1) (imaginary part)

Vz(4i+1:4i)=Vz(4i+1:4i) + ((-Vx(2i)*Vy(2i)>>imm4) +((Vx(2i+1)*Vy(2i+1))>>

ÿimm4);

i=0:(number/2-1) (real part) The result of

complex multiplication retains all precision and is arithmetic right shifted Then add imm=0~15

vrcmulcn.t.rs

- int8x16_t vrcmulcn_s8_rs(int8x16_t, int8x16_t)
- int16x8_t vrcmulcn_s16_rs(int16x8_t, int16x8_t)
- int32x4_t vrcmulcn_s32_rs(int32x4_t, int32x4_t)

>>> Function description: Complex conjugate negative multiplication (-conj(x)*y

Assume Vx, Vy are two parameters, Vz is the return value

round=1<<element_size-1

Tmp(2i+1)= -Vx(2i)*Vy(2i+1)*2+Vx(2i+1)*Vy(2i)*2+round;

i=0:(number/2-1)

Tmp(2i)= -Vx(2i)*Vy(2i)*2-Vx(2i+1)*Vy(2i+1)*2+round;

i=0:(number/2-1)

Tmp(i) retains the full precision of the operation

If Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;

Else if Tmp(i)<-2^(2*element_size-1)

Vz(i)= -2^(element_size-1);

(Continued on next page)

(Continued from previous page)

```
    Else Vz(i)=Tmp(i)[2*element_size-1:element_size];   (Take the high part of the multiplication/addition/subtraction result)
    End i=0:(number-1)
    (Note: Saturation operation is performed after addition and subtraction)
```

vrcmulcna.te

- int16x16_t vrcmulcna_s8_e(int16x16_t, int8x16_t, int8x16_t, const int)
- int32x8_t vrcmulcna_s16_e(int32x8_t, int16x8_t, int16x8_t, const int)
- int64x4_t vrcmulcna_s32_e(int64x4_t, int32x4_t, int32x4_t, const int)

```
>>> Function description: complex conjugate negative multiplication right shift expansion accumulation
    Assume Vz, Vx, Vy, imm4 are 4 parameters, and Vz is the return value
    Vz(4i+3:4i+2)=Vz(4i+3:4i+2) + ((-Vx(2i)*Vy(2i+1))>>imm4) + ((Vx(2i+1)*Vy(2i))>
ÿ>imm4);
    i=0:(number/2-1) (imaginary part)
    Vz(4i+1:4i)=Vz(4i+1:4i) + ((-Vx(2i)*Vy(2i))>>imm4)+ ((-Vx(2i+1)*Vy(2i+1))>>
ÿimm4);
    i=0:(number/2-1) (real part) The result of
    complex multiplication retains all precision and is arithmetic right shifted Then add imm=0~15
```

4.5.6.3 Integer reciprocal, reciprocal square root, e exponential fast calculation and approximation instructions

vrecpe.t && vrecps.t

- sat8x16_t vrecpe_s8(sat8x16_t)
- sat16x8_t vrecpe_s16(sat16x8_t)
- sat32x4_t vrecpe_s32(sat32x4_t)
- usat8x16_t vrecpe_u8(usat8x16_t)
- usat16x8_t vrecpe_u16(usat16x8_t)
- usat32x4_t vrecpe_u32(usat32x4_t)

```
>>> Function description: Take the reciprocal of a vector element.
    Assume Vx is the parameter and Vz is the return value.
    Vz(i) ÿ 1/(Vx(i)) i=0:(number-1)
    (Quickly calculate the reciprocal value of Vx(i))
```

- sat8x16_t vrecps_s8(sat8x16_t, sat8x16_t)
- sat16x8_t vrecps_s16(sat16x8_t, sat16x8_t)
- sat32x4_t vrecps_s32(sat32x4_t, sat32x4_t)
- usat8x16_t vrecps_u8(usat8x16_t, usat8x16_t)
- usat16x8_t vrecps_u16(usat16x8_t, usat16x8_t)

- usat32x4_t vrecps_u32(usat32x4_t, usat32x4_t)

>>> Function description: Vector reciprocal approximation assumes Vx, Vy are 2, Vz is the return value

Vz(i) = 2 – Vx(i) * Vy(i) i=0:(number-1)

vrsqrte.t && vrsqrts.t

- sat8x16_t vrsqrte_s8(sat8x16_t)
- sat16x8_t vrsqrte_s16(sat16x8_t)
- sat32x4_t vrsqrte_s32(sat32x4_t)
- usat8x16_t vrsqrte_u8(usat8x16_t)
- usat16x8_t vrsqrte_u16(usat16x8_t)
- usat32x4_t vrsqrte_u32(usat32x4_t)

>>> Function description: Calculate the square root of the vector element after reciprocal. Assume that Vx is the parameter and Vz is the return value.

Vz(i) ÿ i=0:(number-1) (Quickly calculate the reciprocal square root of Vx(i))

- sat8x16_t vrsqrts_s8(sat8x16_t, sat8x16_t)
- sat16x8_t vrsqrts_s16(sat16x8_t, sat16x8_t)
- sat32x4_t vrsqrts_s32(sat32x4_t, sat32x4_t)
- usat8x16_t vrsqrts_u8(usat8x16_t, usat8x16_t)
- usat16x8_t vrsqrts_u16(usat16x8_t, usat16x8_t)
- usat32x4_t vrsqrts_u32(usat32x4_t, usat32x4_t)

>>> Function description:

Assume Vx and Vy are two parameters, Vz is the return value

Vz(i) = 1.5+ ((-Vx(i)*Vy(i))/2) i=0:(number-1);

vexpe.t

- sat8x16_t vexpe_s8(sat8x16_t)
- sat16x8_t vexpe_s16(sat16x8_t)
- sat32x4_t vexpe_s32(sat32x4_t)
- usat8x16_t vexpe_u8(usat8x16_t)
- usat16x8_t vexpe_u16(usat16x8_t)
- usat32x4_t vexpe_u32(usat32x4_t)

>>> Function description: vector element takes the exponential value of e. Assume Vx is the input and Vz is the return value.

Vz(i) ÿ e^(Vx(i)) i=0:(number-1)

(Quickly calculate the e-index value of Vx(i))

4.5.6.4 Integer shift instructions

vsht.t

- int8x16_t vsht_s8 (int8x16_t, int8x16_t)
- int16x8_t vsht_s16 (int16x8_t, int16x8_t)
- int32x4_t vsht_s32 (int32x4_t, int32x4_t)
- int64x2_t vsht_s64 (int64x2_t, int64x2_t)
- uint8x16_t vsht_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vsht_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vsht_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vsht_u64 (uint64x2_t, uint64x2_t)

>>> Function description: vector left shift

Assume Vx, Vy are parameters, Vz is the return value, and U/S is the sign bit

if Vy(i)[7:0]>0, Vz(i)=Vx(i)<<Vy(i)[7:0];

else Vz(i)=Vx(i)>>|Vy(i)[7:0]|; i=0:(number-1) Use the lower 8-bit data Vy(i)[7:0]

in each element Vy(i) of Vy as the signed shift index; For U, right shift is logical right shift

, For S, the right shift is an arithmetic right shift;

vsht.ts

- int8x16_t vsht_s8_s (int8x16_t, int8x16_t)
- int16x8_t vsht_s16_s (int16x8_t, int16x8_t)
- int32x4_t vsht_s32_s (int32x4_t, int32x4_t)
- int64x2_t vsht_s64_s (int64x2_t, int64x2_t)
- uint8x16_t vsht_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vsht_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vsht_u32_s (uint32x4_t, uint32x4_t)
- uint64x2_t vsht_u64_s (uint64x2_t, uint64x2_t)

>>> Function description: Vector saturation left

shift Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

if Vy(i)[7:0]>0, Vz(i)=sat(Vx(i)<<Vy(i)[7:0]);

else Vz(i)=Vx(i)>>|Vy(i)[7:0]|;

i=0:(number-1) The lower 8-bit data Vy(i)[7:0] in each element Vy(i) is used as the signed shift index; sat determines whether the left shift result overflows according to the bit width of the write-back element. , And according to U/

ÿS saturates the overflow result to the corresponding maximum or minimum value;

For U, right shift is logical right shift, and for S, right shift is arithmetic right shift

vsht.tr

- int8x16_t vsht_s8_r (int8x16_t, int8x16_t)
- int16x8_t vsht_s16_r (int16x8_t, int16x8_t)
- int32x4_t vsht_s32_r (int32x4_t, int32x4_t)
- int64x2_t vsht_s64_r (int64x2_t, int64x2_t)
- uint8x16_t vsht_u8_r (uint8x16_t, uint8x16_t)
- uint16x8_t vsht_u16_r (uint16x8_t, uint16x8_t)
- uint32x4_t vsht_u32_r (uint32x4_t, uint32x4_t)
- uint64x2_t vsht_u64_r (uint64x2_t, uint64x2_t)

>>> Function description: vector round right

shift Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

If Vy(i)[7:0]==0, round=0;

else round=1<<(-Vy(i) [7:0]-1);

end

if Vy(i)[7:0]>0, Vz(i)=Vx(i)<<Vy(i)[7:0];

else Vz(i)=(Vx(i)+round)>>|Vy(i)[7:0]|; i=0:(number-1)

The lower 8-bit data Vy(i)[7:0] in each element Vy(i) is used as the signed shift index; for U, right shift is a logical right shift， For S, the right shift is an arithmetic right shift;

vsht.t.rs

- int8x16_t vsht_s8_rs (int8x16_t, int8x16_t)
- int16x8_t vsht_s16_rs (int16x8_t, int16x8_t)
- int32x4_t vsht_s32_rs (int32x4_t, int32x4_t)
- int64x2_t vsht_s64_rs (int64x2_t, int64x2_t)
- uint8x16_t vsht_u8_rs (uint8x16_t, uint8x16_t)
- uint16x8_t vsht_u16_rs (uint16x8_t, uint16x8_t)
- uint32x4_t vsht_u32_rs (uint32x4_t, uint32x4_t)
- uint64x2_t vsht_u64_rs (uint64x2_t, uint64x2_t)

>>> Function description: vector saturation round

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

If Vy(i)[7:0]==0, round=0; else round=1<<(-

Vy(i) [7:0]-1);

end

if Vy(i)[7:0]>0, Vz(i)=sat(Vx(i)<<Vy(i)[7:0]);

else Vz(i)=(Vx(i)+round)>>|Vy(i)[7:0]|; i=0:(number-1) Use the lower 8-bit data Vy(i)[7:0] in each element of Vy as the signed shift index; sat determines whether the left shift result overflows according to the bit width of the write-back element and returns the result accǫrding to U/

ÿS saturates the overflow result to the corresponding maximum or minimum

value. For U, right shift is a logical right shift.,For S, right shift is an arithmetic right shift.

vshl.t && vshli.t

- int8x16_t vshl_s8 (int8x16_t, int8x16_t)
- int16x8_t vshl_s16 (int16x8_t, int16x8_t)
- int32x4_t vshl_s32 (int32x4_t, int32x4_t)
- int64x2_t vshl_s64 (int64x2_t, int64x2_t)
- uint8x16_t vshl_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vshl_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vshl_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vshl_u64 (uint64x2_t, uint64x2_t)

>>> Function Description: Vector register left shift

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=Vx(i)<<Vy(i)[7:0]; i=0:(number-1) Use the lower 8-bit data Vy(i)

[7:0] in each element Vy(i) of Vy as the unsigned shift index;

- int8x16_t vshli_s8 (int8x16_t, const int)
- int16x8_t vshli_s16 (int16x8_t, const int)
- int32x4_t vshli_s32 (int32x4_t, const int)
- int64x2_t vshli_s64 (int64x2_t, const int)
- uint8x16_t vshli_u8 (uint8x16_t, const int)
- uint16x8_t vshli_u16 (uint16x8_t, const int)
- uint32x4_t vshli_u32 (uint32x4_t, const int)
- uint64x2_t vshli_u64 (uint64x2_t, const int)

>>> Function description: Vector immediate left shift

Assume Vx, imm are two parameters, Vz is the return value

Vz(i)=Vx(i)<<imm; i=0:(number-1) The range of imm is

0 ~ element_size-1

vshl.ts && vshli.ts

- int8x16_t vshl_s8_s (int8x16_t, int8x16_t)
- int16x8_t vshl_s16_s (int16x8_t, int16x8_t)
- int32x4_t vshl_s32_s (int32x4_t, int32x4_t)
- int64x2_t vshl_s64_s (int64x2_t, int64x2_t)
- uint8x16_t vshl_u8_s (uint8x16_t, uint8x16_t)
- uint16x8_t vshl_u16_s (uint16x8_t, uint16x8_t)
- uint32x4_t vshl_u32_s (uint32x4_t, uint32x4_t)
- uint64x2_t vshl_u64_s (uint64x2_t, uint64x2_t)

>>> Function Description: Shift the vector register left to saturate

Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit signed=(T==S); (select according to the element U/S type)
Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
Min=signed? -2^(element_size-1):0;
If (Vx(i)<<Vy(i) [7:0])>Max Vz(i)=Max;
Else if (Vx(i)<<Vy(i)[7:0])<Min Vz(i)=Min;
Else Vz(i)= Vx(i)<<Vy(i) [7:0]; i=0:(number-1)
Use the lower 8-bit data Vy(i)[7:0] in each element Vy(i) of Vy as the unsigned shift index;

- int8x16_t vshli_s8_s (int8x16_t, const int)
- int16x8_t vshli_s16_s (int16x8_t, const int)
- int32x4_t vshli_s32_s (int32x4_t, const int)
- int64x2_t vshli_s64_s (int64x2_t, const int)
- uint8x16_t vshli_u8_s (uint8x16_t, const int)
- uint16x8_t vshli_u16_s (uint16x8_t, const int)
- uint32x4_t vshli_u32_s (uint32x4_t, const int)
- uint64x2_t vshli_u64_s (uint64x2_t, const int)

>>> Function description: vector immediate left shift saturation

Assume Vx, imm are two parameters, Vz is the return value signed=(T==S); (select according to the element U/S type)
Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
Min=signed? -2^(element_size-1):0;
If (Vx(i)<<imm)>Max Vz(i)=Max;
Else if (Vx(i)<<imm)<Min Vz(i)=Min;
Else Vz(i)= Vx(i)<<imm; i=0:(number-1)
The range of imm is 0 ~ element_size-1

vshli.te

- int16x16_t vshli_s8_e (int8x16_t, const int)
- int32x8_t vshli_s16_e (int16x8_t, const int)
- int64x4_t vshli_s32_e (int32x4_t, const int)
- uint16x16_t vshli_u8_e (uint8x16_t, const int)
- uint32x8_t vshli_u16_e (uint16x8_t, const int)
- uint64x4_t vshli_u32_e (uint32x4_t, const int)

>>> Function Description: Vector extended immediate left shift

Assume Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit
Vz(2i+1,2i)=extend(Vx(i))<<imm; i=0:(number-1) extend extends the element to twice the original bit width according to U/S. The range of imm is 0 ~ element_size*2-1

vshr.t && vshri.t

- int8x16_t vshr_s8 (int8x16_t, int8x16_t)
- int16x8_t vshr_s16 (int16x8_t, int16x8_t)
- int32x4_t vshr_s32 (int32x4_t, int32x4_t)
- int64x2_t vshr_s64 (int64x2_t, int64x2_t)
- uint8x16_t vshr_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vshr_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vshr_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vshr_u64 (uint64x2_t, uint64x2_t)

>>> Function Description: Vector register right shift

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=Vx(i)>>Vy(i)[7:0] ; i=0:(number-1)

The lower 8-bit data Vy(i)[7:0] in each element Vy(i) of Vy is used as the unsigned shift index; for U, right shift is a logical right shift, For S, right shift is arithmetic right shift

- int8x16_t vshri_s8 (int8x16_t, const int)
- int16x8_t vshri_s16 (int16x8_t, const int)
- int32x4_t vshri_s32 (int32x4_t, const int)
- int64x2_t vshri_s64 (int64x2_t, const int)
- uint8x16_t vshri_u8 (uint8x16_t, const int)
- uint16x8_t vshri_u16 (uint16x8_t, const int)
- uint32x4_t vshri_u32 (uint32x4_t, const int)
- uint64x2_t vshri_u64 (uint64x2_t, const int)

>>> Function description: vector immediate right shift

Assume Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit

Vz(i)=Vx(i)>>imm; i=0:(number-1)

For U, right shift is logical right shift, For S, right shift is arithmetic right shift. The range of imm is 1 ~ element_size.

vshr.tr && vshri.tr

- int8x16_t vshr_s8_r (int8x16_t, int8x16_t)
- int16x8_t vshr_s16_r (int16x8_t, int16x8_t)
- int32x4_t vshr_s32_r (int32x4_t, int32x4_t)
- int64x2_t vshr_s64_r (int64x2_t, int64x2_t)
- uint8x16_t vshr_u8_r (uint8x16_t, uint8x16_t)
- uint16x8_t vshr_u16_r (uint16x8_t, uint16x8_t)
- uint32x4_t vshr_u32_r (uint32x4_t, uint32x4_t)

- uint64x2_t vshr_u64_r (uint64x2_t, uint64x2_t)

>>> Function description: vector register right shift to round
Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit
If Vy(i)[7:0]==0, round =0;
else round=1<<(Vy(i)[7:0]-1);
Vz(i)=(Vx(i)+round)>>Vy(i)[7:0];
i=0:(number-1)
The lower 8-bit data Vy(i)[7:0] in each element Vy(i) is used as the unsigned shift index; for U, the right shift is a logical right shift For S, right shift is arithmetic right shift

- int8x16_t vshri_s8_r (int8x16_t, const int)
- int16x8_t vshri_s16_r (int16x8_t, const int)
- int32x4_t vshri_s32_r (int32x4_t, const int)
- int64x2_t vshri_s64_r (int64x2_t, const int)
- uint8x16_t vshri_u8_r (uint8x16_t, const int)
- uint16x8_t vshri_u16_r (uint16x8_t, const int)
- uint32x4_t vshri_u32_r (uint32x4_t, const int)
- uint64x2_t vshri_u64_r (uint64x2_t, const int)

>>> Function description: vector immediate left shift round
Assume Vx, imm are two parameters, Vz is the return value, U/S is the sign bit
round=1<<(imm-1);Vz(i)=(Vx(i)+round)>> imm); For U, right shift is logical i=0:(number-1)
right shift, the range of imm is 1 , For S, right shift is arithmetic right shift
~ element_size

vshri.tl

- int16x8_t vshri_s16_l (int16x8_t, const int)
- int32x4_t vshri_s32_l (int32x4_t, const int)
- int64x2_t vshri_s64_l (int64x2_t, const int)
- uint16x8_t vshri_u16_l (uint16x8_t, const int)
- uint32x4_t vshri_u32_l (uint32x4_t, const int)
- uint64x2_t vshri_u64_l (uint64x2_t, const int)

>>> Function Description: Vector immediate value is shifted right to get the lower half
Assume Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit
Tmp(i)=(Vx(i)>> imm)[element_size/2-1:0]; i=0:(number-1) (take the lower half of the result)
Vz(i)={Tmp(2i+1),Tmp(2i)}; i=0:(number/2-1)
(The result is placed in the lower 64
bits of Vz) For U, right shift is a logical right shift. For S, right shift is an arithmetic
right shift. The range of imm is 1 ~ element_size

vshri.t.lr

- int16x8_t vshri_s16_lr (int16x8_t, const int)
- int32x4_t vshri_s32_lr (int32x4_t, const int)
- int64x2_t vshri_s64_lr (int64x2_t, const int)
- uint16x8_t vshri_u16_lr (uint16x8_t, const int)
- uint32x4_t vshri_u32_lr (uint32x4_t, const int)
- uint64x2_t vshri_u64_lr (uint64x2_t, const int)

>>> Function description: vector immediate value right shift round to get the lower half

Assume Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit

round=1<<(imm);

Tmp(i)=((Vx(i)+round)>> (imm+1))[element_size/2-1:0]; i=0:(number-1) (take the lower half of the result)

Vz(i)={Tmp(2i+1),Tmp(2i)}; i=0:(number/2-1)

(The result is placed in the lower 64 bits

of Vz) For U, the right shift is a logical right shift. For S, the right shift is an arithmetic

right shift. The range of oimm is 1 ~ element_size

vshri.t.ls

- int16x8_t vshri_s16_ls (int16x8_t, const int)
- int32x4_t vshri_s32_ls (int32x4_t, const int)
- int64x2_t vshri_s64_ls (int64x2_t, const int)
- uint16x8_t vshri_u16_ls (uint16x8_t, const int)
- uint32x4_t vshri_u32_ls (uint32x4_t, const int)
- uint64x2_t vshri_u64_ls (uint64x2_t, const int)

>>> Function description: Vector immediate right shift to get low half

saturation Assume Vx,imm are two parameters, Vz is the return value, U/S is the

sign bit signed=(T==S); (select according to the element U/S type)

Max=signed? 2^(element_size/2-1)-1: 2^(element_size/2)-1; Min=signed? -2^(element_size/

2-1): 0;

If (Vx(i)>> imm)>Max Tmp(i)=Max;

Else if (Vx(i)>> imm)<Min Tmp(i)=Min;

Else Tmp(i)=(Vx(i)>> imm)[element_size/2-1:0]; (Take the lower half of the result)

End i=0:(number-1)

Vz(i)={Tmp(2i+1),Tmp(2i)}; i=0:(number/2-1) (the result is placed in the lower 64 bits of Vz) For U,

right shift is logical right shift , For S, the right shift is an arithmetic

right shift, the range of imm is 1 ~ element_size

vshri.t.lrs

- int16x8_t vshri_s16_lrs (int16x8_t, const int)
- int32x4_t vshri_s32_lrs (int32x4_t, const int)
- int64x2_t vshri_s64_lrs (int64x2_t, const int)
- uint16x8_t vshri_u16_lrs (uint16x8_t, const int)
- uint32x4_t vshri_u32_lrs (uint32x4_t, const int)
- uint64x2_t vshri_u64_lrs (uint64x2_t, const int)

>>> Function description: vector immediate right shift round to get low half saturation

Assume Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit

round=1<<(oimm-1);signed=(T==S); (select according to element U/S type)

Max=signed? 2^(element_size/2-1)-1: 2^(element_size/2)-1;

Min=signed? -2^(element_size/2-1): 0;

If ((Vx(i)+round)>> oimm)>Max Tmp(i)=Max;

Else if ((Vx(i)+round)>> oimm)<Min Tmp(i)=Min;

Else Tmp(i)=((Vx(i)+round)>> oimm)[element_size/2-1:0]; (Take the lower half of the result)

End i=0:(number-1)

Vz(i)={Tmp(2i+1),Tmp(2i)}; i=0:(number/2-1) (the result is placed in the lower 64 bits of Vz)

For U, right shift is a logical , For S, right shift is arithmetic right shift

right shift. The range of oimm is 1 ~ element_size

vshria.t

- int8x16_t vshria_s8 (int8x16_t, int8x16_t, const int)
- int16x8_t vshria_s16 (int16x8_t, int16x8_t, const int)
- int32x4_t vshria_s32 (int32x4_t, int32x4_t, const int)
- int64x2_t vshria_s64 (int64x2_t, int64x2_t, const int)
- uint8x16_t vshria_u8 (uint8x16_t, uint8x16_t, const int)
- uint16x8_t vshria_u16 (uint16x8_t, uint16x8_t, const int)
- uint32x4_t vshria_u32 (uint32x4_t, uint32x4_t, const int)
- uint64x2_t vshria_u64 (uint64x2_t, uint64x2_t, const int)

>>> Function description: vector immediate right shift accumulation

Assume Vz, Vx, imm are 3 parameters , At the same time, Vz is the return value and U/S is the sign bit

Vz(i)=Vz(i)+(Vx(i)>> imm); i=0:(number-1)

For U, right shift is logical right shift. For S, right shift is arithmetic right shift.

The range of imm is 1 ~ element_size

vshria.tr

- int8x16_t vshria_s8_r (int8x16_t, int8x16_t, const int)

- int16x8_t vshria_s16_r (int16x8_t, int16x8_t, const int)
- int32x4_t vshria_s32_r (int32x4_t, int32x4_t, const int)
- int64x2_t vshria_s64_r (int64x2_t, int64x2_t, const int)
- uint8x16_t vshria_u8_r (uint8x16_t, uint8x16_t, const int)
- uint16x8_t vshria_u16_r (uint16x8_t, uint16x8_t, const int)
- uint32x4_t vshria_u32_r (uint32x4_t, uint32x4_t, const int)
- uint64x2_t vshria_u64_r (uint64x2_t, uint64x2_t, const int)

>>> Function description:

Assume Vz, Vx, imm are 3 parameters , At the same time, Vz is the return value and U/S is the sign bit

round=1<<(imm-1);Vz(i)=Vz(i)+((Vx(i)+round)>> imm) ; i=0:(number-1) For U, right shift is logical right shift For S, right shift is arithmetic right shift Tḥe range of imm is 1 ~ element_size

vexh.t && vexl.t

- int8x16_t vexh_s8 (int8x16_t, int8x16_t, int)
- int16x8_t vexh_s16 (int16x8_t, int16x8_t, int)
- int32x4_t vexh_s32 (int32x4_t, int32x4_t, int)
- int64x2_t vexh_s64 (int64x2_t, int64x2_t, int)
- uint8x16_t vexh_u8 (uint8x16_t, uint8x16_t, unsigned)
- uint16x8_t vexh_u16 (uint16x8_t, uint16x8_t, unsigned)
- uint32x4_t vexh_u32 (uint32x4_t, uint32x4_t, unsigned)
- uint64x2_t vexh_u64 (uint64x2_t, uint64x2_t, unsigned)

>>> Function description: vector immediate right shift to round accumulation

Assume Vz, Vx, ry are parameters , At the same time, Vz is the return value and U/S is the sign bit

imm1=ry[5:0]; imm2=ry[11:6];

Vz(i)={Vx(i)[imm2:imm1],Vz(i)[element_size+imm1-imm2-2:0]};

i=0:(number-1)

element_size > imm2 ÿ imm1 ÿ 0

- int8x16_t vexl_s8 (int8x16_t, int8x16_t, int)
- int16x8_t vexl_s16 (int16x8_t, int16x8_t, int)
- int32x4_t vexl_s32 (int32x4_t, int32x4_t, int)
- int64x2_t vexl_s64 (int64x2_t, int64x2_t, int)
- uint8x16_t vexl_u8 (uint8x16_t, uint8x16_t, unsigned)
- uint16x8_t vexl_u16 (uint16x8_t, uint16x8_t, unsigned)
- uint32x4_t vexl_u32 (uint32x4_t, uint32x4_t, unsigned)
- uint64x2_t vexl_u64 (uint64x2_t, uint64x2_t, unsigned)

>>> Function description: Vector high-order data insertion

Assume Vz, Vx, ry are parameters , At the same time, Vz is the return value and U/S is the sign bit

imm1=ry[5:0]; imm2=ry[11:6];

Vz(i)={Vz(i)[element_size-1: imm2-imm1+1], Vx(i)[imm2:imm1]};

i=0:(number-1)

element_size > imm2 ÿ imm1 ÿ 0

4.5.6.5 Integer Move (MOV), Element Operation, and Bit Operation Instructions

vmtvr.t.1

- int8x16_t vmtvr_s8_1 (int8x16_t, char, const int)
- int16x8_t vmtvr_s16_1 (int16x8_t, short, const int)
- int32x4_t vmtvr_s32_1 (int32x4_t, int, const int)
- uint8x16_t vmtvr_u8_1 (uint8x16_t, unsigned char, const int)
- uint16x8_t vmtvr_u16_1 (uint16x8_t, unsigned short, const int)
- uint32x4_t vmtvr_u32_1 (uint32x4_t, unsigned int, const int)

>>> Function description: Vector single element write

transfer Assume Vz, rx, index are three parameters, At the same time, Vz is the return value and U/S is the sign bit

Vz(index)=Rx[element_size-1:0], the rest of the elements remain unchanged

Index range is 0~(128/element_size -1)

vmtvr.t.2

- int8x16_t vmtvr_s8_2 (int8x16_t, long long, const int)
- int16x8_t vmtvr_s16_2 (int16x8_t, long long, const int)
- int32x4_t vmtvr_s32_2 (int32x4_t, long long, const int)
- uint8x16_t vmtvr_u8_2 (uint8x16_t, long long, const int)
- uint16x8_t vmtvr_u16_2 (uint16x8_t, long long, const int)
- uint32x4_t vmtvr_u32_2 (uint32x4_t, long long, const int)

>>> Function description: Vector two-element write

transfer Assume Vz, rx, index are three parameters, At the same time, Vz is the return value and U/S is the sign bit

Vz(index)=Rx[element_size-1:0], Vz(index+1)=Rx[element_size-1+32:32],ÿ

ÿThe rest of the elements remain unchanged

Index range is 0~(128/element_size -2)

vmfvr.t

- int vmfvr_s8 (int8x16_t, const int)

- int vmfvr_s16 (int16x8_t, const int)
- int vmfvr_s32 (int32x4_t, const int)
- unsigned int vmfvr_u8 (uint8x16_t, const int)
- unsigned int vmfvr_u16 (uint16x8_t, const int)
- unsigned int vmfvr_u32 (uint32x4_t, const int)

>>> Function description: Vector write

transfer assumes Vx, index are 3 parameters, Rz is the return value

Rz = extend_32(Vx(index)); extend_32

extends the value to 32 bits by zero or sign depending on U/S

Index range is 0~(128/element_size -1)

vsext.t

- int vsext_s8 (int8x16_t)
- int vsext_s16 (int16x8_t)
- int vsext_s32 (int32x4_t)
- unsigned int vsext_u8 (uint8x16_t)
- unsigned int vsext_u16 (uint16x8_t)
- unsigned int vsext_u32 (uint32x4_t)

>>> Function Description: Vector data sign bit read and transfer

Assume Vx is the parameter, Rz is the return value, and U/S is the sign bit

If Type=8 for i=0:15, Rz[i]=Vx(i)[7]; end Rz[31:16]=0;

If Type=16 for i=0:7, Rz[i]=Vx(i)[15]; end Rz[31:8]=0;

If Type=32 for i=0:3, Rz[i]=Vx(i)[31]; end Rz[31:4]=0; (Extract the sign bit of each element of Vx , Put them into the lower bits of general register Rz in sequence)

vmov.te

- uint16x16_t vmov_s8_e (uint8x16_t)
- uint32x8_t vmov_s16_e (uint16x8_t)
- uint64x4_t vmov_s32_e (uint32x4_t)
- uint16x16_t vmov_u8_e (uint8x16_t)
- uint32x8_t vmov_u16_e (uint16x8_t)
- uint64x4_t vmov_u32_e (uint32x4_t)

>>> Function description: Vector extended transfer

assumes Vx is the parameter, Vz is the return value, U/S is the sign bit

Vz(i)=extend(Vx(i)); i=0:(number/2-1) extend Zero-extend or sign-

extend the value to twice the element width according to U/S

vmov.tl && vmov.th

- int16x8_t vmov_s16_l (int16x8_t, int16x8_t)
- int32x4_t vmov_s32_l (int32x4_t, int32x4_t)
- int64x2_t vmov_s64_l (int64x2_t, int64x2_t)
- uint16x8_t vmov_u16_l (uint16x8_t, uint16x8_t)
- uint32x4_t vmov_u32_l (uint32x4_t, uint32x4_t)
- uint64x2_t vmov_u64_l (uint64x2_t, uint64x2_t)

>>> Function description: Vector low-bit

transfer assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)={Vx(2i+1)[element_size/2-1:0], Vx(2i)[element_size/2-1:0]};

i=0:(number/2-1)

Vz(number/2+i)={Vy(2i+1)[element_size/2-1:0],Vy(2i)[element_size/2-1:0]};

i=0:(number/2-1)

Take the lower half of the element

- int16x8_t vmov_s16_h (int16x8_t, int16x8_t)
- int32x4_t vmov_s32_h (int32x4_t, int32x4_t)
- int64x2_t vmov_s64_h (int64x2_t, int64x2_t)
- uint16x8_t vmov_u16_h (uint16x8_t, uint16x8_t)
- uint32x4_t vmov_u32_h (uint32x4_t, uint32x4_t)
- uint64x2_t vmov_u64_h (uint64x2_t, uint64x2_t)

>>> Function description: Vector high-bit

transfer assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)={Vx(2i+1)[element_size-1:element_size/2], Vx(2i)[element_size-1:element_
ÿsize/2]};

i=0:(number/2-1)

Vz(number/2+i)={Vy(2i+1)[element_size-1:element_size/2],Vy(2i)[element_size-
ÿ1:element_size/2]};

i=0:(number/2-1)

Take the high half of the element

vmov.t.sl

- int16x8_t vmov_s16_sl (int16x8_t, int16x8_t)
- int32x4_t vmov_s32_sl (int32x4_t, int32x4_t)
- int64x2_t vmov_s64_sl (int64x2_t, int64x2_t)
- uint16x8_t vmov_u16_sl (uint16x8_t, uint16x8_t)
- uint32x4_t vmov_u32_sl (uint32x4_t, uint32x4_t)
- uint64x2_t vmov_u64_sl (uint64x2_t, uint64x2_t)

```
>>> Function description: Vector low-order saturation
        transfer Assume that Vx and Vy are two parameters, Vz is the return value, and U/
        S is the sign bit signed=(T==S); (select according to the element U/S type)
        Max=signed? 2^(element_size/2-1)-1: 2^(element_size/2)-1;
        Min=signed? -2^(element_size/2-1): 0;
        If Vx(i)>Max Tmp1(i)=Max;
        Else if Vx(i)<Min Tmp1(i)=Min;
        Else Tmp1(i)=Vx(i)[element_size/2-1:0];                    (Take the lower half of the elements)
        End i=0:(number-1)
        If Vy(i)>Max Tmp2(i)=Max;
        Else if Vy(i)<Min Tmp2(i)=Min;
        Else Tmp2(i)=Vy(i)[element_size/2-1:0];                    (Take the lower half of the elements)
        End i=0:(number-1)
        Vz(i)={Tmp1(2i+1),Tmp1(2i)};                    i=0:(number/2-1)
        Vz(i+number/2)={Tmp2(2i+1),Tmp2(2i)};                    i=0:(number/2-1)
```

vmov.t.rh

- int16x8_t vmov_s16_rh (int16x8_t, int16x8_t)
- int32x4_t vmov_s32_rh (int32x4_t, int32x4_t)
- int64x2_t vmov_s64_rh (int64x2_t, int64x2_t)
- uint16x8_t vmov_u16_rh (uint16x8_t, uint16x8_t)
- uint32x4_t vmov_u32_rh (uint32x4_t, uint32x4_t)
- uint64x2_t vmov_u64_rh (uint64x2_t, uint64x2_t)

```
>>> Function description: Vector high-order round
        transfer assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit
        round=1<<(elemen_size/2-1)
        Tmp1(i)=(Vx(i)+round)[element_size-1: element_size/2]; i=0:(number-1) ÿ
 ÿ(take the high half of the element)
        Tmp2(i)=(Vy(i)+round)[element_size-1: element_size/2]; i=0:(number-1) ÿ
 ÿ(take the high half of the element)
        Vz(i)={Tmp1(2i+1),Tmp1(2i)};                    i=0:(number/2-1)
        Vz(i+number/2)={Tmp2(2i+1),Tmp2(2i)};                    i=0:(number/2-1)
```

vtrn.t

- int8x32_t vtrn_s8 (int8x16_t, int8x16_t)
- int16x16_t vtrn_s16 (int16x8_t, int16x8_t)
- int32x8_t vtrn_s32 (int32x4_t, int32x4_t)
- uint8x32_t vtrn_u8 (uint8x16_t, uint8x16_t)
- uint16x16_t vtrn_u16 (uint16x8_t, uint16x8_t)

- uint32x8_t vtrn_u32 (uint32x4_t, uint32x4_t)

>>> Function description: vector data cross transmission

Assume Vx, Vy are parameters and Vz is the return value

Vz(2i+1)=Vy(2i); Vz(2i)= Vx(2i); i=0:number/2-1

Vz(2i+1+2*number)= Vy(2i+1), Vz(2i+2*number)=Vx(2i+1); i=0:number/2-1

vrevq && vrevh && vrevw && vrevd

- int8x16_t vrevq_s8 (int8x16_t)
- uint8x16_t vrevq_u8 (uint8x16_t)

>>> Function description: Vector data byte reverse

order Assume Vx is input, Vz is return value

Vz(number-1:0)= {Vx(0), Vx(1), Vx(2),… …Vx(14), Vx(15)};

- int16x8_t vrevh_s16 (int16x8_t)
- uint16x8_t vrevh_u16 (uint16x8_t)

>>> Function description: Vector data half byte reverse

order Assume Vx is input, Vz is return value

Vz(number-1:0)= {Vx(0), Vx(1), Vx(2),… …Vx(6), Vx(7)};

- int32x4_t vrevw_s32 (int32x4_t)
- uint32x4_t vrevw_u32 (uint32x4_t)

>>> Function description: Vector data word reverse

order Assume Vx is input and Vz is return value

Vz(number-1:0)= {Vx(0), Vx(1), Vx(2), Vx(3)};

- int64x2_t vrevd_s64 (int64x2_t)
- uint64x2_t vrevd_u64 (uint64x2_t)

>>> Function description: Vector data double word

reverse order Assume Vx is input, Vz is return value

Vz(number-1:0)= {Vx(0), Vx(1)};

vexti.t && vext.t

- int8x16_t vexti_s8 (int8x16_t, int8x16_t, const int)
- uint8x16_t vexti_u8 (uint8x16_t, uint8x16_t, const int)

>>> Function description: Immediate vector data concatenation

Assume Vx, Vy, imm are 3 parameters and Vz is the return value

If imm[5]==0, Vz (imm[3:0]:0)=Vx(imm[3:0]:0);

(Continued on next page)

(Continued from previous page)

```
(Copy the lower-order elements of Vx to the lower-order elements of Vz)
Else Vz (imm[3:0]:0)=Vx(15:15-imm[3:0]);
(Copy the high-order elements of Vx to the low-order elements of Vz)
If imm[4]==0, Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0); (Copy the lower elements of Vy to
the higher elements of Vz)
Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
(Copy the high-order elements of Vy to the high-order elements of Vz)
where the range of imm[3:0] is 0~14;
```

- int8x16_t vext_s8 (int8x16_t, int8x16_t, int)
- uint8x16_t vext_u8 (uint8x16_t, uint8x16_t, int)

```
>>> Function description: Register vector data splicing
Assume Vx, Vy, Rk are 3 parameters and Vz is the return value
Imm6 = Rk[5:0];
If imm6[5]==0, Vz (imm[3:0]:0)=Vx(imm[3:0]:0);
(Copy the lower-order elements of Vx to the lower-order elements of Vz)
Else Vz (imm[3:0]:0)=Vx(15:15-imm[3:0]);
(Copy the high-order elements of Vx to the low-order elements of Vz)
If imm6[4]==0, Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0); (Copy the lower elements of Vy to
the higher elements of Vz)
Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
(Copy the high-order elements of Vy to the high-order elements of Vz)
where the range of imm[3:0] is 0~14;
```

vtbl.t && vtbx.t

- int8x16_t vtbl_s8 (int8x16_t, int8x16_t)
- uint8x16_t vtbl_u8 (uint8x16_t, uint8x16_t)

```
>>> Function description: Vector data
link assumes Vx, Vy are two parameters, Vz is
the return value if Vy(i)<16 Vz(i)=Vx(Vy(i));
else Vz(i)=8'b0;
i=0:(number-1)
```

- int8x16_t vtbx_s8 (int8x16_t, int8x16_t)
- uint8x16_t vtbx_u8 (uint8x16_t, uint8x16_t)

```
>>> Function description: Vector data link
assumes Vx, Vy are two parameters, Vz is the return value
if Vy(i)<16 Vz(i)=Vx(Vy(i));
else Vz(i)=Vz(i);
i=0:(number-1)
```

vand.t && vandn.t

- int8x16_t vand_s8 (int8x16_t, int8x16_t)
- int16x8_t vand_s16 (int16x8_t, int16x8_t)
- int32x4_t vand_s32 (int32x4_t, int32x4_t)
- int64x2_t vand_s64 (int64x2_t, int64x2_t)
- uint8x16_t vand_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vand_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vand_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vand_u64 (uint64x2_t, uint64x2_t)

>>> Function description: Vector bitwise AND operation

Assume Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=Vx[j] & Vy[j]

- int8x16_t vandn_s8 (int8x16_t, int8x16_t)
- int16x8_t vandn_s16 (int16x8_t, int16x8_t)
- int32x4_t vandn_s32 (int32x4_t, int32x4_t)
- int64x2_t vandn_s64 (int64x2_t, int64x2_t)
- uint8x16_t vandn_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vandn_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vandn_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vandn_u64 (uint64x2_t, uint64x2_t)

>>> Function description: vector bitwise NOT operation

Assume Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=Vx[j] & (!Vy[j])

vxor.t

- int8x16_t vxor_s8 (int8x16_t, int8x16_t)
- int16x8_t vxor_s16 (int16x8_t, int16x8_t)
- int32x4_t vxor_s32 (int32x4_t, int32x4_t)
- int64x2_t vxor_s64 (int64x2_t, int64x2_t)
- uint8x16_t vxor_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vxor_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vxor_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vxor_u64 (uint64x2_t, uint64x2_t)

>>> Function Description: Vector bitwise XOR operation

Assume Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=Vx[j] ^ Vy[j]

vnot.t

- int8x16_t vnot_s8 (int8x16_t, int8x16_t)
- int16x8_t vnot_s16 (int16x8_t, int16x8_t)
- int32x4_t vnot_s32 (int32x4_t, int32x4_t)
- int64x2_t vnot_s64 (int64x2_t, int64x2_t)
- uint8x16_t vnot_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vnot_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vnot_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vnot_u64 (uint64x2_t, uint64x2_t)

>>> Function description: Vector bitwise inversion operation

Assume Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=!Vx[j]

vor.t && vorn.t

- int8x16_t vor_s8 (int8x16_t, int8x16_t)
- int16x8_t vor_s16 (int16x8_t, int16x8_t)
- int32x4_t vor_s32 (int32x4_t, int32x4_t)
- int64x2_t vor_s64 (int64x2_t, int64x2_t)
- uint8x16_t vor_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vor_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vor_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vor_u64 (uint64x2_t, uint64x2_t)

>>> Function description: Vector bitwise

NOT operation Assume Vx, Vy are two

parameters, Vz is the return value for j=0:127 Vz[j]=Vx[j] | Vy[j]

- int8x16_t vorn_s8 (int8x16_t, int8x16_t)
- int16x8_t vorn_s16 (int16x8_t, int16x8_t)
- int32x4_t vorn_s32 (int32x4_t, int32x4_t)
- int64x2_t vorn_s64 (int64x2_t, int64x2_t)
- uint8x16_t vorn_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vorn_u16 (uint16x8_t, uint16x8_t)

- uint32x4_t vorn_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vorn_u64 (uint64x2_t, uint64x2_t)

>>> Function description: Vector bitwise OR operation

Assume Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=Vx[j] | (! Vy[j])

vsel.t

- int8x16_t vsel_s8 (int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vsel_s16 (int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vsel_s32 (int32x4_t, int32x4_t, int32x4_t)
- int64x2_t vsel_s64 (int64x2_t, int64x2_t, int64x2_t)
- uint8x16_t vsel_u8 (uint8x16_t, uint8x16_t, uint8x16_t)
- uint16x8_t vsel_u16 (uint16x8_t, uint16x8_t, uint16x8_t)
- uint32x4_t vsel_u32 (uint32x4_t, uint32x4_t, uint32x4_t)
- uint64x2_t vsel_u64 (uint64x2_t, uint64x2_t, uint64x2_t)

>>> Function description: Vector bit selection

assumes Vx, Vy are two parameters, Vz is the return value

for j=0:127 Vz[j]=Vk[j] ?Vx[j]:Vy[j]

vcls.t && vclz.t

- int8x16_t vcls_s8 (int8x16_t)
- int16x8_t vcls_s16 (int16x8_t)
- int32x4_t vcls_s32 (int32x4_t)
- int64x2_t vcls_s64 (int64x2_t)

>>> Function description: The vector sign bits are the same continuously. Assume

that Vx is the parameter and Vz is the number of consecutive

bits of the return value starting from the MSB that are the same as the element sign bit. , The sign bit is not included in the count

Vz(i)=count_leading_sign_bit(Vx(i)); i=0:(number-1)

- int8x16_t vclz_s8 (int8x16_t)
- int16x8_t vclz_s16 (int16x8_t)
- int32x4_t vclz_s32 (int32x4_t)
- int64x2_t vclz_s64 (int64x2_t)
- uint8x16_t vclz_u8 (uint8x16_t)
- uint16x8_t vclz_u16 (uint16x8_t)
- uint32x4_t vclz_u32 (uint32x4_t)

- uint64x2_t vclz_u64 (uint64x2_t)

>>> Function description: The number of consecutive 0s in the highest bit of the vector. Assume that Vx is the parameter and Vz is the number of consecutive 0s in the return value starting from the MSB.

vcnt1.t

- int8x16_t vcnt1_s8 (int8x16_t)
- int16x8_t vcnt1_s16 (int16x8_t)
- int32x4_t vcnt1_s32 (int32x4_t)
- int64x2_t vcnt1_s64 (int64x2_t)
- uint8x16_t vcnt1_u8 (uint8x16_t)
- uint16x8_t vcnt1_u16 (uint16x8_t)
- uint32x4_t vcnt1_u32 (uint32x4_t)
- uint64x2_t vcnt1_u64 (uint64x2_t)

>>> Function description: vector data 1 count Assume Vx is the parameter, Vz is the return value
count_one calculates the number of 1 bits in the element
Vz(i)=count_one(Vx(i)) ; i=0:(number-1)

vtst.t

- int8x16_t vtst_s8 (int8x16_t, int8x16_t)
- int16x8_t vtst_s16 (int16x8_t, int16x8_t)
- int32x4_t vtst_s32 (int32x4_t, int32x4_t)
- int64x2_t vtst_s64 (int64x2_t, int64x2_t)
- uint8x16_t vtst_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vtst_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vtst_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vtst_u64 (uint64x2_t, uint64x2_t)

>>> Function description: Vector bitwise and set
Assume Vx, Vy are two parameters, Vz is the return value
Vz(i)=(|(Vx(i) & Vy(i))) ?111…11:000…00;
i=0:(number-1)

vdupg.t

- int8x16_t vdupg_s8 (signed char)
- int16x8_t vdupg_s16 (short)
- int32x4_t vdupg_s32 (int)
- uint8x16_t vdupg_u8 (unsigned char)
- uint16x8_t vdupg_u16 (unsigned short)
- uint32x4_t vdupg_u32 (unsigned int)

>>> Function Description: Copy between vector destination register and general source register

Assume Rx is the parameter and Vz is the return value

Vz(i)=Rx[element size-1:0]; i=0:(number-1)

vdup.t.1 && vdup.t.2

- int8x16_t vdup_s8_1 (int8x16_t, const int)
- int16x8_t vdup_s16_1 (int16x8_t, const int)
- int32x4_t vdup_s32_1 (int32x4_t, const int)
- uint8x16_t vdup_u8_1 (uint8x16_t, const int)
- uint16x8_t vdup_u16_1 (uint16x8_t, const int)
- uint32x4_t vdup_u32_1 (uint32x4_t, const int)

>>> Function description: Copy between unary vector source and destination registers

Assume Vx,index is the input and Vz is the return value

Vz(i)=Vx(index); i=0:(number-1)

index=0 ~ (128/element_size -1)

- int8x32_t vdup_s8_2 (int8x32_t, const int)
- int16x16_t vdup_s16_2 (int16x16_t, const int)
- int32x8_t vdup_s32_2 (int32x8_t, const int)
- uint8x32_t vdup_u8_2 (uint8x32_t, const int)
- uint16x16_t vdup_u16_2 (uint16x16_t, const int)
- uint32x8_t vdup_u32_2 (uint32x8_t, const int)

>>> Function description: Copy between binary vector source and destination registers.

Assume that Vx, index is the input and Vz is the return value.

Vz(i)=Rx(index); i=0:number-1

Vz(i)=Rx(index+1); i=number:2*number-1

index=0 ~ (128/element_size -1)

number = 128/element_size

vins.t.1 && vins.t.2

- int8x16_t vins_s8_1 (int8x16_t, int8x16_t, const int, const int)
- int16x8_t vins_s16_1 (int16x8_t, int16x8_t, const int, const int)
- int32x4_t vins_s32_1 (int32x4_t, int32x4_t, const int, const int)
- uint8x16_t vins_u8_1 (uint8x16_t, uint8x16_t, const int, const int)
- uint16x8_t vins_u16_1 (uint16x8_t, uint16x8_t, const int, const int)
- uint32x4_t vins_u32_1 (uint32x4_t, uint32x4_t, const int, const int)

>>> Function description: Unary vector
insertion assumes Vz, Vx, index, index2 are 4 parameters and Vz is the return value
Vz(index2)=Vx(index); the values of other elements in Vz remain unchanged
index=0 ~ (128/element_size -1);
index2=0 ~ (128/element_size -1)

- int8x32_t vins_s8_2 (int8x32_t, int8x32_t, const int, const int)
- int16x16_t vins_s16_2 (int16x16_t, int16x16_t, const int, const int)
- int32x8_t vins_s32_2 (int32x8_t, int32x8_t, const int, const int)
- uint8x32_t vins_u8_2 (uint8x32_t, uint8x32_t, const int, const int)
- uint16x16_t vins_u16_2 (uint16x16_t, uint16x16_t, const int, const int)
- uint32x8_t vins_u32_2 (uint32x8_t, uint32x8_t, const int, const int)

>>> Function description: Binary vector
insertion assumes Vz, Vx, index, index2 are 4 parameters At the same time Vz is the return value
Vz(index2)=Vx(index);
Vz(index2+number) = Vx(index+1)
The rest of the elements in
Vz remain unchanged index=0 ~ (128/element_size -1);
index2=0 ~ (128/element_size -1)
number = 128/element_size

vpkg.t.2

- int8x32_t vpkg_s8_2 (int8x32_t, int8x32_t, const int, const int)
- int16x16_t vpkg_s16_2 (int16x16_t, int16x16_t, const int, const int)
- int32x8_t vpkg_s32_2 (int32x8_t, int32x8_t, const int, const int)
- uint8x32_t vpkg_u8_2 (uint8x32_t, uint8x32_t, const int, const int)
- uint16x16_t vpkg_u16_2 (uint16x16_t, uint16x16_t, const int, const int)
- uint32x8_t vpkg_u32_2 (uint32x8_t, uint32x8_t, const int, const int)

>>> Function description: Binary encapsulation

Assume Vz, Vx, index, index2 are 4 parameters and Vz is the return value

bound=number; bound is used to determine whether it crosses registers, number is the number of elements

Vz(index2)=Vx(index);

if index2+1<bound

Vz(index2+1)=Vx(index+bound);

else Vz(index2+1)=Vx(index+bound);

The rest of the elements in Vz,Vz+1 remain unchanged

index=0 ~ (128/element_size -1);

index2=0 ~ (128/element_size -1)

vitl.t.2 && vdtl.t.2

- int8x32_t vitl_s8_2 (int8x32_t)
- int16x16_t vitl_s16_2 (int16x16_t)
- int32x8_t vitl_s32_2 (int32x8_t)
- uint8x32_t vitl_u8_2 (uint8x32_t)
- uint16x16_t vitl_u16_2 (uint16x16_t)
- uint32x8_t vitl_u32_2 (uint32x8_t)

>>> Function description: Binary interleaving

assumes Vx is the parameter and Vz is the return value

Vz(2i+1,2i)={Vx(i+number),Vx(i)}; i=0:(number-1)

- int8x32_t vdtl_s8_2 (int8x32_t)
- int16x16_t vdtl_s16_2 (int16x16_t)
- int32x8_t vdtl_s32_2 (int32x8_t)
- uint8x32_t vdtl_u8_2 (uint8x32_t)
- uint16x16_t vdtl_u16_2 (uint16x16_t)
- uint32x8_t vdtl_u32_2 (uint32x8_t)

>>> Function Description: Binary Deinterleaving

Assume Vx is the parameter and Vz is the return value

Vz(i) = Vx(2i);Vz(i+number) = Vx(2i+1);

i=0:(number-1)

4.5.6.6 Integer immediate value generation instructions

vmovi.8

- int8x16_t vmovi_s8 (const signed char)
- uint8x16_t vmovi_u8 (const signed char)

>>> Function description: vector immediate data transfer

Assume imm8 is the parameter and Vz is the return value

IMM8 = imm8[7:0]

Vz(i)= IMM8; i=0:(number-1)

vmovi.t16

- uint16x8_t vmovi_u16 (const signed char, const int)
- int16x8_t vmovi_s16 (const signed char, const int)

>>> Function description: vector immediate data

transfer assumes imm8, index is two parameters, Vz is the return value, U/S is the sign bit

IMM16= {8'b0, imm8[7:0]}<<(index*8) (index = 0~1)

If Type=U, Vz(i)= IMM16; i=0:(number-1)

If Type=S, Vz(i)= ~IMM16; i=0:(number-1)

vmovi.t32

- uint32x4_t vmovi_u32 (const signed char, const int)
- int32x4_t vmovi_s32 (const signed char, const int)

>>> Function description: vector immediate data

transfer assumes imm8, index is two parameters, Vz is the return value, U/S is the sign bit

IMM32= {24'b0, imm8[7:0]}<<(index*8) (index= 0~3)

If Type=U, Vz(i)= IMM32; i=0:(number-1)

If Type=S, Vz(i)= ~IMM32; i=0:(number-1)

vmaski.8.l && vmaski.8.h

- int8x16_t vmaski_s8_l (const signed char)
- uint8x16_t vmaski_u8_l (const signed char)

>>> Function description: Vector immediate extension transfer

assumes imm8 is the parameter and Vz is the return value

Vz(i)= {8{imm8[i]}}; i=0:(number/2-1); Set the rest of the elements in Vz to 0

- int8x16_t vmaski_s8_h (const signed char)
- uint8x16_t vmaski_u8_h (const signed char)

>>> Function description: Vector immediate extension transfer

assumes imm8 is the parameter and Vz is the return value

Vz(i+8)= {8{imm8[i]}}; i=0:(number/2-1); the rest of the elements in Vz remain unchanged

vmaski.16

- int16x8_t vmaski_s16 (const signed char)
- uint16x8_t vmaski_u16 (const signed char)

```
>>> Function Description: Vector immediate extension transfer
    Assume imm8 is the parameter and Vz is the return value
    Vz(i)= {16{imm8[i]}}; i=0:(number-1);
```

4.5.6.7 LOAD/STORE Instructions

vld.tn(n=1/2/3/4) && vst.tn(n=1/2/3/4)

- int8x16_t vld_8_1 (int8x16_t*, const int)
- int16x8_t vld_16_1 (int16x8_t*, const int)
- int32x4_t vld_32_1 (int32x4_t*, const int)
- int8x16_t vld_8_2 (int8x16_t*, const int)
- int16x8_t vld_16_2 (int16x8_t*, const int)
- int32x4_t vld_32_2 (int32x4_t*, const int)
- int8x16_t vld_8_3 (int8x16_t*, const int)
- int16x8_t vld_16_3 (int16x8_t*, const int)
- int32x4_t vld_32_3 (int32x4_t*, const int)
- int8x16_t vld_8_3 (int8x16_t*, const int)
- int16x8_t vld_16_3 (int16x8_t*, const int)
- int32x4_t vld_32_3 (int32x4_t*, const int)

```
>>> Function description: Fixed length vector
    load Assume Rx, offset are two parameters, Vz is the
    return value When the address corresponding to rx is nqt aligned with
    element_size, a non-aligned exception is set. size=00/01/10/11 corresponds
    to byte/half word/word/double word Offset=imm7<<size (offset is in
    the range of (0-127)<<size) for j=0:N-1(VLD.TN, N = 1/2/3/4)
    Vz(j)=MEM(Rx+offset+j*(2^(size))) ; end Set the
    remaining elements to 0
```

- void vst_8_1 (int8x16_t*, const int, int8x16_t)
- void vst_16_1 (int16x8_t*, const int, int16x8_t)
- void vst_32_1 (int32x4_t*, const int, int32x4_t)
- void vst_8_2 (int8x16_t*, const int, int8x16_t)
- void vst_16_2 (int16x8_t*, const int, int16x8_t)
- void vst_32_2 (int32x4_t*, const int, int32x4_t)

- void vst_8_3 (int8x16_t*, const int, int8x16_t)
- void vst_16_3 (int16x8_t*, const int, int16x8_t)
- void vst_32_3 (int32x4_t*, const int, int32x4_t)
- void vst_8_4 (int8x16_t*, const int, int8x16_t)
- void vst_16_4 (int16x8_t*, const int, int16x8_t)
- void vst_32_4 (int32x4_t*, const int, int32x4_t)

>>> Function description: Fixed length vector
storage Assume Rx, offset, Vz
When the address corresponding to rx is not an integer multiple of the parameters and set non-alignment exception.
element_size aligned size=00/01/10/11 corresponding to byte/half word/word/
double word offset=imm7<<size (offset range is within (0-127)<<size)
for j=0:N-1 MEM(Rx+offset+j*(2^(size)))=Vz(j); end

vldru.tn(n=1/2/3/4) && vstru.tn(n=1/2/3/4)

- int8x16_t vldru_8_1 (int8x16_t*, int)
- int16x8_t vldru_16_1 (int16x8_t*, int)
- int32x4_t vldru_32_1 (int32x4_t*, int)
- int8x16_t vldru_8_2 (int8x16_t*, int)
- int16x8_t vldru_16_2 (int16x8_t*, int)
- int32x4_t vldru_32_2 (int32x4_t*, int)
- int8x16_t vldru_8_3 (int8x16_t*, int)
- int16x8_t vldru_16_3 (int16x8_t*, int)
- int32x4_t vldru_32_3 (int32x4_t*, int)
- int8x16_t vldru_8_4 (int8x16_t*, int)
- int16x8_t vldru_16_4 (int16x8_t*, int)
- int32x4_t vldru_32_4 (int32x4_t*, int)

>>> Function description: Vector load base address jump update
Assume Rx, Ry are two parameters, Vz is the
return value. When the address corresponding to rx is not , Unaligned exception is set
element_size aligned, size=00/01/10 corresponds to byte/
half word/word for j=0:N-1 Vz(j)=MEM(Rx+j*(2^(size)));end Set the rest of the elements to 0
Rx=Rx+Ry;

- vstru_8_1 (int8x16_t*, int, int8x16_t)
- vstru_16_1 (int16x8_t*, int, int16x8_t)
- vstru_32_1 (int32x4_t*, int, int32x4_t)
- vstru_8_2 (int8x16_t*, int, int8x16_t)

- vstru_16_2 (int16x8_t*, int, int16x8_t)
- vstru_32_2 (int32x4_t*, int, int32x4_t)
- vstru_8_3 (int8x16_t*, int, int8x16_t)
- vstru_16_3 (int16x8_t*, int, int16x8_t)
- vstru_32_3 (int32x4_t*, int, int32x4_t)
- vstru_8_4 (int8x16_t*, int, int8x16_t)
- vstru_16_4 (int16x8_t*, int, int16x8_t)
- vstru_32_4 (int32x4_t*, int, int32x4_t)

```
>>> Function description: Vector storage base address jump
        update Assume Rx, Ry, Vz are three
        parameters When the address corresponding to rx is not , Unaligned exception is set
        element_size aligned size=00/01/10 corresponds to byte/half word/word
        for j=0:N-1 MEM(Rx+j*(2^(size)))=Vz(j); end
        Rx=Rx+Ry;
```

vldu.tn(n=1/2/3/4) && vstu.tn(n=1/2/3/4)

- int8x16_t vldu_8_1 (int8x16_t*)
- int16x8_t vldu_16_1 (int16x8_t*)
- int32x4_t vldu_32_1 (int32x4_t*)
- int8x16_t vldu_8_2 (int8x16_t*)
- int16x8_t vldu_16_2 (int16x8_t*)
- int32x4_t vldu_32_2 (int32x4_t*)
- int8x16_t vldu_8_3 (int8x16_t*)
- int16x8_t vldu_16_3 (int16x8_t*)
- int32x4_t vldu_32_3 (int32x4_t*)
- int8x16_t vldu_8_4 (int8x16_t*)
- int16x8_t vldu_16_4 (int16x8_t*)
- int32x4_t vldu_32_4 (int32x4_t*)

```
>>> Function description: Vector load base address update
        Assume Rx is the parameter and Vz is the return
        value When the address corresponding to rx is not element_size aligned , An unaligned exception is raised.
        size=00/01/10 corresponds to byte/half word/word
        for j=0:N-1 Vz(j)=MEM(Rx+j*(2^(size))); end Set the rest of the elements to 0
        Rx=Rx+N*2^(size);
```

- void vstu_8_1 (int8x16_t*, int8x16_t)
- void vstu_16_1 (int16x8_t*, int16x8_t)

- void vstu_32_1 (int32x4_t*, int32x4_t)
- void vstu_8_2 (int8x16_t*, int8x16_t)
- void vstu_16_2 (int16x8_t*, int16x8_t)
- void vstu_32_2 (int32x4_t*, int32x4_t)
- void vstu_8_3 (int8x16_t*, int8x16_t)
- void vstu_16_3 (int16x8_t*, int16x8_t)
- void vstu_32_3 (int32x4_t*, int32x4_t)
- void vstu_8_4 (int8x16_t*, int8x16_t)
- void vstu_16_4 (int16x8_t*, int16x8_t)
- void vstu_32_4 (int32x4_t*, int32x4_t)

>>> Function description: Vector storage base
address update
Assume Vz, Rx is input When the address corresponding, An unaligned exception is raised.
to rx is not element_size aligned size=00/01/10
corresponding to byte/half word/word for j=0:N-1 MEM(Rx+j*(2^(size)))=Vz(j);
end
Rx=Rx+N*2^(size)

vldm.t && vstm.t

- int8x16_t vldm_8 (int8x16_t*)
- int16x8_t vldm_16 (int16x8_t*)
- int32x4_t vldm_32 (int32x4_t*)
- int8x32_t vldm_8_256 (int8x32_t*)
- int16x16_t vldm_16_256 (int16x16_t*)
- int32x8_t vldm_32_256 (int32x8_t*)

>>> Function Description: Continuous vector loading
Assume Rx is the parameter and Vz is the return
value. When the address corresponding to rx is not aligned to element_size , An unaligned exception is raised.
Vz = MEM(Rx)

- void vstm_8 (int8x16_t*, int8x16_t)
- void vstm_16 (int16x8_t*, int16x8_t)
- void vstm_32 (int32x4_t*, int32x4_t)
- void vstm_8_256 (int8x32_t*, int8x32_t)
- void vstm_16_256 (int16x16_t*, int16x16_t)
- void vstm_32_256 (int32x8_t*, int32x8_t)

>>> Function description: Continuous vector storage

Assume that Rx and Vz are two

parameters. When the address corresponding to rx is not aligned with element_size, An unaligned exception is raised.

MEM(Rx) = Vz

vldmu.t && vstmu.t

- int8x16_t vldmu_8 (int8x16_t*)
- int16x8_t vldmu_16 (int16x8_t*)
- int32x4_t vldmu_32 (int32x4_t*)
- int8x32_t vldmu_8_256 (int8x32_t*)
- int16x16_t vldmu_16_256 (int16x16_t*)
- int32x8_t vldmu_32_256 (int32x8_t*)

>>> Function Description: Continuous vector load base address update

Assume Rx is the parameter and Vz is the return

value. When the address corresponding to rx is not aligned to element_size , An unaligned exception is raised.

Vz = MEM(Rx)

Rx += size(Vz)

- void vstmu_8 (int8x16_t*, int8x16_t)
- void vstmu_16 (int16x8_t*, int16x8_t)
- void vstmu_32 (int32x4_t*, int32x4_t)
- void vstmu_8_256 (int8x32_t*, int8x32_t)
- void vstmu_16_256 (int16x16_t*, int16x16_t)
- void vstmu_32_256 (int32x8_t*, int32x8_t)

>>> Function description: Continuous vector storage base address

update Assume Rx, Vz are two

parameters When the address corresponding to rx is not element_size aligned, An unaligned exception is raised.

MEM(Rx) = Vz

Rx += size(Vz)

vldmru.t && vstmru.t

- int8x16_t vldmru_8 (int8x16_t*, const int)
- int16x8_t vldmru_16 (int16x8_t*, const int)
- int32x4_t vldmru_32 (int32x4_t*, const int)
- int8x32_t vldmru_8_256 (int8x32_t*, const int)
- int16x16_t vldmru_16_256 (int16x16_t*, const int)
- int32x8_t vldmru_32_256 (int32x8_t*, const int)

>>> Function Description: Jump vector load base address update

Assume Rx, Ry are two parameters and Vz is the return value

Vz = MEM(Rx)

Rx += Ry;

- void vstmru_8 (int8x16_t*, int, int8x16_t)
- void vstmru_16 (int16x8_t*, int, int16x8_t)
- void vstmru_32 (int32x4_t*, int, int32x4_t)
- void vstmru_8_256 (int8x32_t*, int, int8x32_t)
- void vstmru_16_256 (int16x16_t*, int, int16x16_t)
- void vstmru_32_256 (int32x8_t*, int, int32x8_t)

>>> Function description: Jump vector storage base address

update Assume Rx, Ry, Vz are 3 parameters

MEM(Rx) = Vz

Rx += Ry

vldx.t

- int8x16_t vldx_8 (int8x16_t*, int)
- int16x8_t vldx_16 (int16x8_t*, int)
- int32x4_t vldx_32 (int32x4_t*, int)

>>> Function Description: Variable length vector loading

Assume Vx and Vy are two parameters and Vz is the

exception. When the address corresponding to rx is not aligned, the return value, then set a non-aligned

element_size aligned, size=00/01/10/corresponds to byte/half word/word/

If Type =8, N= Ry[3:0] (0<N<16)

If Type =16, N= Ry[2:0] (0<N<8)

If Type =32, N= Ry[1:0] (0<N<4)

for j=0:N-1 Vz(j)=MEM(Rx+j*(2^(size))); end Set the remaining elements to 0 If N=0, the result is

unpredictable

vlrw.tn

- int32x4_t vlrw_s32_4 (const int, const int, const int, const int)
- uint32x4_t vlrw_u32_4 (const int, const int, const int, const int)

>>> Function Description: Vector memory read

Assume imm1, imm2, imm3, imm4 are 4 parameters and Vz is the return value

Vz[0] = imm1; Vz[1] = imm2; Vz[2] = imm3; Vz[3] = imm4

4.5.6.8 Floating-point addition and subtraction comparison instructions

vadd.t && vsub.t

- float32x4_t vadd_f32 (float32x4_t, float32x4_t)

>>> Function description: Floating point vector

addition assumes Vx, Vy are two parameters, Vz is the return value

Vz(i)=Vx(i)+Vy(i); i=0:(number-1)

- float32x4_t vsub_f32 (float32x4_t, float32x4_t)

>>> Function description: Floating point vector

subtraction assumes Vx, Vy are two parameters, Vz is the return value

Vz(i)=Vx(i)-Vy(i); i=0:(number-1)

vpadd.t

- float32x4_t vpadd_f32 (float32x4_t, float32x4_t)

>>> Function Description: Vector floating point coupled addition

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=Vx(2i)+Vx(2i+1); i=0:(number/2-1)

Vz(number/2+i)=Vy(2i)+Vy(2i+1); i=0:(number/2-1)

vasx.t && vsax.t

- float32x4_t vasx_f32 (float32x4_t, float32x4_t)

>>> Function Description: Vector floating point cross addition and subtraction

Assume Vx, Vy are two parameters, Vz is the return value

Vz(2i+1) = Vx(2i+1)+Vy(2i); i=0:(number/2-1)

Vz(2i)= Vx(2i)-Vy(2i+1); i=0:(number/2-1)

- float32x4_t vsax_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point cross subtraction and addition

Assume Vx, Vy are two parameters, Vz is the return value

Vz(2i+1) = Vx(2i+1)-Vy(2i); i=0:(number/2-1)

Vz(2i)= Vx(2i)+Vy(2i+1); i=0:(number/2-1)

vabs.t && vsabs.t

- float32x4_t vabs_f32 (float32x4_t)

>>> Function Description: Vector floating point absolute value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=abs(Vx(i)); i=0:number-1

- float32x4_t vsabs_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point subtraction absolute value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=abs(Vx(i)-Vy(i)); i=0:number-1

vneg.t

- float32x4_t vneg_f32 (float32x4_t)

>>> Function description: vector floating point negation

Assume Vx is the parameter and Vz is the return value

Vz(i)=-Vx(i) ; i=0:number-1

vmax.t && vmin.t

- float32x4_t vmax_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point maximum

value Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max((Vx(i),Vy(i)) ; i=0:number-1 max takes the larger value of the two elements

- float32x4_t vmin_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point minimum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=min((Vx(i),Vy(i)) ; i=0:number-1 min takes the smaller value of the two elements

vmaxnm.t && vminnm.t

- float32x4_t vmaxnm_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point norm maximum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max((Vx(i),Vy(i)) ; i=0:number-1

max takes the larger of the two elements;

different from VMAX is , One of the two elements is quiteÿ and takes ÿNaN and the other is a normalized numþerthe value of the normalized number as output.

- float32x4_t vminnm_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point norm minimum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=min((Vx(i),Vy(i)) ; i=0:number-1

min takes the smaller value of the two elements;

When the difference from , One of the two elements is quite ÿ

VMIN is ÿNaN and the other is a normalized numbęr Take the value of the normalized number as output.

vpmax.t && vpmin.t

- float32x4_t vpmax_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point adjacent maximum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max(Vx(2i),Vx(2i+1)); i=0:(number/2-1)

Vz(number/2+i)=max(Vy(2i),Vy(2i+1)); max takes the i=0:(number/2-1)

larger value of the two elements

- float32x4_t vpmin_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point adjacent minimum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=min(Vx(2i),Vx(2i+1)); i=0:(number/2-1)

Vz(number/2+i)=min(Vy(2i),Vy(2i+1)); min takes the i=0:(number/2-1)

smaller value of the two elements

vcmpnez.t && vcmpne.t

- float32x4_t vcmpnez_f32 (float32x4_t)

>>> Function description: Vector floating point is not equal to zero comparison

Assume Vx is the parameter and Vz is the return value

If Vx(i)!=0 Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

- float32x4_t vcmpne_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point equal to zero comparison

Assume Vx, Vy are two parameters, Vz is the return value

If Vx(i)!=Vy(i) Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

vcmphsz.t && vcmphs.t

- float32x4_t vcmphsz_f32 (float32x4_t)

>>> Function description: vector floating point greater than or equal to zero comparison

Assume Vx is the parameter and Vz is the return value

If Vx(i)ÿ0 Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

- float32x4_t vcmphs_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point greater than zero

comparison Assume Vx, Vy are parameters, Vz is the return value

If Vx(i)ÿVy(i) Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

vcmpltz.t && vcmplt.t

- float32x4_t vcmpltz_f32 (float32x4_t)

>>> Function description: Vector floating point less than or equal to zero comparison

Assume Vx is the parameter and Vz is the return value

If Vx(i)<0 Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

- float32x4_t vcmplt_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point less than zero

comparison Assume Vx, Vy are parameters, Vz is the return value

If Vx(i)<Vy(i) Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

vcmphz.t && vcmplsz.t

- float32x4_t vcmmphz_f32 (float32x4_t)

>>> Function description: Vector floating point greater than zero comparison

Assume Vx is the parameter and Vz is the return value

If Vx(i)>0 Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

- float32x4_t vcmplsz_f32 (float32x4_t)

>>> Function description: Vector floating point comparison is less than or equal to zero

Assume Vx is the parameter and Vz is the return value

If Vx(i)ÿ0 Vz(i)=11…111; Else Vz(i)=00…000; i=0:number-1

4.5.6.9 Floating-point multiplication instructions

vmult.t && vmuli.t

- float32x4_t vmul_f32 (float32x4_t, float32x4_t)

>>> Function Description: Vector single-precision multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=Vx(i)*Vy(i); i=0:number-1

- float32x4_t vmuli_f32 (float32x4_t, float32x4_t, const int)

>>> Function Description: Vector single-precision index multiplication

Assume Vx, Vy, index are 3 parameters, Vz is the return value

Vz(i)=Vx(i)*Vy(index); i=0:number-1

index=0 ~ (128/element_size -1);

vmula.t && vmulai.t

- float32x4_t vmula_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function Description: Vector single-precision multiplication and accumulation

Assume Vz, Vx, Vy are three parameters , At the same time Vz is the return value

Vz(i)=Vz(i)+Vx(i)*Vy(i); Note: The i=0:number-1

multiplication result is rounded before accumulation

- float32x4_t vmulai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function Description: Vector single-precision index multiplication and accumulation

Assume Vz, Vx, Vy, index are 4 parameters and,Vz is the return value

Vz(i)=Vz(i)+Vx(i)*Vy(index); Note: The i=0:number-1

multiplication result is rounded before accumulation

index=0 ~ (128/element_size -1);

vmuls.t && vmulsi.t

- float32x4_t vmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function Description: Vector single-precision multiplication and subtraction

Assume Vz, Vx, Vy are three parameters , At the same time Vz is the return value

Vz(i)=Vz(i)-Vx(i)*Vy(i); i=0:number-1

Note: The multiplication result is rounded before being accumulated.

- float32x4_t vmulsi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function description: vector single-precision index multiplication and subtraction

Assume Vz, Vx, Vy, index are 4 parameters and,Vz is the return value

Vz(i)=Vz(i)-Vx(i)*Vy(index)); Note: The multiplication i=0:number-1

result is rounded before cumulative subtraction

index=0 ~ (128/element_size -1);

vfmula.t && vfmuls.t

- float32x4_t vfmula_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function Description: Vector single-precision fused multiply and accumulate

Assume Vz, Vx, Vy are 3 parameters and Vz is the return value

Vz(i)=Vz(i)+Vx(i)*Vy(i); i=0:number-1

Note: The multiplication result retains full precision and is accumulated

- float32x4_t vfmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function Description: Vector single-precision fused multiplication and subtraction

Assume Vz, Vx, Vy are 3 parameters , At the same time Vz is the return value

Vz(i)=Vz(i)-Vx(i)*Vy(i); Note: The multiplication i=0:number-1

result retains all precision and participates in the cumulative subtraction

vfnmula.t && vfnmuls.t

- float32x4_t vfnmula_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: vector floating point fusion multiplication and negative cumulative subtraction

Assume Vz, Vx, Vy are 3 parameters and Vz is the return value

Vz(i)=-Vz(i)-Vx(i)*Vy(i); i=0:number-1

Note: The multiplication result retains full precision and is accumulated

- float32x4_t vfnmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: vector floating point multiplication and subtraction

Assume Vz, Vx, Vy are 3 parameters and Vz is the return value

Vz(i)=-Vz(i)+Vx(i)*Vy(i); Note: The multiplication i=0:number-1

result retains all precision and participates in the cumulative subtraction

vfmulxaa.t && vfmulxaai.t

- float32x4_t vfmulxaa_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: Vector floating point complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy are 3 parameters and Vz is the return value

Vz(2i+1)=Vz(2i+1)+Vx(2i)*Vy(2i+1); i=0:(number/2-1)

Vz(2i)=Vz(2i)+ Vx(2i)*Vy(2i); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

- float32x4_t vfmulxaai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function description: Vector floating point index complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy, index are 4 parameters , At the same time Vz is the return value

(Continued on next page)

(Continued from previous page)

Vz(2i+1)=Vz(2i+1)+Vx(2i)*Vy(2index+1); i=0:(number/2-1)

Vz(2i)=Vz(2i)+ Vx(2i)*Vy(2index); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

index=0 ~ (128/(element_size*2) -1);

vfmulxas.t && vfmulxasi.t

- float32x4_t vfmulxas_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: Vector floating point complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy are 3 parameters and Vz is the return value

Vz(2i+1)=Vz(2i+1)+Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i)=Vz(2i)-Vx(2i+1)*Vy(2i+1); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

- float32x4_t vfmulxasi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function description: Vector floating point index complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy, index are 4 parameters , At the same time Vz is the return value

Vz(2i+1)=Vz(2i+1)+Vx(2i+1)*Vy(2index); i=0:(number/2-1)

Vz(2i)=Vz(2i)-Vx(2i+1)*Vy(2index+1); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

index=0 ~ (128/(element_size*2) -1);

vfmulxss.t && vfmulxssi.t

- float32x4_t vfmulxss_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: Vector floating point complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy are 3 parameters , At the same time Vz is the return value

Vz(2i+1)=Vz(2i+1)-Vx(2i)*Vy(2i+1); i=0:(number/2-1)

Vz(2i)=Vz(2i)-Vx(2i)*Vy(2i); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

- float32x4_t vfmulxssi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function description: Vector floating point index complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy, index are 4 parameters and Vz is the return value

Vz(2i+1)=Vz(2i+1)-Vx(2i)*Vy(2index+1); i=0:(number/2-1)

Vz(2i)=Vz(2i)-Vx(2i)*Vy(2index); i=0:(number/2-1)

Note: The multiplication result retains full precision when added/subtracted

index=0 ~ (128/(element_size*2) -1);

vfmulxsa.t && vfmulxsai.t

- float32x4_t vfmulxsa_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: Vector floating point complex multiplication and accumulation real and imaginary part

calculation Assume Vz, Vx, Vy are 3 parameters , At the same time Vz is the return value

Vz(2i+1)=Vz(2i+1)-Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i)=Vz(2i)+Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: The multiplication result retains full precision and participates in accumulation/subtraction

- float32x4_t vfmulxsai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

>>> Function description: Vector floating point index complex multiplication and accumulation real and imaginary part calculation

Assume Vz, Vx, Vy, index are 4 parameters and,Vz is the return value

Vz(2i+1)=Vz(2i+1)-Vx(2i+1)*Vy(2index); i=0:(number/2-1)

Vz(2i)=Vz(2i)+Vx(2i+1)*Vy(2index+1); i=0:(number/2-1) Note: The multiplication result retains full precision and participates in accumulation/subtraction

index=0 ~ (128/(element_size*2) -1);

vfcmul.t && vfcmula.t

- float32x4_t vfcmul_f32 (float32x4_t, float32x4_t)

>>> Function Description: Vector floating point complex multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Tmp(2i+1) = Vx(2i)*Vy(2i+1);

Tmp(2i) = Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1)+Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i) = Tmp(2i) -Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: The multiplication result of Tmp(i) is rounded and saturated once, and the multiplication and accumulation result in Vz(i) is rounded and saturated again

- float32x4_t vfcmula_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: vector floating point complex multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters, Vz is the return value

Tmp(2i+1) = Vz(2i+1) + Vx(2i)*Vy(2i+1);

Tmp(2i) = Vz(2i) + Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1) + Vx(2i+1)*Vy(2i);

Vz(2i) = Tmp(2i) - Vx(2i+1)*Vy(2i+1); Note: Tmp(i) performs a rounding and saturation operation on the multiplication and accumulation, and Vz(i) performs another rounding and saturation operation on the multiplication and accumulation result.

vfcmulc.t && vfcmulca.t

- float32x4_t vfcmulc_f32 (float32x4_t, float32x4_t)

>>> Function Description: Vector floating point complex conjugate multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Tmp(2i+1) = Vx(2i)*Vy(2i+1);

Tmp(2i) = Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1)-Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i) = Tmp(2i)+Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: The multiplication result of Tmp(i) is rounded and saturated once, and the multiplication and accumulation result in Vz(i) is rounded and saturated again

- float32x4_t vfcmulca_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function Description: Vector floating point complex conjugate multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters, Vz is the return value

Tmp(2i+1) = Vz(2i+1) + Vx(2i)*Vy(2i+1);

Tmp(2i) = Vz(2i) + Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1)-Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i) = Tmp(2i)+Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: Tmp(i) performs a rounding and saturation operation on the multiplication and accumulation, and Vz(i) performs another rounding and saturation operation on the multiplication and accumulation result

vfcmuln.t && vfcmulna.t

- float32x4_t vfcmuln_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point complex negative multiplication Assume Vx, Vy are two parameters, Vz is the return value

Tmp(2i+1) = -Vx(2i)*Vy(2i+1);

Tmp(2i) = -Vx(2i)*Vy(2i);

Vz(2i+1)= Tmp(2i+1) -Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i)= Tmp(2i) +Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: The multiplication result of Tmp(i) is rounded and saturated once, and the multiplication and accumulation result in Vz(i) is rounded and saturated again

- float32x4_t vfcmulna_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: vector floating point complex number negative multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters, Vz is the return value

Tmp(2i+1) = Vz(2i+1) - Vx(2i)*Vy(2i+1);

Tmp(2i) = Vz(2i) - Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1) - Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i)= Tmp(2i) + Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: Tmp(i) performs a rounding and saturation operation on the multiplication and accumulation, and Vz(i) performs another rounding and saturation operation on the multiplication and accumulation result

vfcmulcn.t && vfcmulcna.t

- float32x4_t vfcmulcn_f32 (float32x4_t, float32x4_t)

>>> Function description: vector floating point complex conjugate negative multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Tmp(2i+1) = -Vx(2i)*Vy(2i+1);

Tmp(2i) = -Vx(2i)*Vy(2i);

Vz(2i+1)= Tmp(2i+1) + Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i)= Tmp(2i) -Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: The multiplication result of Tmp(i) is rounded and saturated once, and the multiplication and accumulation result in Vz(i) is rounded and saturated again

- float32x4_t vfcmulcna_f32 (float32x4_t, float32x4_t, float32x4_t)

>>> Function description: vector floating point complex conjugate negative multiplication and accumulation

Assume Vz, Vx, Vy are 3 parameters, Vz is the return value

Tmp(2i+1) = Vz(2i+1) - Vx(2i)*Vy(2i+1);

Tmp(2i) = Vz(2i) - Vx(2i)*Vy(2i);

Vz(2i+1) = Tmp(2i+1) +Vx(2i+1)*Vy(2i); i=0:(number/2-1)

Vz(2i) = Tmp(2i) -Vx(2i+1)*Vy(2i+1); i=0:(number/2-1) Note: Tmp(i) performs a rounding and saturation operation on the multiplication and accumulation, and Vz(i) performs another rounding and saturation operation on the multiplication and accumulation result

4.5.6.10 Floating-point reciprocal, reciprocal square root, e-exponential fast calculation and approximation instructions

vrecpe.t && vrecps.t

- float32x4_t vrecpe_f32 (float32x4_t)

>>> Function Description: Vector floating point reciprocal instruction

assumes Vx is the parameter and Vz is the return value

Vz(i) ÿ1/(Vx(i)) i=0:(number-1) (quickly calculate the reciprocal value of Vx(i))

- float32x4_t vrecps_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point reciprocal approximation

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i) = 2 – Vx(i) * Vy(i) i=0:(number-1)

vrsqrte.t && vrsqrts.t

- float32x4_t vrsqrte_f32 (float32x4_t)

>>> Function description: vector floating point reciprocal square root

Assume Vx is the parameter and Vz is the return value

Vz(i) ÿ1/((Vx(i))^(1/2)) i=0:(number-1) (Quickly calculate the reciprocal square root of Vx(i))

- float32x4_t vrsqrts_f32 (float32x4_t, float32x4_t)

>>> Function description: Vector floating point square root approximation

Assume Vx, Vy are parameters and Vz is the return value

Vz(i) = (3-Vx(i)*Vy(i))/2 i=0:(number-1)

vexpe.t

- float32x4_t vexpe_f32 (float32x4_t)

>>> Function Description: Vector floating point fast e instruction calculation

Assume Vx is the parameter and Vz is the return value

Vz(i) ÿ e^(Vx(i)) i=0:(number-1) (Quickly calculate the e-index value of Vx(i))

4.5.6.11 Floating-point conversion instructions

vdtos.t

- float32x4_t vdtos_f64 (float64x2_t)

>>> Function Description: Vector double-precision floating point single-precision floating point conversion

Assume that Vx is the parameter and Vz is the return value

Vz(i)={double_to_single(Vx(2i+1)), double_to_single(Vx(2i))}; i=0:(number/2-ÿ1);

Convert 64-bit double-precision floating point number to 32-bit single-precision floating point number

vftox.t1.t2 && vxtof.t1.t2 (bit width unchanged)

- int32x4_t vftox_f32_s32 (float32x4_t)
- uint32x4_t vftox_f32_u32 (float32x4_t)

>>> Function Description: Vector floating point fixed point conversion

assumes Vx is the parameter and Vz is the return value

Vz(i)=float_to_fix(Vx(i)); i=0:(number-1);

Convert a (16-bit/32-bit) floating point number to a U/S fixed point number of the same ,
bit width. FCR[20:16].frpos[4:0] specifies the decimal point position of the fixed
point number. 16-bit fixed point numbers use frpos[3:0] to indicate the 1-16-bit
decimal part. 32-bit fixed point numbers use frpos[4:0] to indicate the 1-32-bit decimal part.

- float32x4_t vxtof_s32_f32 (int32x4_t)
- float32x4_t vxtof_u32_f32 (uint32x4_t)

>>> Function Description: Vector fixed-point floating-point conversion

Vx is the parameter and Vz is the return value

Vz(i)=fix_to_float(Vx(i)); i=0:(number-1);

(Continued on next page)

(Continued from previous page)

Convert (16-bit/32-bit) U/S fixed-point numbers to floating-point numbers of the same bit ,
width. FCR[20:16].frpos[4:0] specifies the decimal point position of the fixed-
point number. 16-bit fixed-point numbers use frpos[3:0] to indicate the 1-16-bit
decimal part. 32-bit fixed-point numbers use frpos[4:0] to indicate the 1-32-bit decimal part.

vftox.t1.t2 && vxtof.t1.t2 (bit width expansion)

- float32x8_t vxtof_s16_f32 (int16x8_t)
- float32x8_t vxtof_u16_f32 (uint16x8_t)

>>> Function Description: Vector fixed-point floating-point conversion
assumes Vx is the parameter and Vz is the return value
Vz(i)=fix16_to_single(Vx(i)); i=0:(number-1);
Convert 16-bit U/S fixed-point number to 32-bit single-precision floating-point ,
number FCR[20:16].frpos[4:0] specifies the decimal point position of the fixed-point
number. 16-bit fixed-point number uses frpos[3:0] to indicate the decimal part of 1 to 16 bits.

vftox.t1.t2 && vxtof.t1.t2 (bit width halved)

- int16x8_t vftox_f32_s16 (float32x4_t)
- uint16x8_t vftox_f32_u16 (float32x4_t)

>>> Function description: vector floating point fixed point conversion
Assume Vx is the parameter and Vz is the return value
Vz(i)={single_to_fix16(Vx(2i+1)), single_to_fix16(Vx(2i))}; i=0:(number/2-1); Convert 32-bit single floating point number to 16-bit U/
S fixed point number. FCR[20:16].frpos[4:0] specifies the ,
decimal point position of the fixed point number. 16-bit fixed point
number uses frpos[3:0] to indicate the decimal part of 1 to 16 bits.

vftoi.t1.t2 (bit width unchanged)

- int32x4_t vftoi_f32_s32 (float32x4_t)
- uint32x4_t vftoi_f32_u32 (float32x4_t)

>>> Function Description: Vector floating point integer conversion
Assume Vx is the parameter and Vz is the return value
Vz(i)=float_to_int(Vx(i)); i=0:(number-1); Convert a (16-bit/32-bit)
floating point number to a U/S integer of the same bit width

vftoi.t1.t2 (bit width halved)

- int16x8_t vftoi_f32_s16 (float32x4_t)

- uint16x8_t vftoi_f32_u16 (float32x4_t)

>>> Function Description: Vector floating point integer conversion

assumes Vx is the parameter and Vz is the return value

Vz(i)={single_to_int16(Vx(2i+1)), single_to_fix16(Vx(2i))}; i=0:(number/2-1); Convert 32-bit single floating point number to 16-bit U/S integer

vitof.t1.t2 (same bit width)

- float32x4_t vitof_s32_f32 (int32x4_t)
- float32x4_t vitof_u32_f32 (uint32x4_t)

>>> Function Description: Vector integer floating point conversion

Assume Vx is the parameter and Vz is the return value

Vz(i)=int_to_float(Vx(i)); i=0:(number-1); Convert a (16-bit/32-bit) U/S integer to a floating point number of the same bit width

vitof.t1.t2 (bit width expansion)

- float32x8_t vitof_s16_f32 (int16x8_t)
- float32x8_t vitof_u16_f32 (uint16x8_t)

>>> Function Description: Vector integer floating point conversion

assumes Vx is the parameter and Vz is the return value

Vz(i)=int16_to_single(Vx(i)); i=0:(number-1); Convert 16-bit U/S integer to 32-bit single-precision floating point number

vftoi.t1.t2.rn (round to nearest)

- int32x4_t vftoi_f32_s32_rn (float32x4_t)
- uint32x4_t vftoi_f32_u32_rn (float32x4_t)

>>> Function Description: Vector floating point integer conversion with rounding

Assume Vx is the parameter and Vz is the return value

Vz(i)=single_to_int32(Vx(i)); i=0:(number-1); Convert a 32-bit single-precision floating point number to a 32-bit U/S integer

vftoi.t1.t2.rz (round to zero)

- int32x4_t vftoi_f32_s32_rz (float32x4_t)
- uint32x4_t vftoi_f32_u32_rz (float32x4_t)

>>> Function Description: Vector floating point integer conversion with rounding

Assume Vx is the parameter and Vz is the return value

Vz(i)=single_to_int32(Vx(i)); i=0:(number-1); Convert a 32-bit single-precision floating point number to a 32-bit U/S integer

vftoi.t1.t2.rpi (round to +inf)

- int32x4_t vftoi_f32_s32_rpi (float32x4_t)
- uint32x4_t vftoi_f32_u32_rpi (float32x4_t)

>>> Function Description: Vector floating point integer conversion with rounding

Assume Vx is the parameter and Vz is the return value

Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);

Convert a 32-bit single-precision floating point number to a 32-bit U/S integer

vftoi.t1.t2.rni (round to -inf)

- int32x4_t vftoi_f32_s32_rni (float32x4_t)
- uint32x4_t vftoi_f32_u32_rni (float32x4_t)

>>> Function Description: Vector floating point integer conversion with rounding

Assume Vx is the parameter and Vz is the return value

Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);

Convert a 32-bit single-precision floating point number to a 32-bit U/S integer

4.6 dsp

Currently, the CSKY architecture supports two versions of DSP instruction sets, namely dspv1 and dspv2. DSPV2 contains vector computing instructions, and DSPV1 contains HI and LO registers as operand computing instructions. The DSP instruction bit width is 32 bits. The compiler determines whether the generated target program supports dsp instructions based on the CPU options. For specific support information, please refer to Table 4.2

The compiler generates DSP-type vector instructions in the following situations:

- Vector operation expressions
- Loop optimization
- Use intrinsic functions

The first two are for relatively basic scenarios, while the third is suitable for scenarios that require deep optimization. For detailed instructions, please refer to the following sections of this chapter:

- Vector data type
- Rules for passing parameters and return values of vector types
- Vector operation expressions

- Loop optimization to generate vector instructions (currently only supported in the *GNU* toolchain)
- *Intrinsic* function interface naming rules
- *dspv2* intrinsic interface

4.6.1 Vector Data Types

Vector data types are usually built on top of common data types. For example, the vector data type int8x4_t represents an 8-bit integer data type.

A type consisting of 4 elements has a total bit width of 32 bits. The naming convention is as follows:

```
>                    [element type][element width]x[number of elements]_t
```

The element types are int and uint. When using, you need to reference the header file csky_vdsp.h. The vector data types supported by dspv2 are shown in Table 4.12 :

Table 4.12: dspv2 vector data types

dspv2 32-bit	int8x4_t
int	
	int16x2_t
uint	uint8x4_t
	uint16x2_t

4.6.2 Rules for passing vector type parameters and return values

DSP vector type parameters and return values are passed using ordinary registers.

4.6.3 Vector Operation Expressions

The compiler supports vector operation expressions, which consist of vector type variables and operators. The compiler generates corresponding vector instructions based on these expressions.

4.6.3.1 Definition of vector type variables

There are two ways to define vector type variables:

- The first method is the same as array definition, such as:

```
#include<csky_vdsp.h>

int8x4_t a = {1,2,3,4};
```

- The second method is to define an array first, and then convert the array address into a vector pointer type, such as:

```
#include<csky_vdsp.h>

int a[ ] = {1,2,3,4};
int8x4_t *ap = (int8x4_t *)a;
```

4.6.3.2 Operators

C language uses operators to represent arithmetic operations, and the same is true for variables of vector type.

Currently, the operators supported by vector expressions are as follows:

- Addition: +
- Subtraction:-
- Multiplication: *
- Comparison operators: >, <, !=, >=, <=, ==
- Logical operators: &, |, ^
- Shift operators: », «

Here is a simple example:

```
#include<csky_vdsp.h>

int8x4_t a = {1,2,3,4};
int8x4_t b = {5,6,7,8};
int8x4_t c = {2,4,6,8};

int8x4_t vfunc ()
{
    return a        * b + c;
}
```

4.6.4 Loop optimization to generate vector instructions (currently only supported in the GNU toolchain)

The compiler supports optimizing some loops into vector instructions. The compiler will try to optimize the loop into vector instructions when the following conditions are met:

- Current CPUs support vector instructions
- The optimization level is -O1 or above, and option -ftree-loop-vectorize is added

  (This option is enabled by default with -O3)

For example, the following loop:

```
void svfun1 (char *a,char *b,char *c)
{
    for (int i = 0;i < 4;i++)
        c[i] = a[i] + b[i]; /*scalar operation*/
}
```

If the current CPU supports 32-bit vector addition instructions, after loop optimization is enabled, the optimized code above is as shown in the following pseudo code:

```
#include<csky_vdsp.h>

int8x4_t svfun2 (int8x4_t va,int8x4_t vb,int8x4_t vc)
{
        int8x4_t vc = va + vb; /* vector operation */
        return vc;
}
```

4.6.5 Intrinsic function interface naming rules

The function name of the intrinsic interface is basically consistent with the instruction name. If the instruction name contains ".", it will be replaced by "_" in the function name. For example, the intrinsic interface function name corresponding to the instruction padd.8 is padd_8. The parameter and return value types of the function are determined by the data type of the instruction operand. For example, the instruction padd.8 Rz, Rx, Ry, its function is to add the 8-bit wide 4-element integer vectors in two ordinary registers and put the result into the ordinary register rz. Therefore, the declaration of the function padd_8 is as follows:

```
int8x4_t padd_8(int8x4_t __a, int8x4_t __b)
```

The first parameter is of type int8x4_t, the second parameter is of type int8x4_t, and the return value is of type int8x4_t.

4.6.6 DSPv2 intrinsic interface

dspv2 instructions can be divided into the following parts:

- Integer addition, subtraction, and comparison instructions
- Integer shift instructions
- Other operation instructions

4.6.6.1 Integer addition, subtraction and comparison instructions

padd.t && psub.t

- int8x4_t padd_8 (int8x4_t, int8x4_t)
- int16x2_t padd_16 (int16x2_t, int16x2_t)

```
>>> Function Description: Vector addition
        Assuming parameters Vx, Vy, return value Vz
        Vz(i)=Vx(i)+Vy(i);                    i=0:(number-1)
```

- int8x4_t psub_8 (int8x4_t, int8x4_t)
- int16x2_t psub_16 (int16x2_t, int16x2_t)

```
>>> Function Description: Vector Subtraction
        Assuming parameters Vx, Vy, return value Vz
        Vz(i)=Vx(i)-Vy(i);                    i=0:(number-1)
```

padd.ts && psub.ts

- uint8x4_t padd_u8_s (uint8x4_t, uint8x4_t)
- uint16x2_t padd_u16_s (uint16x2_t, uint16x2_t)
- int8x4_t padd_s8_s (int8x4_t, int8x4_t)
- int16x2_t padd_s16_s (int16x2_t, int16x2_t)

```
>>> Function Description: Vector Saturation Addition
        Assume Vx and Vy are two parameters, Vz is the return value, and U/S indicates whether it is signed or not.
        signed=(T==S); (selected according to element U/S type)
        Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;
        Min=signed ? -2^(element_size-1) : 0;
        If Vx(i)+Vy(i)>Max                    Vz(i)=Max;
        Else if Vx(i)+Vy(i)<Min                    Vz(i)=Min;
        Else Vz(i)= Vx(i)+Vy(i);
        End                 i=0:(number-1)
```

- uint8x4_t psub_u8_s (uint8x4_t, uint8x4_t)
- uint16x2_t psub_u16_s (uint16x2_t, uint16x2_t)
- int8x4_t psub_s8_s (int8x4_t, int8x4_t)
- int16x2_t psub_s16_s (int16x2_t, int16x2_t)

```
>>> Function Description: Vector saturation subtraction
        Assume Vx and Vy are two parameters, Vz is the return value, and U/S indicates whether it is signed or not.
        signed=(T==S); (selected according to element U/S type)
        Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;
        Min=signed ? -2^(element_size-1) : 0;
        If Vx(i)-Vy(i)>Max                    Vz(i)=Max;
        Else if Vx(i)-Vy(i)<Min                    Vz(i)=Min;
        Else Vz(i)= Vx(i)-Vy(i);
        End                 i=0:(number-1)
```

paddh.t && psubh.t

- int8x4_t paddh_s8 (int8x4_t, int8x4_t)
- int16x2_t paddh_s16 (int16x2_t, int16x2_t)
- uint8x4_t paddh_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t paddh_u16 (uint16x2_t, uint16x2_t)

>>> Function description: Additive average

operation assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)=(Vx(i)+Vy(i))>>1; i=0:number-1 For U, right shift is

logical right shift For S, right shift is arithmetic right shift

- int8x4_t psubh_s8 (int8x4_t, int8x4_t)
- int16x2_t psubh_s16 (int16x2_t, int16x2_t)
- uint8x4_t psubh_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t psubh_u16 (uint16x2_t, uint16x2_t)

>>> Function description: Subtraction average

operation assumes Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit

Vz(i)=(Vx(i)-Vy(i))>>1; i=0:number-1 For U, right shift is

logical right shift For S, right shift is arithmetic right shift

pcmp[ne/hs/lt].t

- int8x4_t pcmpne_8 (int8x4_t, int8x4_t)
- int16x2_t pcmpne_16 (int16x2_t, int16x2_t)

>>> Function description: Vector elements are not equal.

Assume that Vx and Vy are two parameters and Vz is the return value.

If Vx(i)!=Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x4_t pcmphs_s8 (int8x4_t, int8x4_t)
- int16x2_t pcmphs_s16 (int16x2_t, int16x2_t)
- uint8x4_t pcmphs_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t pcmphs_u16 (uint16x2_t, uint16x2_t)

>>> Function Description: Vector elements are greater than or equal to

Assume Vx, Vy are two parameters, Vz is the return value

If Vx(i)>=Vy(i) Vz(i)=11…111;

Else Vz(i)=00…000;

i=0:number-1

- int8x4_t pcmplt_s8 (int8x4_t, int8x4_t)
- int16x2_t pcmplt_s16 (int16x2_t, int16x2_t)
- uint8x4_t pcmplt_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t pcmplt_u16 (uint16x2_t, uint16x2_t)

```
>>> Function description: vector element is less
        than assuming Vx, Vy are two parameters, Vz is the return value
        If Vx(i)<Vy(i) Vz(i)=11…111;
        Else Vz(i)=00…000;
        i=0:number-1
```

4.6.6.2 Integer shift instructions

pasri.t

- int16x2_t pasri_s16 (int16x2_t, const int)

```
>>> Function Description: Vector immediate arithmetic right shift
        Assume Vx, imm are two parameters, Vz is the return value
        Vz(i)=Vx(i)>>imm; i=0:(number-1)
        The range of imm is 1 ~ element_size
```

pasr.t

- int16x2_t pasr_s16 (int16x2_t, int)

```
>>> Function Description: Vector register arithmetic right shift
        Assume Vx, Rx are two parameters, Vz is the return value
        imm = Rx[4:0]
        Vz(i)=Vx(i)>>Rx; i=0:(number-1)
```

plsri.t

- uint16x2_t plsri_u16 (uint16x2_t, const int)

```
>>> Function Description: Vector immediate logical right shift
        Assume Vx, imm are two parameters, Vz is the return value
        Vz(i)=Vx(i)>>imm; i=0:(number-1)
        The range of imm is 1 ~ element_size
```

plsr.t

- uint16x2_t plsr_u16 (uint16x2_t, int)

>>> Function Description: Vector immediate logical right shift

Assume Vx, Rx are two parameters, Vz is the return value

imm = Rx[4:0]

Vz(i)=Vx(i)>>imm; i=0:(number-1)

plsli.t

- int16x2_t plsli_s16 (int16x2_t, const int)

>>> Function description: Vector immediate left shift

Assume Vx, imm are two parameters, Vz is the return value

Vz(i)=Vx(i)<<imm; i=0:(number-1)

The range of imm is 1 ~ element_size

plsl.t

- int16x2_t plsl_s16 (int16x2_t, int)

>>> Function Description: Vector immediate left shift

Assume Vx, Rx are two parameters, Vz is the return value

imm = Rx[4:0]

Vz(i)=Vx(i)<<imm; i=0:(number-1)

pasri.tr

- int16x2_t pasri_s16_r (int16x2_t, const int)

>>> Function description: The result of a vector immediate arithmetic right shift is rounded

Assume Vx, imm are two parameters, Vz is the return value

round=1<<(imm-1)

Vz(i)=(Vx(i)+round)>>imm; i=0:(number-1)

The range of imm is 1 ~ element_size

pasr.tr

- int16x2_t pasr_s16_r (int16x2_t, int)

>>> Function description: Vector immediate arithmetic right shift result is rounded.

Assume Vx and Rx are two parameters and Vz is the return value.

IF(Rx) == 0

round = 0

ELSE

round=1<<(imm-1)

(Continued on next page)

(Continued from previous page)

```
    imm = Rx[4:0]
    Vz(i)=(Vx(i)+round)>>imm; i=0:(number-1)
```

plsri.tr

- uint16x2_t plsri_u16_r (uint16x2_t, const int)

```
>>> Function description: The result of logical right shift of vector immediate value is rounded
    Assume Vx, imm are two parameters, Vz is the return value
    round=1<<(imm-1)
    Vz(i)=(Vx(i)+round)>>imm; i=0:(number-1) The range of imm is 1 ~
    element_size
```

plsr.tr

- uint16x2_t plsr_u16_r (uint16x2_t, int)

```
>>> Function description: The result of the vector immediate logical right shift is rounded.
    Assume that Vx and Rx are two parameters and Vz is the return value.
    IF(Rx) == 0
        round = 0
    ELSE
        round=1<<(imm-1)
    imm = Rx[4:0]
    Vz(i)=(Vx(i)+round)>>imm; i=0:(number-1)
```

plsli.ts

- int16x2_t plsli_s16_s (int16x2_t, const int)
- uint16x2_t plsli_u16_s (uint16x2_t, const int)

```
>>> Function description: vector immediate left shift saturation
    Assume Vx, imm are two parameters, Vz is the return
    value signed=(T==S); (select according to the element U/S type)
    Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
    Min=signed? -2^(element_size-1):0;
    If (Vx(i)<<imm)>Max Vz(i)=Max;
    Else if (Vx(i)<<imm)<Min Vz(i)=Min;
    Else Vz(i)= Vx(i)<<imm;                    i=0:(number-1)
    The range of imm is 1 ~ element_size
```

plsl.ts

- int16x2_t plsl_s16_s (int16x2_t, int)
- uint16x2_t plsl_u16_s (uint16x2_t, int)

```
>>> Function description: vector immediate left shift saturation
        Assume Vx, Rx are two parameters, Vz is the return value
        imm = Rx[4:0]
        signed=(T==S); (selected according to the element U/S type)
        Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
        Min=signed? -2^(element_size-1):0;
        If (Vx(i)<<imm)>Max Vz(i)=Max;
        Else if (Vx(i)<<imm)<Min Vz(i)=Min;
        Else Vz(i)= Vx(i)<<imm;                                  i=0:(number-1)
        The range of imm is 0 ~ element_size-1
```

4.6.6.3 Other operation instructions

pasx.t

- int16x2_t pasx_16 (int16x2_t, int16x2_t)

```
>>> Function description: Vector offset addition and
        subtraction Assume Vx and Vy are two parameters, Vz is the return value
        Vz(2i+1) = Vx(2i+1)+Vy(2i);
        Vz(2i)= Vx(2i)-Vy(2i+1); i=0:(number/2-1)
```

pasx.ts

- int16x2_t pasx_s16_s (int16x2_t, int16x2_t)
- uint16x2_t pasx_u16_s (uint16x2_t, uint16x2_t)

```
>>> Function description: Vector offset addition and
        subtraction Assume that Vx and Vy are two parameters, Vz is the return value, and U/S
        is the sign bit signed=(T==S); (select according to the element U/S type)
        Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
        Min=signed? -2^(element_size-1):0; If (Vx(2i+1)
        +Vy(2i))>Max Vz(2i+1)=Max;
        Else if (Vx(2i+1) +Vy(2i))<Min Vz(2i+1)=Min;
        Else Vz(2i+1) = Vx(2i+1)+Vy(2i);
        End i=0:(number/2-1)
        If (Vx(2i)-Vy(2i+1))>Max Vz(2i)=Max;
        Else if (Vx(2i)-Vy(2i+1))<Min Vz(2i)=Min;
        Else Vz(2i)= Vx(2i)-Vy(2i+1);
        End i=0:(number/2-1)
```

psax.t

- int16x2_t psax_16 (int16x2_t, int16x2_t)

>>> Function description: Vector offset subtraction and addition Assume that Vx and Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(2i+1)= Vx(2i+1)-Vy(2i);

Vz(2i)= Vx(2i)+Vy(2i+1); i=0:(number/2-1)

psax.ts

- int16x2_t psax_s16_s (int16x2_t, int16x2_t)
- uint16x2_t psax_u16_s (uint16x2_t, uint16x2_t)

>>> Function description: Vector offset subtraction and addition Assume Vx, Vy are two parameters, Vz is the return value, U/S is the sign bit signed=(T==S); (select according to the element U/S type)

Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;

Min=signed? -2^(element_size-1):0;

If (Vx(2i+1)-Vy(2i))>Max Vz(2i+1)=Max;

Else if (Vx(2i+1)-Vy(2i))<Min Vz(2i+1)=Min;

Else Vz(2i+1)= Vx(2i+1)-Vy(2i);

End i=0:(number/2-1)

If (Vx(2i)+Vy(2i+1))>Max Vz(2i)=Max;

Else if (Vx(2i)+Vy(2i+1))<Min Vz(2i)=Min;

Else Vz(2i)= Vx(2i)+Vy(2i+1);

End i=0:(number/2-1)

pasxh.t

- int16x2_t pasxh_s16(int16x2_t, int16x2_t)
- uint16x2_t pasxh_u16(uint16x2_t, uint16x2_t)

>>> Function description: Take the average value after vector misalignment addition and subtraction. Assume that Vx and Vy are two parameters, Vz is the return value, and U/S is the sign bit.

Vz(2i+1)=(Vx(2i+1)+Vy(2i))>>1;

Vz(2i)=(Vx(2i)-Vy(2i+1))>>1; i=0:(number/2-1) For U, right shift is logical right shift For S, right shift is arithmetic right shift

psaxh.t

- int16x2_t vsaxh_s16(int16x2_t, int16x2_t)
- uint16x2_t vsaxh_u16(uint16x2_t, uint16x2_t)

>>> Function description: Get the average value after vector misalignment subtraction and addition

Assume Vx, Vy are two parameters, Vz is the return value, and U/S is the sign bit

Vz(2i+1)=(Vx(2i+1)-Vy(2i)) >>1;

Vz(2i)=(Vx(2i)+Vy(2i+1)) >>1; i=0:(number/2-1) For U, right shift is logical right shift , For S, right shift is arithmetic right shift

pmax.t && pmin.t

- int8x4_t pmax_s8 (int8x4_t, int8x4_t)
- int16x2_t pmax_s16 (int16x2_t, int16x2_t)
- uint8x4_t pmax_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t pmax_u16 (uint16x2_t, uint16x2_t)

>>> Function description: vector element takes the maximum value

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=max((Vx(i),Vy(i)) ; i=0:number-1 max takes the larger value of the two elements

- int8x4_t pmin_s8 (int8x4_t, int8x4_t)
- int16x2_t pmin_s16 (int16x2_t, int16x2_t)
- uint8x4_t pmin_u8 (uint8x4_t, uint8x4_t)
- uint16x2_t pmin_u16 (uint16x2_t, uint16x2_t)

>>> Function description: Minimum value of vector elements

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=min((Vx(i),Vy(i)) ; i=0:number-1

min takes the smaller value of the two elements

pext.te

- int16x4_t pext_s8_e (int8x4_t)
- uint16x4_t pext_u8_e (uint8x4_t)

>>> Function description: Vector expansion

assumes Vx is the parameter, Vz is the return value, and U/S is the sign bit

Vz(i)=extend(Vx(i)); i=0:(number/2-1) extend Zero-extend or sign-extend the value to twice the element width according to U/S

pextx.te

- int16x4_t pextx_s8_e (int8x4_t)
- uint16x4_t pextx_u8_e (uint8x4_t)

>>> Function description: Vector interleaved

expansion assumes Vx is the parameter, Vz is the return value, U/S is the sign bit

Vz(3) = extend(Vx(3))

Vz(2) = extend(Vx(1))

Vz(1) = extend(Vx(2))

Vz(0) = extend(Vx(0))

extend Zero-extend or sign-extend the value to twice the element width according to U/S

pclipi.t

- int16x2_t pclipi_s16 (int16x2_t, const int)
- uint16x2_t pclipi_u16 (uint16x2_t, const int)

>>> Function description: Vector clipping to get saturated

value Assume Vx, imm4 are two parameters, Vz is the return value, U/S is the sign bit

U: Max=2^(imm4)-1, Min=0;

S: Max=2^(imm4-1)-1, Min=-2^(imm4-1);

Regardless of T is U/S, Vx(i) is always considered as a signed number If Vx(i)>Max Vz(i)=Max;

else if Vx(i)<Min Vz(i)=Min;

else Vz(i)=Vx(i);

end i=0:number-1

U:imm4 range is 0 ~ (element_size-1)

S:imm4 range is 1 ~ (element_size)

pclip.t

- int16x2_t pclip_s16 (int16x2_t, const int)
- uint16x2_t pclip_u16 (uint16x2_t, const int)

>>> Function description: Vector clipping to saturation value

Assume Vx, Ry are two parameters, Vz is the return value, U/S is the sign bit, imm4 = Ry[3:0]

U: Max=2^(imm4)-1, Min=0;

S: Max=2^(imm4-1)-1, Min=-2^(imm4-1); Regardless of T is

U/S, Vx(i) is always considered as a signed number If Vx(i)>Max Vz(i)=Max;

else if Vx(i)<Min Vz(i)=Min;

else Vz(i)=Vx(i);

end i=0:number-1

U:imm4 range is 0 ~ (element_size-1)

S:imm4 range is 1 ~ (element_size)

pabs.ts

- int8x4_t pabs_s8_s(int8x4_t)

- int16x2_t pabs_s16_s(int16x2_t)

>>> Function description: vector element saturation absolute

value Assume Vx is the parameter, Vz is the return value, U/S is the sign bit

If Vx(i)==-2^(element_size-1) Vz(i)= 2^(element_size-1)-1;

Else Vz(i)=abs(Vx(i));

End i=0:number-1

pneg.ts

- int8x4_t pneg_s8_s (int8x4_t)
- int16x2_t pneg_s16_s (int16x2_t)

>>> Function description: Vector element saturation is negative.

Assume Vx is the parameter and Vz is the return value.

If Vx(i)==-2^(element_size-1) Vz(i)= 2^(element_size-1)-1;

Else Vz(i)=-Vx(i));

End i=0:number-1

pmul.t

- int32x2_t pmul_s16 (int16x2_t, int16x2_t)
- uint32x2_t pmul_u16 (uint16x2_t, uint16x2_t)

>>> Function Description: Vector element extension multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Vz(i)=(Vx(i)*Vy(i))[2*element_size-1:0];

The multiplication result takes full precision, that is, twice the element width

pmulx.t

- int32x2_t pmulx_s16 (int16x2_t, int16x2_t)
- uint32x2_t pmulx_u16 (uint16x2_t, uint16x2_t)

>>> Function Description: Vector element interleaved extension multiplication

Assume Vx, Vy are two parameters, Vz is the return value

Vz(1) = Vx(1) X Vy(0)

Vz(0) = Vx(0) X Vy(1)

The multiplication result takes full precision, that is, twice the element width

prmul.t

- int32x2_t prmul_s16 (int16x2_t, int16x2_t)

>>> Function Description: Vector extension with saturated fractional multiplication

Assume Vx, Vy are two parameters, Vz is the return value

If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))

Vz(i)= 2^(2*element_size-1)-1;

Else Vz(i)= (Vx(i)*Vy(i))[2*element_size-1:0];

(The multiplication result takes full precision, which is twice the element width)

End i=0:(number-1)

prmulx.t

- int32x2_t prmulx_s16 (int16x2_t, int16x2_t)

>>> Function Description: Vector interleaved extension with saturated fractional multiplication

Assume Vx, Vy are two parameters, Vz is the return value

IF(Vx(1) == 0x8000 && Vy(0) == 0x8000)

Vz(1) = 0x7FFFFFFF

ELSE

Vz(1) = (Vx(1) X Vy(0)) << 1

IF(Vx(0) == 0x8000 && Vy(1) == 0x8000)

Vz(0) = 0x7FFFFFFF

ELSE

Vz(0) = (Vx(0) X Vy(1)) << 1

prmul.th

- int16x2_t prmul_s16_h (int16x2_t, int16x2_t)

>>> Function description: Vector extension with saturated fractional multiplication to take the high half

Assume Vx, Vy are two parameters, Vz is the return value

If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))

Vz(i)= 2^(element_size-1)-1;

Else Vz(i)= Vx(i)*Vy(i)[2*element_size-2:element_size-1];

End i=0:(number-1)

prmul.t.rh

- int16x2_t prmul_s16_rh (int16x2_t, int16x2_t)

>>> Function description: Vector expansion with saturation fractional multiplication result with rounding to take the high

half Assume Vx, Vy are two parameters, Vz is the return value

round=1<<(element_size-2);

If (Vx(i)== -2^(element_size-1)) && (Vy(i)== -2^(element_size-1))

Vz(i)= 2^(element_size-1)-1;

(Continued on next page)

(Continued from previous page)

```
        Else Vz(i)= (Vx(i)*Vy(i) + round)[2*element_size-2:element_size-1];
        End          i=0:(number-1)
```

prmulx.th

- int16x2_t prmulx_s16_h (int16x2_t, int16x2_t)

```
>>> Function description: Vector interleaved extension with saturated fractional multiplication
        Assume that Vx and Vy are two parameters and Vz is the return value
    IF(Vx(1) == 0x8000 && Vy(0) == 0x8000)
        Vz(1) = 0x7FFF
    ELSE
        Vz(1) = (Vx(1) X Vy(0))[2*element_size-2:element_size-1]
    IF(Vx(0) == 0x8000 && Vy(1) == 0x8000)
        Vz(0) = 0x7FFF
    ELSE
        Vz(0) = (Vx(0) X Vy(1))[2*element_size-2:element_size-1]
```

prmulx.t.rh

- int16x2_t prmulx_s16_h (int16x2_t, int16x2_t)

```
>>> Function Description: Vector interleaved extension with saturated fractional multiplication
        Assume Vx, Vy are two parameters, Vz is the return value
    round=1<<(element_size-2);
    IF(Vx(1) == 0x8000 && Vy(0) == 0x8000)
        Vz(1) = 0x7FFF
    ELSE
        Vz(1) = (Vx(1) X Vy(0) + round)[2*element_size-2:element_size-1]
    IF(Vx(0) == 0x8000 && Vy(1) == 0x8000)
        Vz(0) = 0x7FFF
    ELSE
        Vz(0) = (Vx(0) X Vy(1) + round)[2*element_size-2:element_size-1]
```

psabsa.t

- uint32_t psabsa_u8(uint8x4_t, uint8x4_t)

```
>>> Function description: Take the absolute value after subtracting vector , Then add
        elements. Assume Vx, Vy are parameters and Vz is the return value.
    Vz = 0;
    Vz= Vz+abs(Vx(i)-Vy(i)) ; i=0:number-1
```

psabsaa.t

- uint32_t psabsaa_u8(uint32_t, uint8x4_t, uint8x4_t)

>>> Function description: Take the absolute value after subtracting vector , Then add elements. Assume Rz, Vx, Vy are parameters and Vz is the return value.

Vz = Rz;

Vz= Vz+abs(Vx(i)-Vy(i)) ; i=0:number-1

4.7 minilibc

4.7.1 math

4.7.1.1 Basic operations

- **fabs, fabsf**

Defined in header <math.h>

float	fabsf(float arg);	(1)	(since C99)
double	fabs(double arg);	(2)	

1-2) Calculate the absolute value of the parameter *arg*

parameter

arg - floating point argument

Return Value

Returns the absolute value of the argument *arg (|arg|)*

- **fmod, fmodf**

Defined in header <math.h>

float	fmodf(float x, **float** y);	(1)	(since C99)
double	fmod(double x, **double** y);	(2)	

1-2) Calculate the remainder of the parameter *x*/*y* , using the formula *x* - *n* * *y* is obtained, n is rounded towards **0** , and the sign of the returned value is the same as that *of x* .

parameter

x, y – floating point parameters

Return Value

Returns the remainder of the parameter *x*/*y*

- **remainder, remainderf**

Defined in header <math.h>

float	remainderf(float x, **float** y);	(1) (since C99)
double	remainder(double x, **double** y);	(2) (since C99)

1-2) Calculate the remainder of the parameter *x*/*y* , using the formula *x* - *n* * *y* is obtained, n is rounded to the nearest integer, and the sign of the returned value cannot be guaranteed to be the same as that of *x* .

parameter

x, *y* – floating point parameters

Return Value

Returns the remainder of the parameter *x*/*y*

- **remquo, remquof**

Defined in header <math.h>

float	remquof(float x, **float** y, **int** *quo);	(1) (sinceÿ
ÿC99)		
double	remquo(double x, **double** y, **int** *quo);	(2) (sinceÿ
ÿC99)		

1-2) *Similar to remainder(),* calculate the remainder of parameter *x*/*y* and store the quotient in parameter *quo* .

parameter

x, *y* – floating point parameters

quo - quotient

Return Value

Returns the remainder of parameter *x*/*y* and stores the quotient in parameter *quo* .

- **fma, fmaf**

Defined in header <math.h>

float	fmaf(float x, **float** y, **float** z); fma(double x, **double** y,	(1)	(since C99)
double	**double** z);	(2)	(since C99)

1-2) Calculate the result of the parameter *(x * y) + z* .

parameter

x, y, z – floating point parameters

Return Value

Returns the result of the parameter *(x * y) + z* .

- **fmax, fmaxf**

Defined in header <math.h>

float	fmaxf(float x, **float** y);	(1)	(since C99)
double	fmax(double x, **double** y);	(2)	(since C99)

1-2) Returns the larger of its two arguments.

parameter

x, y – floating point parameters

Return Value

Returns the larger of its two arguments.

- **fmin, fminf**

Defined in header <math.h>

float	fminf(float x, **float** y);	(1)	(since C99)
double	fmin(double x, **double** y);	(2)	(since C99)

1-2) Returns the smaller of its two arguments.

parameter

x, y – floating point parameters

Return Value

Returns the smaller of its two arguments.

- **fdim, fdimf**

Defined in header <math.h>

float	fdimf(float x, **float** y);	(1)	(since C99)
double	fdim(double x, **double** y);	(2)	(since C99)

1-2) Returns the positive difference of two parameters. If x > y, returns x - y, otherwise returns +0.

parameter

x, y – floating point parameters

Return Value

Returns the positive difference of its two arguments.

- **nan, nanf**

Defined in header <math.h>

float	**nanf(const char** *unused);	(1)	(since C99)
double	**nan(const char** *unused);	(2)	(since C99)

1-2) Return not-a-number value

parameter

unused - constant character pointer parameter

Return Value

Returns not-a-number value

4.7.1.2 Exponential Operation

- **exp, expf**

Defined in header <math.h>

float	expf(float arg); (1)	(since C99)
double	exp(double arg); (2)	

1-2) Calculate e raised to the power of arg

parameter

arg - floating point argument

Return Value

Returns e raised to the power of arg.

- **exp2, exp2f**

Defined in header <math.h>

float	exp2f(float n);	(1)	(since C99)
double	exp2(double n);	(2)	(since C99)

1-2) Calculate 2 to the power of n

parameter

n - floating point parameter

Return Value

Returns 2 raised to the power of n.

- **expm1, expm1f**

Defined in header <math.h>

float	expm1f(float arg);	(1)	(since C99)
double	expm1(double arg);	(2)	(since C99)

1-2) Calculate e raised to the power of arg - 1

parameter

arg - floating point argument

Return Value

Returns e raised to the power of arg - 1.

- **log, logf**

Defined in header <math.h>

float	logf(float arg); (1)		(since C99)
double	log(double arg); (2)		

1-2) Calculate the logarithm of arg with base e as the real number

parameter

arg - floating point argument

Return Value

Returns the logarithm of arg to base e.

- **log10, log10f**

Defined in header <math.h>

float	log10f(float arg);	(1)	(since C99)
double	log10(double arg);	(2)	

1-2) Calculate the logarithm of arg to base 10

parameter

arg - floating point argument

Return Value

Returns the base 10 logarithm of arg.

- **log1p, log1pf**

Defined in header <math.h>

float	log1pf(float arg);	(1)	(since C99)
double	log1p(double arg);	(2)	(since C99)

1-2) Calculate the logarithm of 1+arg with base e as the real number

parameter

arg - floating point argument

Return Value

Returns the logarithm of 1+arg to base e.

- **log2, log2f**

Defined in header <math.h>

float	log2f(float arg); log2(double	(1)	(since C99)
double	arg);	(2)	(since C99)

1-2) Calculates the logarithm of arg to base 2.

parameter

arg - floating point argument

Return Value

Returns the base 2 logarithm of arg.

4.7.1.3 Exponentiation

- **sqrt, sqrtf**

Defined in header <math.h>

float	sqrtf(float arg);	(1)	(since C99)
double	sqrt(double arg);	(2)	

1-2) Calculate the square root of arg

parameter

arg - floating point argument

Return Value

Returns the square root of arg.

- **cbrt, cbrtf**

Defined in header <math.h>

float	cbrtf(float arg);	(1)	(since C99)
double	cbrt(double arg);	(2)	(since C99)

1-2) Calculate the cube root of arg

parameter

arg - floating point argument

Return Value

Returns the cube root of arg.

- **hypot, hypotf**

Defined in header <math.h>

float	hypotf(float x, **float** y);	(1)	(since C99)
double	hypot(double x, **double** y);	(2)	(since C99)

1-2) Calculate the square root of the sum of the squares of x and y

parameter

x - the floating point parameter

y - floating point parameter

Return Value

Returns the square root of the sum of the squares of x and y.

- **pow, powf**

Defined in header <math.h>

float powf(float base, **float** exponent);	(1)	(since C99)
double pow(double base, **double** exponent);	(2)	

1-2) Calculate the exponent of base

parameter

base - floating point base parameter

exponent - floating point exponent parameter

Return Value

Returns base raised to the power of exponent.

4.7.1.4 Trigonometric and Hyperbolic Operations

- **sin, sinf**

Defined in header <math.h>

float **double**	sinf(float arg); (1) sin(double arg); (2)	(since C99)

1-2) Calculate the sine of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the sine of arg (radians).

- **cos, cosf**

Defined in header <math.h>

float **double**	cosf(float arg); (1) cos(double arg); (2)	(since C99)

1-2) Calculate the cosine of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the cosine of arg (radians).

- **tan, tanf**

Defined in header <math.h>

float **double**	tanf(float arg); (1) tan(double arg); (2)	(since C99)

1-2) Calculate the tangent of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the tangent of arg (radians).

- **asin, asinf**

Defined in header <math.h>

float	asinf(float arg);	(1)	(since C99)
double	asin(double arg);	(2)	

1-2) Calculate the arc sine of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the arc sine of arg (radians).

- **acos, acosf**

Defined in header <math.h>

float	acosf(float arg); acos(double	(1)	(since C99)
double	arg);	(2)	

1-2) Calculate the arc cosine of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the arccosine of arg (radians).

- **atan, atanf**

Defined in header <math.h>

float	atanf(float arg);	(1)	(since C99)
double	atan(double arg);	(2)	

1-2) Calculate the arc tangent of arg (radians)

parameter

arg - floating point representation of the angle

Return Value

Returns the arc tangent of arg (radians).

- **atan2, atan2f**

Defined in header <math.h>

float	atan2f(float y, **float** x);	(1)	(since C99)
double	atan2(double y, **double** x);	(2)	

1-2) Calculate the inverse tangent of y/x (radians)

parameter

y - a floating point representation of the angle

x - a floating point representation of the angle

Return Value

Returns the arc tangent of y/x (radians).

- **sinh, sinhf**

Defined in header <math.h>

float	sinhf(float arg);	(1)	(since C99)
double	sinh(double arg);	(2)	

1-2) Calculate the hyperbolic sine of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the hyperbolic sine of arg.

- **cosh, coshf**

Defined in header <math.h>

float	coshf(float arg);	(1)	(since C99)
double	cosh(double arg);	(2)	

1-2) Calculate the hyperbolic cosine of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the hyperbolic cosine of arg.

- **tanh, tanhf**

Defined in header <math.h>

float	tanhf(float arg);	(1)	(since C99)
double	tanh(double arg);	(2)	

1-2) Calculate the hyperbolic tangent of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the hyperbolic tangent of arg.

- **asinh, asinhf**

Defined in header <math.h>

float	asinhf(float arg);	(1)	(since C99)
double	asinh(double arg);	(2)	(since C99)

1-2) Calculates the inverse hyperbolic sine of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the inverse hyperbolic sine of arg.

- **acosh, acoshf**

Defined in header <math.h>

float	acoshf(float arg);	(1)	(since C99)
double	acosh(double arg);	(2)	(since C99)

1-2) Calculates the inverse hyperbolic cosine of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the inverse hyperbolic cosine of arg.

- **atanh, atanhf**

Defined in header <math.h>

float	atanhf(float arg);	(1)	(since C99)
double	atanh(double arg);	(2)	(since C99)

1-2) Calculate the inverse hyperbolic tangent of arg

parameter

arg - floating point representation of the angle

Return Value

Returns the inverse hyperbolic tangent of arg.

4.7.1.5 Error and Gamma Operation

- **erf, erff**

Defined in header <math.h>

float	erff(float arg); (1)		(since C99)
double	erf(double arg); (2)		(since C99)

- **erfc, erfcf**

Defined in header <math.h>

float	erfcf(float arg); (1)		(since C99)
double	erfc(double arg); (2)		(since C99)

- **tgamma, tgammaf**

Defined in header <math.h>

float	tgammaf(float arg); tgamma(double	(1)	(since C99)
double	arg);	(2)	(since C99)

4.7.1.6 Floating-point operations

- **ldexp, ldexpf**

Defined in header <math.h>

float	ldexpf(float arg, **int** exp); ldexp(double arg, **int**	(1)	(since C99)
double	exp);	(2)	

1-2) Calculate arg * 2 to the power of exp

parameter

arg - floating point argument

exp - integer parameter

Return Value

Returns arg * 2 raised to the power of exp.

- **scalbn, scalbnf**

Defined in header <math.h>

float	scalbnf(float arg, **int** exp);	(1) (since C99)
double	scalbn(double arg, **int** exp);	(2) (since C99)
float	scalblnf(float arg, **long** exp); scalbln(double arg,	(5) (since C99)
double	**long** exp);	(6) (since C99)

1-2, 5-6) Calculate arg * FLT_RADIX raised to the power of exp

parameter

arg - floating point argument

exp - integer parameter

Return Value

Returns arg * FLT_RADIX raised to the power of exp.

- **ilogb, ilogbf**

Defined in header <math.h>

int ilogbf(float arg);		(1)	(since C99)
int ilogb(double arg);		(2)	(since C99)

1-2) Extracts the value of the unbiased exponent from the floating-point argument arg and returns it as a signed integer value.

parameter

arg - floating point argument

Return Value

Extracts the value of the unbiased exponent from the floating-point argument arg and returns it as a signed integer value.

- **logb, logbf**

Defined in header <math.h>

float	logbf(float arg);	(1)	(since C99)
double	logb(double arg);	(2)	(since C99)

1-2) Extracts the value of the unbiased base-independent exponent from the floating-point argument arg and returns it as a floating-point value.

parameter

arg - floating point argument

Return Value

Extracts the value of the unbiased base-independent exponent from the floating-point argument arg and returns it as a floating-point value.

- **frexp, frexpf**

Defined in header <math.h>

float	frexpf(float arg, **int*** exp);	(1)	(since C99)
double	frexp(double arg, **int*** exp);	(2)	

1-2) Decomposes the given floating point value x into normalized fractions and integer powers of 2.

parameter

arg - floating point argument

exp - integer parameter

Return Value

Factors the given floating point value x into its normalized fraction and integer powers of 2.

- **modf, modff**

Defined in header <math.h>

float	modff(float arg, **float*** iptr);	(1)	(since C99)
double	modf(double arg, **double*** iptr);	(2)	

1-2) Decomposes the given floating-point value arg into its integer and fractional parts, each of which has the same type and sign as arg.

parameter

arg - floating point argument

iptr - pointer to a floating point value, used to store the integer part

Return Value

Decomposes the given floating-point value arg into its integral and fractional parts, each of which has the same type and sign as arg.

- **nextafter, nextafterf**

Defined in header <math.h>

float	nextafterf(float from, **float** to);	(1)	(since C99)
double	nextafter(double from, **double** to);	(2)	(since C99)

1-2) First, convert the two arguments to the type of the function, and then return the next representable value from in the direction of to. If from is equal to to, then Return to.

parameter

from - floating point parameter

to - the floating point argument

Return Value

First, the two arguments are converted to the type of the function, and then the next representable value from is returned in the direction of to.

Then return to.

- **copysign, copysignf**

Defined in header <math.h>

float	copysignf(float x, **float** y); copysign(double x,	(1)	(since C99)
double	**double** y);	(2)	(since C99)

1-2) Compose a floating-point value using the magnitude of x and the sign of y.

parameter

x - the floating point parameter

y - floating point parameter

Return Value

Compose a floating-point value using the magnitude of x and the sign of y.

4.7.1.7 Approximate operations

- **ceil, ceilf**

Defined in header <math.h>

float	ceilf(float arg);	(1)	(since C99)
double	ceil(double arg);	(2)	

1-2) Calculate the smallest integer value not less than arg

parameter

arg - floating point argument

Return Value

Returns the smallest integer value not less than arg.

- **floor, floorf**

Defined in header <math.h>

float	floorf(float arg);	(1)	(since C99)
double	floor(double arg);	(2)	

1-2) Calculate the maximum integer value not greater than arg

parameter

arg - floating point argument

Return Value

Returns the largest integer value not greater than arg.

- **round, roundf**

Defined in header <math.h>

float	roundf(float arg);	(1)	(since C99)
double	round(double arg);	(2)	(since C99)
long	lroundf(float arg);	(5)	(since C99)
long	lround(double arg);	(6)	(since C99)
long long	llroundf(float arg); (9)		(since C99)
long long	llround(double arg); (10)		(since C99)

1-2) Calculate the floating point number closest to arg

5-6, 9-10) Calculate the integer closest to arg

parameter

arg - floating point argument

Return Value

Returns the value closest to arg.

- **trunc, truncf**

Defined in header <math.h>

float	truncf(float arg);	(1)	(since C99)
double	trunc(double arg);	(2)	(since C99)

1-2) Computes the largest integer whose magnitude is not greater than arg.

parameter

arg - floating point argument

Return Value

Returns the largest integer value not greater than arg.

- **nearbyint, nearbyintf**

Defined in header <math.h>

float	nearbyintf(float arg); nearbyint(double	(1)	(since C99)
double	arg);	(2)	(since C99)

1-2) Rounds the floating point argument arg to an integer value in floating point format.

parameter

arg - floating point argument

Return Value

Returns the integer value of the parameter arg.

- **rint, rintf**

Defined in header <math.h>

float rintf(float arg);		(1)	(since C99)
double rint(double arg); **long** lrintf(float		(2)	(since C99)
arg);		(5)	(since C99)
long lrint(double arg);		(6)	(since C99)
long long llrintf(float arg);	(9)	(since C99)	
long long llrint(double arg);	(10)	(since C99)	

1-2) Rounds the floating point argument arg to an integer value in floating point format.

5-6, 9-10) Rounds the floating point argument arg to an integer value in integer format.

parameter

arg - floating point argument

Return Value

Returns the integer value of the parameter arg.

Chapter 5 XuanTie 900 Series CPU Programming

The XuanTie 900 series CPU is a processor developed based on the RISC-V architecture. This chapter mainly introduces the RISC-V related Architecture-specific usage, such as instruction set selection, assembly programming, and instruction intrinsic interfaces.

This chapter contains the following parts:

- How to add options for processors
- Xuantie small size runtime library *libcc-rt*
- *pthread* multithreading (currently only supported in the *GNU* toolchain)
- *C/C++* language extensions
- Dynamic linker name
- General purpose coprocessor extension *C intrinsic* interface
- Instructions for using *RISC-V Vector*

5.1 How to add options for processors

Currently, Xuantie has developed a variety of RISC-V architecture processors, which are uniformly compiled and developed using the Xuantie RISC-V tool chain and are compiled through different options. Generates target code for a specific processor.

The XuanTie LLVM compiler supports the generation of target files for the corresponding model through the -mcpu=<CPU model> option. It is also compatible with the methods of specifying -march, -mabi and -mtune respectively. -march can be obtained through "-mcpu=<CPU model> -v", and -mabi and -mtune are shown in Table 5.1 . It is recommended to use the -mcpu option.

Starting from V2.6.0, the compiler supports -mcpu=<CPU model> to select all CPU-related features. It is also compatible with specifying -march, -mabi and -mtune separately. It is recommended to use the -mcpu option.

For the selection of -march and -mabi, since the V2.2.0 (GNU) version, the implementation of RVV Instrinsic has been upgraded from SIMD to VECTOR, the corresponding options are also different, and since the V3.0.0 (GNU) version, the default ISA-SPEC has been updated from 2.2 to 20191213, and some archs have been adjusted, so the following is divided into three tables. The specific correspondence is shown in Table 5.1 after (including) V3.0.0 (GNU), in Table 5.2 from V2.2.0 (GNU) to V2.10.2 (GNU) (where c908 is supported starting from V2.4 (GNU)), and before V2.2.0 (GNU) as shown in Table 5.3 .

Table 5.1: Correspondence between CPU and options (V3.0.0 (GNU) and later versions)

	CPU	-march	-mabi -mtune
e902 Series	e902	rv32ec_zicsr_zifencei_xtheadse	ilp32e e902
	e902m	rv32emc_zicsr_zifencei_xtheadse	ilp32e e902

Continued on next page

Table 5.1 – Continued from previous page

	CPU	-march	-mabi -mtune
	e902t	rv32ec_zicsr_zifencei_xtheadse	ilp32e e902
	e902mt	rv32emc_zicsr_zifencei_xtheadse	ilp32e e902
e906 Series	e906	rv32imac_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32 e906
	e906f	rv32imafc_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32f e906
	e906fd	rv32imafdc_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32d e906
	e906p	rv32imacp_zicntr_zicsr_zifencei_zihpm_zpsfoperand_xtheade	ilp32 e906
	e906fp	rv32imafc_pzpsfoperand_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32f e906
	e906fdp	rv32imafdc_pzpsfoperand_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32d e906
e907 Series	e907	rv32imac_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32 e907
	e907f	rv32imafc_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32f e907
	e907fd	rv32imafdc_zicntr_zicsr_zifencei_zihpm_xtheade	ilp32d e907
	e907p	rv32imacp_zicntr_zicsr_zifencei_zihpm_zpsfoperand_xtheade	ilp32 e907
	e907fp	rv32imafcp_zicntr_zicsr_zifencei_zihpm_zpsfoperand_xtheade	ilp32f e907
	e907fdp	rv32imafdcp_zicntr_zicsr_zifencei_zihpm_zpsfoperand_xtheade	ilp32d e907
c906 Series	c906	rv64imac_zicntr_zicsr_zifencei_zihpm_xtheadc	lp64 c906v
	c906fd	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc	lp64d c906v
	c906fdv	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc_xtheadvec tor	lp64d c906v
c907 Series	c907	rv64imac_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei_z ihintntl_zihintpause_zihpm_zawrs_zca_zcb_zba_zbb_zbc_zbs_ssc ofpmf_sstc_svinval_svnapot_svpbmt_xtheadc	lp64 c907
	c907fd	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei _zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_z cd_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot_svpbmt_xthe adc	lp64d c907
	c907fdv	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_ zcd_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscofpmf_sstc_svi nval_svnapot_svpbmt_xtheadc_xtheadvdot	lp64d c907
	c907fdvm	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_ zcd_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscofpmf_sstc_svi nval_svnapot_svpbmt_xtheadc_xtheadmatrix_xtheadvdot	lp64d c907
	c907-rv32	rv32imac_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei_z ihintntl_zihintpause_zihpm_zawrs_zca_zcb_zba_zbb_zbc_zbs_ssc ofpmf_sstc_svinval_svnapot_svpbmt_xtheadc	ilp32 c907
	c907fd-rv32	rv32imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei _zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_z cd_zcf_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot_svpbmt_ xtheadc	ilp32d c907
	c907fdv-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	ilp32d c907

Continued on next page

Table 5.1 – Continued from previous page

	CPU	-march	-mabi -mtune
		i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_	
		zcd_zcf_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscofpmf_sstc	
		_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	
	c907fdvm-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	ilp32d c907
		i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_	
		zcd_zcf_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscofpmf_sstc	
		_svinval_svnapot_svpbmt_xtheadc_xtheadmatrix_xtheadvdot	
c908 Series	c908	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	lp64d c908
		pause_zihpm_zfh_zba_zbb_zbc_zbs_sstc_svinval_svnapot_svpbmt_	
		xtheadc	
	c908v	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	lp64d c908
		tpause_zihpm_zfh_zba_zbb_zbc_zbs_zvfh_sstc_svinval_svnapot_s	
		vpbmt_xtheadc_xtheadvdot	
	c908-rv32	rv32imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	ilp32d c908
		pause_zihpm_zfh_zba_zbb_zbc_zbs_sstc_svinval_svnapot_svpbmt_	
		xtheadc	
	c908v-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	ilp32d c908
		tpause_zihpm_zfh_zba_zbb_zbc_zbs_zvfh_sstc_svinval_svnapot_s	
		vpbmt_xtheadc_xtheadvdot	
	c908i	rv64imac_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihintpa	lp64 c908
		use_zihpm_zba_zbb_zbc_zbs_sstc_svinval_svnapot_svpbmt_xthead	
		c	
c910 Series	c910	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc	lp64d c910
	c920	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc_xtheadvec	lp64d c910
		tor	
	r910	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc	lp64d c910
	r920	rv64imafdc_zicntr_zicsr_zifencei_zihpm_zfh_xtheadc_xtheadvec	lp64d c910
		tor	
c910v2 Series	c910v2	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei	lp64d c910
		_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_z	
		cd_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot_svpbmt_xthe	
		adc	
	c920v2	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	lp64d c910
		i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_	
		zcd_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscofpmf_sstc_svi	
		nval_svnapot_svpbmt_xtheadc_xtheadvdot	
c910v3 Series	c910v3	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei	lp64d c910
		_zihintntl_zihintpause_zihpm_zimop_zawrs_zfa_zfbfmin_zfh_zca	
		_zcb_zcd_zcmop_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot	
		_svpbmt_xtheadc	
	c920v3	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	lp64d c910

Continued on next page

Table 5.1 – Continued from previous page

	CPU	-march	-mabi -mtune
		i_zihintntl_zihintpause_zihpm_zimop_zawrs_zfa_zfbfmin_zfh_zc	
		a_zcb_zcd_zcmop_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscof	
		pmf_sstc_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	
	c910v3-cp	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei	lp64d c910
		_zihintntl_zihintpause_zihpm_zimop_zawrs_zfa_zfbfmin_zfh_zca	
		_zcb_zcd_zcmop_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot	
		_svpbmt_xtheadcmo_xtheadsync_xxtccef_xxtccei	
	c920v3-cp	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	lp64d c910
		i_zihintntl_zihintpause_zihpm_zimop_zawrs_zfa_zfbfmin_zfh_zc	
		a_zcb_zcd_zcmop_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwma_zvfh_sscof	
		pmf_sstc_svinval_svnapot_svpbmt_xtheadcmo_xtheadsync_xxtccef	
		_xxtccei_xxtccev	
r908 Series	r908	rv64imac_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihintpa	lp64 c908
		use_zihpm_zimop_zca_zcb_zcmop_zba_zbb_zbc_zbs_sstc_svinval_s	
		vnapot_svpbmt_xtheadc_xtheadfpp	
	r908fd	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	lp64d c908
		pause_zihpm_zimop_zca_zcb_zcd_zcmop_zfh_zba_zbb_zbc_zbs_sstc	
		_svinval_svnapot_svpbmt_xtheadc_xtheadfpp	
	r908fdv	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	lp64d c908
		tpause_zihpm_zimop_zca_zcb_zcd_zcmop_zfh_zba_zbb_zbc_zbs_zvf	
		h_sstc_svinval_svnapot_svpbmt_xtheadc_xtheadfpp_xtheadvdot	
	r908-rv32	rv32imac_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihintpa	ilp32 c908
		use_zihpm_zimop_zca_zcb_zcmop_zba_zbb_zbc_zbs_sstc_svinval_s	
		vnapot_svpbmt_xtheadc_xtheadfpp	
	r908fd-rv32	rv32imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	ilp32d c908
		pause_zihpm_zimop_zca_zcb_zcd_zcf_zcmop_zfh_zba_zbb_zbc_zbs_	
		sstc_svinval_svnapot_svpbmt_xtheadc_xtheadfpp	
	r908fdv-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	ilp32d c908
		tpause_zihpm_zimop_zca_zcb_zcd_zcf_zcmop_zfh_zba_zbb_zbc_zbs	
		_zvfh_sstc_svinval_svnapot_svpbmt_xtheadc_xtheadfpp_xtheadvd	
		ot	
	r908-cp	rv64imac_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihintpa	lp64 c908
		use_zihpm_zimop_zca_zcb_zcmop_zba_zbb_zbc_zbs_sstc_svinval_s	
		vnapot_svpbmt_xtheadcmo_xtheadfpp_xtheadsync_xxtccei	
	r908fd-cp	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	lp64d c908
		pause_zihpm_zimop_zca_zcb_zcd_zcmop_zfh_zba_zbb_zbc_zbs_sstc	
		_svinval_svnapot_svpbmt_xtheadcmo_xtheadfpp_xtheadsync_xxtcc	
		ef_xxtccei	
	r908fdv-cp	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	lp64d c908
		tpause_zihpm_zimop_zca_zcb_zcd_zcmop_zfh_zba_zbb_zbc_zbs_zvf	
		h_sstc_svinval_svnapot_svpbmt_xtheadcmo_xtheadfpp_xtheadsync	

Continued on next page

Table 5.1 – Continued from previous page

	CPU	-march	-mabi -mtune
		_xxtccef_xxtccei_xxtccev	
	r908-cp-rv32	rv32imac_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihintpa	ilp32 c908
		use_zihpm_zimop_zca_zcb_zcmop_zba_zbb_zbc_zbs_sstc_svinval_s	
		vnapot_svpbmt_xtheadcmo_xtheadfpp_xtheadsync_xxtccei	
	r908fd-cp-rv32	rv32imafdc_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihint	ilp32d c908
		pause_zihpm_zimop_zca_zcb_zcd_zcf_zcmop_zfh_zba_zbb_zbc_zbs_	
		sstc_svinval_svnapot_svpbmt_xtheadcmo_xtheadfpp_xtheadsync_x	
		xtccef_xxtccei	
	r908fdv-cp-rv32 rv32	imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	ilp32d c908
		tpause_zihpm_zimop_zca_zcb_zcd_zcf_zcmop_zfh_zba_zbb_zbc_zbs	
		_zvfh_sstc_svinval_svnapot_svpbmt_xtheadcmo_xtheadfpp_xthead	
		sync_xxtccef_xxtccei_xxtccev	
c920v2.c908v series	c920v2.c908v	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicsr_zifencei_zihin	lp64d c920v2.c908v
		tpause_zihpm_zfh_zba_zbb_zbc_zbs_zvfh_sstc_svinval_svnapot_s	
		vpbmt_xtheadc_xtheadvdot	

Remark:

1. **The default -misa-spec of the XuanTie GNU compiler for versions V3.0.0 and above is 20191213.**
2. **Extend XTheadVector to be compatible with intrinsic 0p10. The compatible option is -mrvv-v0p10-compatible.**

Table 5.2: Correspondence between CPU and options (V2.2.0 (GNU) to V2.10.2 (GNU))

	CPU	-march	-mabi -mtune
e902 Series	e902	rv32ecxtheadse	ilp32e e902
	e902m	rv32emcxtheadse	ilp32e e902
	e902t	rv32ecxtheadse	ilp32e e902
	e902mt	rv32emcxtheadse	ilp32e e902
e906 Series	e906	rv32imacxtheade	ilp32 e906
	e906f	rv32imafcxtheade	ilp32f e906
	e906fd	rv32imafdcxtheade	ilp32d e906
	e906p	rv32imacpzpsfoperand_xtheade	ilp32 e906
	e906fp	rv32imafcpzpsfoperand_xtheade	ilp32f e906
	e906fdp	rv32imafdcpzpsfoperand_xtheade	ilp32d e906
e907 Series	e907	rv32imacxtheade	ilp32 e907
	e907f	rv32imafcxtheade	ilp32f e907
	e907fd	rv32imafdcxtheade	ilp32d e907
	e907p	rv32imacpzpsfoperand_xtheade	ilp32 e907
	e907fp	rv32imafcpzpsfoperand_xtheade	ilp32f e907
	e907fdp	rv32imafdcpzpsfoperand_xtheade	ilp32d e907

Continued on next page

Table 5.2 – Continued from previous page

	CPU	-march	-mabi -mtune
c906 Series	c906	rv64imacxtheadc	lp64 c906
	c906fd	rv64imafdc_zfh_xtheadc	lp64d c906
	c906fdv	rv64imafdcv0p7_zfh_xtheadc	lp64d c906
c910 Series	c910	rv64imafdc_zfh_xtheadc	lp64d c910
	c920	rv64imafdcv0p7_zfh_xtheadc	lp64d c920
	r910	rv64imafdc_zfh_xtheadc	lp64d r910
	r920	rv64imafdcv0p7_zfh_xtheadc	lp64d r920
c908 Series	c908	rv64imafdc_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc_ zbs_svinval_svnapot_svpbmt_xtheadc	lp64d c908
	c908i	rv64imac_zicbom_zicbop_zicboz_zihintpause_zba_zbb_zbc_zbs_sv inval_svnapot_svpbmt_xtheadc	lp64 c908
	c908v	rv64imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc _zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	lp64d c908
	c908-rv32	rv32imafdc_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc_ zbs_svinval_svnapot_svpbmt_xtheadc	ilp32d c908
	c908v-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc _zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	ilp32d c908
c910v2 Series	c910v2	rv64imafdc_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause _zawrs_zfa_zfbfmin_zfh_zca_zcb_zcd_zba_zbb_zbc_zbs_svinval_s vnapot_svpbmt_xtheadc	lp64d c910v2
	c920v2	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus e_zawrs_zfa_zfbfmin_zfh_zca_zcb_zcd_zba_zbb_zbc_zbs_zvfbfmin _zvfbfwma_svinval_svpbmt_xtheadc_xtheadvdot	lp64d c920v2
c920v2.c908v series	c920v2.c908v	rv64imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc _zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	lp64d c920v2.c908v
c907 Series	c907	rv64imac_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause_z awrs_zcb_zba_zbb_zbc_zbs_svinval_svnapot_svpbmt_xtheadc	lp64 c907
	c907fd	rv64imafdc_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause _zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_svinval_svnapot_s vpbmt_xtheadc	lp64d c907
	c907fdv	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus e_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwm a_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	lp64d c907
	c907fdvm	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus e_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwm a_svinval_svnapot_svpbmt_xtheadc_xtheadmatrix_xtheadvdot	lp64d c907
	c907-rv32	rv32imac_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause_z awrs_zcb_zba_zbb_zbc_zbs_svinval_svnapot_svpbmt_xtheadc	ilp32 c907
	c907fd-rv32	rv32imafdc_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause _zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_svinval_svnapot_s vpbmt_xtheadc	ilp32d c907

Continued on next page

Table 5.2 – Continued from previous page

	CPU	-march	-mabi -mtune
	c907fdv-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus	ilp32d c907
		e_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwm	
		a_svinval_svnapot_svpbmt_xtheadc_xtheadvdot	
	c907fdvm-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus	ilp32d c907
		e_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwm	
		a_svinval_svnapot_svpbmt_xtheadc_xtheadmatrix_xtheadvdot	

Remark:

1. The -mcpu option is recommended for both the XuanTie LLVM compiler and the XuanTie GNU compiler V2.6.0 (inclusive) and above.

Convenient and compatible.

2. The Xuantie LLVM compiler follows the community rules. The vector instruction set v does not include the half-precision vector instruction set zvfh by default. If you need to use it, you need

Add zvfh0p1 to the -march option and add -menable-experimental-extensions; the p extension in -march is

p0p94.

Table 5.3: Correspondence between CPU and options (before V2.2.0 (GNU))

	CPU	-march(before V2.0.3)	-march(after V2.0.3(inclusive))	-mabi	-mtune
e902 Seriese902		rv32ecxtheadse	same left	ilp32e	e902
	e902m	rv32emcxtheadse	same left	ilp32e	e902
	e902t	rv32ecxtheadse	same left	ilp32e	e902
	e902mt	rv32emcxtheadse	same left	ilp32e	e902
e906 seriese906		rv32imacxtheade	same left	ilp32	e906
	e906f	rv32imafcxtheade	same left	ilp32f	e906
	e906fd	rv32imafdcxtheade	Same as left	ilp32d	e906
	e906p	rv32imacpzp64_xtheade	rv32imacpzpsfoperand_xtheade	ilp32	e906
	e906fp	rv32imafcpzp64_xtheade ilp32f e906	rv32imafcpzpsfoperand_xtheade		
	e906fdp	rv32imafdcpzp64_xtheade	rv32imafdcpzpsfoperand_xtheade ilp32d e906		
e907 series e907 same left same left		rv32imacxtheade		ilp32	e907
	e907f	rv32imafcxtheade		ilp32f	e907
	e907fd	rv32imafdcxtheade	Same as left	ilp32d	e907
	e907p	rv32imacpzp64_xtheade	rv32imacpzpsfoperand_xtheade	ilp32	e907
	e907fp	rv32imafcpzp64_xtheade ilp32f e907	rv32imafcpzpsfoperand_xtheade		
	e907fdp	rv32imafdcpzp64_xtheade	rv32imafdcpzpsfoperand_xtheade ilp32d e907		
c906 series c906 same left same left same left same left same left		rv64imacxtheadc		lp64	c906
	c906fd	rv64imafdcxtheadc		lp64d	c906
	c906fdv	rv64imafdcvxtheadc		lp64dv	c906
c910 series c910		rv64imafdcxtheadc		lp64d	c910
	c920	rv64imafdcvxtheadc		lp64dv	c920
	r910	rv64imafdcxtheadc		lp64d	r910

Continued on next page

Table 5.3 – Continued from previous page

	CPU	-march(before V2.0.3) -march(after V2.0.3(inclusive)) -mabi -mtune			
	r920	rv64imafdcvxtheadc	Same as left	lp64dv r920	

Note: In the XuanTie V2.0.3 (GNU) version, the P extension instruction is upgraded to version 0.9.4, the zp64 instruction set is changed to zpsfoperand, and the others remain unchanged.

The specific functions and usage of the -mcpu, -march, -mabi and -mtune options are described in the following sections:

- *-mcpu* option (supported by XuanTie *V2.6 (GNU))*
- *-march* option
- *-mabi* option
- *-mtune* option

5.1.1 -mcpu option (supported by XuanTie V2.6 (GNU))

The -mcpu option selects arch, tune and default abi according to the correspondence in Table 5.1 and Table 5.2. Abi can be selected by the -mabi option. It is a non-default abi.

5.1.2 -march Option

The -march option is used to specify the instruction set used to generate the target file. Different characters correspond to different instruction sets. The corresponding relationship is shown in Table 5.4 . rv32imacxthead indicates that the current target file uses the 32-bit integer basic instruction set, integer multiplication and division instruction set, atomic operation instruction set, compression instruction set and meta-instruction set. Iron performance enhancement instruction set. Except for the extended instruction set, the rest of the instruction sets are RISC-V standard instruction sets. For more detailed description, see RISC-V ISA SPEC.

Table 5.4: Instruction set corresponding to strings

Character Name Instruction Set	
rv32i	32-bit integer basic instruction set
rv32e	Embedded 32-bit integer basic instruction set (basically the same as rv32i but only uses 16 registers)
rv64i	64-bit integer basic instruction set
m	Integer multiplication and division instruction set
a	Atomic operation instruction set
f	Single-precision floating-point instruction set
d	Double-precision floating-point instruction set
c	Compressed instruction set (i.e. 16-bit instruction set)
p	Packed-SIMD instruction set
zba	Address calculation instruction set (B extended instruction subset)
zb	Basic bit manipulation instruction set (B extended instruction subset)
zbc	carry-less multiplication instruction set (instruction subset of the B extension)
zb	Single bit operation instruction set (B extended instruction subset)

Continued on next page

Table 5.4 – Continued from previous page

Character Name Instruction Set	
zihintpause Pause prompt instruction set	
v	Vector Instruction Set
f	Half-precision vector instruction set
v0p7	Vector instruction set version v0.7.1
xtheadcmo Instruction set for Xuantie cache management	
xtheadsync Instruction set for synchronization of Hematite multiprocessors	
xtheadba	Instruction set for calculating black iron address
xtheadbb	Instruction set for basic bit operations
xtheadbs	Black Iron Bit Instruction Set
xtheadcondmov Black Iron conditional move instruction set	
xtheadmemidx Black Iron integer memory operation instruction set	
xtheadmempair Black Iron double integer register memory operation instruction set	
xtheadfmemidx Xuantie floating point memory operation instruction set	
xtheadmac Black Iron multiply accumulate instruction set	
xtheadfmv Black Iron double floating point high-order data transfer instruction set	
xtheadint	Black Iron Acceleration Interrupt Instruction Set
xtheadvdot Black Iron vector dot operation enhanced instruction set	
xtheadvector Instruction set for black iron vector	
xtheadmatrix Black Iron matrix operation enhanced instruction set	
xtheadfpp Black Iron Fast Peripheral Instruction Set	
xxtccei	Black Iron Coprocessor Integer Instruction Set
xxtccef	Black Iron Coprocessor Floating Point Instruction Set
xxtccev	Black Iron Coprocessor Vector Instruction Set
xtheadc	Xuantie C series performance enhancement instruction set, equivalent to xtheadba_xtheadbb_xtheadbs_xtheadcmo_xtheadc
	ondmov_xtheadfmemidx_xtheadfmv_xtheadmac_xtheadmemidx_xthead
	mempair_xtheadsync
xtheade	Xuantie E series performance enhancement instruction set, equivalent to xtheadba_xtheadbb_xtheadbs_xtheadcmo_xtheadc
	ondmov_xtheadfmv_xtheadint_xtheadmac_xtheadmemidx_xtheadsync
xtheadse	Xuantie Small E series performance enhancement instruction set, equivalent to xtheadcmo
rv32g	Abbreviation of rv32imafd_zicsr_zifencei
rv64g	Abbreviation of rv64imafd_zicsr_zifencei

Note: In the specification of the T-HEAD extension instruction, the T-HEAD extension instruction set such as xtheadc is split into multiple sub-extensions. For details, see T-HEAD Extension Spec.

5.1.3 -mabi Option

The -mabi option is used to specify the ABI (Application Binary Interface) rules for generating target files. A brief description is shown in Table 5.5. A more detailed description is shown in Table 5.5.

For details, see RISC-V ABI SPEC.

Table 5.5: ABI rules description

Name	Description
ilp32e	rv32e Parameter passing rules for 16 registers
ilp32	When using rv32i, all types use the parameter integer register parameter passing rules
ilp32f	rv32i, single-precision floating-point type parameters use the parameter passing rules of floating-point registers
ilp32d	rv32i, single-precision and double-precision floating-point type parameters use the parameter passing rules of floating-point registers
is used,	When rv64i is used, all types use the parameter integer register parameter passing rules. When lp64
lp64f	is used, single-precision floating-point type parameters use the floating-point register parameter passing rules. When
lp64d	is used, single-precision and double-precision floating-point type parameters use the floating-point register parameter passing rules.

5.1.4 -mtune Option

The -mtune option does not affect the correctness of program execution, but specifying the corresponding tune can generate a cost model and pipeline optimized for a specific CPU. Target program. For the current mtune options for all CPUs of Xuantie, please refer to Table 5.1 and Table 5.2 .

5.2 Xuantie small-size runtime library libcc-rt

A runtime library is a library used by compilers to implement built-in operations and built-in functions of a programming language to provide runtime support for the language program. This library generally includes some basic mathematical operations. For example, when the instruction set does not contain floating-point instructions, floating-point calculations in the programming language need to call soft floating-point functions in the library.

In the Xuantie GNU series compilers, libgcc is the default runtime library, and its floating-point calculation complies with the IEEE754 standard. libcc-rt is another runtime library developed by Xuantie for embedded customers who have a deep demand for code size. It is basically similar to libgcc in terms of functionality. At the same time, in order to obtain the optimal code size, some of its floating-point calculation functions are incompatible with the libgcc interface. For details, please refer to the section on the differences between libcc-rt and *libgcc* floating-point calculation .

5.2.1 How to use libcc-rt

Add the option -mccrt when linking to use libcc-rt instead of libgcc.

Note: Currently, this option is only available for the E902, E906 and E907 series.

5.2.2 Differences between libcc-rt and libgcc in floating point calculations

In order to achieve better code size optimization, libcc-rt implements floating-point interfaces differently from libgcc. See below for specific differences.

surface:

Table 5.6: libcc-rt floating point interface differences

	libcc-rt implements the standard	libgcc implements the standard
1. For values where the result precision underflows, 0 is returned instead of the denormalized number.	a. For floating-point arithmetic operations (addition, subtraction, multiplication, and division), 0 is returned for values with underflow	to return denormalized numbers
	b. When the floating-point precision is converted and the precision is reduced (double ÿ float), if underflow occurs, 0 is returned a. For floating-point	Returns a denormalized number
2. For values whose result precision overflows, an unexpected overflow value is returned	arithmetic operations (addition, subtraction, multiplication, and division), an unexpected overflow value is returned for values with overflow b. When the floating-point precision	When overflow occurs, the corresponding signed infinity value is returned
	is converted and the precision is reduced (double ÿ float), an unexpected overflow value is returned for values with overflow c. When the floating-point value is converted to an integer, when the	When overflow occurs, the corresponding signed infinity value is returned
	floating-point value is greater than the maximum value of the integer to be represented, the low-order truncation of the binary of the actual value corresponding to the integer is returned a. For floating-	For overflow values, returns the saturated value of the corresponding integer type.
3. For non-numbers and infinities in the calculation process and results, treat them as normalized numbers with corresponding exponents. reason	point arithmetic operations (addition, subtraction, multiplication, and division), when there is a non-number or infinity in the operand, it is treated as a normalized number with the corresponding exponent and the operation is	When adding or subtracting infinity from other numbers, the value is still the infinity. When subtracting infinities or when a non-number is involved in an operation, Nan is returned.
	continued b. When the floating-point precision is converted and the precision is increased (float ÿ double), for the infinity, non-number, and non-normalized number of the low-precision value, the high-precision number is treated as a normalized number and its exponent is processed	Infinities and non-numbers are converted to the corresponding high-precision infinities and non-numbers.
	c. Floating point comparison instructions, treating non-numbers as corresponding ordered floating points for comparison	When there is a non-number (Nan) in the numbers involved in the comparison, the comparison result is considered to be
		unordered: This result is different from greater than, less than, and equal to. Therefore, in the floating-point comparison functions that judge equality, greater than, greater than or equal to, less than, and less than or equal to, the comparison result is considered to fail;
		Conversely, in the function un-ord that judges whether it is unordered, the comparison result is considered to be successful. When two
4. For floating-point addition and subtraction, when the operands are opposite numbers and the result is 0, the sign bit of 0 is the same as the first operand. 5. For floating-point		negative 0s are subtracted (negative 0 plus negative 0 is equivalent), negative 0 is returned, and positive 0 is returned in other cases. When a non-zero value
division, if the divisor or dividend has ±0, a 0 with the appropriate sign is returned, and the division by 0 exception is not recorded.		is divided by 0, an infinite value with a suitable sign is obtained.

5.2.3 Examples of differences between libcc-rt and libgcc in floating-point calculations

This section uses C language to illustrate the differences between libcc-rt and libgcc in floating-point calculation to help readers better understand.

Difference 1:

For values where the result underflows, 0 is returned instead of the denormalized number.

```
#include <stdio.h>
```

(Continued on next page)

(Continued from previous page)

```
/*
    |s| e                   |                       t                          |
     0 00000001 00000000000000000000000
*/
int minest_float_value = 0x00800000;

/*
    |s|              e            |           t          |
     0 01101111111 1111...111
*/
long long double_minor_float_value = 0x37ffffffffffffff;

int main() {
    float res;
    float fa = *(float*)&minest_float_value;

    // Test for underflow in floating point arithmetic
    res = fa * 0.5; printf("The
    result of arithmetic floating point underflow is : %f\n", res); printf(" The result
    in hexadecimal is: 0x%08x\n\n", *(int*)&res);



    double da = *(double*)&double_minor_float_value;
    res = (float)da;

    //Test underflow during floating
    point conversion printf("The result of floating point conversion
    underflow is : %f\n", res); printf(" The result in hexadecimal is: 0x%08x\n\n", *(int*)&res);

    return 0;
}
```

Operation results display:

```
$riscv64-unknown-elf-gcc underflow.c -march=rv32imac -mabi=ilp32 -mccrt
$ qemu-riscv32 a.out
The result of a floating point underflow in an arithmetic operation is: 0.000000
        The result is expressed in hexadecimal: 0x00000000

When underflow occurs in floating point conversion, the result is: 0.000000
        The result is expressed in hexadecimal: 0x00000000

$riscv64-unknown-elf-gcc underflow.c -march=rv32imac -mabi=ilp32
$ qemu-riscv32 a.out
```

(Continued on next page)

(Continued from previous page)

```
The result of a floating point underflow in an arithmetic operation is: 0.000000
        The result is expressed in hexadecimal: 0x00400000

When underflow occurs in floating point conversion, the result is: 0.000000
        The result is expressed in hexadecimal: 0x00400000
```

Difference 2:

For values whose result overflows, an unexpected overflow value is returned.

```
#include <stdio.h>

/*
    |s| e                   |                       t                       |
     0 11111110 11111111111111111111111
*/
int big_float_value = 0x7f7fffff;

/*
    |s|               e             |            t            |
     0 10011111111 0000...001
*/
long long double_greater_float_value = 0x4ff0000000000001;

/*
    |s| e                   |                       t                       |
     0 10100000 00000000000000000000001
*/
int float_greater_int_value = 0x50000001;

int main() {
    float res;
    float fa = *(float*)&big_float_value;

    // Test for overflow in floating point arithmetic
    res = fa * 2;
    printf("The result of arithmetic floating point overflow is : %f\n", res);
    printf(" The result is expressed in hexadecimal: 0x%08x\n\n", *(int*)&res);


    double da = *(double*)&double_greater_float_value;
    res = (float)da;

    //Test for overflow during floating point conversion
```

(Continued on next page)

(Continued from previous page)

```
    printf("When overflow occurs in floating point conversion, the result is : %f\n",
    res); printf(" The result is expressed in hexadecimal: 0x%08x\n\n", *(int*)&res);


    float fb = *(float*)&float_greater_int_value;
    int ires = fb;

    //Test the overflow when converting floating
    point to integer printf("The result of the overflow when converting floating point to integer
    printf("                    is : %d\n", ires); The result is expressed in hexadecimal: 0x%08x\n\n", ires);

    return 0;
}
```

Operation results display:

```
$riscv64-unknown-elf-gcc overflow.c -march=rv32imac -mabi=ilp32 -mccrt $qemu-riscv32 a.out The result of arithmetic
floating point overflow is:
6805646932770577196234083669690338508 80.000000 The result is expressed in hexadecimal: 0x7fffffff


When a floating point conversion overflows, the result is:
        -1.000000 The result is expressed in hexadecimal: 0xbf800000

When overflow occurs when converting floating point to integer, the result is: 1024
          The result is expressed in hexadecimal: 0x00000400

$riscv64-unknown-elf-gcc overflow.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out The result
of arithmetic floating point overflow is: inf
        The result is expressed in hexadecimal: 0x7f800000

When a floating point conversion overflows, the result is: inf
        The result is expressed in hexadecimal: 0x7f800000

When overflow occurs during conversion from floating point to integer, the
          result is: 2147483647 The result is expressed in hexadecimal: 0x7fffffff
```

Difference

3: Non-numbers and infinities in the calculation process and results are treated as normalized numbers with corresponding exponents

```
#include <stdio.h>

/*
```

(Continued on next page)

(Continued from previous page)

```
    |s| e                 |                   t                    |
      0 11111101 00000000000000000000000
*/
int quarter_inf_float_value = 0x7e800000;

/*
    |s| e                 |                   t                    |
      0 11111111 00000000000000000000000
*/
int inf_float_value = 0x7f800000;

/*
    |s| e                 |                   t                    |
      0 11111111 00000000000000000000001
*/
int nan_float_value = 0x7f800001;


int main() {
    float res;
    float fqinf = *(float*)&quarter_inf_float_value;
    float finf = *(float*)&inf_float_value;
    float fnan = *(float*)&nan_float_value;


    //Test floating point arithmetic operations involving non-numbers and infinity
    res = finf - fqinf;
    printf("The result of infinite minus a large value is : %f\n", res);
    printf(" The result is expressed in hexadecimal: 0x%08x\n\n", *(int*)&res);
    res = fnan / finf;
    printf(" The result of arithmetic operation involving non-numbers is : %f\n", res);
    printf(" The result is expressed in hexadecimal: 0x%08x\n\n", *(int*)&res);


    double dress;
    //Test the processing of infinity and non-number when converting floating point to high precision
    dres = finf;
    printf(" The result of converting low-precision infinity to high-precision is : %f\n", dres);
    printf("                The result is expressed in hexadecimal: 0x%08x\n\n", *(long long*)&dres);
    dres = fnan;
    printf(" The result of converting low-precision non-number to high-precision is : %f\n", dres);
    printf("                The result is expressed in hexadecimal: 0x%08x\n\n", *(long long*)&dres);
```

(Continued on next page)

(Continued from previous page)

```
    //Test floating point unordered
    comparison char *cmp_res = finf < fnan ? "Floating point positive infinity is less than floating point positive non-number" :
                                            "Floating point positive infinity is not less than floating point positive non-number";
    printf(cmp_res);
    printf("\n"); cmp_res = finf
    > fnan ? "Floating point positive infinity is greater than floating point positive non-number" : "Floating point
                                        positive infinity is not greater than floating point positive non-number";
    printf(cmp_res);
    printf("\n\n");

    return 0;
}
```

Operation results display:

```
$riscv64-unknown-elf-gcc input_inf_or_nan.c -march=rv32imac -mabi=ilp32 -mccrt $qemu-riscv32 a.out The result of infinitely
subtracting a large value is:
25521177519070384759753095557382615859 2.000000 The result is expressed in hexadecimal: 0x7f400000

The result of arithmetic operation involving non-numbers is: 1.000000
    The result is expressed in hexadecimal: 0x3f800002

The result of converting low-precision infinity to high-precision is: 340282366920938463463374607431768211456.000000 The
        result is expressed in hexadecimal: 0x0001d008

The result of converting low-precision non-number to high-precision is: 340282407485757670766715455326270783488.000000
        The result is expressed in hexadecimal: 0x0001d008

Floating point positive infinity is less than floating point positive non-number

Floating point positive infinity is not greater than floating point positive non-number

$riscv64-unknown-elf-gcc input_inf_or_nan.c -march=rv32imac -mabi=ilp32
$ qemu-riscv32 a.out
The result of infinitely subtracting a large value
is: inf The result is expressed in hexadecimal as: 0x7f800000

The result of arithmetic operation involving non-numbers is: nan
    The result is expressed in hexadecimal: 0x7fc00000

The result of converting low-precision infinity to high-precision is: inf
        The result is expressed in hexadecimal: 0x0001e008
```

(Continued on next page)

(Continued from previous page)

```
The result of converting low-precision non-number to high-precision is: nan
        The result is expressed in hexadecimal: 0x0001e008

Floating point positive infinity is not less than floating point positive non-number

Floating point positive infinity is not greater than floating point positive non-number
```

Difference

4: In floating-point addition and subtraction operations, when the operands are opposite numbers and the result is 0, the sign bit of 0 is the same as the addend (minuend)

```
#include <stdio.h>

/*
    |s| e                 |                   t                       |
     0 10000000 00000000000000000000000
*/
int float_p2_value = 0x40000000;

/*
    |s| e                 |                   t                       |
     1 10000000 00000000000000000000000
*/
int float_n2_value = 0xc0000000;

int main() {
    float res;
    float p2 = *(float*)&float_p2_value;
    float n2 = *(float*)&float_n2_value;

    //When the result of addition and subtraction
    is 0, the symbol of 0
    res = p2 - p2; printf("+2 - +2 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);
    res = n2 - n2;
    printf("-2 - -2 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);
    res = p2 + n2;
    printf("+2 + -2 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);
    res = n2 + p2;
    printf("-2 + +2 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);

    printf("\n");
    return 0;
}
```

Operation results display:

```
$riscv64-unknown-elf-gcc cancel_to_zero.c -march=rv32imac -mabi=ilp32 -mccrt $qemu-riscv32 a.out +2 - +2 =
0.000000, the result is
expressed in hexadecimal: 0x00000000 -2 - -2 = -0.000000, the result is expressed
in hexadecimal: 0x80000000 +2 + -2 = 0.000000, the result is expressed in
hexadecimal: 0x00000000 -2 + +2 = -0.000000, the result is expressed in
hexadecimal: 0x80000000

$riscv64-unknown-elf-gcc cancel_to_zero.c -march=rv32imac -mabi=ilp32
$ qemu-riscv32 a.out
+2 - +2 = 0.000000, the result is expressed in hexadecimal: 0x00000000
-2 - -2 = 0.000000, the result is expressed in hexadecimal: 0x00000000
+2 + -2 = 0.000000, the result is expressed in hexadecimal: 0x00000000
-2 + +2 = 0.000000, the result is expressed in hexadecimal: 0x00000000
```

Difference

5: Floating-point division, if the divisor or dividend has ±0, returns 0 with the appropriate sign, and does not record the division by zero exception

```
#include <stdio.h>

/*
    |s| e                  |                    t                        |
     0 00000000 00000000000000000000000
*/
int float_p0_value = 0x00000000;

/*
    |s| e                  |                    t                        |
     1 00000000 00000000000000000000000
*/
int float_n0_value = 0x80000000;

int main() {
    float res;
    float p0 = *(float*)&float_p0_value;
    float n0 = *(float*)&float_n0_value;

    //Test result when divisor is 0
    res = 1.0 / p0;
    printf("+value / +0 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res); res = -1.0 / n0;
    printf("-value / -0 = %f,
    \tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);
    res = 1.0 / n0;
    printf("+value / -0 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res); res = -1.0 / p0;
```

(Continued on next page)

(Continued from previous page)

```
    printf("-value / +0 = %f,\tThe result is expressed in hexadecimal: 0x%08x\n", res, *(int*)&res);

    printf("\n");
    return 0;
}
```

Operation results display:

```
$riscv64-unknown-elf-gcc input_with_zero.c -march=rv32imac -mabi=ilp32 -mccrt $qemu-riscv32 a.out +value / +0 = 0.000000,
the result is expressed in
hexadecimal: 0x00000000 -value / -0 = 0.000000, the result is expressed in hexadecimal:
0x00000000 The result is expressed in hexadecimal: 0x80000000 The result is
+value / -0 = -0.000000,                    expressed in hexadecimal: 0x80000000
-value / +0 = -0.000000,

$riscv64-unknown-elf-gcc input_with_zero.c -march=rv32imac -mabi=ilp32
$ qemu-riscv32 a.out
+value / +0 = inf,                  The result is expressed in hexadecimal:
hexadecimal: 0x7f800000             0x7f800000 The result is expressed in
-value / -0 = inf, +value / -0 = -inf, The result is expressed in hexadecimal: 0xff800000
-value / +0 = -inf, The result is expressed in hexadecimal: 0xff800000
```

5.3 pthread multithreading (currently only supported in the GNU toolchain)

The implementation of multithreading function is related to the specific platform/OS. The multithreading implementation supported by various OS is different, such as Linux, rtems, vxWorks, zephyr, FreeRTOS, RT-Thread, etc.

Some functions on the compiler tool, such as C++11 language standard multithreading support: std::thread, are also implemented by calling the multithreading function interface implemented by the platform/OS, mainly involving the relevant interface in the pthread.h header file. Therefore, in order to ensure the correctness of the multithreading function on the compiler tool, it is necessary to ensure the consistency of the data structure size on both the compiler tool and the platform/OS.

5.3.1 Main data structure

```
pthread_mutex_t
pthread_cond_t
pthread_mutexattr_t
pthread_condattr_t
pthread_attr_t
```

5.3.2 Size consistency test

```
#include <stdlib.h>
#include <pthread.h>

int main()
{
        if (__THEAD_SIZEOF_PTHREAD_MUTEX_T != sizeof(pthread_mutex_t))
                abort();
        if (__THEAD_SIZEOF_PTHREAD_COND_T != sizeof(pthread_cond_t))
                abort();
        if (__THEAD_SIZEOF_PTHREAD_MUTEXATTR_T != sizeof(pthread_mutexattr_t))
                abort();
        if (__THEAD_SIZEOF_PTHREAD_CONDATTR_T != sizeof(pthread_condattr_t))
                abort();
        if (__THEAD_SIZEOF_PTHREAD_ATTR_T != sizeof(pthread_attr_t))
                abort();

        return 0;
}
```

5.4 C/C++ Language Extensions

This chapter introduces the C/C++ language extensions of Xuantie, such as function attributes, variable types, intrinsic interfaces, etc.

5.4.1 Nested interrupt function properties

The nested interrupt function is enabled by adding __attribute__((interrupt("THead-interrupt-nesting"))) when declaring or defining a function. Its function is to generate an interrupt handler function that supports nested interrupts in machine mode. Compared with ordinary interrupt handlers, at the beginning of the function, in addition to saving the general register scene, it also saves macuse, mepc and mstatus, and re-enables interrupts; at the end of the function, in addition to restoring the general register scene, it also restores macuse, mepc and mstatus, and uses mret to return.

When the target CPU contains the xtheadint instruction set (included in xtheade, ipush/ipop instructions), __attribute__((interrupt("THead-interrupt-nesting"))) and __attribute__((interrupt)) or __attribute__((interrupt("machine"))) have the same effect, and ipush/ipop instructions will be used first; When the target CPU does not contain the xtheadint instruction set, taking E902 as an example, the nested interrupt function will generate the following function header and tail.

```
// function prologue
addi    sp,sp,-28
sw      t0,24(sp)
csrr    t0,mepc
sw      t0,20(sp)
csrr    t0,mcause
sw      t0,16(sp)
```

(Continued on next page)

(Continued from previous page)

```
csrr    t0,mstatus
sw      t0,12(sp)
sw      a4,8(sp)
sw      a5,4(sp)
csrsi mstatus,8

// function epilogue
csrw    mstatus,t0
l       t0,16(sp)
csrw    mcause,t0
l       t0,20(sp)
csrw    mepc,t0
l       t0,24(sp)
l       a4,8(sp)
l       a5,4(sp)
addi    sp,sp,28
mret
```

5.4.2 __bf16 Data Type

The __bf16 data type is supported starting with the BlackTie GNU compiler V2.8.0 and the BlackTie LLVM compiler V1.0.0-beta. Currently the __bf16 data type It has been merged into the ABI document by the community, please refer to RISC-V Calling Conventions.

In rv32 and rv64, its length and alignment are both 2 bytes, and it follows the arithmetic rules of regular floating-point types (such as float types).

5.5 Dynamic Linker Name

In versions prior to the BlackTie GNU compiler V2.8.0 and the BlackTie LLVM compiler V1.0.0-beta, in order to more conveniently handle multilib scenarios, when When the target CPU contains the XThead extension, the name of the dynamic linker will include thead. This will cause the community version of the program and dynamic library to be different from our To solve this problem, starting with the XuanTie GNU compiler V2.8.0 and XuanTie LLVM compiler V1.0.0-beta, the dynamic linker will No longer contain special characters such as xthead, consistent with community rules, as shown in Table 5.7 .

Table 5.7: Dynamic linker name comparison

	Before GCC V2.8.0 and LLVM V1.0.0-beta	After GCC V2.8.0 and LLVM V1.0.0-beta (inclusive)
march contains xtheadc	ld-linux-riscv[xlen]xthead-[abi].so.1	ld-linux-riscv[xlen]-[abi].so.1
march contains xtheadc and v0p7	ld-linux-riscv[xlen]v0p7_xthead-[abi].so.1	ld-linux-riscv[xlen]-[abi].so.1
march contains xtheadc and v	ld-linux-riscv[xlen]v_xthead-[abi].so.1	ld-linux-riscv[xlen]-[abi].so.1

Note: [xlen] refers to the bit width of the target CPU, which is 64 for rv64 and 32 for rv32; [abi] is the ABI specified by the -mabi option, such as lp64d, lp64, etc.

5.5.1 Compatibility Issues

Normally, when the name of the dynamic linker changes, simply generating a soft link pointing to the new dynamic linker name as the old dynamic linker name will cause an error when both the program and the dynamic library it depends on contain dynamic linker information and the names are different. We have solved this problem in the dynamic linker in the Black Iron GNU compiler V2.8.0 and Black Iron LLVM compiler V1.0.0-beta. When you need to be compatible with programs or dynamic libraries compiled by old versions of the compiler, you can solve this problem by creating a soft link.

Take C920 as an example. Its dynamic linker is /lib/ld-linux-riscv64-lp64d.so.1. First, use the ls -l command to view the actual file it points to. Assuming that the file it points to is /lib64/lp64d/ld-2.33.so, use ln -s to create a new soft link. The specific command is as follows:

```
ln -s /lib64/lp64d/ld-2.33.so /lib/ld-linux-riscv64v0p7_xthead-lp64d.so.1
```

5.6 Generic Coprocessor Extension C Intrinsic Interface

This section introduces the intrinsic interface of the XuanTie general coprocessor extension (including Xxtccei, Xxtccev, Xxtccef), including general naming rules and C intrinsics. The complete set of interfaces.

These interfaces are declared in the riscv_xt_cce.h header file.

Note: The encoding domain of the Xuantie general coprocessor extension (Xxtccei, Xxtccev, Xxtccef) conflicts with other xtheadc extensions except xtheadcmo (see the instruction set corresponding to the string for extension information) and cannot be used at the same time. If the CPU you are using contains the Xuantie general coprocessor extension, it means that the CPU does not contain other xtheadc extensions except xtheadcmo. If used, unexpected behavior errors will occur.

5.6.1 Naming conventions

The Xuantie general coprocessor extended C intrinsic interface is coded as follows:

```
__riscv_xt_{INSTRUCTION_MNEMONIC}_{BASE_TYPE}_{ROUND_MODE}_{POLICY}{(...)
```

- INSTRUCTION_MNEMONIC is the instruction mnemonic specified in the generic coprocessor interface documentation, such as cpx0 and vcpx4.
- BASE_TYPE indicates the type of the operand. Since Xxtccev only contains full register operations, only the EEW part is used, which is one of i8 | i16 | i32 | i64 | u8 | u16 | u32 | u64 | bf16 | f16 | f32 | f64. The type of Xxtccef is one of bf16 | f16 | f32 | f64. Only the intrinsics of Xxtccev and Xxtccef contain this part.
- ROUND_MODE is applicable only to Xxtccev with floating-point vector operands and an explicit floating-point rounding mode intrinsic, rm.
- POLICY is empty or mu, which applies only to Xxtccev.
  - No suffix: indicates a vector operation without a mask (vm=1); – _mu
    suffix: indicates a vector operation with a mask (vm=0) and is mask-undisturbed.

Similar to the RISC-V standard vector extension C intrinsic interface, the Xxtccev intrinsic interface also provides an implicit (overloaded) naming scheme, omitting BASE_TYPE, ROUND_MODE, POLICY section.

5.6.2 Interface parameters

idx is an integer constant in the range [0, 3].

imm10 is an integer constant in the range [0, 1023].

imm5 is an integer constant in the range [0, 31].

5.6.3 Xxtccei interface

```
void __riscv_xt_cpx0(unsigned idx, unsigned long rs1, unsigned imm10);
void __riscv_xt_cpx1(unsigned idx, unsigned long rs1);
unsigned long __riscv_xt_cpx2(unsigned idx, unsigned long rs1, unsigned imm5);
unsigned long __riscv_xt_cpx3(unsigned idx, unsigned long rs1); void __riscv_xt_cpx4(unsigned
idx, unsigned long rs1, unsigned long rs2,
                                    unsigned imm5);
void __riscv_xt_cpx5(unsigned idx, unsigned long rs1, unsigned long rs2);
unsigned long __riscv_xt_cpx6(unsigned idx, unsigned long rs1,
                                                unsigned long rs2);
void __riscv_xt_cpx7(unsigned idx, unsigned long rs3, unsigned long rs1,
                                    unsigned long rs2);
unsigned long __riscv_xt_cpx8(unsigned idx, unsigned long rd, unsigned long rs1,
                                                unsigned long rs2);
unsigned long __riscv_xt_cpx9(unsigned idx, unsigned long rd, unsigned long rs1,
                                                unsigned imm10);
unsigned long __riscv_xt_cpx10(unsigned idx, unsigned imm10);
```

5.6.4 Xxtccev Explicit (Non-overloaded) Interface

5.6.4.1 Basic Set

```
void __riscv_xt_vcpx0_i8(unsigned idx, vint8m1_t vs2); void
__riscv_xt_vcpx0_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vs2);
void __riscv_xt_vcpx0_i16(unsigned idx, vint16m1_t vs2);
void __riscv_xt_vcpx0_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vs2);
void __riscv_xt_vcpx0_i32(unsigned idx, vint32m1_t vs2); void
__riscv_xt_vcpx0_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vs2);
void __riscv_xt_vcpx0_i64(unsigned idx, vint64m1_t vs2);
void __riscv_xt_vcpx0_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vs2);
void __riscv_xt_vcpx0_u8(unsigned idx, vuint8m1_t vs2); void
__riscv_xt_vcpx0_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vs2);
void __riscv_xt_vcpx0_u16(unsigned idx, vuint16m1_t vs2);
void __riscv_xt_vcpx0_u16_mu(unsigned idx, vbool16_t mask, vuint16m1_t vs2);
void __riscv_xt_vcpx0_u32(unsigned idx, vuint32m1_t vs2);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx0_u32_mu(unsigned idx, vbool32_t mask, vuint32m1_t vs2);
void __riscv_xt_vcpx0_u64(unsigned idx, vuint64m1_t vs2);
void __riscv_xt_vcpx0_u64_mu(unsigned idx, vbool64_t mask, vuint64m1_t vs2); void __riscv_xt_vcpx0_f32(unsigned idx,
vfloat32m1_t vs2);
void __riscv_xt_vcpx0_f32_rm(unsigned idx, vfloat32m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0_f32_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs2);
void __riscv_xt_vcpx0_f32_rm_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                                unsigned frm);
void __riscv_xt_vcpx0_f64(unsigned idx, vfloat64m1_t vs2);
void __riscv_xt_vcpx0_f64_rm(unsigned idx, vfloat64m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0_f64_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs2); void __riscv_xt_vcpx0_f64_rm_mu(unsigned
idx, vbool64_t mask, vfloat64m1_t vs2,
                                unsigned frm);
vint8m1_t __riscv_xt_vcpx1_i8(unsigned idx, vint8m1_t vs2);
vint8m1_t __riscv_xt_vcpx1_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vd,
                                 vint8m1_t vs2);
vint16m1_t __riscv_xt_vcpx1_i16(unsigned idx, vint16m1_t vs2);
vint16m1_t __riscv_xt_vcpx1_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vd,
                                   vint16m1_t vs2);
vint32m1_t __riscv_xt_vcpx1_i32(unsigned idx, vint32m1_t vs2);
vint32m1_t __riscv_xt_vcpx1_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vd,
                                   vint32m1_t vs2);
vint64m1_t __riscv_xt_vcpx1_i64(unsigned idx, vint64m1_t vs2); vint64m1_t
__riscv_xt_vcpx1_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vd,
                                   vint64m1_t vs2);
vuint8m1_t __riscv_xt_vcpx1_u8(unsigned idx, vuint8m1_t vs2);
vuint8m1_t __riscv_xt_vcpx1_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                                  vuint8m1_t vs2);
vuint16m1_t __riscv_xt_vcpx1_u16(unsigned idx, vuint16m1_t vs2);
vuint16m1_t __riscv_xt_vcpx1_u16_mu(unsigned idx, vbool16_t mask,
                                    vuint16m1_t vd, vuint16m1_t vs2);
vuint32m1_t __riscv_xt_vcpx1_u32(unsigned idx, vuint32m1_t vs2);
vuint32m1_t __riscv_xt_vcpx1_u32_mu(unsigned idx, vbool32_t mask,
                                    vuint32m1_t vd, vuint32m1_t vs2);
vuint64m1_t __riscv_xt_vcpx1_u64(unsigned idx, vuint64m1_t vs2); vuint64m1_t
__riscv_xt_vcpx1_u64_mu(unsigned idx, vbool64_t mask,
                                    vuint64m1_t vd, vuint64m1_t vs2);
vfloat32m1_t __riscv_xt_vcpx1_f32(unsigned idx, vfloat32m1_t vs2);
vfloat32m1_t __riscv_xt_vcpx1_f32_rm(unsigned idx, vfloat32m1_t vs2, unsigned frm);

vfloat32m1_t __riscv_xt_vcpx1_f32_mu(unsigned idx, vbool32_t mask,
                                     vfloat32m1_t vd, vfloat32m1_t vs2);
vfloat32m1_t __riscv_xt_vcpx1_f32_rm_mu(unsigned idx, vbool32_t mask,
```

(Continued on next page)

(Continued from previous page)

```
                                                    vfloat32m1_t vd, vfloat32m1_t vs2,
                                                    unsigned frm);
vfloat64m1_t __riscv_xt_vcpx1_f64(unsigned idx, vfloat64m1_t vs2); vfloat64m1_t
__riscv_xt_vcpx1_f64_rm(unsigned idx, vfloat64m1_t vs2,
                                                    unsigned frm);
vfloat64m1_t __riscv_xt_vcpx1_f64_mu(unsigned idx, vbool64_t mask,
                                                    vfloat64m1_t vd, vfloat64m1_t vs2);
vfloat64m1_t __riscv_xt_vcpx1_f64_rm_mu(unsigned idx, vbool64_t mask,
                                                    vfloat64m1_t vd, vfloat64m1_t vs2,
                                                    unsigned frm);
void __riscv_xt_vcpx2_i8(unsigned idx, vint8m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2_i8_mu(unsigned
idx, vbool8_t mask, vint8m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_i16(unsigned idx, vint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_i32(unsigned idx, vint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_i64(unsigned idx, vint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_u8(unsigned idx, vuint8m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2_u8_mu(unsigned
idx, vbool8_t mask, vuint8m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_u16(unsigned idx, vuint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_u16_mu(unsigned idx, vbool16_t mask, vuint16m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_u32(unsigned idx, vuint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_u32_mu(unsigned idx, vbool32_t mask, vuint32m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_u64(unsigned idx, vuint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_u64_mu(unsigned idx, vbool64_t mask, vuint64m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_f32(unsigned idx, vfloat32m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2_f32_rm(unsigned
idx, vfloat32m1_t vs2, unsigned imm5,
                                                unsigned frm);
void __riscv_xt_vcpx2_f32_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_f32_rm_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                                                unsigned imm5, unsigned frm);
void __riscv_xt_vcpx2_f64(unsigned idx, vfloat64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_f64_rm(unsigned idx, vfloat64m1_t vs2, unsigned imm5,
```

(Continued on next page)

(Continued from previous page)

```
                                                unsigned frm);
void __riscv_xt_vcpx2_f64_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_f64_rm_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs2,
                                                unsigned imm5, unsigned frm);
vint8m1_t __riscv_xt_vcpx3_i8(unsigned idx, vint8m1_t vs2, unsigned imm5);
vint8m1_t __riscv_xt_vcpx3_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vd,
                                                vint8m1_t vs2, unsigned imm5);
vint16m1_t __riscv_xt_vcpx3_i16(unsigned idx, vint16m1_t vs2, unsigned imm5);
vint16m1_t __riscv_xt_vcpx3_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vd,
                                                vint16m1_t vs2, unsigned imm5); vint32m1_t
__riscv_xt_vcpx3_i32(unsigned idx, vint32m1_t vs2, unsigned imm5);
vint32m1_t __riscv_xt_vcpx3_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vd,
                                                vint32m1_t vs2, unsigned imm5);
vint64m1_t __riscv_xt_vcpx3_i64(unsigned idx, vint64m1_t vs2, unsigned imm5); vint64m1_t
__riscv_xt_vcpx3_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vd,
                                                vint64m1_t vs2, unsigned imm5);
vuint8m1_t __riscv_xt_vcpx3_u8(unsigned idx, vuint8m1_t vs2, unsigned imm5);
vuint8m1_t __riscv_xt_vcpx3_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                                                vuint8m1_t vs2, unsigned imm5);
vuint16m1_t __riscv_xt_vcpx3_u16(unsigned idx, vuint16m1_t vs2, unsigned imm5);
vuint16m1_t __riscv_xt_vcpx3_u16_mu(unsigned idx, vbool16_t mask,
                                                vuint16m1_t vd, vuint16m1_t vs2,
                                                unsigned imm5);
vuint32m1_t __riscv_xt_vcpx3_u32(unsigned idx, vuint32m1_t vs2, unsigned imm5);
vuint32m1_t __riscv_xt_vcpx3_u32_mu(unsigned idx, vbool32_t mask,
                                                vuint32m1_t vd, vuint32m1_t vs2,
                                                unsigned imm5);
vuint64m1_t __riscv_xt_vcpx3_u64(unsigned idx, vuint64m1_t vs2, unsigned imm5);
vuint64m1_t __riscv_xt_vcpx3_u64_mu(unsigned idx, vbool64_t mask,
                                                vuint64m1_t vd, vuint64m1_t vs2,
                                                unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3_f32(unsigned idx, vfloat32m1_t vs2,
                                                unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3_f32_rm(unsigned idx, vfloat32m1_t vs2,
                                                unsigned imm5, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx3_f32_mu(unsigned idx, vbool32_t mask,
                                                vfloat32m1_t vd, vfloat32m1_t vs2,
                                                unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3_f32_rm_mu(unsigned idx, vbool32_t mask,
                                                vfloat32m1_t vd, vfloat32m1_t vs2,
                                                unsigned imm5, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx3_f64(unsigned idx, vfloat64m1_t vs2,
```

(Continued on next page)

(Continued from previous page)

```
                                                    unsigned imm5);
vfloat64m1_t __riscv_xt_vcpx3_f64_rm(unsigned idx, vfloat64m1_t vs2,
                                                    unsigned imm5, unsigned frm); vfloat64m1_t
__riscv_xt_vcpx3_f64_mu(unsigned idx, vbool64_t mask,
                                                    vfloat64m1_t vd, vfloat64m1_t vs2,
                                                    unsigned imm5);
vfloat64m1_t __riscv_xt_vcpx3_f64_rm_mu(unsigned idx, vbool64_t mask,
                                                        vfloat64m1_t vd, vfloat64m1_t vs2,
                                                        unsigned imm5, unsigned frm);
void __riscv_xt_vcpx4_i8(unsigned idx, vint8m1_t vs3, vint8m1_t vs2,
                                                vint8m1_t vs1);
void __riscv_xt_vcpx4_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vs3,
                                                    vint8m1_t vs2, vint8m1_t vs1);
void __riscv_xt_vcpx4_i16(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                                                vint16m1_t vs1);
void __riscv_xt_vcpx4_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vs3,
                                                    vint16m1_t vs2, vint16m1_t vs1);
void __riscv_xt_vcpx4_i32(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                                                vint32m1_t vs1);
void __riscv_xt_vcpx4_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vs3,
                                                    vint32m1_t vs2, vint32m1_t vs1);
void __riscv_xt_vcpx4_i64(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                                                vint64m1_t vs1);
void __riscv_xt_vcpx4_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vs3,
                                                    vint64m1_t vs2, vint64m1_t vs1);
void __riscv_xt_vcpx4_u8(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2,
                                                vuint8m1_t vs1);
void __riscv_xt_vcpx4_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
                                                    vuint8m1_t vs2, vuint8m1_t vs1);
void __riscv_xt_vcpx4_u16(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2,
                                                vuint16m1_t vs1);
void __riscv_xt_vcpx4_u16_mu(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                                                    vuint16m1_t vs2, vuint16m1_t vs1);
void __riscv_xt_vcpx4_u32(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                                                vuint32m1_t vs1);
void __riscv_xt_vcpx4_u32_mu(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                                                    vuint32m1_t vs2, vuint32m1_t vs1);
void __riscv_xt_vcpx4_u64(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                                                vuint64m1_t vs1);
void __riscv_xt_vcpx4_u64_mu(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                                                    vuint64m1_t vs2, vuint64m1_t vs1);
void __riscv_xt_vcpx4_f32(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                                vfloat32m1_t vs1);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx4_f32_rm(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                             vfloat32m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4_f32_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                             vfloat32m1_t vs2, vfloat32m1_t vs1);
void __riscv_xt_vcpx4_f32_rm_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                vfloat32m1_t vs2, vfloat32m1_t vs1,
                                unsigned frm);
void __riscv_xt_vcpx4_f64(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                          vfloat64m1_t vs1);
void __riscv_xt_vcpx4_f64_rm(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                             vfloat64m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4_f64_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                             vfloat64m1_t vs2, vfloat64m1_t vs1);
void __riscv_xt_vcpx4_f64_rm_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                vfloat64m1_t vs2, vfloat64m1_t vs1,
                                unsigned frm);
vint8m1_t __riscv_xt_vcpx5_i8(unsigned idx, vint8m1_t vs2, vint8m1_t vs1);
vint8m1_t __riscv_xt_vcpx5_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vd,
                                 vint8m1_t vs2, vint8m1_t vs1);
vint16m1_t __riscv_xt_vcpx5_i16(unsigned idx, vint16m1_t vs2, vint16m1_t vs1);
vint16m1_t __riscv_xt_vcpx5_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vd,
                                   vint16m1_t vs2, vint16m1_t vs1);
vint32m1_t __riscv_xt_vcpx5_i32(unsigned idx, vint32m1_t vs2, vint32m1_t vs1); vint32m1_t __riscv_xt_vcpx5_i32_mu(unsigned
idx, vbool32_t mask, vint32m1_t vd,
                                   vint32m1_t vs2, vint32m1_t vs1);
vint64m1_t __riscv_xt_vcpx5_i64(unsigned idx, vint64m1_t vs2, vint64m1_t vs1);
vint64m1_t __riscv_xt_vcpx5_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vd,
                                   vint64m1_t vs2, vint64m1_t vs1);
vuint8m1_t __riscv_xt_vcpx5_u8(unsigned idx, vuint8m1_t vs2, vuint8m1_t vs1);
vuint8m1_t __riscv_xt_vcpx5_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                                  vuint8m1_t vs2, vuint8m1_t vs1);
vuint16m1_t __riscv_xt_vcpx5_u16(unsigned idx, vuint16m1_t vs2,
                                 vuint16m1_t vs1);
vuint16m1_t __riscv_xt_vcpx5_u16_mu(unsigned idx, vbool16_t mask,
                                    vuint16m1_t vd, vuint16m1_t vs2,
                                    vuint16m1_t vs1);
vuint32m1_t __riscv_xt_vcpx5_u32(unsigned idx, vuint32m1_t vs2,
                                 vuint32m1_t vs1);
vuint32m1_t __riscv_xt_vcpx5_u32_mu(unsigned idx, vbool32_t mask,
                                    vuint32m1_t vd, vuint32m1_t vs2,
                                    vuint32m1_t vs1);
vuint64m1_t __riscv_xt_vcpx5_u64(unsigned idx, vuint64m1_t vs2,
                                 vuint64m1_t vs1);
```

(Continued on next page)

(Continued from previous page)

```
vuint64m1_t __riscv_xt_vcpx5_u64_mu(unsigned idx, vbool64_t mask,
                                    vuint64m1_t vd, vuint64m1_t vs2,
                                    vuint64m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5_f32(unsigned idx, vfloat32m1_t vs2,
                                  vfloat32m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5_f32_rm(unsigned idx, vfloat32m1_t vs2,
                                     vfloat32m1_t vs1, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx5_f32_mu(unsigned idx, vbool32_t mask,
                                     vfloat32m1_t vd, vfloat32m1_t vs2,
                                     vfloat32m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5_f32_rm_mu(unsigned idx, vbool32_t mask,
                                        vfloat32m1_t vd, vfloat32m1_t vs2,
                                        vfloat32m1_t vs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx5_f64(unsigned idx, vfloat64m1_t vs2,
                                  vfloat64m1_t vs1);
vfloat64m1_t __riscv_xt_vcpx5_f64_rm(unsigned idx, vfloat64m1_t vs2,
                                     vfloat64m1_t vs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx5_f64_mu(unsigned idx, vbool64_t mask,
                                     vfloat64m1_t vd, vfloat64m1_t vs2,
                                     vfloat64m1_t vs1);
vfloat64m1_t __riscv_xt_vcpx5_f64_rm_mu(unsigned idx, vbool64_t mask,
                                        vfloat64m1_t vd, vfloat64m1_t vs2,
                                        vfloat64m1_t vs1, unsigned frm);
void __riscv_xt_vcpx6_i8(unsigned idx, vint8m1_t vs3, vint8m1_t vs2,
                         int8_t rs1);
void __riscv_xt_vcpx6_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vs3,
                            vint8m1_t vs2, int8_t rs1);
void __riscv_xt_vcpx6_i16(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                          int16_t rs1);
void __riscv_xt_vcpx6_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vs3,
                             vint16m1_t vs2, int16_t rs1);
void __riscv_xt_vcpx6_i32(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                          int32_t rs1);
void __riscv_xt_vcpx6_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vs3,
                             vint32m1_t vs2, int32_t rs1);
void __riscv_xt_vcpx6_u8(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2, uint8_t rs1);

void __riscv_xt_vcpx6_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
                            vuint8m1_t vs2, uint8_t rs1);
void __riscv_xt_vcpx6_u16(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2, uint16_t rs1);

void __riscv_xt_vcpx6_u16_mu(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                             vuint16m1_t vs2, uint16_t rs1);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx6_u32(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                          uint32_t rs1);
void __riscv_xt_vcpx6_u32_mu(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                             vuint32m1_t vs2, uint32_t rs1);
vint8m1_t __riscv_xt_vcpx7_i8(unsigned idx, vint8m1_t vs2, int8_t rs1);
vint8m1_t __riscv_xt_vcpx7_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vd,
                                 vint8m1_t vs2, int8_t rs1);
vint16m1_t __riscv_xt_vcpx7_i16(unsigned idx, vint16m1_t vs2, int16_t rs1);
vint16m1_t __riscv_xt_vcpx7_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vd,
                                   vint16m1_t vs2, int16_t rs1);
vint32m1_t __riscv_xt_vcpx7_i32(unsigned idx, vint32m1_t vs2, int32_t rs1); vint32m1_t
__riscv_xt_vcpx7_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vd,
                                   vint32m1_t vs2, int32_t rs1);
vuint8m1_t __riscv_xt_vcpx7_u8(unsigned idx, vuint8m1_t vs2, uint8_t rs1);
vuint8m1_t __riscv_xt_vcpx7_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                                  vuint8m1_t vs2, uint8_t rs1);
vuint16m1_t __riscv_xt_vcpx7_u16(unsigned idx, vuint16m1_t vs2, uint16_t rs1);
vuint16m1_t __riscv_xt_vcpx7_u16_mu(unsigned idx, vbool16_t mask,
                                    vuint16m1_t vd, vuint16m1_t vs2,
                                    uint16_t rs1);
vuint32m1_t __riscv_xt_vcpx7_u32(unsigned idx, vuint32m1_t vs2, uint32_t rs1);
vuint32m1_t __riscv_xt_vcpx7_u32_mu(unsigned idx, vbool32_t mask,
                                    vuint32m1_t vd, vuint32m1_t vs2,
                                    uint32_t rs1);
void __riscv_xt_vcpx8_i8(unsigned idx, vint8m1_t vs3, vint8m1_t vs2,
                         unsigned imm5);
void __riscv_xt_vcpx8_i8_mu(unsigned idx, vbool8_t mask, vint8m1_t vs3, vint8m1_t vs2, unsigned imm5);

void __riscv_xt_vcpx8_i16(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                          unsigned imm5);
void __riscv_xt_vcpx8_i16_mu(unsigned idx, vbool16_t mask, vint16m1_t vs3, vint16m1_t vs2, unsigned imm5);

void __riscv_xt_vcpx8_i32(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                          unsigned imm5);
void __riscv_xt_vcpx8_i32_mu(unsigned idx, vbool32_t mask, vint32m1_t vs3, vint32m1_t vs2, unsigned imm5);

void __riscv_xt_vcpx8_i64(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                          unsigned imm5);
void __riscv_xt_vcpx8_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vs3, vint64m1_t vs2, unsigned imm5);

void __riscv_xt_vcpx8_u8(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2,
                         unsigned imm5);
void __riscv_xt_vcpx8_u8_mu(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
```

(Continued on next page)

(Continued from previous page)

```
                                        vuint8m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_u16(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2,
                                        unsigned imm5); void
__riscv_xt_vcpx8_u16_mu(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                                        vuint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_u32(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                                        unsigned imm5); void
__riscv_xt_vcpx8_u32_mu(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                                        vuint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_u64(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                                        unsigned imm5); void
__riscv_xt_vcpx8_u64_mu(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                                        vuint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_f32(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                        unsigned imm5);
void __riscv_xt_vcpx8_f32_rm(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                        unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8_f32_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                        vfloat32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_f32_rm_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                        vfloat32m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8_f64(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                        unsigned imm5);
void __riscv_xt_vcpx8_f64_rm(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                        unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8_f64_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                        vfloat64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_f64_rm_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                        vfloat64m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx9_f32(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                        float fs1);
void __riscv_xt_vcpx9_f32_rm(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                        float fs1, unsigned frm);
void __riscv_xt_vcpx9_f32_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                        vfloat32m1_t vs2, float fs1);
void __riscv_xt_vcpx9_f32_rm_mu(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                        vfloat32m1_t vs2, float fs1, unsigned frm);
void __riscv_xt_vcpx9_f64(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                        double fs1);
void __riscv_xt_vcpx9_f64_rm(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                        double fs1, unsigned frm);
void __riscv_xt_vcpx9_f64_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                        vfloat64m1_t vs2, double fs1);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx9_f64_rm_mu(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                vfloat64m1_t vs2, double fs1, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx10_f32(unsigned idx, vfloat32m1_t vs2, float fs1); vfloat32m1_t __riscv_xt_vcpx10_f32_rm(unsigned
idx, vfloat32m1_t vs2, float fs1,
                                unsigned frm);
vfloat32m1_t __riscv_xt_vcpx10_f32_mu(unsigned idx, vbool32_t mask,
                                vfloat32m1_t vd, vfloat32m1_t vs2,
                                float fs1);
vfloat32m1_t __riscv_xt_vcpx10_f32_rm_mu(unsigned idx, vbool32_t mask,
                                vfloat32m1_t vd, vfloat32m1_t vs2,
                                float fs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx10_f64(unsigned idx, vfloat64m1_t vs2, double fs1);
vfloat64m1_t __riscv_xt_vcpx10_f64_rm(unsigned idx, vfloat64m1_t vs2,
                                double fs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx10_f64_mu(unsigned idx, vbool64_t mask,
                                vfloat64m1_t vd, vfloat64m1_t vs2,
                                double fs1);
vfloat64m1_t __riscv_xt_vcpx10_f64_rm_mu(unsigned idx, vbool64_t mask,
                                vfloat64m1_t vd, vfloat64m1_t vs2,
                                double fs1, unsigned frm);
```

5.6.4.2 RV64 section

The following intrinsic interfaces are only available on RV64 platforms.

```
void __riscv_xt_vcpx6_i64(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                          int64_t rs1);
void __riscv_xt_vcpx6_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vs3,
                             vint64m1_t vs2, int64_t rs1);
void __riscv_xt_vcpx6_u64(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                          uint64_t rs1);
void __riscv_xt_vcpx6_u64_mu(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                             vuint64m1_t vs2, uint64_t rs1);
vint64m1_t __riscv_xt_vcpx7_i64(unsigned idx, vint64m1_t vs2, int64_t rs1);
vint64m1_t __riscv_xt_vcpx7_i64_mu(unsigned idx, vbool64_t mask, vint64m1_t vd,
                                   vint64m1_t vs2, int64_t rs1);
vuint64m1_t __riscv_xt_vcpx7_u64(unsigned idx, vuint64m1_t vs2, uint64_t rs1);
vuint64m1_t __riscv_xt_vcpx7_u64_mu(unsigned idx, vbool64_t mask,
                                    vuint64m1_t vd, vuint64m1_t vs2,
                                    uint64_t rs1);
```

5.6.4.3 Zvfh/Zvfhmin extension

The following intrinsic interfaces are available only when the Zvfh or Zvfhmin extension is supported.

```
void __riscv_xt_vcpx0_f16(unsigned idx, vfloat16m1_t vs2);
void __riscv_xt_vcpx0_f16_rm(unsigned idx, vfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0_f16_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs2); void __riscv_xt_vcpx0_f16_rm_mu(unsigned
idx, vbool16_t mask, vfloat16m1_t vs2,
                                                unsigned frm);
vfloat16m1_t __riscv_xt_vcpx1_f16(unsigned idx, vfloat16m1_t vs2);
vfloat16m1_t __riscv_xt_vcpx1_f16_rm(unsigned idx, vfloat16m1_t vs2, unsigned frm);

vfloat16m1_t __riscv_xt_vcpx1_f16_mu(unsigned idx, vbool16_t mask,
                                                    vfloat16m1_t vd, vfloat16m1_t vs2);
vfloat16m1_t __riscv_xt_vcpx1_f16_rm_mu(unsigned idx, vbool16_t mask,
                                                        vfloat16m1_t vd, vfloat16m1_t vs2,
                                                        unsigned frm);
void __riscv_xt_vcpx2_f16(unsigned idx, vfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_f16_rm(unsigned idx, vfloat16m1_t vs2, unsigned imm5,
                                                unsigned frm);
void __riscv_xt_vcpx2_f16_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs2,
                                                unsigned imm5);
void __riscv_xt_vcpx2_f16_rm_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs2, unsigned imm5, unsigned frm);

vfloat16m1_t __riscv_xt_vcpx3_f16(unsigned idx, vfloat16m1_t vs2,
                                                unsigned imm5);
vfloat16m1_t __riscv_xt_vcpx3_f16_rm(unsigned idx, vfloat16m1_t vs2,
                                                    unsigned imm5, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx3_f16_mu(unsigned idx, vbool16_t mask,
                                                    vfloat16m1_t vd, vfloat16m1_t vs2,
                                                    unsigned imm5);
vfloat16m1_t __riscv_xt_vcpx3_f16_rm_mu(unsigned idx, vbool16_t mask,
                                                        vfloat16m1_t vd, vfloat16m1_t vs2,
                                                        unsigned imm5, unsigned frm);
void __riscv_xt_vcpx4_f16(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                                            vfloat16m1_t vs1);
void __riscv_xt_vcpx4_f16_rm(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                                                vfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4_f16_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                                                vfloat16m1_t vs2, vfloat16m1_t vs1);
void __riscv_xt_vcpx4_f16_rm_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                                                vfloat16m1_t vs2, vfloat16m1_t vs1,
                                                unsigned frm);
vfloat16m1_t __riscv_xt_vcpx5_f16(unsigned idx, vfloat16m1_t vs2,
                                                vfloat16m1_t vs1);
```

(Continued on next page)

(Continued from previous page)

```
vfloat16m1_t __riscv_xt_vcpx5_f16_rm(unsigned idx, vfloat16m1_t vs2,
                                     vfloat16m1_t vs1, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx5_f16_mu(unsigned idx, vbool16_t mask,
                                     vfloat16m1_t vd, vfloat16m1_t vs2,
                                     vfloat16m1_t vs1);
vfloat16m1_t __riscv_xt_vcpx5_f16_rm_mu(unsigned idx, vbool16_t mask,
                                        vfloat16m1_t vd, vfloat16m1_t vs2,
                                        vfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx8_f16(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                          unsigned imm5);
void __riscv_xt_vcpx8_f16_rm(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2, unsigned imm5, unsigned frm);

void __riscv_xt_vcpx8_f16_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                             vfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_f16_rm_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3, vfloat16m1_t vs2, unsigned imm5, unsigned
                                frm);
void __riscv_xt_vcpx9_f16(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                          _Float16 fs1);
void __riscv_xt_vcpx9_f16_rm(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                             _Float16 fs1, unsigned frm);
void __riscv_xt_vcpx9_f16_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                             vfloat16m1_t vs2, _Float16 fs1);
void __riscv_xt_vcpx9_f16_rm_mu(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                                vfloat16m1_t vs2, _Float16 fs1, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx10_f16(unsigned idx, vfloat16m1_t vs2,
                                   _Float16 fs1);
vfloat16m1_t __riscv_xt_vcpx10_f16_rm(unsigned idx, vfloat16m1_t vs2,
                                      _Float16 fs1, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx10_f16_mu(unsigned idx, vbool16_t mask,
                                      vfloat16m1_t vd, vfloat16m1_t vs2,
                                      _Float16 fs1);
vfloat16m1_t __riscv_xt_vcpx10_f16_rm_mu(unsigned idx, vbool16_t mask,
                                         vfloat16m1_t vd, vfloat16m1_t vs2,
                                         _Float16 fs1, unsigned frm);
```

5.6.4.4 Zvfbfmin extension

The following intrinsic interfaces are only available when the Zvfbfmin extension is supported.

```
void __riscv_xt_vcpx0_bf16(unsigned idx, vbfloat16m1_t vs2); void
__riscv_xt_vcpx0_bf16_rm(unsigned idx, vbfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0_bf16_mu(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2);
void __riscv_xt_vcpx0_bf16_rm_mu(unsigned idx, vbool16_t mask,
```

(Continued on next page)

(Continued from previous page)

```
                                                vbfloat16m1_t vs2, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx1_bf16(unsigned idx, vbfloat16m1_t vs2);
vbfloat16m1_t __riscv_xt_vcpx1_bf16_rm(unsigned idx, vbfloat16m1_t vs2, unsigned frm);

vbfloat16m1_t __riscv_xt_vcpx1_bf16_mu(unsigned idx, vbool16_t mask,
                                       vbfloat16m1_t vd, vbfloat16m1_t vs2);
vbfloat16m1_t __riscv_xt_vcpx1_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                          vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                          unsigned frm);
void __riscv_xt_vcpx2_bf16(unsigned idx, vbfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2_bf16_rm(unsigned idx, vbfloat16m1_t vs2, unsigned imm5,
                              unsigned frm);
void __riscv_xt_vcpx2_bf16_mu(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2,
                              unsigned imm5);
void __riscv_xt_vcpx2_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                 vbfloat16m1_t vs2, unsigned imm5,
                                 unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx3_bf16(unsigned idx, vbfloat16m1_t vs2,
                                    unsigned imm5);
vbfloat16m1_t __riscv_xt_vcpx3_bf16_rm(unsigned idx, vbfloat16m1_t vs2,
                                       unsigned imm5, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx3_bf16_mu(unsigned idx, vbool16_t mask,
                                       vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                       unsigned imm5);
vbfloat16m1_t __riscv_xt_vcpx3_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                          vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                          unsigned imm5, unsigned frm);
void __riscv_xt_vcpx4_bf16(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                           vbfloat16m1_t vs1);
void __riscv_xt_vcpx4_bf16_rm(unsigned idx, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, vbfloat16m1_t vs1,
                              unsigned frm);
void __riscv_xt_vcpx4_bf16_mu(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, vbfloat16m1_t vs1);
void __riscv_xt_vcpx4_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                 vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                                 vbfloat16m1_t vs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx5_bf16(unsigned idx, vbfloat16m1_t vs2,
                                    vbfloat16m1_t vs1);
vbfloat16m1_t __riscv_xt_vcpx5_bf16_rm(unsigned idx, vbfloat16m1_t vs2,
                                       vbfloat16m1_t vs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx5_bf16_mu(unsigned idx, vbool16_t mask,
                                       vbfloat16m1_t vd, vbfloat16m1_t vs2,
```

(Continued on next page)

(Continued from previous page)

```
                                                    vbfloat16m1_t vs1);
vbfloat16m1_t __riscv_xt_vcpx5_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                          vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                          vbfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx8_bf16(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                           unsigned imm5);
void __riscv_xt_vcpx8_bf16_rm(unsigned idx, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8_bf16_mu(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                 vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                                 unsigned imm5, unsigned frm);
void __riscv_xt_vcpx9_bf16(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                           __bf16 fs1);
void __riscv_xt_vcpx9_bf16_rm(unsigned idx, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, __bf16 fs1, unsigned frm);
void __riscv_xt_vcpx9_bf16_mu(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                              vbfloat16m1_t vs2, __bf16 fs1);
void __riscv_xt_vcpx9_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                 vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                                 __bf16 fs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx10_bf16(unsigned idx, vbfloat16m1_t vs2,
                                     __bf16 fs1);
vbfloat16m1_t __riscv_xt_vcpx10_bf16_rm(unsigned idx, vbfloat16m1_t vs2,
                                        __bf16 fs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx10_bf16_mu(unsigned idx, vbool16_t mask,
                                        vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                        __bf16 fs1);
vbfloat16m1_t __riscv_xt_vcpx10_bf16_rm_mu(unsigned idx, vbool16_t mask,
                                           vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                           __bf16 fs1, unsigned frm);
```

5.6.5 Xxtccev Implicit (Overloaded) Interface

5.6.5.1 Basic Set

```
void __riscv_xt_vcpx0(unsigned idx, vint8m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool8_t mask, vint8m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vint16m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vint16m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vint32m1_t vs2);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx0(unsigned idx, vbool32_t mask, vint32m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vint64m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool64_t mask, vint64m1_t vs2); void __riscv_xt_vcpx0(unsigned idx,
vuint8m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool8_t mask, vuint8m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vuint16m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vuint16m1_t vs2); void __riscv_xt_vcpx0(unsigned idx,
vuint32m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool32_t mask, vuint32m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vuint64m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool64_t mask, vuint64m1_t vs2); void __riscv_xt_vcpx0(unsigned idx,
vfloat32m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vfloat32m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0(unsigned idx, vbool32_t mask, vfloat32m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                      unsigned frm);
void __riscv_xt_vcpx0(unsigned idx, vfloat64m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vfloat64m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0(unsigned idx, vbool64_t mask, vfloat64m1_t vs2); void __riscv_xt_vcpx0(unsigned idx,
vbool64_t mask, vfloat64m1_t vs2,
                      unsigned frm);
vint8m1_t __riscv_xt_vcpx1(unsigned idx, vint8m1_t vs2);
vint8m1_t __riscv_xt_vcpx1(unsigned idx, vbool8_t mask, vint8m1_t vd,
                           vint8m1_t vs2);
vint16m1_t __riscv_xt_vcpx1(unsigned idx, vint16m1_t vs2);
vint16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vint16m1_t vd,
                            vint16m1_t vs2);
vint32m1_t __riscv_xt_vcpx1(unsigned idx, vint32m1_t vs2);
vint32m1_t __riscv_xt_vcpx1(unsigned idx, vbool32_t mask, vint32m1_t vd,
                            vint32m1_t vs2);
vint64m1_t __riscv_xt_vcpx1(unsigned idx, vint64m1_t vs2); vint64m1_t
__riscv_xt_vcpx1(unsigned idx, vbool64_t mask, vint64m1_t vd,
                            vint64m1_t vs2);
vuint8m1_t __riscv_xt_vcpx1(unsigned idx, vuint8m1_t vs2);
vuint8m1_t __riscv_xt_vcpx1(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                            vuint8m1_t vs2);
vuint16m1_t __riscv_xt_vcpx1(unsigned idx, vuint16m1_t vs2);
vuint16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vuint16m1_t vd,
                             vuint16m1_t vs2);
vuint32m1_t __riscv_xt_vcpx1(unsigned idx, vuint32m1_t vs2);
vuint32m1_t __riscv_xt_vcpx1(unsigned idx, vbool32_t mask, vuint32m1_t vd,
                             vuint32m1_t vs2);
vuint64m1_t __riscv_xt_vcpx1(unsigned idx, vuint64m1_t vs2);
```

(Continued on next page)

(Continued from previous page)

```
vuint64m1_t __riscv_xt_vcpx1(unsigned idx, vbool64_t mask, vuint64m1_t vd,
                                                    vuint64m1_t vs2);
vfloat32m1_t __riscv_xt_vcpx1(unsigned idx, vfloat32m1_t vs2); vfloat32m1_t __riscv_xt_vcpx1(unsigned
idx, vfloat32m1_t vs2, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx1(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                                                    vfloat32m1_t vs2);
vfloat32m1_t __riscv_xt_vcpx1(unsigned idx, vbool32_t mask, vfloat32m1_t vd, vfloat32m1_t vs2, unsigned frm);

vfloat64m1_t __riscv_xt_vcpx1(unsigned idx, vfloat64m1_t vs2);
vfloat64m1_t __riscv_xt_vcpx1(unsigned idx, vfloat64m1_t vs2, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx1(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                                                    vfloat64m1_t vs2);
vfloat64m1_t __riscv_xt_vcpx1(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                                                    vfloat64m1_t vs2, unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vint8m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2(unsigned idx,
vbool8_t mask, vint8m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vint16m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool32_t mask, vint32m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool64_t mask, vint64m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vuint8m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2(unsigned idx,
vbool8_t mask, vuint8m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vuint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vuint16m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vuint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool32_t mask, vuint32m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vuint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool64_t mask, vuint64m1_t vs2,
                                    unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vfloat32m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2(unsigned idx,
vfloat32m1_t vs2, unsigned imm5,
                                    unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                                    unsigned imm5);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx2(unsigned idx, vbool32_t mask, vfloat32m1_t vs2,
                      unsigned imm5, unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vfloat64m1_t vs2, unsigned imm5); void __riscv_xt_vcpx2(unsigned
idx, vfloat64m1_t vs2, unsigned imm5,
                      unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vbool64_t mask, vfloat64m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool64_t mask, vfloat64m1_t vs2,
                      unsigned imm5, unsigned frm);
vint8m1_t __riscv_xt_vcpx3(unsigned idx, vint8m1_t vs2, unsigned imm5);
vint8m1_t __riscv_xt_vcpx3(unsigned idx, vbool8_t mask, vint8m1_t vd, vint8m1_t vs2, unsigned imm5);

vint16m1_t __riscv_xt_vcpx3(unsigned idx, vint16m1_t vs2, unsigned imm5);
vint16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vint16m1_t vd,
                            vint16m1_t vs2, unsigned imm5);
vint32m1_t __riscv_xt_vcpx3(unsigned idx, vint32m1_t vs2, unsigned imm5);
vint32m1_t __riscv_xt_vcpx3(unsigned idx, vbool32_t mask, vint32m1_t vd,
                            vint32m1_t vs2, unsigned imm5);
vint64m1_t __riscv_xt_vcpx3(unsigned idx, vint64m1_t vs2, unsigned imm5); vint64m1_t __riscv_xt_vcpx3(unsigned
idx, vbool64_t mask, vint64m1_t vd,
                            vint64m1_t vs2, unsigned imm5);
vuint8m1_t __riscv_xt_vcpx3(unsigned idx, vuint8m1_t vs2, unsigned imm5);
vuint8m1_t __riscv_xt_vcpx3(unsigned idx, vbool8_t mask, vuint8m1_t vd, vuint8m1_t vs2, unsigned imm5);

vuint16m1_t __riscv_xt_vcpx3(unsigned idx, vuint16m1_t vs2, unsigned imm5);
vuint16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vuint16m1_t vd,
                             vuint16m1_t vs2, unsigned imm5); vuint32m1_t
__riscv_xt_vcpx3(unsigned idx, vuint32m1_t vs2, unsigned imm5);
vuint32m1_t __riscv_xt_vcpx3(unsigned idx, vbool32_t mask, vuint32m1_t vd,
                             vuint32m1_t vs2, unsigned imm5);
vuint64m1_t __riscv_xt_vcpx3(unsigned idx, vuint64m1_t vs2, unsigned imm5); vuint64m1_t
__riscv_xt_vcpx3(unsigned idx, vbool64_t mask, vuint64m1_t vd,
                             vuint64m1_t vs2, unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx, vfloat32m1_t vs2, unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx, vfloat32m1_t vs2, unsigned imm5,
                              unsigned frm);
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                              vfloat32m1_t vs2, unsigned imm5);
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                              vfloat32m1_t vs2, unsigned imm5, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx, vfloat64m1_t vs2, unsigned imm5);
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx, vfloat64m1_t vs2, unsigned imm5,
                              unsigned frm);
```

(Continued on next page)

(Continued from previous page)

```
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                              vfloat64m1_t vs2, unsigned imm5);
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                              vfloat64m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vint8m1_t vs3, vint8m1_t vs2,
                      vint8m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool8_t mask, vint8m1_t vs3, vint8m1_t vs2,
                      vint8m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                      vint16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vint16m1_t vs3,
                      vint16m1_t vs2, vint16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                      vint32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool32_t mask, vint32m1_t vs3,
                      vint32m1_t vs2, vint32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                      vint64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool64_t mask, vint64m1_t vs3,
                      vint64m1_t vs2, vint64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2,
                      vuint8m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
                      vuint8m1_t vs2, vuint8m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2,
                      vuint16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                      vuint16m1_t vs2, vuint16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                      vuint32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                      vuint32m1_t vs2, vuint32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                      vuint64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                      vuint64m1_t vs2, vuint64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                      vfloat32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                      vfloat32m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                      vfloat32m1_t vs2, vfloat32m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
```

(Continued on next page)

(Continued from previous page)

```
                                    vfloat32m1_t vs2, vfloat32m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                    vfloat64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                    vfloat64m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, vfloat64m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, vfloat64m1_t vs1, unsigned frm);
vint8m1_t __riscv_xt_vcpx5(unsigned idx, vint8m1_t vs2, vint8m1_t vs1);
vint8m1_t __riscv_xt_vcpx5(unsigned idx, vbool8_t mask, vint8m1_t vd,
                                    vint8m1_t vs2, vint8m1_t vs1);
vint16m1_t __riscv_xt_vcpx5(unsigned idx, vint16m1_t vs2, vint16m1_t vs1);
vint16m1_t __riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vint16m1_t vd,
                                    vint16m1_t vs2, vint16m1_t vs1);
vint32m1_t __riscv_xt_vcpx5(unsigned idx, vint32m1_t vs2, vint32m1_t vs1);
vint32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vint32m1_t vd,
                                    vint32m1_t vs2, vint32m1_t vs1);
vint64m1_t __riscv_xt_vcpx5(unsigned idx, vint64m1_t vs2, vint64m1_t vs1); vint64m1_t __riscv_xt_vcpx5(unsigned idx,
vbool64_t mask, vint64m1_t vd,
                                    vint64m1_t vs2, vint64m1_t vs1);
vuint8m1_t __riscv_xt_vcpx5(unsigned idx, vuint8m1_t vs2, vuint8m1_t vs1);
vuint8m1_t __riscv_xt_vcpx5(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                                    vuint8m1_t vs2, vuint8m1_t vs1);
vuint16m1_t __riscv_xt_vcpx5(unsigned idx, vuint16m1_t vs2, vuint16m1_t vs1);
vuint16m1_t __riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vuint16m1_t vd,
                                    vuint16m1_t vs2, vuint16m1_t vs1);
vuint32m1_t __riscv_xt_vcpx5(unsigned idx, vuint32m1_t vs2, vuint32m1_t vs1);
vuint32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vuint32m1_t vd,
                                    vuint32m1_t vs2, vuint32m1_t vs1);
vuint64m1_t __riscv_xt_vcpx5(unsigned idx, vuint64m1_t vs2, vuint64m1_t vs1); vuint64m1_t __riscv_xt_vcpx5(unsigned idx,
vbool64_t mask, vuint64m1_t vd,
                                    vuint64m1_t vs2, vuint64m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vfloat32m1_t vs2, vfloat32m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vfloat32m1_t vs2, vfloat32m1_t vs1,
                                    unsigned frm);
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                                    vfloat32m1_t vs2, vfloat32m1_t vs1);
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                                    vfloat32m1_t vs2, vfloat32m1_t vs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx5(unsigned idx, vfloat64m1_t vs2, vfloat64m1_t vs1);
vfloat64m1_t __riscv_xt_vcpx5(unsigned idx, vfloat64m1_t vs2, vfloat64m1_t vs1,
                                    unsigned frm);
```

(Continued on next page)

(Continued from previous page)

```
vfloat64m1_t __riscv_xt_vcpx5(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                              vfloat64m1_t vs2, vfloat64m1_t vs1);
vfloat64m1_t __riscv_xt_vcpx5(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                              vfloat64m1_t vs2, vfloat64m1_t vs1, unsigned frm);
void __riscv_xt_vcpx6(unsigned idx, vint8m1_t vs3, vint8m1_t vs2, int8_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool8_t mask, vint8m1_t vs3, vint8m1_t vs2,
                      int8_t rs1); void
__riscv_xt_vcpx6(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                      int16_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool16_t mask, vint16m1_t vs3,
                      vint16m1_t vs2, int16_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                      int32_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool32_t mask, vint32m1_t vs3,
                      vint32m1_t vs2, int32_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2,
                      uint8_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
                      vuint8m1_t vs2, uint8_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2,
                      uint16_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                      vuint16m1_t vs2, uint16_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                      uint32_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                      vuint32m1_t vs2, uint32_t rs1);
vint8m1_t __riscv_xt_vcpx7(unsigned idx, vint8m1_t vs2, int8_t rs1);
vint8m1_t __riscv_xt_vcpx7(unsigned idx, vbool8_t mask, vint8m1_t vd,
                           vint8m1_t vs2, int8_t rs1);
vint16m1_t __riscv_xt_vcpx7(unsigned idx, vint16m1_t vs2, int16_t rs1); vint16m1_t __riscv_xt_vcpx7(unsigned idx,
vbool16_t mask, vint16m1_t vd,
                            vint16m1_t vs2, int16_t rs1);
vint32m1_t __riscv_xt_vcpx7(unsigned idx, vint32m1_t vs2, int32_t rs1);
vint32m1_t __riscv_xt_vcpx7(unsigned idx, vbool32_t mask, vint32m1_t vd,
                            vint32m1_t vs2, int32_t rs1); vuint8m1_t
__riscv_xt_vcpx7(unsigned idx, vuint8m1_t vs2, uint8_t rs1);
vuint8m1_t __riscv_xt_vcpx7(unsigned idx, vbool8_t mask, vuint8m1_t vd,
                            vuint8m1_t vs2, uint8_t rs1);
vuint16m1_t __riscv_xt_vcpx7(unsigned idx, vuint16m1_t vs2, uint16_t rs1);
vuint16m1_t __riscv_xt_vcpx7(unsigned idx, vbool16_t mask, vuint16m1_t vd,
                             vuint16m1_t vs2, uint16_t rs1);
vuint32m1_t __riscv_xt_vcpx7(unsigned idx, vuint32m1_t vs2, uint32_t rs1);
```

(Continued on next page)

(Continued from previous page)

```
vuint32m1_t __riscv_xt_vcpx7(unsigned idx, vbool32_t mask, vuint32m1_t vd,
                             vuint32m1_t vs2, uint32_t rs1);
void __riscv_xt_vcpx8(unsigned idx, vint8m1_t vs3, vint8m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool8_t mask, vint8m1_t vs3, vint8m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vint16m1_t vs3, vint16m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vint16m1_t vs3,
                      vint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vint32m1_t vs3, vint32m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool32_t mask, vint32m1_t vs3,
                      vint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool64_t mask, vint64m1_t vs3,
                      vint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vuint8m1_t vs3, vuint8m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool8_t mask, vuint8m1_t vs3,
                      vuint8m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vuint16m1_t vs3, vuint16m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vuint16m1_t vs3,
                      vuint16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vuint32m1_t vs3, vuint32m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool32_t mask, vuint32m1_t vs3,
                      vuint32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                      vuint64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                      unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                      vfloat32m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                      vfloat32m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
```

(Continued on next page)

(Continued from previous page)

```
                                    unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                    unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                    float fs1);
void __riscv_xt_vcpx9(unsigned idx, vfloat32m1_t vs3, vfloat32m1_t vs2,
                                    float fs1, unsigned frm); void
__riscv_xt_vcpx9(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                    vfloat32m1_t vs2, float fs1);
void __riscv_xt_vcpx9(unsigned idx, vbool32_t mask, vfloat32m1_t vs3,
                                    vfloat32m1_t vs2, float fs1, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                    double fs1);
void __riscv_xt_vcpx9(unsigned idx, vfloat64m1_t vs3, vfloat64m1_t vs2,
                                    double fs1, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, double fs1);
void __riscv_xt_vcpx9(unsigned idx, vbool64_t mask, vfloat64m1_t vs3,
                                    vfloat64m1_t vs2, double fs1, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx10(unsigned idx, vfloat32m1_t vs2, float fs1);
vfloat32m1_t __riscv_xt_vcpx10(unsigned idx, vfloat32m1_t vs2, float fs1,
                                          unsigned frm);
vfloat32m1_t __riscv_xt_vcpx10(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                                          vfloat32m1_t vs2, float fs1);
vfloat32m1_t __riscv_xt_vcpx10(unsigned idx, vbool32_t mask, vfloat32m1_t vd,
                                          vfloat32m1_t vs2, float fs1, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx10(unsigned idx, vfloat64m1_t vs2, double fs1); vfloat64m1_t __riscv_xt_vcpx10(unsigned
idx, vfloat64m1_t vs2, double fs1,
                                          unsigned frm);
vfloat64m1_t __riscv_xt_vcpx10(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                                          vfloat64m1_t vs2, double fs1);
vfloat64m1_t __riscv_xt_vcpx10(unsigned idx, vbool64_t mask, vfloat64m1_t vd,
                                          vfloat64m1_t vs2, double fs1, unsigned frm);
```

5.6.5.2 RV64 section

The following intrinsic interfaces are only available on RV64 platforms.

```
void __riscv_xt_vcpx6(unsigned idx, vint64m1_t vs3, vint64m1_t vs2,
                      int64_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool64_t mask, vint64m1_t vs3,
                      vint64m1_t vs2, int64_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vuint64m1_t vs3, vuint64m1_t vs2,
                      uint64_t rs1);
void __riscv_xt_vcpx6(unsigned idx, vbool64_t mask, vuint64m1_t vs3,
                      vuint64m1_t vs2, uint64_t rs1);
vint64m1_t __riscv_xt_vcpx7(unsigned idx, vint64m1_t vs2, int64_t rs1); vint64m1_t __riscv_xt_vcpx7(unsigned idx,
vbool64_t mask, vint64m1_t vd,
                                    vint64m1_t vs2, int64_t rs1);
vuint64m1_t __riscv_xt_vcpx7(unsigned idx, vuint64m1_t vs2, uint64_t rs1);
vuint64m1_t __riscv_xt_vcpx7(unsigned idx, vbool64_t mask, vuint64m1_t vd, vuint64m1_t vs2, uint64_t rs1);
```

5.6.5.3 Zvfh/Zvfhmin extension

The following intrinsic interfaces are available only when the Zvfh or Zvfhmin extension is supported.

```
void __riscv_xt_vcpx0(unsigned idx, vfloat16m1_t vs2); void __riscv_xt_vcpx0(unsigned
idx, vfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vfloat16m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vfloat16m1_t vs2,
                      unsigned frm);
vfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vfloat16m1_t vs2);
vfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vfloat16m1_t vs2, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2);
vfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vfloat16m1_t vs2, unsigned imm5,
                      unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vfloat16m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vfloat16m1_t vs2,
                      unsigned imm5, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vfloat16m1_t vs2, unsigned imm5);
vfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vfloat16m1_t vs2, unsigned imm5,
                              unsigned frm);
vfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2, unsigned imm5);
vfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2, unsigned imm5, unsigned frm);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx4(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      vfloat16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      vfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                      vfloat16m1_t vs2, vfloat16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vfloat16m1_t vs3, vfloat16m1_t vs2, vfloat16m1_t vs1,
                      unsigned frm);
vfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vfloat16m1_t vs2, vfloat16m1_t vs1);
vfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vfloat16m1_t vs2, vfloat16m1_t vs1,
                              unsigned frm);
vfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2, vfloat16m1_t vs1);
vfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                              vfloat16m1_t vs2, vfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                      vfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                      vfloat16m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      _Float16 fs1);
void __riscv_xt_vcpx9(unsigned idx, vfloat16m1_t vs3, vfloat16m1_t vs2,
                      _Float16 fs1, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                      vfloat16m1_t vs2, _Float16 fs1);
void __riscv_xt_vcpx9(unsigned idx, vbool16_t mask, vfloat16m1_t vs3,
                      vfloat16m1_t vs2, _Float16 fs1, unsigned frm);
vfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vfloat16m1_t vs2, _Float16 fs1);
vfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vfloat16m1_t vs2, _Float16 fs1,
                               unsigned frm);
vfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                               vfloat16m1_t vs2, _Float16 fs1);
vfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vbool16_t mask, vfloat16m1_t vd,
                               vfloat16m1_t vs2, _Float16 fs1, unsigned frm);
```

5.6.5.4 Zvfbfmin extension

The following intrinsic interfaces are only available when the Zvfbfmin extension is supported.

```
void __riscv_xt_vcpx0(unsigned idx, vbfloat16m1_t vs2); void __riscv_xt_vcpx0(unsigned
idx, vbfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2);
void __riscv_xt_vcpx0(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2,
                      unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbfloat16m1_t vs2);
vbfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbfloat16m1_t vs2, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2);
vbfloat16m1_t __riscv_xt_vcpx1(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2, unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vbfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbfloat16m1_t vs2, unsigned imm5,
                      unsigned frm);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2,
                      unsigned imm5);
void __riscv_xt_vcpx2(unsigned idx, vbool16_t mask, vbfloat16m1_t vs2,
                      unsigned imm5, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbfloat16m1_t vs2, unsigned imm5);
vbfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbfloat16m1_t vs2, unsigned imm5,
                               unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2, unsigned imm5);
vbfloat16m1_t __riscv_xt_vcpx3(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2, unsigned imm5, unsigned frm); void
__riscv_xt_vcpx4(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      vbfloat16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      vbfloat16m1_t vs1, unsigned frm);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, vbfloat16m1_t vs1);
void __riscv_xt_vcpx4(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, vbfloat16m1_t vs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vbfloat16m1_t vs2,
                               vbfloat16m1_t vs1);
vbfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vbfloat16m1_t vs2,
                               vbfloat16m1_t vs1, unsigned frm); vbfloat16m1_t
__riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2, vbfloat16m1_t vs1);
vbfloat16m1_t __riscv_xt_vcpx5(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                               vbfloat16m1_t vs2, vbfloat16m1_t vs1,
                               unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      unsigned imm5);
```

(Continued on next page)

(Continued from previous page)

```
void __riscv_xt_vcpx8(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      unsigned imm5, unsigned frm);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, unsigned imm5);
void __riscv_xt_vcpx8(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, unsigned imm5, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      __bf16 fs1);
void __riscv_xt_vcpx9(unsigned idx, vbfloat16m1_t vs3, vbfloat16m1_t vs2,
                      __bf16 fs1, unsigned frm);
void __riscv_xt_vcpx9(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, __bf16 fs1);
void __riscv_xt_vcpx9(unsigned idx, vbool16_t mask, vbfloat16m1_t vs3,
                      vbfloat16m1_t vs2, __bf16 fs1, unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vbfloat16m1_t vs2, __bf16 fs1); vbfloat16m1_t __riscv_xt_vcpx10(unsigned
idx, vbfloat16m1_t vs2, __bf16 fs1,
                                unsigned frm);
vbfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                                vbfloat16m1_t vs2, __bf16 fs1);
vbfloat16m1_t __riscv_xt_vcpx10(unsigned idx, vbool16_t mask, vbfloat16m1_t vd,
                                vbfloat16m1_t vs2, __bf16 fs1, unsigned frm);
```

5.6.6 Xxtccef Interface

5.6.6.1 Basic Set

```
void __riscv_xt_fcpx0_f32(unsigned idx, float fs1);
float __riscv_xt_fcpx1_f32(unsigned idx, float fs1); void __riscv_xt_fcpx2_f32(unsigned
idx, float fs1, float fs2);
float __riscv_xt_fcpx3_f32(unsigned idx, float fs1, float fs2);
void __riscv_xt_fcpx4_f32(unsigned idx, float fs3, float fs1, float fs2);
float __riscv_xt_fcpx5_f32(unsigned idx, float fd, float fs1, float fs2);
float __riscv_xt_fcpx6_f32(unsigned idx, float fs2, unsigned imm5);

void __riscv_xt_fcpx0_f64(unsigned idx, double fs1);
double __riscv_xt_fcpx1_f64(unsigned idx, double fs1);
void __riscv_xt_fcpx2_f64(unsigned idx, double fs1, double fs2);
double __riscv_xt_fcpx3_f64(unsigned idx, double fs1, double fs2);
void __riscv_xt_fcpx4_f64(unsigned idx, double fs3, double fs1, double fs2);
double __riscv_xt_fcpx5_f64(unsigned idx, double fd, double fs1, double fs2);
double __riscv_xt_fcpx6_f64(unsigned idx, double fs2, unsigned imm5);
```

5.6.6.2 Zfhmin extension

The following intrinsic interfaces are only available when the Zfhmin extension is supported.

```
void __riscv_xt_fcpx0_f16(unsigned idx, _Float16 fs1);
_Float16 __riscv_xt_fcpx1_f16(unsigned idx, _Float16 fs1);
void __riscv_xt_fcpx2_f16(unsigned idx, _Float16 fs1, _Float16 fs2);
_Float16 __riscv_xt_fcpx3_f16(unsigned idx, _Float16 fs1, _Float16 fs2);
void __riscv_xt_fcpx4_f16(unsigned idx, _Float16 fs3, _Float16 fs1,
                                                    _Float16 fs2);
_Float16 __riscv_xt_fcpx5_f16(unsigned idx, _Float16 fd, _Float16 fs1,
                                                          _Float16 fs2);
_Float16 __riscv_xt_fcpx6_f16(unsigned idx, _Float16 fs2, unsigned imm5);
```

5.6.6.3 Zfbfmin extension

The following intrinsic interfaces are only available when the Zfbfmin extension is supported.

```
void __riscv_xt_fcpx0_bf16(unsigned idx, __bf16 fs1);
__bf16 __riscv_xt_fcpx1_bf16(unsigned idx, __bf16 fs1); void __riscv_xt_fcpx2_bf16(unsigned
idx, __bf16 fs1, __bf16 fs2);
__bf16 __riscv_xt_fcpx3_bf16(unsigned idx, __bf16 fs1, __bf16 fs2);
void __riscv_xt_fcpx4_bf16(unsigned idx, __bf16 fs3, __bf16 fs1, __bf16 fs2);
__bf16 __riscv_xt_fcpx5_bf16(unsigned idx, __bf16 fd, __bf16 fs1, __bf16 fs2); __bf16 __riscv_xt_fcpx6_bf16(unsigned idx, __bf16
fs2, unsigned imm5);
```

5.6.7 Code Examples

The following is a C code example that shows how to use the Xxtccef extended intrinsic interface.

```
#include <riscv_xt_cce.h>

double test_fcpx3_f64(double a, double b)
{
    return __riscv_xt_fcpx3_f64(1, a, b); // idx=1, fs1=a, fs2=b
}
```

The code first includes the header file riscv_xt_cce.h, which declares the intrinsic function in the general coprocessor extension. This code calls the __riscv_xt_fcpx3_f64 interface, which is the double-precision floating-point type interface corresponding to the fcpx3 instruction. The idx parameter is 1, indicating that the instruction is sent to the coprocessor numbered 1. The fs1 parameter is a and the fs2 parameter is b. These two parameters are the two source operands of the fcpx3 instruction. The value of the destination register of the fcpx3 instruction is the return value of this function.

Take the C920V3 (c920v3-cp) that supports the general coprocessor interface extension as an example. Save the above code as test.c and use The riscv64-unknown-linux-gnu-gcc -mcpu=c920v3-cp -c test.c command compiles the target file test.o.

5.7 RISC-V Vector Usage Instructions

XuanTie GNU compiler versions V3.0.0 and later (inclusive) support RISC-V Vector Intrinsic V1.0-RC2 and support switching RVV variable types to Fixed-length mode, compatible with RISC-V Vector Intrinsic V0.10 used in the old version. Supports RISC-V Vector V1.0 automatic vectorization function

XuanTie LLVM compiler versions V2.0.0 and later (inclusive) support RISC-V Vector Intrinsic V1.0-RC2. Support RISC-V Vector V1.0 automatic Vectorized functionality.

Note: The Intrinsic interface list of RISC-V Vector Intrinsic V0.10 includes "Xuantie 900 Series RVV-1.0 Intrinsic Manual.pdf" and "Xuantie 900 Series RVV-0.7.1 Intrinsic Manual.pdf".

5.7.1 How to use RISC-V Vector V1.0 Intrinsic

RISC-V Vector V1.0 can be directly referenced in the XuanTie GNU/LLVM compiler RISC-V Vector Intrinsic V1.0-RC2 Perform Intrinsic programming, As shown in Example 1. You can also add the option -mrvv-v0p10-compatible to support RISC-V Vector Intrinsic V0.10, as shown in Example 2.

Example 1:

```
#include <riscv_vector.h>

void
add_vectorized (int *c, int *a, int *b, int N)
{
    vint32m1_t va;
    vint32m1_t vb;
    vint32m1_t vc;

    size_t gvl;
    for (; N > 0; N -= gvl)
    {
        gvl = __riscv_vsetvl_e32m1(N);
        va = __riscv_vle32_v_i32m1(a, gvl);
        a += gvl;
        vb = __riscv_vle32_v_i32m1(b, gvl);
        b += gvl;
        vc = __riscv_vadd_vv_i32m1(va, vb, gvl);
        __riscv_vse32_v_i32m1(c, vc, gvl);
        c += gvl;
    }
}
```

Compile options:

```
-mcpu=c908v -O2
```

The generated instruction sequence:

```
add_vectorized:
        ble     a3,zero,.L8
.L3:
        vsetvli a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        subw    a3,a3,a5
        sh2add  a1,a5,a1
        vle32.v v2,0(a2)
        sh2add  a2,a5,a2
        vadd.vv v1,v1,v2
        vse32.v v1,0(a0)
        sh2add  a0,a5,a0
        bgt     a3,zero,.L3
.L8:
        ret
```

Example 2:

```
#include <riscv_vector.h>

void
add_vectorized (int *c, int *a, int *b, int N)
{
    vint32m1_t va;
    vint32m1_t vb;
    vint32m1_t vc;

    size_t gvl;
    for (; N > 0; N -= gvl)
    {
        gvl = vsetvl_e32m1(N);
        va = vle32_v_i32m1(a, gvl);
        a += gvl;
        vb = vle32_v_i32m1(b, gvl);
        b += gvl;
        vc = vadd_vv_i32m1(va, vb, gvl);
        vse32_v_i32m1(c, vc, gvl);
        c += gvl;
    }
}
```

Compile options:

```
-mcpu=c908v -O2 -mrvv-v0p10-compatible
```

The generated instruction sequence:

```
add_vectorized:
        ble     a3,zero,.L8
.L3:
        vsetvli a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        subw    a3,a3,a5
        sh2add  a1,a5,a1
        vle32.v v2,0(a2)
        sh2add  a2,a5,a2
        vadd.vv v1,v1,v2
        vse32.v v1,0(a0)
        sh2add  a0,a5,a0
        bgt     a3,zero,.L3
.L8:
        ret
```

5.7.2 How to use RISC-V Vector V1.0 automatic vectorization

Both the GNU and LLVM compilers support automatic vectorization. The GNU compiler needs to enable the compilation option -mrvv-auto-vectorize, and the LLVM compiler needs to enable the compilation option -mrvv-auto-vectorize.

The translator is not enabled by default on the BlackTie series CPUs. It can be enabled with the options "-mllvm -xt-enable-vectorization", as shown in Example 3.

Example 3:

```
#include <riscv_vector.h>

void
add_scalar (int *restrict a, int *restrict b, int N)
{
    for (int i = 0; i < N; i++)
        a[i] = 10 + b[i];
}
```

XuanTie GNU compiler compilation options:

```
-mcpu=c908v -O2 -mrvv-auto-vectorize
```

XuanTie LLVM compiler compilation options:

```
-mcpu=c908v -O2 -mllvm -xt-enable-vectorization
```

The generated instruction sequence:

```
add_vectorized:
        ble     a3,zero,.L8
.L3:
        vsetvli a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        subw    a3,a3,a5
        sh2add  a1,a5,a1
        vle32.v v2,0(a2)
        sh2add  a2,a5,a2
        vadd.vv v1,v1,v2
        vse32.v v1,0(a0)
        sh2add  a0,a5,a0
        bgt     a3,zero,.L3
.L8:
        ret
```

Note: RISC-V Vector V0.7.1 does not support automatic vectorization.

5.7.3 How to use RISC-V Vector V0.7.1 Intrinsic

In the XuanTie GNU compiler, v0p7 and xtheadvector are equivalent. Please refer to the XuanTie ISA extension specification In the example 4, you can use the XTheadvector related content to perform intrinsic programming. You can also add the option -mrvv-v0p10-compatible to support RISC-V Vector Intrinsic V0.10, as shown in example 5.

Example 4:

```
#include <riscv_vector.h>

void
add_vectorized (int *c, int *a, int *b, int N)
{
    vint32m1_t va;
    vint32m1_t vb;
    vint32m1_t vc;

    size_t gvl;
    for (; N > 0; N -= gvl)
    {
        gvl = __riscv_vsetvl_e32m1(N);
        va = __riscv_vle32_v_i32m1(a, gvl);
        a += gvl;
        vb = __riscv_vle32_v_i32m1(b, gvl);
        b += gvl;
```

(Continued on next page)

(Continued from previous page)

```
        vc = __riscv_vadd_vv_i32m1(va, vb, gvl);
        __riscv_vse32_v_i32m1(c, vc, gvl);
        c += gvl;
    }
}
```

Compile options:

```
-mcpu=c906fdv -O2
```

The generated instruction sequence:

```
add_vectorized:
        ble     a3,zero,.L8
.L3:
        vsetvli a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        subw    a3,a3,a5
        sh2add  a1,a5,a1
        vle32.v v2,0(a2)
        sh2add  a2,a5,a2
        vadd.vv v1,v1,v2
        vse32.v v1,0(a0)
        sh2add  a0,a5,a0
        bgt     a3,zero,.L3
.L8:
        ret
```

Example 5:

```
#include <riscv_vector.h>

void
add_vectorized (int *c, int *a, int *b, int N)
{
    vint32m1_t va;
    vint32m1_t vb;
    vint32m1_t vc;

    size_t gvl;
    for (; N > 0; N -= gvl)
    {
        gvl = vsetvl_e32m1(N);
        va = vle32_v_i32m1(a, gvl);
        a += gvl;
```

(Continued on next page)

(Continued from previous page)

```
        vb = vle32_v_i32m1(b, gvl);
        b += gvl;
        vc = vadd_vv_i32m1(va, vb, gvl);
        vse32_v_i32m1(c, vc, gvl);
        c += gvl;
    }
}
```

Compile options:

```
-mcpu=c906fdv -O2 -mrvv-v0p10-compatible
```

The generated instruction sequence:

```
add_vectorized:
        ble     a3,zero,.L8
        .align 2
.L3:
        th.vsetvli      a5,a3,e32,m1
        th.vle.v        v1,0(a1)
        th.vle.v        v2,0(a2)
        slli    a4,a5,2
        subw    a3,a3,a5
        add     a1,a1,a4
        th.vadd.vv      v1,v1,v2
        add     a2,a2,a4
        th.vse.v        v1,0(a0)
        add     a0,a0,a4
        bgt     a3,zero,.L3
.L8:
        ret
```

Note: The XuanTie LLVM compiler does not support XTheadvector, only v0p7.

5.7.4 How to use RISC-V Vector V1.0/V0.7.1 Intrinsic fixed length

In versions of the XuanTie GNU compiler V3.0.0 and later (inclusive), RISC-V Vector V0.7.1 and RISC-V Vector V1.0 both support fixed-length mode programming. The length is specified by zvl in arch. For example, when the length is 128, the arch is zvl128b. Example 6:

```
#include <riscv_vector.h>

/* Sizeless objects with global scope. */
```

(Continued on next page)

(Continued from previous page)

```
vint8m1_t global_rvv_sc;
static vint8m1_t local_rvv_sc;
extern vint8m1_t extern_rvv_sc;
__thread vint8m1_t tls_rvv_sc;
_Atomic vint8m1_t atomic_rvv_sc;

struct rvv_fixed
{
    vfloat32m1_t a;
    vfloat32m1_t b;
} x;
```

RISC-V Vector V0.7.1 compilation options:

```
-march=rv64gcv0p7_zvl128b -mabi=lp64d -mrvv-vector-bits=zvl -mrvv-v0p10-compatible
```

RISC-V Vector V1.0 Compilation Options

```
-march=rv64gcv_zvl128b -mabi=lp64d -mrvv-vector-bits=zvl
```

Note: The XuanTie LLVM compiler does not support fixed-length mode programming.

Chapter 6 Linking object files to generate executable files

The linker combines the various object files to generate the final executable file. The content in the object file can be adjusted in order and position through the linker description file. Its basic command is

```
csky-elfabiv2-ld options input-file-list
```

in:

optionsLink **options**

input-file-list all input object files

This chapter contains the following parts:

6.1 How to link libraries

Library is the most commonly used means of packaging Application Programming Interface (API), which is divided into static library and dynamic library.

• Static library: A collection of object files, that is, a file packaged from many target files, usually with ".a" as the file extension.

• Dynamic library: also known as dynamic shared objects (DSO), abbreviated as common objects, generally with ".so" as the file extension.

6.1.1 Generation of library files

• Static library generation: First use the compiler to generate each object file, and then use csky-elfabiv2-ar to package and generate library files

```
csky-elfabiv2-gcc -c csky_a.c -o part_a.o
csky-elfabiv2-gcc -c csky_b.c -o part_b.o
csky-elfabiv2-ar -r libcsky.a part_a.o part_b.o
```

• Dynamic library generation: Use the compiler and add options -shared -fPIC to generate library files (supported only by Linux tools)

```
csky-linux-gnuabiv2-gcc -shared -fPIC csky.c -o libcsky.so
```

6.1.2 Link Library

Whether it is a static library or a dynamic library, their naming follows a unified rule, that is, lib[name].[a/so]. The way to link a library is to add the option -l[name] and -L [libpath]. where libpath refers to the path where the lib[name].[a/so] file is located.

For example, if you need to link libcsky.a, you need to add the link option -lcsky -L [libcsky.a path]

6.2 Memory layout of code segment and data segment in target file

Each object file is composed of multiple segments such as code segments and data segments. The linking process is actually to merge similar segments of each input object file. The process of generating the final executable file after construction is shown in Figure 6.1.

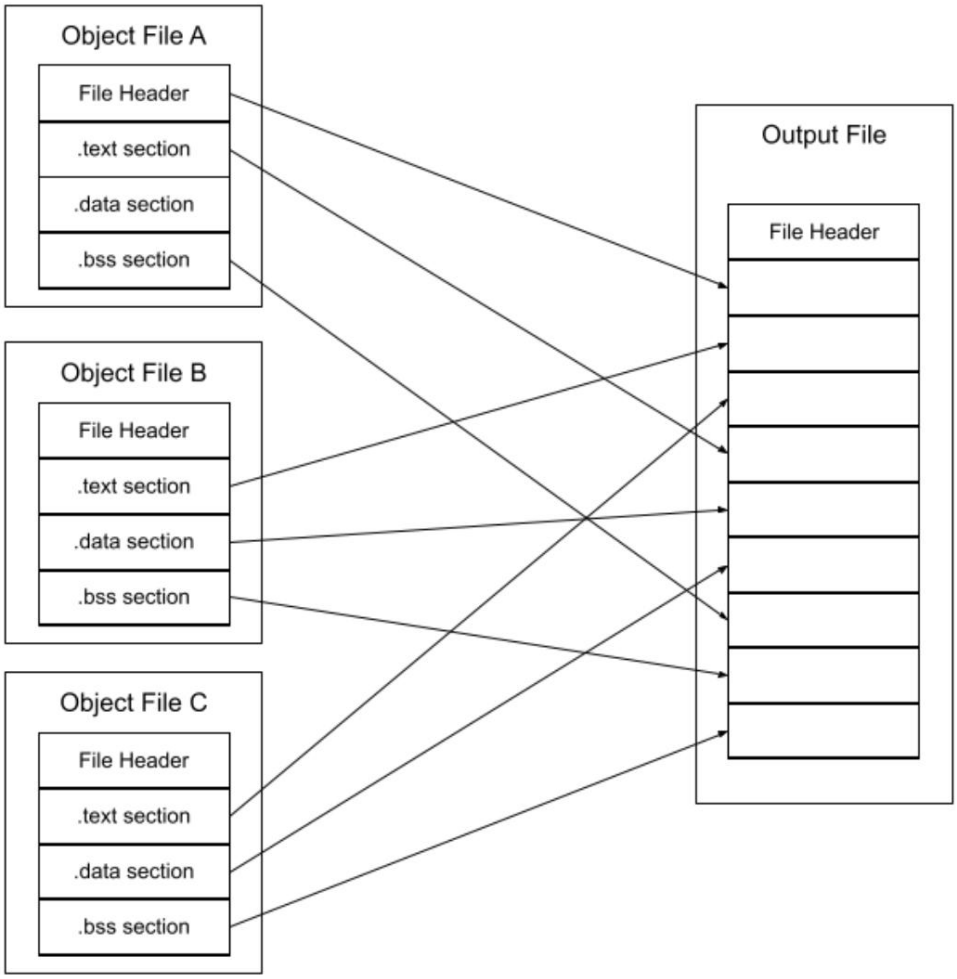

Figure 6.1: Linking process

In order to accurately control the layout of the input file segments in the output file, the linker designed a linker script to complete this arduous task.

Specified by the linker option -T [linkscript]. If you do not specify a link script, the linker will use the default link script to control the link process, which will

After merging, they are placed at a fixed address, which generally makes it difficult to meet the needs of embedded software developers.

A simple linker script looks like this:

```
ENTRY(__start)

MEMORY
{
        INST : ORIGIN = 0x00000000      ,   LENGTH = 0x00020000      /* ROM */
        DATA : ORIGIN = 0x00400000      ,   LENGTH = 0x00004000      /* RAM */
        EEPROM : ORIGIN = 0x00600000    ,   LENGTH = 0x00010000
}
PROVIDE (__stack = 0x00404000 - 0x8);
SECTIONS
{
        .text : {
                        .  = ALIGN(0x4);
                        *crt0.o(.exp_table)
                        *(.text*)
                        .  = ALIGN (0x10);
        } > INST
        .rodata :       {
                        .  = ALIGN(0x4);
                        *(.rdata*)
                        .  = ALIGN(0x10);
        } > DATA
        .data : {
                        .  = ALIGN(0x4);
                        *(.data*)
                        .  = ALIGN(0x10);
        } > DATA
        .bss :      {
                        .  = ALIGN(0x4);
                        *(.bss*)
                        *(COMMON)
                        .  = ALIGN(0x10);
        } > DATA
}
```

Linker scripts contain a lot of complex syntax to control the generation of executable files, and several commonly used syntaxes control the memory layout of segments in the target file.

The memory layout is the address where the segment is stored in the target file. The address is divided into two types: virtual address and load address:

- Virtual address: VMA, Virtual Memory Address, represents the address of code or data at runtime
- Load address: LMA, Load Memory Address

In most cases, VMA and LMA are the same, but in some embedded systems, especially those where the program is placed in ROM, LMA and

The VMA is different. The method of specifying the output segment of the target file is shown in the above code.

1. Use the MEMORY expression to define the memory area. The format is as follows:

```
MEMORY
    {
        name [(attr)] : ORIGIN = origin, LENGTH = len
         …
    }
```

2. Use "> [memory region]" in the section expression to specify the VMA of the output segment. The format is as follows:

```
section:
    {
        output-section-command
        output-section-command
         …
    } [>region] [AT>lma_region]
```

3. If not specified, the VMA and LMA of the output segment are the same. If you want to specify the LMA, you can use "AT>[memory region]" in the section expression.

6.3 View the memory layout of the generated target file through ckmap

---

Note: The XuanTie 900 series toolchain currently does not support this function.

If you want to view the memory layout and other detailed information of the final generated target file, you can add the link option -ckmap=[output file name], Generate a ckmap file for viewing. ckmap mainly contains the following five parts:

1. Section Cross References

List all the calling relationships between sections in the following format:

```
([File A name])([Section name]) refers to ([File B name])([Section name]) for [Symbol name]
```

This part is composed of multiple of the above statements, indicating that a certain section of file A references a symbol defined in a certain section of file B.

2. Removing Unused input sections from the image

List all sections deleted after enabling the link option –gc-sections, in the following format:

```
Removing [segment name]([file name]), ([size] bytes).
...
[number] unused seciton(s) (total [size] bytes) removed from the image.
```

3. Image Symbol Table

List all local symbols and global symbols separately, and display the symbol's address, attribute, size, and segment name. The format is as follows:

```
Local Symbols
Symbol Name                                  Value    Type Size Section
...
Global Symbols
Symbol Name                                  Value    Type Size Section
...
```

The values of Type are as follows:

- w: Weak, weak symbol
- d: Debug, some auxiliary symbols needed for debugging, such as file names, segment names
- F: Function, function name
- f: filename, file name
- O: zero, symbolic name of the bss segment

4. Memory Map of the image

Displays the entry address of the target file and the memory layout of the input file segment in the output file in the following format:

```
Image Entry point: [Entry address]
Region [memory region name] (Base: [starting address], Size: [actual size], Max: [maximum value of the memory region])
Base Addr        Size        Type Attr    Idx Section Name                Object
 [Start Address] [Size]       [Type] [Attributes]  [Section number] [Section name]  [file name]
...
```

Each input segment linked to a memory region (Region) will be displayed in the above table, where Type has the following types:

- Code: Code snippet
- Data: Data segment
- PAD: for aligning padded areas
- LD_GEN: linker generated code

Attr has the following types:

- RO: Read Only, read only
- RW: Read Write, readable and writable

5. Image component sizes

Count the size of each input file's data in the target file, the format is as follows:

```
Code          RO Data       RW Data       ZI Data       Debug       Objectÿ
 ÿName
 [code segment size] [read-only data segment size] [read-write data segment size] [bss segment size] [debug information segment size]ÿ
 ÿ[Enter file name]
```

Chapter 7 Optimization

This chapter mainly introduces how to use the XuanTie compiler tool to optimize code size or performance, and the impact of the optimization level on the use of debugging functions.

This chapter contains the following parts:

7.1 Link-time Optimization

Link Time Optimization (LTO) enables the compiler to store the generated internal data structures (GIMPLE or LLVM IR) on disk, so that all compilation units can be optimized as a whole. The compiler will output the internal data structure to a special section of the .o file. When these .o files are linked together, the linker will collect the information in all these special sections. With this information, the optimizer can determine the dependencies between these modules and achieve more optimization. For example, the following is the source code of two C code files foo.c and bar.c. The main function in foo.c calls the foo function, and the foo function calls the bar function in bar.c, and the bar function is just a simple addition operation:

```
int foo(int fa, int fb)
{
        return bar(fa, fb);
}

int main()
{
        return foo(12, 3);
}
```

```
int bar(int a, int b)
{
        return a + b;
}
```

The Xuantie compiler uses the link-time optimization function through the -flto option. It should be noted that we must use this option both when compiling and linking the program, such as:

```
csky-elfabiv2-gcc -c -O2 -flto foo.c
csky-elfabiv2-gcc -c -O2 -flto bar.c
csky-elfabiv2-gcc -o myprog -flto -O2 foo.o bar.o
```

Another more common way is:

```
csky-elfabiv2-gcc -o myprog -flto -O2 foo.c bar.c
```

In this example, the functions in the two files call each other. Without LTO, the disassembled code of the foo function calling the bar function is as follows:

```
foo:
        push    r15
        bsr     bar
        pop     r15

main:
        push    r15
        movi    r1, 3
        movi    r0, 12
        bsr     foo
        pop     r15
```

When LTO optimization is enabled, the main function does not need to call the foo function and then the bar function, but directly returns the result of the addition.

Of course, the instructions are better optimized when LTO is turned on:

```
bar.constprop.0:
        movi    r0, 15
        rts

main:
        push    r15
        bsr     bar.constprop.0
        pop     r15
```

7.2 Effects of Optimization Options on Debug Information

The optimized code has a certain impact on the debugging information, so we need to balance the functions of the two in certain situations.

Choosing to optimize performance will also have a different impact on the final result of the optimization.

Generally speaking, the relationship between the code compiled with the compiler option -O0 and the debugging information is the most accurate, and all the generated code structures can directly correspond to the corresponding

As the optimization level increases, the correspondence between the compiled code and the source code will become less and less, because the code optimized by the compiler has

Of course, users can also use the optimization option -Og to optimize the code according to their own needs if they do not want to use -O0.

Code, try to ensure the accuracy of code and debugging information.

7.3 Code Optimization Suggestions

This section mainly introduces some excellent programming experience and related technologies to improve the portability, efficiency and robustness of C and C++ code.

This section contains the following parts:

- Loop iteration condition optimization
- Loop unrolling optimization
- Reduce function parameter passing

7.3.1 Loop Iteration Condition Optimization

Loop structures are a common type of operation in programs. A lot of computing time will be consumed in these loops, so in time-sensitive environments, you need to pay close attention to these areas.

When writing the loop end condition judgment, please refer to the following coding guidelines first:

- Use simple conditionals
- The loop iteration variable is decremented to zero
- Use a counter of type **unsigned int**
- Use the iteration variable not equal to zero as the loop exit condition

The following two comparative examples introduce the loop operation of calculating n!. By comparing the generated assembly code, it can be seen that the code using the self-decrement operation can use one **The jbnez** assembly instruction is used to achieve the comparison jump function, and two instructions are required for the self-increment operation, first the comparison (cmp) and then the jump (jbf):

```
csky-elfabiv2-gcc -Os -S loop.c
```

- Increment operation:

```
int fact1(int n)
{
        int i, fact = 1;
        for (i = 1; i <= n; i++)
                fact *= i;
        return (fact);
}
```

```
fact1:
        movi        a3, 1
        mov         a2, a3
.L2:
        cmplt a0, a2
        jbf         .L3
        mov         a0, a3
        rts
```

(Continued on next page)

(Continued from previous page)

```
.L3:
        mult        a3, a3, a2
        addi        a2, a2, 1
        jbr         .L2
```

• Decrement operation:

```
int fact2(int n)
{
        unsigned int i, fact = 1;
        for (i = n; i != 0; i--)
                fact *= i;
        return (fact);
}
```

```
fact2:
        mov         a3, a0
        movi        a0, 1
.L2:
        jbnez a3, .L3
        rts
.L3:
        mult        a0, a0, a3
        subi        a3, a3, 1
        jbr         .L2
```

7.3.2 Loop Unrolling Optimization

After some short loops are unrolled, the performance can be improved, but the code will be correspondingly larger. After the loop is unrolled, the number of loops will be reduced, which will also reduce The number of branch jump instructions executed is reduced. Some small short loops will be fully expanded, and the loop cost will completely disappear. Under the optimization level -O3 , loop expansion In some cases, the feature is automatically enabled; otherwise, you need to manually unroll the loop in your code.

The following two comparative examples introduce the loop operation of data copying. In the case of loop unrolling, the performance loss caused by jumps can be reduced. In general, the operation The performance of the loop will be better than that of the unrolled loop, but the disadvantage is that the code size will increase:

```
csky-elfabiv2-gcc -Os -S loop.c
```

• Loop not unrolled:

```
int countbit1(unsigned int n, char *d, char *s)
{
        int bits = 0;
        while (n != 0)
```

(Continued on next page)

(Continued from previous page)

```
    {
        d[bits] = s[bits];
        bits++;
        n -= 1;
    }
    return bits;
}
```

```
countbit1:
    addu    a3, a2, a0
.L2:
    cmpne a2, a3
    jbt     .L3
    rts
.L3:
    ld.b    t0, (a2, 0)
    st.b    t0, (a1, 0)
    addi    a2, a2, 1
    addi    a1, a1, 1
    jbr     .L2
```

- Loop unrolling:

```
int countbit2(unsigned int n, char *d, char *s)
{
    int bits = 0;
    while (n != 0)
    {
        d[bits+0] = s[bits+0];
        d[bits+1] = s[bits+1];
        d[bits+2] = s[bits+2];
        d[bits+3] = s[bits+3];
        bits += 4;
        n -= 4;
    }
    return bits;
}
```

```
countbit2:
    movi    a3, 0
.L2:
    cmpne a0, a3
    jbt     .L3
```

(Continued on next page)

(Continued from previous page)

```
    rts
.L3:
    ld.b    t0, (a2, 0)
    st.b    t0, (a1, 0)
    ld.b    t0, (a2, 1)
    st.b    t0, (a1, 1)
    ld.b    t0, (a2, 2)
    st.b    t0, (a1, 2)
    ld.b    t0, (a2, 3)
    st.b    t0, (a1, 3)
    addi    a3, a3, 4
    addi    a2, a2, 4
    addi    a1, a1, 4
    jbr     .L2
```

7.3.3 Reduce function parameter passing

The following points should be noted when passing function parameters:

- In the case of abiv2, the function has 4 integer parameter registers. If the hard floating point function is used, there will be 4 floating point registers.
  In the case of multiple calls, the number of parameters should be reduced as much as possible so that the number can be within the range of registers to improve efficiency.
- In the C++ case, the implicit this pointer of a non-static function is passed via r0, so the number of parameter registers is reduced by one accordingly.
- Put the related parameters into a structure, and then pass the structure pointer to pass the parameters when calling the function. This reduces
  The number of registers used.
- Reduce the use of long long type parameters, which occupy 2 registers.
- When using soft floating point, reduce the use of double type parameters.

7.4 KO file size optimization

The current mainstream RISCV compilers, including the Xuantie compiler, will enable the relax function by default when compiling programs so that the programs can be optimized during linking. The KO file is a special ELF file. Many of its symbols are external symbols and cannot be determined during linking. Enabling the relax function will not only fail to optimize the KO file, but will increase the size of the KO file.

In addition, the Xuantie compiler optimizes the size of the KO file for the scenario where the relax function is turned off, and deletes some redundant information.

Therefore, when using the BlackTie compiler, you can add the -mno-relax option to reduce the size of the KO file.

Chapter 8 Programming Essentials

This chapter introduces several problems that developers often encounter during the development process, mainly including the following contents:

- Peripheral registers
- Impact of *Volatile* on Compilation Optimization
- Use of function stack
- *Inline* functions
- *Memory Barriers*
- Specify variables and function *sections*
- Assign functions and data to absolute addresses
- Delayed operation
- Custom *C* language standard input and output streams
- Basic *ABI* description
- Variable synchronization
- Notes on self-modifying code
- Use inline assembly
- *Newlib* implements reentrancy

8.1 Peripheral Registers

The operation of peripheral registers is a frequent scenario in embedded software development. Due to its special features (such as easy to be optimized by the compiler), this section is dedicated to This article introduces common development methods related to peripheral registers to help developers avoid unnecessary troubles.

8.1.1 Peripheral register description

This section explains how to use C language to describe peripheral registers so that the code can still be compiled into correct and efficient code when the compiler is optimized. The instruction sequence is kept nice and readable. Here is how:

1. Define the peripheral register flag macro, define the peripheral as volatile type, and modify the input peripheral register with the const attribute, so that the register It can only be read. When writing a value into it, the compiler will report a warning message.

```
#define __I volatile const
#define __O volatile
#define __IO volatile
```

2. Define the data structure corresponding to the peripheral register programming model and modify these registers with the corresponding IO. For example:

```
typedef struct {
 ...
        __I uint32_t RXD;
        __O uint32_t TXD;
        __IO uint32_t STATUS;
        __I uint32 RESERVERD[5];
...
}Device_Uart_Type;
```

3. Define the operation macros of peripheral registers, for example:

```
#define SOC_UART0    ((Device_Uart_Type *) SOC_UART0_BASE)
#define SOC_UART1    ((Device_Uart_Type *) SOC_UART1_BASE)
#define SOC_UART2    ((Device_Uart_Type *) SOC_UART2_BASE)
```

4. After defining the above macros, you can reference these macros in the program to read and write peripheral registers. For example:

```
Receive_buf[0] = SOC_UART0->RXD;
SOC_UART0->TXD = Receive_buf[0];
While (!(SOC_UART0->STATUS & UART_SENT_BIT));
```

Of course, users can further define SOC_UART0->TXD as Uart0_TXD to facilitate the operation of peripheral registers in the code.

8.1.2 Peripheral Bit Field Operations

Peripheral registers are usually split into multiple parts representing different functions. Therefore, in order to facilitate the control of peripheral registers, each part of the peripheral register can be represented by the bit field of the structure. Each field has a domain name, and operations are performed according to the domain name in the program. The following is an example of operating peripheral registers through the structure bit field:

```
//---------- SPI control register 0
typedef volatile union {
    unsigned int Word;
    struct {
        unsigned DSS        :4;
        unsigned FRF        :2;
        unsigned SPO        :1;
        unsigned SPH        :1;
```

(Continued on next page)

(Continued from previous page)

```
        unsigned SCR                :8;
        unsigned                    :16;
    } Bits;
} SPI_CR0_STR;

#define HMS_SPI_BASE                0x40003800
#define _SPI_CR0                    *(SPI_CR0_STR *)(HMS_SPI_BASE + 0x000) //SPI Controlÿ
 ÿRegister 0
#define SPI_CR0                     (_SPI_CR0).Word
#define SPI_CR0_DSS                 (_SPI_CR0).Bits.DSS
#define SPI_CR0_FRF                 (_SPI_CR0).Bits.FRF
#define SPI_CR0_SPO                 (_SPI_CR0).Bits.SPO
#define SPI_CR0_SPH                 (_SPI_CR0).Bits.SPH
#define SPI_CR0_SCR                 (_SPI_CR0).Bits.SCR

void test ()
{
        SPI_CR0_DSS = 7;            //8-bit data size
        SPI_CR0_FRF = 0;            //SPI frame mode
        SPI_CR0_SPO = 0;
        SPI_CR0_SPH = 0;            //SPI mode 00
        SPI_CR0_SCR = 0;            //clock post-scaler=1
}
```

8.1.3 -fstrict-volatile-bitfields Option

In addition to adding volatile when defining types, in most cases you also need to add the -fstrict-volatile-bitfields option when compiling to ensure that the generated pointers are The command operates the bit field with word as the minimum reading and writing unit.

8.2 Impact of Volatile on Compilation Optimization

(1) Variables that are used repeatedly will not be cached.

If the keyword Volatile is not added before the variable, the compiler will cache the value of the variable through a register if it finds that the variable has been used more than twice in a row. Instead of reading from the initial memory location each time. Volatile indicates that the value of the variable may be changed externally and needs to be re-accessed each time it is used, so the compiler The server does not perform cache optimization.

(2) No optimizations such as constant merging and constant propagation are performed.

The compiler's data flow analysis will analyze the assignment and use of variables to perform optimizations such as constant merging and constant propagation, further eliminating dead code. When these optimizations are needed, you can disable them by using the Volatile keyword. For example, in the following code, the if condition will not always be true.

```
Volatile int i = 1;
if (i)
 ...
```

8.3 Use of Function Stack

In C/C++, the stack is frequently used. The stack can store the following:

1. Local variables. Due to the limited number of registers, some local variables cannot be stored in registers, and the stack will allocate space for these variables.
2. Overflowing parameters. Because the number of parameters of the called function is greater than the number of parameter registers or because the parameter bit width is large, one parameter register is not enough to store the entire parameter, resulting in the parameter register being able to store only part of the parameters, and the remaining parameters or part of the parameters are overflowed, and their values are stored in the stack.
3. Return values that cannot be passed by registers. If the bit width of the return value is greater than the sum of the bit widths of the registers storing the return value, the return value is stored in the stack, such as a parameter of a structure type whose bit width is greater than 8 bytes.
4. The original value of the register. The original value of some registers needs to be protected, so using these registers to store local variables of the function requires the registers to be The original value is pushed onto the stack, and the stack value is stored into the register before the called function returns to the main function.
5. The return address of the function.
6. In addition to the above points, if the alloc() function is used, it will also occupy some memory in the stack.

In general, it is difficult to estimate the stack usage because code dependencies change as the program executes. Here is a way to estimate the stack usage:

1. Compilation uses -fstack-usage, which will generate a file with the suffix .su, which can be used to view the size of the stack.
2. Use the option –callgraph when linking, which will generate an html file to view the size of the stack.
3. Use the debugger to set a watchpoint at a stack location and observe the hit situation.

In order to avoid using the stack as much as possible, the program can reduce the demand for the stack, which is generally solved by the following methods:

1. Avoid local variables that are structure types or arrays.
2. Avoid recursive algorithms.
3. Don’t write too many variables in a function.
4. Use block scope and define variables in the required scope.

8.4 Inline functions

Inline functions are a way to balance code size and performance. GCC can use keywords such as "inline" to instruct the compiler to inline the required functions, but Whether to inline is determined by the compiler; in addition, the inline function can be forced to be inlined through the attribute keyword.

The definition of an inline function is usually placed in a header file because the compiler needs to know the content of the function definition when optimizing the inline function. The content and the call to the function must be in the same file, and the header file must be included when using the inline function.

8.4.1 Inlining

The keyword for declaring an inline function is "inline". If it is C90, use "__inline__", which is defined as follows:

```
inline int fun1(int x,int y)
{
    return x+y;
}

int fun2 (int xx, int yy)
{
    return 2*fun1(2,6);
}
```

8.4.2 Forced Inlining

When GCC does not inline any function, you can use the attribute keyword "always_inline" to force inlining. The specific declaration is as follows:

```
inline void foo (const char) __attribute__((always_inline));
```

8.4.3 Mixed use of inline functions and external calls

When there are inline function definitions and external function definitions of the same function in C language, the compiler only selects the code of the inline function.

8.5 Memory Barriers

A memory barrier instruction is a synchronization point in the random memory access operation of the CPU or compiler, so that all read and write operations before this point can be executed before the operation after this point can be started.

In CSKY, the definition of memory fence is relatively simple. It is actually a section of inline assembly instruction. The specific definition is as follows:

```
#define MEMORY_BARRIER __asm (""":::"memory");
```

8.6 Variable and Function Section Specification

Variables and functions can be assigned to a specific section through the gcc section attribute. The method of using it is to add __attribute__ ((section (“<section name>”))) to the definition or declaration of the variable or function. Please refer to the following example.

Assign functions to specific sections

```
extern void foobar (void) __attribute__ ((section (".bar")));
```

Assign variables to a specific section

```
char stack[10000] __attribute__ ((section (".STACK"))) = { 0 };
```

8.7 Assigning functions and data to absolute addresses

The method to specify functions and data to absolute addresses is:

1. First assign the function or data to a special section. See the method for assigning variables and function *sections* .
2. Then, when linking, modify the link script and assign the section to the corresponding address. See the memory layout of the code segment and data segment in the target file for the method.

The following example assigns the variable stack to address 0x500000.

• Definition of variables

```
char stack[10000] __attribute__ ((section (".STACK"))) = { 0 };
```

• Linker script modification, add memory region STACKR in MEMORY, and assign the STACK segment to the STACKR region in SECTIONS

```
ENTRY(__start)

MEMORY
{
        INST : ORIGIN = 0x00000000      ,  LENGTH = 0x00020000      /* ROM */
        DATA : ORIGIN = 0x00400000      ,  LENGTH = 0x00004000      /* RAM */
        STACKR : ORIGIN = 0x00500000    ,  LENGTH = 0x00010000
        EEPROM : ORIGIN = 0x00600000    ,  LENGTH = 0x00010000
}
PROVIDE (__stack = 0x00404000 - 0x8);
SECTIONS
{
        .text : {
                        .  = ALIGN(0x4);
                        *crt0.o(.exp_table)
                        *(.text*)
                        .  = ALIGN (0x10);
        } > INST
        .stack :        {
                .  = ALIGN(0x4);
                *(.STACK)
        } > STACKR
        .rodata :       {
                        .  = ALIGN(0x4);
                        *(.rdata*)
                        .  = ALIGN(0x10);
```

(Continued on next page)

(Continued from previous page)

```
        } > DATA
        .data : {
                        .  = ALIGN(0x4);
                        *(.data*)
                        .  = ALIGN(0x10);
        } > DATA
        .bss :      {
                        .  = ALIGN(0x4);
                        *(.bss*)
                        *(COMMON)
                        .  = ALIGN(0x10);
        } > DATA
}
```

8.8 Delayed Operation

In some MCU applications, there needs to be a certain interval between two operations. Some developers prefer to use an empty loop body to achieve a certain Delay (the specific delay time is calculated based on the number of instructions executed). This style of code has the following disadvantages:

1. The operation is dangerous. A useless empty operation is often deleted by the compiler directly, so there is no delay effect.
2. The delay time will vary with the optimization options of the compilation
3. Portability of functions at different frequencies

The following methods are recommended:

Abstract a delay function, such as delay_us(int val), which is written in assembly or inline assembly to prevent compiler optimization from affecting it. Delay functions such as the delay_us function can adjust the number of instructions executed according to the current system operating frequency to adapt to different system operating frequencies.

Assuming the system operates at 20Mhz, the delay function of 801 can be implemented as follows:

```
.text
.align 2
.global delay_us .type
delay_us, @function
delay_us:
        cmplti a0, 1
        bt .L2
.L1:
        subi a0, 1
        cmplti a0, 1
        .rept 17
        nop
        .endr
```

(Continued on next page)

(Continued from previous page)

```
        bf .L1
.L2:
        rts
.size delay_us, .-delay_us
```

8.9 Customizing C language standard input and output streams

Due to the particularity of embedded platform development, in the C language standard library provided by the current CSKY platform, the standard input scanf and output function printf will use hook functions fgetc and fputc to implement the corresponding input and output functions, thereby improving the flexibility of development. When developing, users need to provide user-defined fgetc and fputc functions if necessary.

8.10 Basic ABI Description

When users mix C and assembly code for development, they need to be concerned with the application binary interface, which describes how parameters are passed. How to store the return value. This section will briefly introduce the mechanism of passing and returning basic data type variables in the second edition of CSKY ABI.

8.10.1 Function parameter passing

There are two types of parameter transfer: one is basic data type (char, short, int, long, etc.), the other is aggregate type (array, structure, class, etc.). For the aggregate type parameter transfer method, please refer to *CSKY* Application Binary Interface Specification (Second Edition) .

First of all, it should be noted that the csky series compiler will perform expansion operations on the type of size less than 32 bits so that its size can be stored in a 32-bit register. The current CSKY ABI specification stipulates that the first four parameters with a size less than 4 bytes can be passed using registers r0-r3, and the remaining parameters are passed using stack slots. Take the following code as an example,

```
void bar(char ch, short sh, int i, long l, int rem1)
{
    int res = ch + sh + i + l + rem1;
}
```

ch, sh, i, l will be stored in the registers r0, r1, r2, r3 (or alias a0, a1, a2, a3) in sequence, and rem1 will be stored in the stack slot at the location of sp-8.

8.10.2 Function return value transmission

In addition to specifying how to pass data to the called function (callee), ABI also specifies how to read data from the called function (callee). This function requires the calling function and the called function to be implemented in a coordinated manner. The called function places a certain type of data in a predetermined position (r0/r1) or stack slot according to the ABI specification. The following example:

```
int bar(int a, int b)
{
```

(Continued on next page)

(Continued from previous page)

```
    return a + b;
}
```

The return value in the above code will be placed in the r0 register to be used by the calling function.

When the called function returns a value of aggregate type, such as a large structure array, the registers in the csky series CPU cannot provide enough space for the return value to be passed. The following example:

```
type struct
{
    int a;
    int b;
    int c;
    int d;
    int e;
}St;

St bar(int a, int b)
{
    St st;
    st.a = a;
    st.b = b;
    st.c = a + b;
    st.d = a - b;
    st.e = a          * b;
    return st;
}
```

The above code shows the rules for passing the return value of the aggregate type. According to the second edition of the CSKY ABI specification, the return value of the aggregate type will be implemented by indirect transmission. First, the calling function allocates a space in the calling function's call stack for the return value, and then passes the entry address of the space as the first parameter of the function call to the called function, and the called function uses the address to implement various operations. The equivalent C language code after the transformation is similar to the following form:

```
type struct
{
    int a;
    int b;
    int c;
    int d;
    int e;
}St;

void bar(St* st, int a, int b)
{
    st->a = a;
    st->b = b;
```

(Continued on next page)

(Continued from previous page)

```
    st->c = a + b;
    st->d = a - b;
    st->e = a       * b;
}
```

8.11 Variable Synchronization

Variable synchronization is a common problem in application development. Volatile declarations can be used to achieve variable synchronization. In multi-task programming, the CSKY architecture provides users with The idly4 instruction is provided to mask interrupts.

8.11.1 Using volatile synchronization variables

In the process of developing applications, we often encounter the problem of variable synchronization. For example, in a certain application scenario, the interrupt service program that receives data After that, the data is put into the receiving buffer and the global variable Received_flag is changed to notify the main program to process; the main program continuously reads the variable. When it is set, the processing function is called to process the data in the receiving buffer. Users often ignore the particularity of this global variable and directly use the ordinary application This may cause the application to fall into an infinite loop. Why is this happening?

Let's briefly analyze the reason. After a global variable is compiled and linked, the compiler tool will allocate a piece of memory space for the variable as its value. Storage space. When the user accesses (reads) the variable in C language, the compiler tool will generate the corresponding memory access instruction to obtain the value of the variable memory space. But when we repeatedly write the same statement in C language and keep reading the value of the same variable, how will the compiler handle this situation? When optimization is enabled, the compiler generates a corresponding memory access instruction for each C statement to obtain the memory value in turn. When optimization is enabled, the compiler generates only one memory access instruction. The instructions that are stored will always use the original value for processing and calculation, because the compiler believes that the value of the memory will not change in the current scope.

How to deal with this situation correctly? The simplest way is to use volatile to modify this global variable (Received_flag). The compiler will Variable access operations generate memory access instructions to obtain the latest value in memory for calculation.

8.11.2 Variable synchronization in multi-tasking programming

How to access and modify the same variable in different tasks (or threads) is a key issue in multi-task programming. In the structure, special instructions (idly4) are provided for users to use in such scenarios.

The instruction function of idly4 is defined as follows. After this instruction is executed, the execution of the following four instructions will block interrupts. If an exception occurs during the execution, the interrupt will be set. The high flag bit (C bit) notifies the user that an abnormality has occurred. The application example is as follows:

```
bmaski r1, 32                    ; Get all 1's constant
lrw r2, Semaphore                ; Get the semaphore pointer in memory
idly4                            ; Start an uninterruptible 4-instruction queue
ld.w r3,(r2,0)                   ; Read the semaphore from memory
bt Sequence_failed               ; Check for exceptions (optional)
or r1,r1                         ; No-Op
st.w r1,(r2,0) bt                ; (id) Set semaphore to all 1's
Semaphore_corrupted ; Check if an exception occurs (optional)
cmpnei r3,0                      ;Semaphore test
```

8.12 Notes on Self-Modifying Code

In BootLoader, it is usually necessary to move a section of code from the storage address to its execution address and jump to the address; in some scenarios, it is necessary to dynamically change the instruction code and jump to the execution, which can be considered as self-modifying code behavior.

In the CSKY architecture, since it is von Neumann and Harvard compatible, when doing such operations, it is necessary to consider the consistency of CPU Cache and memory content. That is, after the modification is completed, the content in the D Cache needs to be cleared to the memory, and the content in the I Cache needs to be invalidated before the final jump operation can be performed.

8.13 Using Inline Assembly

The basic format of the embedded assembly of the XuanTie CPU tool chain conforms to the basic syntax of GNU gcc. If not specified in the following content, the XuanTie 800 series instructions are used as examples, but the same syntax rules apply to the XuanTie 900 series.

8.13.1 asm format

Use the "asm" keyword to indicate a section of source code written in assembly language.

The basic format of an asm segment is as follows:

```
asm ("assembly code");
```

Example:

```
/* Assign the value in r1 to r0 */
asm volatile ("mov r0, r1");

/* Multiple inline assembly lines*/ asm
volatile ("mov r0, r1\nmov r1, r0");

/* Multiple inline assembly asm , And use the optional \t to make the generated assembly code more friendly */
volatile ("mov r0, r1\n\tmov r1, r0");
```

The assembly code enclosed in brackets must follow a specific format:

- Commands must be enclosed in quotes
- If more than one instruction is included, each line of assembly language code must be separated by a new line character. Often, tab characters are also included to help indent the assembly language code. Language codes, to make lines of code easier to read.

The second rule is needed because the compiler takes the assembly code in the asm section verbatim and places it in the assembly code generated for the program. Each assembly language instruction must be on a single line - hence the need to include newline characters.

Note: If you do not want the compiler to optimize inline assembly, you can add the volatile keyword to prevent the compiler from optimizing, that is, asm volatile ("assembly code"). It is not necessary, but it is necessary in most cases, so volatile is added in the examples in this manual.

8.13.2 Extended asm format

The basic asm format provides an easy way to create assembly code, but has its limitations:

- All input and output values must use global variables of the C program.
- Extreme care must be taken not to change the value of any registers in inline assembly code.

The gcc compiler provides an extended format of the asm segment to help solve these problems. The format of the asm extended version is as follows:

asm (“assembly code”: output locations : input operands : changed registers);

This format consists of four parts, separated by colons:

- Assembly code: Inline assembly code using the same syntax as the basic asm format
- Output locations: A list of registers and memory locations containing the output values of the inline assembly code, in the format described in: Specifying input values and outputs value
- Input operands: A list of registers and memory locations containing input values to the inline assembly code, in the format described in: Specifying input values and output value
- Changed registers: A list of any other registers that were changed by the inline code

In the extended asm format, not all of these parts must appear. If the assembly code does not generate output values, this part must be empty, but two colons must be used to separate the assembly code from the input operands. If the inline assembly code does not change the value of the register, the last colon can be ignored. Example:

```
int a=10, b;
asm volatile ("mov r1, %1\n\t"
  "mov %0, r1"
  : "=r"(b)
  : "r"(a)
  : "r1");
```

8.13.2.1 Specifying input and output values

The format of the input and output value lists is:

“constraint” (variable)

Where variable is a C variable declared in the program. In the extended asm format, both local and global variables can be used. The constraint definition stores the variable in Where (for input values) or from where to transfer the variable (for output values). Use it to define whether to store the variable in a register or in a memory location.

The constraint is composed of a single string, see: gcc constraint related code

In addition to these constraints, the output value also contains a constraint modifier, which instructs the compiler how to handle the output value, see: gcc output modifiers

Note! "Constraints" can be used to specify the registers used to store variables, but they do not perform implicit type conversions at the C language level.

Observe the following E906P use case:

```
static short MAX16(short *a, short *b)
{
    short __result;
    __asm__(
        "#MAXW select the bigger one from 2 reg\n\t"
        "maxw %0, %1, %2\n\t"
        : "=r"(__result)
        : "r"(*a), "r"(*b)
    );
    return __result;
}
```

The purpose of this code is to find the larger value of two shorts. The author hopes that a and b will be sign-extended to the register width in the inline assembly, but in fact the compiler only guarantees that the lower 16 bits of a and b are valid, while their upper bits (the upper 16 bits for 32-bit systems and the upper 48 bits for 64-bit systems) are undefined. In order to fix the problem caused by the above code, you can try to explicitly convert a and b at the C language level and modify it to the following code:

```
static short MAX16(short *a, short *b)
{
    short __result;
    __asm__(
        "#MAXW select the bigger one from 2 reg\n\t" "maxw %0, %1, %2\n\t"

        : "=r"(__result)
        : "r"((int)*a), "r"((int)*b)
    );
    return __result;
}
```

8.14 Newlib implements reentrancy

There are some global variables, linked lists or queues in the functions implemented by newlib. When the system is multi-threaded, these variables, linked lists or queues may cause thread contention problems. Therefore, newlib provides some locks and interfaces to allow users to connect to specific operating systems, so that the data structures used by newlib are thread-safe and reentrant.

```
#include <sys/lock.h>

struct __lock {
    char unused;
};

struct __lock __lock___sinit_recursive_mutex;
struct __lock __lock___sfp_recursive_mutex;
struct __lock __lock___atexit_recursive_mutex;
```

(Continued on next page)

(Continued from previous page)

```
struct __lock __lock___at_quick_exit_mutex;
struct __lock __lock___malloc_recursive_mutex;
struct __lock __lock___env_recursive_mutex; struct __lock
__lock___tz_mutex;
struct __lock __lock___dd_hash_mutex;
struct __lock __lock___arc4random_mutex;

void
__retarget_lock_init (_LOCK_T *lock)
{
}

void
__retarget_lock_init_recursive(_LOCK_T *lock)
{
}

void
__retarget_lock_close(_LOCK_T lock)
{
}

void
__retarget_lock_close_recursive(_LOCK_T lock)
{
}

void
__retarget_lock_acquire (_LOCK_T lock)
{
}

void
__retarget_lock_acquire_recursive (_LOCK_T lock)
{
}

int
__retarget_lock_try_acquire(_LOCK_T lock)
{
    return 1;
}
```

(Continued on next page)

(Continued from previous page)

```
int
__retarget_lock_try_acquire_recursive(_LOCK_T lock)
{
    return 1;
}

void
__retarget_lock_release (_LOCK_T lock)
{
}

void
__retarget_lock_release_recursive (_LOCK_T lock)
{
}
```

To implement the above file, perform the following steps:

1. Define the struct __lock structure according to the system's lock definition
2. Define the above mutex locks
3. Implement __retarget_lock_acquire, __retarget_lock_release, __retarget_lock_acquire_recursive, __retarget_lock_release_recursive, __retarget_lock_close_recursive, and __retarget_lock_init_recursive. As the other interfaces are not called by the C library at present, you can keep empty function implementations.

Finally, compile the above C file into an object file and add the object file to the link command.

Chapter 9 Use of Binary Tools

In addition to the compiler, assembler, linker, and debugger, the XuanTie toolchain also includes many binary tools:

- *-addr2line - Get the source code file name and line number based on the program address.
- *-ar - create, modify and extract static libraries (archives).
- *-c++filt - Get the original symbol name of overloaded C++ symbols.
- *-gprof - Displays program profiling information.
- *-nm - Lists the symbols in a given object file.
- *-objcopy - copies and converts object files.
- *-objdump - Displays information about object files.
- *-ranlib - Generates an index of the contents of a static library (archive).
- *-readelf - Displays information about object files in ELF format.
- *-size - lists the section sizes of object files or static libraries.
- *-strings - Lists all printable strings in the file.
- *-strip - removes symbol information from object files, reducing file size.

The above tools are supported by both the GNU toolchain and the LLVM toolchain. The program name of the GNU toolchain is the prefix (Table 2.1) plus the component name, such as riscv64-unknown-linux-addr2line; the program name of the LLVM toolchain is llvm- plus the component name, such as llvm-addr2line. Some of these tools are often used in development. The following two sections only use two of them:

- View and analyze common information of *ELF* files
- How to generate *bin* and *hex* files

9.1 Viewing and analyzing common information of ELF files

ELF files are target files in ELF (Executable and Linkable Format) format. You can view relevant information through the *-readelf command. Different options can view different information.

1. -S

Displays the segment information of the program in the following format:

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [1].text               PROGBITS        00000000 001000 0022c0 00  AX  0   0  1024
  [2] .rodata            PROGBITS        00400000 004000 000550 00   A  0   0  4
  [3].data               PROGBITS        00400550 004550 000230 00  WA  0   0  4
  [4].bss                NOBITS          00400780 004780 000090 00  WA  0   0  4
```

in,

- Name: segment name
- Type: segment type, its common values are as follows
  - NOBITS: Program data that does not need to be stored in the file. Generally, the .bss segment has the NOBITS attribute.
  - PROGBITS: Corresponding to NOBITS, program data is stored in the file. Code segments and data segments other than the .bss segment are generally of this type.
  - SYMTAB: symbol table, usually the type of the .symtab section
  - STRTAB: string table, usually the type of the .strtab section
- Addr: The starting address of the segment
- Off: The segment is offset in the file
- Size: segment size, in bytes
- Flg: segment attribute, its common values are as follows
  - W: Write, writable
  - A: Alloc, which needs to be loaded into the content during execution
  - XÿExecute, executable
  - M: Merge, the linker thinks it can be merged, and the linker will try to merge the compressed segments
  - S: Strings, the segment content is a string
- Al: segment alignment requirement, in byte

2. -l

Display program header information in the following format:

```
  Type           Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x022c0 0x022c0 RE 0x1000
  LOAD           0x004000 0x00400000 0x00400000 0x00780 0x00810 RW 0x1000

Section to Segment mapping:
  Segment Sections...
    00     .text
    01     .rodata .data .bss
```

The so-called program is composed of multiple sections, which belong to the same program and have the same attributes. When the program is executed, the file is based on the program.

The meanings of the various fields of Program are as follows:

- Type: program type, usually LOAD, indicating that the program needs to be loaded into memory

- Offset: the offset of the program in the file
- VirtAddr: the running address of the program
- PhysAddr: the loading address of the program
- FileSiz: The size of the program in the file
- MemSiz: The size of the program loaded into memory
- Flg: program attribute, with the following values
  - R: Readable
  - E: Executable
  - W: Writable
- Align: the requirements of the program

3. -s

Display the program's symbol table in the following format:

```
Symbol table '.symtab' contains 170 entries:
   Num:    Value Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     ...
    96: 00001a38   386 FUNC    GLOBAL DEFAULT    1 printf
     ...
```

This option will list all symbols of the program and related information of the symbols. The meaning of each field is as follows:

- Value: the address of the symbol
- Size: The size of the symbol (such as the size of a function or variable)
- Type: symbol type, with the following common values
  - FUNC: function name
  - OBJECT: variable name
  - FILE: file name
  - NOTYPE: Symbol with no declared type
- Bind: The scope of the symbol, with the following common values
  - LOCAL: local symbol
  - GLOBAL: global symbols, which can be accessed by other files
  - WEAK: weak symbol
- Name: symbol name

9.2 How to generate bin and hex files

Bin files are binary files with no address marks inside. Generally, when programming with a programmer, the program starts from address zero. If it is loaded into memory and run, it can be loaded to the run address when linking. Bin files can be obtained by converting ELF files. The command is as follows:

```
*-*-objcopy -O binary [input ELF file ] [output bin file ]
```

Hex files are often used to store programs or data to ROM or EPROM. They can be obtained by converting bin files. The command is as follows:

```
*-*-objcopy -I binary -O ihex [input bin file ] [output hex file ]
```

Chapter 10: Charts

10.1 gcc constraint related code

10.1.1 CSKY Architecture Related Constraints

Constraint
Description a Use r0 - r7
registers b Use r0 - r15
registers c Use c
register y Use hi or lo registers
l Use lo register h Use
hi register v Use vector
register z Use sp register

10.1.2 RISC-V Architecture Related Constraints

Constraint
Description f Use floating point
I register 12-bit signed immediate value
J Integer zero
K 5-bit unsigned immediate value
A is the memory address stored in the general register

10.1.3 gcc public constraint code

Constraint
Description m Use the memory
location of the variable r Use any available
general purpose register i Use
an immediate integer value g Use any available register or memory location

10.1.4 gcc output modifiers

Output Modifier	Description
+	Can read and write operands
=	Only operands can be written
%	If necessary, the order of the operands can be swapped.
&	Operands may be deleted or reused before the inline function completes.