Gentei CPU Software Development Guide

Xuantie

November 05, 2024

Notice

The purchased products, services and features are stipulated by the contract made between C-SKY and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Hangzhou C-SKY MicroSystems Co., LTD

Address: Room 201, 2/F, Building 5No. 699,

Netshang Road, Hangzhou, Zhejiang,

China Website: www.xrvm.cn

Edition本 History

blockdof printing	descriptive	dates
1.1	Fix some vdsp intrinsic interface misdescriptions	2020.02.11
1.2	Added RISC-V series CPU programming chapter to fix some writing errors.	2020.05.07
1.3	Add vector floating point instruction description	2020.05.11
1.4	Modify the naming convention for T-Head extension commands Added c906	2020.06.20
1.5	Add libcc-rt description section	2020.08.05
1.6	Add abi type lp64dv	2020.08.09
1.7	Added e907 related CPUs Added 900 series P extension related CPUs	2020.12.26
1.8 table	Revise some writing errors in the documentation Added RISC-V architecture embedded assembly constraint Moved "Inline Assemblies" from Chapter 3 to Chapter 7 and added new notes	2021.04.22
1.9	Added c920	2021.05.26
2.0	New CPUs r910 and r920 Support for Xtreamer 900 Series Version V2.0 Tools Add Basalt 800 series dsp chapter	2021.07.20
2.1	Added CPU c908 and c908v.	2021.11.13
2.2	New -mcpu option support for Gentech 900 series Added newlib for reentrancy Support for Gentei 900 Series Version V2.6 Tools	2020.05.26
3.0	Support for Gentei LLVM compiler	2023.05.24
3.1	Added CPUs c910v2, c920v2 and c908i New section on properties of nested interrupt functions New section on bf16 type descriptions Added section on description of dynamic linker names Fix missing zfh march for c906fd, c910, r910	2023.10.08 2024.03.18
3.2	Fix missing xtheadvdot in march for c908v Added CPU c920v2.c908v Added c907 series CPU Updated march for c910v2 series and c908 series. Delete ABI lp64dv and lp64v Added KO file size optimization section	
3.3	Fix zcb omission in arch for C907 Added CPU c910v3, c920v3, c910v3-cp, c920v3-cp and r908 series.	2024.07.13

3.4	New General Purpose Coprocessor Extension C intrinsic Interface New RISC-V Vector Usage Note	2024.11.1

Corresponding Tools Edition

Format	Edition number
Gentei 800 Series GNU Tools	V3.10
Gentei 900 Series GNU Tools	V3.0.0
Xantei LLVM Tools	V2.0.0

Software Development Guide

Chapter 1 Introduction to the toolchain

The traditional definition of a tool chain usually includes a compiler, assembler, linker, etc. All of these components work together to realize the translation process from C/C++ source code to executable files, as shown in Figure 1.1 The compiler's processing of the input source file includes: 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 messages to alert the program developer that there are syntactic or semantic errors in the corresponding source file. Due to the design principle of separation of libraries in modern computer software, it is often necessary for a linker to link multiple target files and several static and dynamic shared libraries into a complete executable file. Together, the three parts form a toolset for the program developer to use.

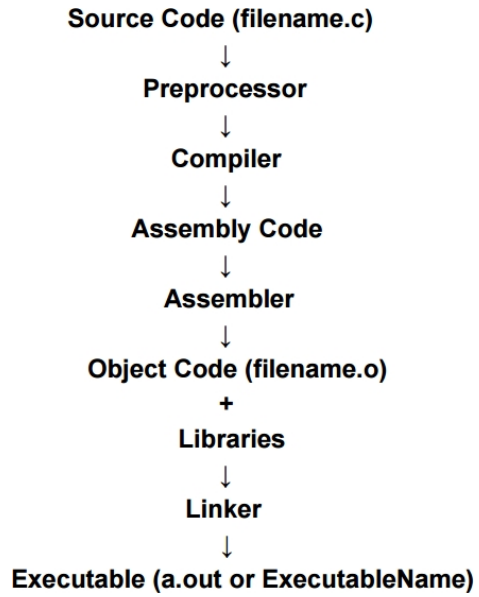


Figure 1.1: Compiler Processing of Input Source Files

Currently XuanTie has two toolchains, one based on the GNU toolchain and the other based on the LLVM compilation framework. Unless otherwise specified, the descriptions and examples in the following sections apply to both toolchains. Special sections can be found in the following sections:

- Instructions for using the *GNU* worker具 chain
- *LLVM* worker具 chain usage instructions

Chapter 2 Instructions for Using the GNU Toolchain

Currently Gentei has two GNU toolchains, one for the CSKY family of CPUs and the other for the RISC-V family of CPUs. this chapter contains the following sections:

- Worker具 Chain component version, name and language support
- General Options Description
- Passing options to assembler, linker
- Worker具 Chain Error and Warning Messages

2.1 Toolchain Component Version本 , Name and Language Support

The current releases of the CSKY series of toolchains are based on GCC 6.3, and the RISC-V series of toolchains based on GCC 10.2. The specific prefixes of the toolchains are shown Table 2.1, and the names of the toolchain's main components are shown Table 2.2. They support the C language version as shown in Table 2.3, and the C++ language version as shown Table 2.4, and the language standards are selectable by the option The language standard can be selected with the option -std=[language standard name].

Table 2.1: Toolchain Names

CPU Series	Targeted system platforms	C Library	Toolchain name prefix	Older version name prefixes
Genetsu 600 Series	elf	minilibc	csky-elf-	not have
	linux (computer)	glibc	csky-linux-gnu-	csky-linux-
		uclibc	csky-linux-uclibc-	csky-linux-

Genetsu 800 Series	elf	minilibc	csky-elfabiv2	csky-abiv2-elf-
		newlib	csky-noneabiv2-	not have
	linux (computer)	glibc	csky-linux-gnuabiv2	csky-abiv2-linux-
		uclibc	csky-linux-uclibcabiv2-	csky-abiv2-linux-
Basic Iron 900 Series	elf	newlib	riscv64-unknown-elf-	not have
	linux (computer)	glibc	riscv64-unknown-linux-	not have

Remarks.

1. The new version of the tool is compatible with the old toolchain names, and examples of the new toolchain names are used consistently throughout this document.
2. If not otherwise specified, the examples contained in this manual are suitable for use with the full range of basalt tools and are given as examples of one of these tools.

Table 2.2: Component Names

assemblies	name (of a thing)
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 Standards 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 Options Description

In order to avoid going through the entire toolchain, this section describes some of the most frequently used command line options for the average application developer. For a detailed description of the command line options, refer to the GCC Command Line online documentation.

2.2.1 Compiler Commands

1. Options for controlling the output

-E

only C/C++ preprocessing, not subsequent syntax analysis, code generation, etc.

-fsyntax-only

Typically used to control the compiler to perform only up to semantic checking, without performing subsequent processes. This option is usually used to test the input source file for compliance with the C/C++ language standard, without producing any files.

-c

Lets the compiler generate only the target file (usually file

-S

with a .o suffix), without performing the subsequent linking

-v

process. Controls the compiler to generate only assembly

code.

-
###

View the compiler version and prints out the compiler compiled commands, assembler commands and linker commands. And also prints out the search directory for header files.

This option must be used only after the gcc command, and is similar in function to the -v option, but differs from -v in that it does not perform the actual compilation, assembly, or linking process, but only prints the commands to be executed. Developers can use this option to get the compiler's compilation commands for debugging the source program. Note that normally debugging gcc does not give access to the compiler's process. gcc is a driver control program that generates different commands to call the compiler (cc1 for C programs, cc1plus for C++ programs), assembler (as), and linker (ld) to achieve the desired purpose.

--version

Displays the version number of the GCC being used.

2. language standard

-std

This option is used to control the language standards supported by the compiler. Currently the default language standards used by the Gentei compiler are gnu99 (for C) and gnu++14 (for C++). Of course, developers can force the compiler to use a specific standard by using this option.

3. Debugging Format Support

-g

Controls the compiler's insertion of natively supported debugging information (in dwarf format by default) into object or assembly code for use by the gdb debugger. This option is typically used during the development phase of a program to assist the developer in debugging and testing. It is recommended that this option be turned off for releases.

-ggdb

Controls the compiler to generate debugging information that conforms to the gdb debugger format.

-gdwarf

Controls the compiler to generate debugging information that conforms to the dwarf debugging format.

-gcoff

Controls the compiler to generate debugging information in coff format.

-gxcoff

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

4. Control Diagnostic Message Format

When the compiler parses the input source file, if there is a syntax or semantic error in the input source file, the compiler will generate a number of diagnostic messages to prompt the developer for the location of the source file where the diagnostic message occurs and the diagnostic category (fatal, error, warning), as well as giving detailed diagnostic description information.

-fmessage-lengt=n

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

-fdiagnostics-show-location=once

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

-fdiagnostics-show-location=every-line

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

-fno-diagnostics-color

The control compiler does not colorize diagnostic messages.

-fno-diagnostics-show-option

The control compiler does not generate diagnostic message description information.

-fno-diagnostics-show-caret

The control compiler does not generate the^ symbol to indicate where to generate it.

5. Optimization Options

These options are typically used to control the compiler's optimization strategy, telling the compiler whether it should perform certain optimizations to increase the speed of the program, such as loop unrolling, or whether it should avoid certain optimizations to reduce the size of the target file so that the target program can run more efficiently on embedded devices with small memory sizes.

-O0

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

O-O-1.

These two options have the same meaning and are usually used to control the compiler to perform some of the more rewarding optimizations, while achieving the goal of reducing code size and execution time.

-Og

-Os

-O2

-O3

while maintaining fast compilation and a good debugging experience.

Optimize the debugging experience by providing reasonable optimization levels

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

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

This optimization option is typically used to tell the compiler to reduce the size of the object code as much as possible while maintaining performance. It will take the size of the target code from the

Remove some of the optimization strategies that increase the size of the target code from the full set of options turned on by O2.

-Ofast

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

2.2.2 Assembler Commands

For most developers, there are few opportunities to use the assembler directly. Therefore, this section describes the more commonly used options. For options not covered in this section, developers can consult the CSKY or RISC-V assembler development manuals.

-g --gen-debug

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

-o

Specify the filename of the destination file for output, default is a.out.

2.2.2.1 CSKY Assembler-Specific Options

-march= architecture

Generating object code for a CPU of the specified architecture, e.g.: *-march=ck803* will tell the compiler to generate instructions supported by ck803 series CPUs, but not ck810 series CPUs.

-mcpu=CPU model.

Specify to generate optimized code for the CPU characteristic to be selected, e.g. *-mcpu=ck803er1* will enable the dspv2 instruction.

-m{no-}ljump

Converts the destination address of a jbf, jbt, or jbr instruction beyond the instruction's offset range into a jmpi instruction, which is turned off by default.

2.2.2.2 RISC-V assembler specific options

-march= architecture

Generate object code for the CPU of the specified architecture, e.g.: *-march=rv32imac* will tell the compiler to generate an instruction set that contains the base integer instruction set (Target code for 32-bit RISC-V series CPUs with the "I" instruction set, the atomic operation instruction set ("A" instruction set) and the compression instruction set ("C" instruction set). The target code for the 32-bit RISC-V series CPUs.

2.2.3 typical example

This section describes in detail the usage of each of the above options and what they do, using CSKY as an example, and the methodology described is also applicable to RISC-V. This section uses bar.h and bar.c files, with the bar.h header file declaring a function the signature *int sum(int num)*, and the function being implemented and called in bar.c. The contents of the bar.h file are as follows

```
#ifndef
BAR_H
#define
BAR_H

/**
 * Performs an  operation, calculating the sum from 1 to len, if the input len
   is less than or equal to 0.
 * Return 0 .
 */
int sum(int len).
#endif
```

The bar.c file has the following code

```
#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 commands are as follows

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

After the above command is executed, an executable file named bar will be created in the directory where the source file is located. Using the qemu-system-cskyv2 emulator (which is used to simulate the execution of elf-formatted files) you can simulate the execution of this file, and the corresponding output will be displayed on a standard output device (usually a terminal or a command line window).

In some development scenarios, e.g., debugging a source code file. Often the file will include 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 preprocessor files to avoid searching the unknown header file search directory.

To achieve the above, the developer needs to manually pass the -E option to the compiler to tell the gcc compiler that it only needs to perform preprocessing, and the gcc compiler will generate a file with the same name as the source file, but with a .i suffix (for C, or .ii if the source is in C++), with the following commands

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

The output preprocessor file is named bar.i and reads as follows. It is clear that the compiler will extract the corresponding function declaration from the header file to the location where the function is to be used, thus realizing one of the important principles of the C/C++ language-variables or functions must be declared before they can be 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, e.g., to see if the corresponding generated assembly code is correct. In addition to disassembling the generated object code with the csky-elfabiv2-objdump utility, a more direct way to deal with this is to pass the -S option to the compiler, in which case 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 command is shown below.

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

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

```
        subi sp. sp, 4
        st.w l4, (sp,
                 0)
        mov  l4, sp
        (com
        puti
        ng)
        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)
        cmpl a2. a3
        t
        jbt  .L7
        subi a3. l4, 4
        ld.w a3. (a3,
                 0)
.L5.
        mov  a0, a3
        (com
        puti
        ng)
        mov  sp. l4
        (com
        puti
        ng)
        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. Take the above assembly as an example, when -O2 is turned on, you can observe the following assembly code and find that the overall size of the program is reduced very significantly, while the instructions are more concise, as shown in the figure:

```
# To save space, omit the code for the main function and remove
extraneous 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 see a significant reduction in the number of instructions in the generated assembly file compared to not turning on optimization.

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

2.3 Passing options to assembler, linker

GCC execution includes pre-compilation, compilation, assembly, and linking by default, and it automatically calls the pre-processor, assembler, and linker. In some cases, the developer needs to pass options to each component, as described Table 3.4:

Table 2.5: Passing Options for Individual Components

GCC Options	corresponds English -ity, -ism, -ization
-Wp, [parameters]	Passing parameters to the preprocessor
-Wa, [parameters]	Passing parameters to the assembler
-Wl, [Parameters]	Passing parameters to the linker
-L [Path]	Add path for linker to find libraries

2.4 Toolchain Errors and Warning Messages

In order to better improve the program developer's development efficiency and improve the stability of the program runtime, most toolchains will provide a good program checking mechanism, in the illegal, there are potential security or performance hazards in the location of the code fragment to generate diagnostic information, print out easy to read and understand diagnostic information for the program developer, including error messages (error), warning messages (warning) and Partial modification suggestions (notes).

This chapter will focus on the classification and format of the diagnostic information output by the CSKY series and RISC-V series toolchains, and use several examples to show how to repair code fragments based on the diagnostic information (this chapter uses the CSKY series compilers as examples, but the rules are generally applicable to the RISC-V series compilers).

This chapter contains the following sections:

- Format of Compiler Error and Warning Messages

- Options for compiler diagnostic information
- Controlling Error and Warning Messages with *pragma* Preprocessing Commands
- Other Workers具 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且 is the category of diagnostic messages that most developers come across on a daily basis. This category of messages is usually generated by the front-end of the compiler (lexical analysis, syntax analysis, semantic checking) to inform the developer about the presence of numbers, identifiers, linguistic constructs, and illegal operations on variables of a certain type in the source program being compiled that do not conform to the C standard (or to the GNU extended syntax).

2.4.1.1 Error message format

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

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

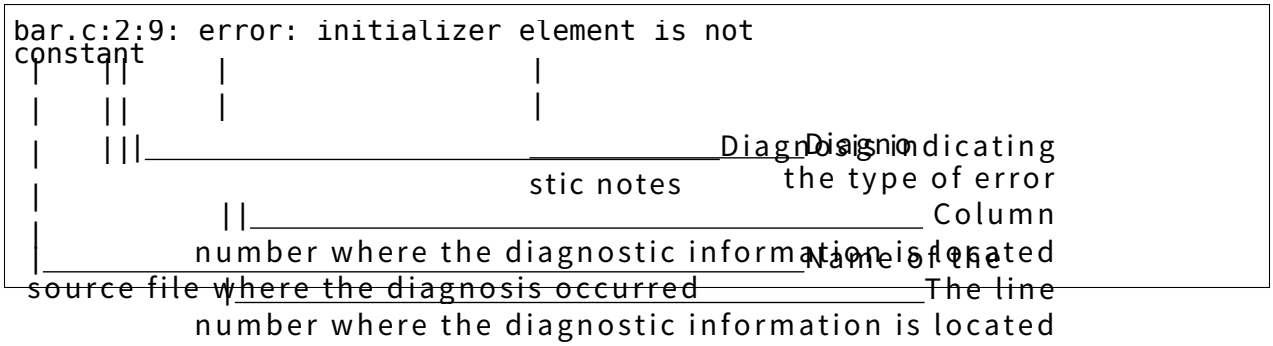
Use the following shell command to compile. The *-fsyntax-only* option is used to tell the compiler to perform only front-end actions, not back-end optimization and code generation.

```
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.



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 perform only front-end actions, not 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)
^~~
```

The warning diagnostic message format will be displayed as follows.

```
too.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)

```
|    ||_____________________________________ This warning occurs in
column 1 of the source file
|    |_____________________________________This warning occurs at line 1
of the source file
|_____________________________________This warning occurs in the
source file foo.c
```

2.4.1.3 Other diagnostic information formats

In addition to reporting errors and warning messages, the compiler will often, in some cases, provide some degree of error-fixing advice to the program developer on how to fix the error. This is illustrated by the following code snippet, in which the main function calls the malloc function to allocate 10 bytes of space on the heap and then assigns it the value of 1. Note that the code does not include <stdlib.h>.

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

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

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

As you can see from the diagnostic information above, in addition to the warning message, there is also a note message, which usually advises the developer on how to use include.
'<stdlib.h>' to fix the problem.

2.4.2 Options for compiler diagnostic information

Often in embedded development, in order to minimize the risk of potential

vulnerabilities in the program, it is necessary to ensure that the source program is compiled without any warnings and errors. However, a significant portion of developers will ignore the warning messages generated by the compiler, and且 more importantly, when the compiler generates warning messages, the compilation process will continue as usual. This masks the possibility of future pitfalls. To avoid this, the compiler provides a number of options to fulfill this need, the most important of which is -Werror, which will convert all diagnostic warnings into errors to avoid hiding the error.

At the same time, the compiler also provides some options to control the compiler to generate warning messages, suggesting that the program developer needs to pay attention to the place, it is more appropriate to use the -Wall option, the option will be all the warnings will be turned on, instead of the default off (the default state of the output warning messages depends on the compiler version). However, sometimes the developer only needs to convert specific warnings into error messages, and the compiler provides the -Wxxx (xxx is the name of the warning option, such as the implicit-function-declaration mentioned in the previous section) option to turn on specific warnings for each warning option.

Some legacy code displays certain warnings as error messages for compatibility with different GCC versions. To avoid compilation failures with newer versions of GCC, you can turn off such warning messages with -Wnoxxx (xxx is the name of the warning option), e.g. -Wnoimplicit-function-declaration.

2.4.3 Controlling Error and Warning Messages with pragma Preprocessing Commands

In addition to controlling the compiler's diagnostic message output via command line options, the compiler also supports the #pragma preprocessing option in source files for more flexible control over the display of diagnostic messages. In addition, customized error or warning messages can be output via #pragma, and the diagnostic options can be selectively turned off for all or a specified portion of the diagnostics.

2.4.3.1 #pragma GCC error

This option is used for program developers to set custom error diagnostic messages 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"
```

Compiling with the csky-elfabiv2-gcc command displays the error message:

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

As you can see from the above results, #pragma GCC error gives the compiler the flexibility to 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 a specific, customized warning message, displayed in essentially the same format. Again, the following code snippet is used as an example to illustrate the situation.

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

Compiling with the csky-elfabiv2-gcc command displays the warning message:

```
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 note diagnostic message, not a warning or error message.

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

Compiling with the csky-elfabiv2-gcc command displays the warning message:

```
pragam-message.c:1:9: note: #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 above are normally used to control the output of customized diagnostic messages from the compiler, but this option is used to tell the compiler to ignore, display, or display a warning option as an error when it meets this command while compiling a source file, as in the following example.

```
#pragma GCC diagnostic ignored "-Wimplicit-int"
bar()  // This is where "return type defaults to 'int '" should have
appeared.
{
  #pragma GCC diagnostic warning "-Wimplicit-function-declaration" //
  enable -Wimplicit-
↪function-declaration option
  int *ptr=  (int*)malloc(sizeof(int));  // Display the warning
  *ptr= 1;
  #pragma GCC diagnostic error "-Wimplicit-function-declaration"      //
  Will -Wimplicit-
↪function-declaration Display as error
  memset(ptr, 0, sizeof(int)); // show warning as error
  return 0;
}
```

The following diagnostic message appears when compiled with csky-elfabiv2-gcc. Observing the following message makes it obvious that the above preprocessing instructions play a role.

```
implicit.c: In function 'bar'.
implicit.c:5:20: Warning:implicit☞ explicit function 'malloc' [-
  Wimplicit-function-declaration] int *ptr =
  (int*)malloc(sizeof(int));         // Show warning
                ^~~~~~
implicit.c:5:20:   Warning: implicit☞ is incompatible with the built-in
function 'malloc'.
implicit.c:5:20: note: include '<stdlib.h>' or provide a declaration of
'malloc'
implicit.c:8:3: Error: implicit☞ implicit-function 'memset' [-Werror=
  implicit-function-declaration] memset(ptr, 0, sizeof(int)); // will
  be warning as an error
  ^~~~~~
implicit.c:8:3:   Warning: implicit☞ is incompatible with the built-in
function 'memset'.
implicit.c:8:3: note: include '<string.h>' or provide a declaration of
'memset'
```

2.4.4 Options for other tools to control error and warning messages

2.4.4.1 Options for assembler control of diagnostic information

The developer in some cases when using the assembler alone, as with the compiler, also needs to control the output behavior of the output diagnostic messages, e.g., not outputting a warning message or displaying a warning message as an error.

1. -W

Hide warning messages

2. -warn

Does not hide warning messages

3. -fatal-warnings

Display warnings as errors

4. -Z

Generate target file even if there are errors

Chapter 3 Instructions for Using the LLVM Toolchain

The Gentei LLVM compiler is a high-performance, highly reliable toolchain based on the open-source LLVM 15. It is based on the XuanTie processor for deep co-optimization of hardware and software, while optimizing performance and code density for a variety of commonly used domains. Currently, Gentei LLVM compiler focuses on C, C++ programming language support, supports the full range of CSKY and RISCV processors, and supports both linux and windows platforms.

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

General instructions for the use of CLANG can be found at https://releases.llvm.org/15.0.0/tools/clang/docs/UsersManual.html

This section describes some special or common usage instructions, and contains the following main sections:

- Work具 Chain component names and language support
- General Options Description
- Passing options to assembler, 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, and their support for the C language version is shown in Table 3.2, and for the C++ language version is shown in Table 3.3, where the language standard can be selected with the option -std=[language standard name].

Table 3.1: Component Names

assemblies	name (of a thing)
C Compiler	clang
C++ compiler	clang++
assembler	llvm-mc
linker	[Prefix]-ld

Note: Currently the linkers used in the LLVM toolchain are those in the GNU suite, see 具 for linker 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 Standards 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 Options Description

This section introduces some of the most frequently used command line options for general application developers. For a detailed description of the command line options, please refer to the CLANG command line online documentation.

1. Optimization Options

   Options in this category are typically used to control the compiler's optimization strategy, informing the compiler whether it should perform certain optimizations to increase the speed of the program or reduce the size of the object code.

   -O0

   The default optimization level, no optimization is performed to reduce compilation time, and

且 does not cause inaccurate debugging information to facilitate debugging.

O-O-1.

These two options have the same meaning and are usually used to control the compiler to perform some of the more rewarding optimizations, while achieving the goal of reducing code size and execution time.

-Og

Optimize the debugging experience by providing reasonable optimization levels while

-O2 maintaining fast compilation and a good debugging experience.

-O3 This option will enable optimization options that require more compilation time to significantly increase the execution speed of the program. Note that this option usually increases the size of the target code.

-Os

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

-Oz

This optimization option is typically used to tell the compiler to reduce the size of the object code as much as possible while maintaining performance. It will take the size of the target code from the

Remove some of the optimization strategies that increase the size of the target code from the full set of options turned on by O2.

Turn on more options for optimizing code size on top of Os.

-Ofast

Enable other aggressive optimizations on top of O3 that may violate some strict language standards.

2. Options for controlling the output

-E

only C/C++ preprocessing, not subsequent syntax analysis, code generation, etc.

-fsyntax-only

Typically used to control the compiler to perform only up to semantic checking, without performing subsequent processes. This option is usually used to test the input source file for compliance with the C/C++ language standard, without producing any files.

-c

Lets the compiler generate only the target file (usually file

-S

with a .o suffix), without performing the subsequent linking

-### process. Controls the compiler to generate only assembly

code.

This option is similar in function to the -v option, but differs from -v in that it does not perform the actual compiling, assembling, and linking processes, but only prints the execution commands.

3. Message Viewing Options

-v

View the compiler version and prints out the compiler compiled commands, assembler commands and linker commands. And also prints out the search directory for header files.

--version

Displays the version number.

3.3 Passing options to assembler, linker

The compiler execution includes pre-compilation, compilation, assembly, and linking by default, and it automatically calls the pre-processor, assembler, and linker. In some cases, the developer needs to pass options to each component, as described Table 3.4:

Table 3.4: Passing Options for Individual Components

CLANG Options	corresponds English -ity, -ism, -ization
-Wp,/-Xpreprocessor [Arguments]	Passing parameters to the preprocessor
-Wa,/-Xassembler [Arguments]	Passing parameters to the assembler
-Wl,/-Xlinker [parameters]	Passing parameters to the linker
-L [Path]	Add path for linker to find libraries

3.4 compatibility

Currently Clang includes the GCC Most of the of GCC are currently included in Clang, and common compatibility and migration issues can be found in the official document https://clang.llvm.org/compatibility.html.

Chapter 4 Programming the Gentei 800 Series CPU

Gentei 800 series CPUs are processors developed based on the CSKY architecture. This chapter mainly introduces the CSKY architecture that is involved in C and C++ programming. Architecture-related special usages such as cpu selection, instruction set selection, assembly programming, vdsp and dsp instruction intrinsic interfaces, and minilibc.

This chapter contains the following sections:

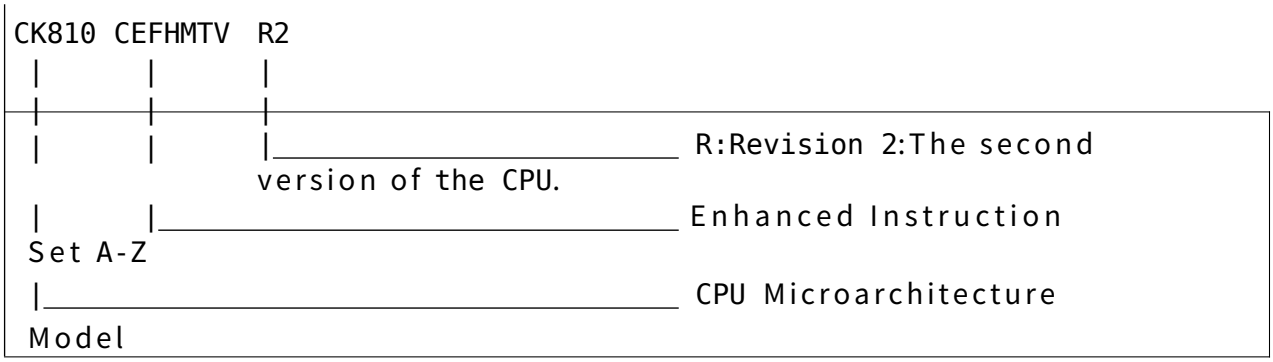
- How to add the corresponding option to the processor
- Instruction Set Introduction
- How to use hard floating point instructions
- assembly language programming
- *vdsp*
- *dsp*
- *minilibc*

4.1 How to add the corresponding option to the processor

Currently our CSKY supports many different cpu models, you can check all the cpu models supported by the current tool by using the GCC or CLANG option, execute the following command at the command line:

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

The result of the command can be seen for all cpu models supported by the compiler, which follow a basic set of naming rules, such as:



The meaning of the Enhanced Instruction Set symbols is shown Table 4.1:

Table 4.1: Meaning of Enhanced Instruction Set Symbols

Enhanced Instruction Set Notation	full name	clarification
C	Crypto enhance	cryptographic enhancement
E	EDSP	DSP enhancement
F	FPU	floating point
H	Shield	Physical Anti-Attack
M	Memory enhance	Storage Enhancement
T	TEE	Credible implementation environment
V	VDSP	Vector DSP

In general, we compile certain projects by specifying the appropriate cpu for compilation with the -mcpu compilation option:

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

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

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

4.2 Instruction Set Introduction

The previous section has introduced the architectures that CSKY has, and 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 sets, CSKY has two dsp instruction sets, two vdsp instruction sets, and three floating-point instruction sets, as described in the following sections.

4.2.1 dsp instruction set

There are two dsp instruction sets, dsp 1.0 and dsp 2.0, and the correspondence between the instruction sets and CPUs is shown Table 4.2:

Table 4.2: Correspondence between dsp instruction set and cpu

CPU Model	dsp Command Set Version
ck803 Series CPU with 'e' Label	dsp 1.0
ck803r1 Above (including r1) CPUs with 'e' labeling	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 vdsp instruction sets, vdspv1 and vdspv2, and the correspondence between the instruction sets and CPUs is shown Table 4.3:

Table 4.3: vdsp instruction set and cpu correspondence

CPU Model	vdsp Command Set Version
ck805 series CPU	vdspv2
ck810 series CPUs with 'v' labeling	vdspv1
ck860 Series CPU with 'v' Label	vdspv2

4.2.3 floating point instruction set

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

Table 4.4: Floating Point Instruction Set and CPU Correlation

CPU Model	Floating Point Instruction Set version
ck610 series CPUs with 'f' labeling	fpuv1
ck803 Series CPU with 'f' Label	fpuv2 single precision floating point
ck804 Series CPU with 'f' Label	fpuv2 single precision floating point
ck805 Series CPU with 'f' Label	fpuv2 single precision floating point
ck807 Series CPU with 'f' Label	fpuv2
ck810 series CPUs with 'f' labeling	fpuv2
ck860 Series CPU with 'f' Label	fpuv3

Note: If you need the compiler to compile hardware floating point instructions, you need to add additional options besides adding the correct cpu model, see How to Use Hard Floating Point Instructions.

4.2.4 CPU version本 and base instruction set

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

Table 4.5: Relationship Between CPU Version and Base Instruction Set

CPU Model	Basic Instruction Set Description
ck803r1	Added to the ck803 base instruction set: mul.u32 mul.s32 mula.u32 mula.s32 mula.32.l mulall.s16.s
ck803r2	Added to the ck803r1 base instruction set: bnezad
ck803r3	Added to ck803r2 base instruction set: divul divsl

4.3 How to use hard floating point instructions

Currently some of our cpu's support hardware floating point units, how can we get the gcc compiler to generate code with hardware floating point instructions? It can be done based on the *csky cpu* feature to control the compiler's generation of code containing hardware floating-point instructions:

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

We currently support a variety of floating-point ABI rules, controlled via the **-mfloat-abi** compilation option:

soft: using software

floating point **hard**:

using hardware

floating point

softfp: same as hard, but does not use floating-point registers for arguments and return values.

Remarks: For floating-point control, the compiler is compatible with older versions, -msoft-float is equivalent to -mfloat-abi=soft, -mhard-float is equivalent to -mfloat- abi=hard.

4.4 assembly language programming

Some developers need to hand-write assembly files and compile them into target files, typically using the following basic commands:

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

This chapter contains the following sections:

- assembly instruction format
- Preprocessing of assembly documents
- assembly pseudo-instruction
- register alias

4.4.1 assembly instruction format

The format of an assembly instruction is divided into two parts, the instruction name and the operand name, separated by spaces, as follows.

```
>                    Instruction name Operand 1, Operand 2···
```

where the types of operands are as in Table 4.6:

Table 4.6: Types of Operands in Assembly Instructions

operand type	writing format	typical example
general-purpose register	Generic register name, see Register Alias for details	abs r1
v1 Floating Point Registers	fr0-fr31	fabss fr1
v2 Floating Point Registers	vr0-vr15	fabss vr0-vr15
v2 Vector Register	As above, the floating-point module and the vector module in abiv2 use the same set of registers.	vabs.8 vr1
Memory Addresses with Immediate Number 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	bsr functionname
control register	cr<z, sel> (group sel, register z)	mtcr r1, cr<0, 0>
General Purpose Register Sequence	rx-ry, rz…	push r4-r11,r15

Remarks: 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 of assembly documents

When an assembly file contains some C macros (such as #define, #include, #if, etc.) and comments, it must be preprocessed. GCC determines whether an assembly file needs to be preprocessed based on its suffix:

- When the suffix is (.S), it means that the file contains macros that need to be preprocessed.
- When the suffix is (.s), it means that the file contains only assembly instructions that do not need to be preprocessed. For example, an assembly file (test.S) containing macro instructions

is shown below:

```
#define P          /* As with C syntax,
2                  macro definitions */
movi t0,P
```

You can get a preprocessed assembly file by adding the -E option to gcc. The command and the resulting file are shown below:

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

File test.s:

```
movi t0, 2
```

Note: Don't confuse #include#if, etc. with .include,if, etc. #include#if, etc. are C macros that need to be handled by the preprocessor, and .include,
.if, etc. are assembly instructions that need only be processed by the assembler.

4.4.3 assembly pseudo-instruction

An assembly source program contains, in addition to assembly instructions, pseudo-instructions, which have no counterpart in the CPU instruction set. Assembly pseudo-instructions can be expanded into one or more assembly instructions. The reasons for using pseudo-instructions fall into three main categories:

1. The uncertainty of the offset distance of the target address of a jump instruction relative to the instruction itself results in the assembler's decision as to which jump instruction to use; the
2. Make the writing of some commands 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 Pseudo-Instructions

bogus instruction	Expanded Instructions	descriptive	CPU
clrc	cmpne r0,r0	Clear the C bit to zero	full
cmplei rd,n	cmplti rd, n+1	Comparison of signed immediate numbers Use less than compatible with less than or equal to	full
cmpls rd,rs	cmphs rs, rd	Unsigned Comparison of Immediate Numbers Use greater than or equal to compatibility less than or equal to	full
cmpgt rd,rs	cmplt rs, rd	Comparison of signed immediate numbers with less than compatible greater than or equal to	full
jbsr label	abiv1: bsr label maybe jsri label abiv2.	Jump to subroutine	full

	bsr label		
jbr label	abiv1: br label maybe jmpi label abiv2. br label	unconditional jump	full
jbf label	abiv1. bf label maybe bt 1f jmpi label 1:… abiv2. bf label (16/32 bit) or bt 1f (16-bit) br/jmpi label (32-bit) 1:…	C bit is 0 Jump	full

continued on next page

Table 4.7 - continued from previous page

bogus instruction	Expanded Instructions	descriptive	CPU
jbt label	abiv1. bt label maybe bf 1f jmpi label 1.… abiv2. bt label (16/32 bit) or bf 1f (16-bit) br/jmpi label (32- bit) 1.…	C bit is 1 Jump	full
rts	jmp r15	Return from subroutine	full
neg rd	abiv1: rsubi rd,0 abiv2: not rd, rd addi rd, 1	take the opposite number	full
rotlc rd,1	addc rd,rd	addition with a rounding error	full
rotri rd,imm	rotli rd, 32-imm	Immediate number cyclic left shift	full
setc	cmphs r0,r0	Setting the C Bit	full
tstle rd	cmplti rd,1	Test register value is non-positive	full
tstlt rd	btsti rd,31	Test register value is negative	full
tstne rd	cmplnei rd,0	Test register value is a non-zero number	full
bgeni rz,imm	movi rz,immpow immpow is imm power of 2.	Setting the imm bit of the register to 1, other positions 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	rz=rx	V2.0
	maybe lsli rz,rx,0	If rz and rx are both r0~r15, then it is mov (computing) If rz or rx is	

continued on next page

Table 4.7 - continued from previous page

bogus instruction	Expanded Instructions	descriptive	CPU
sexth rz,rx	sext rz,rx,15,0	Take the first word of rx and Signed extensions to rz	V2.0
zextb rz,rx	zext rz,rx,7,0	Take the first byte of rx and Unsigned extensions to rz	V2.0
zexth rz,rx	zext rz,rx,15,0	Take the first word of rx and Unsigned extensions to rz	V2.0
lrw rz,imm32	movih rz,imm32_hi16 ori rz, rz,imm32_lo16	Load a 32-bit immediate number into the register. tool	V2.0
jbez rx,label	bez rx,label maybe bnez rx,1f br/jmpi label (32-bit) 1.…	If rx is equal to zero, jump to subroutine preface	v2.0
jbnez rx,label	bnez rx,label maybe bez rx,1f br/jmpi label (32-bit) 1.…	If rx is not equal to zero, jump to sub programs	v2.0
jbhz rx,label	bhz rx,label maybe blsz rx,1f br/jmpi label (32-bit) 1.…	If rx is greater than zero, jump to subroutine preface	v2.0
jblsz rx,label	blsz rx,label maybe bhz rx,1f br/jmpi label (32-bit) 1.…	If rx is less than or equal to zero, jump to the subroutines	v2.0
jblz rx,label	blz rx,label maybe bhsz rx,1f br/jmpi label (32-bit) 1.…	If rx is less than zero, jump to subroutine preface	v2.0

jbhsz rx,label	bhsz rx,label maybe blz rx,1f br/jmpi label (32-bit) 1.…	If rx is greater than or equal to zero, jump to the subroutines	v2.0

4.4.4 register alias

Many general-purpose registers (r0-r31) are supported with aliases that help to improve the readability and compatibility of assembly code in certain situations, as shown in Table 4.8 and Table 4.9:

Table 4.8: CSKY ABI V1 Register Aliases

V1 Register Name	nickname	descriptive
r2-r3	a0-a1	Passing a parameter/passing a return value
r4-r7	a2-a5	pass on a questionnaire
r8-r13	l0-l5	Storing local variables (which need to be saved and restored at the end of the function header when used)
r14	l10/gb	Stores local variables/stores GOT table base address when compiled with the PIC option.
r15	lr	Store the return address
r16-r19	l6-l9	Storing local variables (which need to be saved and restored at the end of the function header when used)
r20-r25	t0-t5	Storing temporary data (use without saving and restoring at the head and tail of the function)
r31	tls	TLS Register

Table 4.9: CSKY ABI V2 Register Aliases

V2 Register Name	nickname	descriptive
r0-r1	a0-a1	Passing a parameter/passing a return value
r2-r3	a2-a3	pass on a questionnaire
r4-r11	l0-l7	Storing local variables (which need to be saved and restored at the end of the function header when used)
r12-r13	t0-t1	Storing temporary data (use without saving and restoring at the head and tail of the function)
r14	sp	stack pointer (computing)
r15	lr	Store the return address
r16-r17	l8-l9	Storing local variables (which need to be saved and restored at the end of the function header when used)
r18-r25	t2-t9	Storing temporary data (use without saving and restoring at the head and tail of the function)
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 text section base address
r30	svbr	Storage handler base address
r31	tls	TLS Register

Remarks: Passing registers can be used as temporary registers when they are not used, and do not need to be saved and restored at the end of the function header when they are used.

4.5 vdsp

Currently, the CSKY architecture supports two versions of the VDSP instruction set, vdspv1 and vdspv2. vdspv1 can be configured with 64-bit and 128-bit bitwidths, and the compiler controls the bitwidth of the generated object code with the option -mvdsp-width=<size> (default 128). vdspv2 has a bitwidth of 128 bits. ck810 uses vdspv1 and ck860 and ck805 use vdspv2. vdspv2 is 128 bits wide. ck810 uses vdspv1, ck860 and ck805 use vdspv2.

The compiler determines whether the generated target program supports the vdsp instruction according to the CPU option, where ck810 CPUs that support vdsp are:

ck810v, ck810fv, ck810tv, ck810ftv (i.e., the 810

contains the cpu for v). ck860 CPUs that support

vdsp are:

The ck860v ck860fv (i.e., the

860 includes the cpu for v) All

CPUs in the ck805 support

vdsp.

In the following cases, the compiler generates vector instructions:

- vector operator expression
- circular optimization
- Using the intrinsic function

Of these, the first two are for more basic scenarios, while the third applies to scenarios that require deep optimization. Detailed descriptions can be found in the following sections of this chapter:

- vector data type (computing)
- Passing rules for parameters and return values of vector types
- vector operator expression
- Loop-optimized generation of vector instructions *(*currently only supported in the *GNU* worker具chain*)*
- *intrinsic* function interface naming convention
- *intrinsic* interface to *vdspv2*

4.5.1 vector data type (computing)

Vector datatypes are usually based on common datatypes, e.g., the vector datatype int8x8_t represents an integer datatype with 8-bit elements, a type consisting of 8 elements, and it has a total bit width of 64 bits. The naming convention is as follows.

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

The element types are int, uint, or float, and the header file csky_vdsp.h is required to reference the vector data types supported by vdspv1 and vdspv2 as shown in the table. 4.10, 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 Passing rules for parameters and return values of vector types

By default or with the option -mfloat-abi=soft/softfp turned on, arguments and return values of vector types are still passed using normal registers.

With the option -mfloat-abi=hard enabled, arguments and return values of vector types are no longer passed using normal registers. Parameters of vector types are passed through registers vr0-vr3 Passed, when the vector type more than 4 arguments, the remaining arguments are passed through the stack. The return value of a vector type through register vr0.

4.5.3 vector operator expression

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

4.5.3.1 Definition of Vector Type Variables

Vector type variables are defined in two ways:

- The first way is the same as an array definition, e.g.:

```
#include<csky_vdsp.h>

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

- In the second way, an array is defined, and then the array address is converted to a vector pointer

type, e.g:

```
#include<csky_vdsp.h>

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

4.5.3.2 operator (computing)

C uses operators to represent arithmetic operations, and this is also true for variables of vector type. Currently, the operators supported by vector expressions are shown below:

- Addition:+
- Subtraction: -
- Multiplication:*
- Comparison operators:> ,< , ! =,>= ,<= .==
- Logical operators: &,| .^
- Shift operator: ","

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 partial loops to generate vector instructions. The compiler tries to optimize loops into vector instructions when the following conditions are met:

- Current CPU support for vector instructions
- Optimization level is -O1 or above and且 add option -ftree-loop-vectorize

(This option is turned on by default for -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 operations */
}
```

If the current CPU supports 128-bit vector addition instructions, with loop optimization turned on, the above code is optimized as shown in the pseudo-code below:

```
#include <csky_vdsp.h>

int32x4_t svfun2 (int32x4_t va,int32x4_t vb)
{
    int32x4_t vc=  va+  vb;   /* Vector operations */
    return vc.
}
```

4.5.5 intrinsic function interface naming convention

The intrinsic interface function name is basically the same as the instruction name, if the instruction name contains ".", it will be replaced by "_" in the function name. If the name of the instruction contains ".", it will be replaced by "_" in the function name, for example, the instruction vmfvr.u32 corresponds to the function name of the intrinsic interface, vmfvr_u32. The parameter and return value types of the function are determined by the data types of the instruction operands, for example, the instruction vmfvr.u32 rz,vr[index], its function is to transfer the index element of the vector register to the first element of the vector register, the function is to transfer the index element to the first element of the vector register. index], whose function is to transfer the index element of a vector register to the general register rz, the function vmfvr_u32 is declared as follows:

```
uint32_t vmfvr_u32 (uint32x4_t__ a, const int32_t__ b);
```

where 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 interface to vdspv2

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

- ck860v ck860fv (i.e. 860 with v cpu)
- All CPUs of ck805

The commands of vdspv2 can be divided into the following sections:

- Integer addition and subtraction, comparison instructions
- Integer Multiply Instruction
- Integer inverse, inverse square, *e* exponential fast arithmetic and approximation instructions
- Integer shift instruction
- Integer Move *(MOV)*, Element Operation, Bit Operation Instructions

- Integer Immediate Number Generation Instruction
- *LOAD/STORE* Command
- Floating Point Add/Subtract Compare Instruction
- floating-point multiply instruction
- Floating-point inverse, inverse square, *e* exponential fast arithmetic and approximation instructions
- floating-point conversion instruction

4.5.6.1 Integer addition and subtraction, comparison instructions

vadd.t && vsu b.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.t.e && vsub.t.e

- 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 of the first
    parameter zero
    extension,the second
    vector addition
    assumption parameter
    Vx,Vy, the return 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 parameter
    has signed extension,
    second vector addition
    assumes parameters
    Vx,Vy, return 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 parameter zero
    extension,second
    vector subtraction
    assumption parameter
    Vx,Vy, return 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 have
    signed extensions, and
    second, vector subtraction
    assumes parameters Vx,Vy,
    and return value Vz.
    Vz(i)=extend(Vx(i))-extend(Vy(i)) i=0:number-1
```

vadd.t.h && vsub.t.h

- 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: Add vectors at high level
    The result of the addition is taken from the
    upper half of the elements and put in order
    into the lower half of the return value vector
    assuming that Vx,Vy are the 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: Add vectors at high level
    The result of the subtraction is taken from
    the upper half of the elements and put in
    order into the lower half of the return value
    vector assuming that Vx,Vy are the 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.t.s && vsub.t.s

- 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
    Suppose Vx,Vy are the two parameters, Vz is the return value, and U/S
    denotes with or without sign
    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); Vz(i)=Max; Else if
    Vx(i)+Vy(i)<Min
    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
    Suppose Vx,Vy are the two parameters, Vz is the return value, and U/S
    denotes with or without sign
    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); If Vx(i)-Vy(i)>Max; Else
    if Vx(i)-Vy(i)<Min
    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: Add the result with
rounding to take the high half of the result
    Suppose Vx,Vy are two parameters and Vz is the
    return value.                                                    i=0:(number-
    round=1<<(element_size/2-1);                                     1)
    Vz(i)=(Vx(i)+Vy(i)+round)>>[element_size-
    1:element_size/2];
    (Addition with rounding takes the higher half)
    The result is placed sequentially in
    the lower half of the destination
    register Vz (default)
```

- 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,

_t h uint32x4_t)

- uint64x2_t asub_u64_rh (uint64x2_t, uint64x2_t)

>>> Function Description: Subtract the result with rounding to take the high half

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

round=1<<(element_size/2-1);

Vz[i]=((Vx[i]+Vy[i]+round)>>(element_size-1))[element_size/2];

(Addition with rounding takes the higher half)

The result is placed sequentially in the lower half of the destination register Vz (default)

i=0:(number-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 averaging
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S
    is the sign bit.
    Vz(i)=(Vx(i)+Vy(i))>>1; i=0:number-1
    For U, the right shift is a logical right shift, and for S, the right shift
    is an 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: Subtractive averaging
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S
    is the sign bit.
    Vz(i)=(Vx(i)-Vy(i))>>1; i=0:number-1
    For U, the right shift is a logical right shift, and for S, the right shift
    is an arithmetic right shift.
```

vaddh.t.r && vsubh.t.r

- 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: Additive averaging and rounding operations

Suppose 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, the right shift is a logical right shift, and for S, the right shift is an 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 averaging and rounding operations

Suppose 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, the right shift is a logical right shift, and for S, the right shift is an arithmetic right shift.

vadd.t.x & vsub.t.x

- 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

Suppose 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 Extends the value zero or sign to twice the element bit 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

Suppose 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 Extends the value zero or sign to twice the element bit width depending on the 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: Add proximity elements to a vector.
    Suppose Vx,Vy are two parameters and 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)
```

vpadd.t.s

- 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: Saturated addition of neighboring
elements of a vector.
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
    the sign bit.
    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(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.t.e

- `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: Extended addition of neighboring elements
of a vector.
    Suppose 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 extends the value zero or sign to
    twice the element's bit width according to U/S
```

vpadda.t.e

- 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: Extend and accumulate
the proximity elements of a vector.
    Suppose Vx,Vy are two parameters, Vz is the return      i=0:(number/
    value, and U/S is the sign bit.
    Vz(2i+1:2i) =Vz(2i+1:2i)
    +extend(Vx(2i))+extend(Vx(2i+1));
→1)
    extend Extends the value zero or sign to twice the
    element bit width depending on the U/S.
```

vsax.t.s

- 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 mismatch subtraction and addition
    Suppose Vx,Vy are two parameters, Vz
    is the return value, U/S is the sign
    bit signed=(T==S); ( chosen
    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.t.s

- 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: Add or subtract vectors in the wrong place
Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
the sign bit.
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(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: Average the vector after subtracting and adding the mismatches.

Suppose Vx,Vy are two parameters, Vz is the return value, and 0/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, the right shift is a logical right shift, and for S, the right shift is an 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: Average the vectors after adding and subtracting the mismatches.
Suppose 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, the right shift is a logical right shift, and for S, the right shift is an arithmetic 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
elements take
absolute values
assuming Vx is the
parameter and Vz is
the return value.
Vz(i)=abs(Vx(i)); i=0:number-1
```

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

```
>>> Function Description:Absolute value of vector element saturation
    Assume that Vx is the argument, Vz is the return value, and 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.t.s

- 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
    Suppose Vx,Vy are the parameters, Vz is the return value, and 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.t.e

- 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:Expand and subtract the vector elements to
get the absolute value.
    Suppose Vx,Vy are the 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 Extend the value zero or
    sign to 2 times the element's bit width
    according to U/S
```

vsabsa.t

- int8x16_t vsabsa_s8(int8x16_t, int8x16_t,)
- int16x8_t vsabsa_s16(int16x8_t, int16x8_t,)
- int32x4_t vsabsa_s32(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:  Subtract the
    vector elements to get the absolute
    value, and then add them up. Suppose
    Vz,Vx,Vy are the 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.t.e

- int16x16_t vsabsa_s8_e(int16x16_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:Subtract the
    vector elements after expansion, take the
    absolute value, and then add the
    assumption that Vz, Vx, Vy are the
    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 extends the value zero-
    extended or sign-extended to twice the element's
    bit-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: Vector
    elements are
    negative assuming
    Vx is the parameter
    and Vz is the return
    value.
    Vz(i)=-Vx(i), i=0:number-1
```

vneg.t.s

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

```
>>> Function
    description: Vector
    elements saturated
    with negative
    assumptions 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: Maximize vector elements
    Suppose Vx,Vy are two parameters and 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: Minimize vector elements
    Suppose Vx,Vy are two parameters and Vz is the return value.
    Vz(i)=min((Vx(i),Vy(i)) ;
    i=0:number-1 min takes the
    smaller of the values of 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:
    Vector near the element
    to take the maximum
    value of the assumption
    that Vx, Vy are two
    parameters, Vz is the
    return value of the
    Vz(i)=max(Vx(2i),Vx(2i+1)).  i=0:(number/2-1)
    Vz(number/2+i)=max(Vy(2i),Vy(2i+1));
    i=0:(number/2-1)
    max takes the greater of the two elements
```

- `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:
    Vector proximity to the
    element to take the
    minimum value of the
    assumption that Vx, Vy
    are two parameters, Vz is
    the return value of the
    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 of the
    two elements
```

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

- 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 not equal
    to 0 Assume Vx is the
    argument 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 less
    than or equal to 0
    Assume Vx is the
    parameter, Vz is the
    return value If
    Vx(i)≤0 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
    with elements less
    than 0. Suppose 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 vcmphz_s8 (int8x16_t)
- int16x8_t vcmphz_s16 (int16x8_t)
- int32x4_t vcmphz_s32 (int32x4_t)

```
>>> Function
    Description:Vector
    with elements
    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
    greater than or equal
    to 0 assume Vx is the
    parameter, Vz is the
    return value If Vx(i)
    ≥ 0 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: The vector element is less than
    Suppose Vx,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
```

- 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 with elements less than or equal to
    Suppose Vx,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
```

- 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 greater than or equal to
    Suppose Vx,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
```

- 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: A vector whose elements are greater than
    Suppose Vx,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
```

- 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: The elements of the vector are not equal to
    Suppose Vx,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 cropping
to take saturation values
    Suppose Vx,imm6 are two parameters, Vz
    is the return value, and U/S is the sign bit.
    U: Max=2^(imm6)-1, Min=0.                          Vz(i)=Ma
    S: Max=2^(imm6-1)-1, Min=-2^(imm6-1).              x.
    else    Vz(i)=Vx(i);
    end       i=0:number-1
    Regardless of whether T is U/S,treat
    Vx(i) as always signed If Vx(i)>Max
    The range of U:imm6 is 0~
    else if Vx(i)<Min Vz(i)=Min.
    (element_size-1)
    The range of S:imm6 is 1~
    (element_size)
```

4.5.6.2 Integer Multiply Instruction

vmul.t && vmu li.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: Multiply vector elements
    Suppose Vx,Vy are two parameters, Vz is the return value, and 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)
- int16x8_t vmuli_s16 (int16x8_t, int16x8_t, const)
- int32x4_t vmuli_s32 (int32x4_t, int32x4_t, const)
- 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: Multiply vector elements
    Suppose Vx,Vy,index are the three parameters, Vz is the return value,
    and U/S is the sign bit.
    Vz(i)=Vx(i)*Vy(index));
    i=0:number-1 index ranges from 0
    to (128/element_size -1)
```

vmul.t.h && vmuli.t.h

- 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:
    Multiply vector elements
    to take the high part of
    the assumption that Vx, Vy
    are two parameters, Vz is
    the return value of the
    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)
- int16x8_t vmuli_s16_h (int16x8_t, int16x8_t, const)
- int32x4_t vmuli_s32_h (int32x4_t, int32x4_t, const)
- 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: Multiply vector elements by their higher
half
    Suppose Vx,Vy,index are the three parameters and Vz is the return
    value.
    Vz(i)=(Vx(i)*Vy(index)) [2*element_size-1:element_size];
    i=0:number-1
    Take the high part of the multiplication result
    The range of index is 0 to (128/element_size -1).
```

vmul.t.e && vmuli.t.e

- 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:Extended multiplication of vector elements
    Suppose Vx,Vy are two parameters and Vz is the return value.
    Vz(i)=(Vx(i)*Vy(i))[2*element_size-1:0];    i=0:(number-1)
    The multiplication result is taken to full precision, i.e., 2 timesthe
    bit width of the element.
```

- int16x16_t vmuli_s8_e (int8x16_t, int8x16_t, const)
- int32x8_t vmuli_s16_e (int16x8_t, int16x8_t, const)
- int64x4_t vmuli_s32_e (int32x4_t, int32x4_t, const)
- 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:Extended multiplication of vector elements
    Suppose Vx,Vy,index are the three parameters and Vz is the return
    value.
    Vz(i)=(Vx(i)*Vy(index))[2*element_size-1:0];      i=0:(number-1)
    The multiplication result is taken to full precision, i.e., 2 timesthe
    bit width of the element.
    The range of index is 0 to (128/element_size -1).
```

vmula.t && vmulai.t

- int8x16_t vmula_s8 (int8x16_t, int8x16_t,)
- int16x8_t vmula_s16 (int16x8_t, int16x8_t,)
- int32x4_t vmula_s32 (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: Multiply and accumulate vector elements
    Suppose Vz,Vx,Vy are three
    parameters,and also Vz is the return
    value Vz(i)=Vz(i)+Vx(i)*Vy(index));
    i=0:number-1 index range 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)
- int32x4_t vmulai_s32 (int32x4_t, int32x4_t, int32x4_t, const)
- 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: Multiply and accumulate vector elements
    Suppose Vz,Vx,Vy,index 4 parameters,
    and Vz is the return value
    Vz(i)=Vz(i)+Vx(i)*Vy(index));
    i=0:number-1 index range is
    0~(128/element_size -1)
```

vmula.t.e && vmulai.t.e

- int16x16_t vmula_s8_e (int16x16_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: Extended multiply-accumulate vector
elements.
    Suppose Vz,Vx,Vy 3 parameters and 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 is taken to full precision, i.e., 2 timesthe
    bit width of the element.
```

- int16x16_t vmulai_s8_e (int16x16_t, int8x16_t, int8x16_t, const)
- int32x8_t vmulai_s16_e (int32x8_t, int16x8_t, int16x8_t, const)
- int64x4_t vmulai_s32_e (int64x4_t, int32x4_t, int32x4_t, const)
- 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: Extended multiply-accumulate vector
elements.
    Suppose Vz,Vx,Vy,index 4 parameters and 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 is taken to full precision, i.e., 2 times the
    bit width of the element.
    The range of index is 0 to (128/element_size -1).
```

vmuls.t && vmulsi.t

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

- int32x4_t vmuls_s32 (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: Multiply and Decrement Vector Elements
    Suppose 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: Multiply and Decrement Vector Elements
    Suppose Vz,Vx,Vy,index 4 parameters and Vz is the return value.
    Vz(i)=Vz(i)-Vx(i)*Vy(index));
    i=0:number-1 index ranges from 0 to
    (128/element_size -1)
```

vmuls.t.e && vmulsi.t.e

- 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 decimation
    Suppose Vz,Vx,Vy are 3 parameters and 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 is taken to full precision, i.e., 2 timesthe
    bit width of the element.
```

- 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)

- int64x4_t vmulsi_s32_e (int64x4_t, int32x4_t, int32x4_t, const)
- 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 decimation
    Suppose Vz,Vx,Vy,index 4 parameters and 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-
↪1)
    The multiplication result is taken to full precision, i.e., 2 timesthe
    bit width of the element.
    The range of index is 0 to (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: chain multiply
accumulate vectors
    Suppose Vx,Vy are two parameters, Vz is the
    return value, and U/S is the sign bit.
    Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0].                i=number/4-
    1
    i=0:number-1 multiplication result taken to full
    precision, i.e., 2 timesthe element bit-width
    Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+T
    mp[4i]);
```

extend means to extend the result to the bit width of the destination element according to U/S, i.e., 4 timesthe bit-width of the element.

- 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: chain multiply accumulate vectors
with indexes
    Suppose Vx,Vy,index three parameters, Vz is the return
    value, and U/S is the sign bit.
    Tmp(4i+1)=(Vx(4i+1)*Vy(4*index+1))[2*element_size-
    1:0];                                                    i=0:number/4
    Tmp(4i)=(Vx(4i)*Vy(4*index))[2*element_size-1:0];        -1
    Tmp(4i+2)=(Vx(4i+2)*Vy(4*index+2))[2*element_si          i=0:number/4
    ze-1:0];                                                 -1
    Tmp(4i+3)=(Vx(4i+3)*Vy(4*index+3))[2*element_si          i=0:number/4
    ze-1:0];                                                 -1 i=number/4-
    extend means to extend the result to the bit-width of the
    destination element according to U/S, i.e., 4 timesthebit-
    width of the sourceoperand element.
    The result is multiplied to full precision, i.e., 2
    timesthebit width of the element.
    Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+T
    mp[4i]);
```

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 Multiply Accumulate Add
Suppose Vz,Vx,Vy 3 parameters and Vz is the return value and U/S is the sign bit.
Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0]; i=0:number-1
The multiplication result is taken to full precision, i.e., 2 timesthebit width of the element.
Vz(4i+3:4i)= 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 result to the bit-width of the destination element according to U/S, i.e., 4 timesthebit-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)
- 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 multiply accumulate plus
Suppose Vz,Vx,Vy,index 4 parameters, while Vz is the return value and 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 is taken to full precision, i.e., 2 timesthe bit width of the element.
Vz(4i+3:4i)= 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 result to the bit-width of the destination element according to U/S, i.e., 4 timesthebit-width of the source operand element index is in the range 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 expansion with
    saturated decimal
    multiplication assuming
    Vx,Vy are the two
    parameters and 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 is taken to full
    precision, i.e., 2 times the element's bit
    width )
    End     i=0:(number-1)
```

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

```
>>> Function Description: Vector
    expansion with index and
    saturated decimal
    multiplication assuming
    Vx,Vy,index are the three
    parameters and 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 is taken to full
    precision, i.e., 2 times the element's bit
    width )
    End     i=0:(number-1)
    The range of index is 0 to (128/element_size -1).
```

vrmulh.t.s && vrmulhi.t.s

- 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: Multiply vectors with saturated high
semidecimal numbers.
    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]; (Multiplication result is taken
    to be higher)
    End     i=0:(number-1)
```

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

```
>>> Function Description: Vector
    with index and saturation take
    the high half decimal
    multiplication assuming that Vx,
    Vy, index is 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]; (Multiplication result is taken to
    be higher)
    End     i=0:(number-1)
    The range of index is 0 to (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
    with saturated high half
    rounded decimal
    multiplication assuming that
    Vx,Vy are the 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]; (Multiplication result is taken to
    be higher)
    End     i=0:(number-1)
```

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

```
>>> Function Description: Vector with
    index and saturation take the high
    half rounded decimal multiplication
    assuming that Vx, Vy, index is three
    parameters, Vz is the return value of
    the
    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]; (Multiplication result is taken to be
    higher)
    End     i=0:(number-1)
    The range of index is 0 to (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 saturated take the high half
    rounded decimal multiply
    accumulate assuming that 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) preserves full precision of
    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 multiply-accumulate result)
    End   i=0:(number-1)
    (Note: Saturation is performed after totalization)
```

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

```
>>> Function Description: Vector with
    index with saturation take the high
    half rounded decimal multiply
    accumulate assuming that Vx, Vy,
    index is 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) preserves full precision of 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 multiply-accumulate result)
    End   i=0:(number-1)
    (Note: Saturation is performed after totalization)
    The range of index is 0 to (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 saturated take the high half
    of the rounded decimal
    multiplication and decimation
    assumptions 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) preserves full precision of
    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 multiply-accumulate result ) End
    i=0:(number-1)
    (Note: Saturation is performed after totalization)
```

- 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 saturated take the high
    half rounded decimal multiply
    accumulate hypothesis Vx, Vy, index
    is three 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) preserves 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 multiply-accumulate result ) End
i=0:(number-1)
(Note: Saturation is performed after totalization)
The range of index is 0 to (128/element_size -1).
```

vrmulshr.t.e && vrmulshri.t.e

- 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 Expansion with
Shifted Decimal
Multiplication Suppose
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 is arithmetically right-shifted after
retaining full precision          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 expansion with
shifted decimal multiplication
assuming 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) multiplication result
retains full precision after arithmetic
right shift                       imm4=0~15
index range is 0~(128/element_size -1)
```

vrmulsa.t.e && vrmulsai.t.e

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

>>> Function Description: Vector Expansion with Shifted Decimal Multiplication Accumulation

Suppose Vz,Vx,Vy,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 is shifted arithmetically to the right after retaining full precision, and then summed. 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 expansion with shifted
decimal multiplication and accumulation
    Suppose Vz,Vx,imm,index 4 parameters and Vz is the return value.
    Vz(i)=Vz(i)+ ((Vx(i)*Vx+1(index))>>imm);
    i=0:(number-1) multiplication result retains all
    the precision after arithmetic shift to the
    right,and then accumulates                imm=0~15
    index range is 0~(128/element_size -1)
```

vrmulss.t.e && vrmulssi.t.e

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

```
>>> Function Description: Vector Expansion with Shifted Decimal
Multiplication and Negative Accumulation
    Suppose Vz,Vx,Vy,imm are 4 parameters and Vz is the return value.
    Vz(i)=Vz(i)+ ((-Vx(i)*Vy(i))>>imm); i=0:(number-1)
    The result of the multiplication is shifted arithmetically to the
    right after retaining full precision,and then decimated.    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 expansion with shifted
    decimal multiplication and negative
    accumulation Suppose
    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 result of the multiplication is shifted
    arithmetically to the right after retaining
    full precision,and then decimated.
    imm=0~15 index range is 0~(128/element_size
    -1)
```

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

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

```
>>> Function Description: Vector with saturated
    complex real imaginary cross-multiply
    cumulative cumulative take the high half
    rounded assuming that Vz, Vx, Vy are parameters,
    while Vz is the return value of the
    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;
    Tmp(i) preserves 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)

```
i=0:(number/2-1)
```

(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 multiply-accumulate result ) End
i=0:(number-1)
(Note: Saturation is performed after totalization)
```

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

```
>>> Function Description: Vector with index with
saturated complex real imaginary cross multiply
accumulate accumulate take the high half rounded
assuming that Vz,Vx,Vy,index are 4 parameters, while Vz
is the return value of the
round=1<<(element_size-1);
Tmp(2i+1)=Vz(2i+1)<<element_size+Vx(2i)*Vy(2index+1)*2+round;
i=0:(number/2-
↪1)
Tmp(2i)=Vz(2i)<<element_size+ Vx(2i)*Vy(2index)*2+round;
i=0:(number/2-1) Tmp(i) preserves 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 multiply-accumulate result ) End
i=0:(number-1)
(Note: Saturation is performed after totalization)
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,)
- int16x8_t vrmulxas_s16_rs(int16x8_t, int16x8_t,)
- int32x4_t vrmulxas_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function Description: Vector with saturated complex real part of the imaginary part of the cross-multiplication cumulative addition and subtraction to take the high half of the rounding assumption that Vz, Vx, Vy are three parameters, while Vz is the return value of the
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) preserves 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 the multiply-accumulate-subtract) 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)

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

```
>>> Function Description: Vector with index with
    saturated complex real part imaginary part cross
    multiply accumulate accumulate subtract take the
    high half rounded assuming that Vz,Vx,Vy,index are 4
    parameters, while 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-
↪1)
    Tmp(2i)=Vz(2i)<<element_size-Vx(2i+1)*Vy(2index+1)*2+round;
    i=0:(number/2-1) Tmp(i) preserves 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 the multiply-accumulate-subtract) 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,)
- int16x8_t vrmulxss_s16_rs(int16x8_t, int16x8_t,)
- int32x4_t vrmulxss_s32_rs(int32x4_t, int32x4_t, int32x4_t)

>>> Function Description: Vector with saturated complex real part of the imaginary part of the cross-multiplication cumulative cumulative to take the high half of the rounding assumption that Vz, Vx, Vy are three parameters, while Vz is the return value of the
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) preserves full precision of 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 the multiply-accumulate-subtract) 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)
- int16x8_t vrmulxssi_s16_rs(int16x8_t, int16x8_t, int16x8_t, const)
- int32x4_t vrmulxssi_s32_rs(int32x4_t, int32x4_t, int32x4_t, const)

>>> Function Description: Vector with index with saturated complex real part of the imaginary part of the cross multiply accumulate accumulate take the high half rounded assuming that Vz, Vx, Vy, index is 4 parameters, while Vz is the return value of the
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-
→1)
   Tmp(2i)=Vz(2i)<<element_size-Vx(2i)*Vy(2index)*2+round;
   i=0:(number/2-1) Tmp(i) preserves 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 the multiply-accumulate-subtract) 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,)
- int16x8_t vrmulxsa_s16_rs(int16x8_t, int16x8_t,)
- int32x4_t vrmulxsa_s32_rs(int32x4_t, int32x4_t,)

```
>>> Function Description: Vector with saturated
   complex real imaginary cross multiply
   accumulate accumulate take the high half
   rounded assuming that Vz, Vx, Vy are parameters,
   while Vz is the return value of the
   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) preserves the full precision of the operation
```

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

```
>>> Function Description: Vector with index with
    saturated complex real imaginary cross multiply
    accumulate accumulate take the high half rounded
    assuming that Vz,Vx,Vy,index are 4 parameters, while Vz
    is the return value of the
    round=1<<(element_size-1);
    Tmp(2i+1)=Vz(2i+1)<<element_size-Vx(2i+1)*Vy(2index)*2+round;
    i=0:(number/2-
→1)
    Tmp(2i)=Vz(2i)<<element_size+Vx(2i+1)*Vy(2index+1)*2+round;
    i=0:(number/2-1) Tmp(i) preserves 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 the multiply-accumulate-subtract) End
          i=0:(number-1)
    (Note: Saturation 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: Multiply complex numbers
    Suppose Vx,Vy are two parameters and 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) preserves 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
    addition/subtraction ) End                    i=0:(number-1)
    (Note: Saturation is performed after addition and subtraction)
```

vrcmula.t.e

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

```
>>> Function Description: Multiply complex numbers with right shift
and expansion
    Suppose Vz,Vx,Vy,imm are 4 parameters and 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))>>
'→imm).
    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))>>)
'→imm).
    i=0:(number/2-1) (real part)
    The result of multiplying complex numbers is shifted
    arithmetically to the right after retaining full precision, and then
    summed.                                          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
    assuming Vx,Vy are the
    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+r
    ound; i=0:(number/ 2-1)
    Tmp(i) preserves 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
    addition/subtraction ) End                    i=0:(number-1)
    (Note: Saturation is performed after addition and subtraction)
```

vrcmulca.t.e

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

```
>>> Function Description: Complex conjugate multiply right shift
expand accumulate
    Suppose Vz,Vx,Vy,imm 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))>
'->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))>>
'->imm4).
    i=0:(number/2-1) (real part)
    The result of multiplying complex numbers is shifted
    arithmetically to the right after retaining full precision, and then
    summed.                                          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: Multiply complex numbers by (-x)*y
    Suppose Vx,Vy are two parameters and 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) preserves 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
addition/subtraction ) End                    i=0:(number-1)
(Note: Saturation is performed after addition and subtraction)
```

vrcmulna.t.e

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

```
>>> Function Description:Complex number take negative
multiplication right shift expanding bit cumulative
    Suppose Vz,Vx,Vy,imm4 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))>>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))>>)
→imm4).
    i=0:(number/2-1) (real part)
    The result of multiplying complex numbers is shifted
    arithmetically to the right after retaining full precision,and then
    summed.                                          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
    assuming that Vx, Vy are two
    parameters, Vz is the return
    value of the
    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) preserves 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 result of multiplication and
addition/subtraction ) End                          i=0:(number-1)
(Note: Saturation is performed after addition and subtraction)
```

vrcmulcna.t.e

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

```
>>> Function Description: Complex
conjugate take negative multiply
right shift expand accumulate
suppose Vz,Vx,Vy,imm4 are
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)+
((Vx(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))>>)
'→imm4).
i=0:(number/2-1) (real part)
The result of multiplying complex numbers is shifted
arithmetically to the right after retaining full precision,and then
summed.                                              imm=0~15
```

4.5.6.3 Integer inverse, inverse square, e exponential fast arithmetic and approximation instructions

vrecpe.t && vrec ps.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: The
    vector element is
    the inverse of the
    assumption that Vx
    is the parameter and
    Vz is the return
    value.
    Vz(i) ≈ 1/(Vx(i))
    i=0:(number-1) (quick
    calculation of the inverse
    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: Approximate the inverse of a vector
    Suppose Vx,Vy are two two, 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:
Square the inverse of a
vector element.                    (Quickly calculate the
    Suppose Vx is a                inverse square of Vx(i)
    parameter and Vz is a          )
    return value.
    Vz(i)≈     i=0:(number-
    1)
```

- 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.
    Suppose Vx,Vy are two parameters and 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:
    The vector element
    takes the exponential
    value of e. Assume that
    Vx is the input and Vz is
    the return value.
    Vz(i) ≈  e^(Vx(i))
    i=0:(number-1) (quick
    calculation of e-index value
    for Vx(i))
```

4.5.6.4 Integer shift instruction

vsh t.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: Shift Vector Left
    Suppose Vx,Vy are the 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)
    the lower 8-bit data Vy(i)[7:0] in each element
    Vy(i) of Vy as the signed shift index; for U, the
    right-shift is a logical right-shift, and for S, the
    right-shift is an arithmetic right-shift; the right-
    shift is a logical right-shift.
```

vsht.t.s

- 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
    Suppose Vx,Vy are two parameters, Vz is the return value, and 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) Take the lower 8-bit data Vy(i)[7:0] in each
    element Vy(i) of Vy as the signed shift index; sat determines
    whether the left-shift result is overflowed according to the bit-
    width of the write-back element and determines whether the left-
    shift result is overflowed according to the U/
↪S saturates the overflow result to the corresponding maximum or
minimum value; the
    For U, the right shift is a logical right shift, for S, the right shift
    is an arithmetic right shift.
```

vsht.t.r

- 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: Shift vector round to the right
    Suppose 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)=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) of Vy as the signed shift index; for U, the
    right-shift is a logical right-shift, and for S, the
    right-shift is an arithmetic right-shift; the right-
    shift is a logical 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
    Suppose 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)
    The lower 8-bit data Vy(i)[7:0] in each element
    Vy(i) of Vy is used as the signed shift index; sat
    determines whether the left-shift result overflows
    according to the write-back element bit-width, and
    determines whether the left-shift result overflows
    according to the U/
↵S saturates the overflow result to the corresponding maximum or
minimum value
    For U, the right shift is a logical right shift, and for S, the 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: Left Shift Vector Registers
    Suppose 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)
    with 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: Immediate vector left shift
    Suppose Vx,imm are two parameters and Vz is the return value.
    Vz(i)=Vx(i)<<imm.
    i=0:(number-1) imm ranges
    from 0 ~ element_size-1
```

vshl.t.s && vshli.t.s

- 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: Saturate Vector Registers with Left Shift
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
    the sign bit.
    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) [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)
    with 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:
    Immediate vector left
    shift saturation assuming
    that Vx,imm are two
    parameters, Vz is the return
    value of the
    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)<<imm)>Max
    Vz(i)=Max; Else if
    (Vx(i)<<imm)<Min)   Vz(i)=Min;
    Else if (Vx(i)<<imm)<Min)
    Else   Vz(i)= Vx(i)<<imm.
    i=0:(number-1) imm ranges from 0 ~
    element_size-1
```

- 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 Extension Immediate Left Shift
    Suppose 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 element
    to 2xoriginal bit width based on 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: Shift vector registers to the right.
    Suppose 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; for U, the
    right shift is a logical right shift, for S, the right
    shift is an 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: Shift vector immediately to the right
    Suppose 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, the right shift is a logical right shift, and for S, the right shift
    is an arithmetic right shift.
    The range of imm is 1~ element_size
```

vshr.t.r && vshri.t.r

- 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: Shift vector register right to get round
Suppose 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);
Vz(i)=(Vx(i)+round)>>Vy(i)[
7:0]; i=0:(number-1)
with the lower 8-bit data Vy(i)[7:0] in each element Vy(i) of Vy as the unsigned shift index.
For U, the right shift is a logical right shift, and for S, the right shift is an 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: Immediate vector left shift to round
Suppose Vx,imm are two parameters, Vz is the return value, and U/S is the sign bit.
round=1<<(imm-1);Vz(i)=(Vx(i)+round)>> imm). i=0:(number-1)
For U, the right shift is a logical right shift, and for S, the right shift is an arithmetic right shift.
The range of imm is 1~ element_size

vshri.t.l

- 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: Shift the vector immediately to the right and take the lower half of the vector.

Suppose Vx,imm are the 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 64bit of Vz)

For U, the right shift is a logical right shift, and for S, the 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: Shift the vector immediately round to the
lower half of the vector.
    Suppose Vx,imm are the 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 lower half of result)
    Vz(i)={Tmp(2i+1),Tmp(2i)};
    i=0:(number/2-1) (the result is
    placed in the lower 64bit of Vz)
    For U, the right shift is a logical right shift, and 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: Shift the vector immediately to the right and take the lower half-saturation.

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

```
signed=(T==S); (selected 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)>> imm)>Max
Tmp(i)=Max; Else if (Vx(i)>>
imm)<Min             Tmp(i)=Min;
Else if (Vx(i)>> imm)<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 64bit of Vz)
```

For U, the right shift is a logical right shift, and 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: Immediately shift the vector right round to
the lower half-saturation.
    Suppose Vx,imm are the two parameters, Vz is the return value, and U/S
    is the sign bit.
    round=1<<(oimm-1);signed=(T==S); ( selected 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 64bit of Vz)
    For U, the right shift is a logical right shift, and for S, the right shift
    is an arithmetic right shift.
    The range of oimm is 1 ~ element_size
```

vshria.t

- int8x16_t vshria_s8 (int8x16_t, int8x16_t, const)
- int16x8_t vshria_s16 (int16x8_t, int16x8_t, const)
- int32x4_t vshria_s32 (int32x4_t, int32x4_t, const)
- int64x2_t vshria_s64 (int64x2_t, int64x2_t, const)
- 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: Accumulate vectors with immediate right
shift.
    Suppose Vz,Vx,imm three parameters, 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 imm ranges
    from 1 ~ element_size
```

vshria.t.r

- `int8x16_t vshria_s8_r (int8x16_t, int8x16_t, const )`

- int16x8_t vshria_s16_r (int16x8_t, int16x8_t, const)
- int32x4_t vshria_s32_r (int32x4_t, int32x4_t, const)
- int64x2_t vshria_s64_r (int64x2_t, int64x2_t, const)
- 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.
    Suppose Vz,Vx,imm three parameters, 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, the right shift is a logical right shift, and for S, the right shift
    is an arithmetic right shift.
    The range of imm is 1~ element_size
```

vexh.t && vexl.t

- int8x16_t vexh_s8 (int8x16_t, int8x16_t,)
- int16x8_t vexh_s16 (int16x8_t, int16x8_t,)
- int32x4_t vexh_s32 (int32x4_t, int32x4_t,)
- int64x2_t vexh_s64 (int64x2_t, int64x2_t,)
- 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 Round
Accumulate
    Suppose Vz,Vx,ry are the parameters, while 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≥ imm1≥ 0
```

- int8x16_t vexl_s8 (int8x16_t, int8x16_t,)
- int16x8_t vexl_s16 (int16x8_t, int16x8_t,)
- int32x4_t vexl_s32 (int32x4_t, int32x4_t,)

- int64x2_t vexl_s64 (int64x2_t, int64x2_t,)
- 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: Insert vector high data
    Suppose Vz,Vx,ry are the parameters, while 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≥ imm1≥ 0
```

4.5.6.5 Integer Move (MOV), Element

Operation, Bit Operation

Instructions vm tvr.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: Write transfer of a single element of a
vector
    Suppose Vz,rx,index are the three parameters, while Vz is the return
    value and U/S is the sign bit.
    Vz(index)=Rx[element_size-1:0], rest of elements unchanged
    Index range is 0~(128/element_size -1)
```

vmtvr.t.2

- int8x16_t vmtvr_s8_2 (int8x16_t, long , const int)
- int16x8_t vmtvr_s16_2 (int16x8_t, long , const int)
- int32x4_t vmtvr_s32_2 (int32x4_t, long , const int)
- uint8x16_t vmtvr_u8_2 (uint8x16_t, long , const int)
- uint16x8_t vmtvr_u16_2 (uint16x8_t, long , const int)
- uint32x4_t vmtvr_u32_2 (uint32x4_t, long , const int)

```
>>> Function Description: Write transfer of two elements of a vector
    Suppose Vz,rx,index are the three parameters, while 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],␣
↪the 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
    Suppose Vx,index are 3 parameters and Rz is the return value.
    Rz=extend_32(Vx(index));
    extend_32 Extends the value zero or
    sign to 32 bits according to 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 Symbol Bit Read Transfer
    Assume that 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; If
    Type=32 for i=0:3, Rz[i]=Vx(i)[31]; end
    Rz[31:4]=0 end Rz[31:4]=0;
    (Extract the sign bit of each element of Vx and place it in turn in the
    lower bit of the general-purpose register Rz.)
```

vmov.t.e

- 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 Extension Transfer

Assume that Vx is the argument, Vz is the return value, and U/S is the sign bit

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

extend extends the value zero or sign to two timesthe element bit width according to U/S

vmov.t.l && vmov.t.h

- 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
    Suppose Vx,Vy are two parameters, Vz is the return value, and 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 Level Transfer
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
    the sign bit.
    Vz(i)={Vx(2i+1)[element_size-1:element_size/2], Vx(2i)[element_size-
    1:element_
→size/2]};
    i=0:(number/2-1)
    Vz(number/2+i)={Vy(2i+1)[element_size-
    1:element_size/2],Vy(2i)[element_size-
→1:element_size/2]};
    i=0:(number/2-1)
    Takes the higher 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 saturation transfer
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
    the sign bit.
    signed=(T==S); ( selected 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)>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 element )
    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 element )
    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 Rounds Transfer
    Suppose Vx,Vy are two parameters, Vz is the return value, and 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␣
↪(take the high half of the element)
    Tmp2(i)=(Vy(i)+round)[element_size-1: element_size/2]; i=0:(number-
    1␣
↪(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-
    transfer assuming
    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 bytes inverted
    assuming that Vx is
    the input and Vz is the
    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 in half-
    byte reverse order
    assuming that Vx is the
    input and Vz is the
    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 inverted
    assuming Vx is the
    input and Vz is the
    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
    inverted assuming Vx
    is the input and Vz is
    the return value.
    Vz(number-1:0)= {Vx(0), Vx(1)};
```

vexti.t && vext.t

- int8x16_t vexti_s8 (int8x16_t, int8x16_t, const)
- uint8x16_t vexti_u8 (uint8x16_t, uint8x16_t, const int)

```
>>> Function Description: Immediate Vector Data Splice
    Suppose 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 numbered elements
of Vx to the lower numbered elements
of Vz) Else Vz (imm[3:0]:0)=Vx(15:15-
imm[3:0]);
(Copy the high elements of Vx to the low elements of Vz )
If imm[4]==0 ,Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0);
(Copy the lower number of elements of Vy to the upper number of
elements of Vz )
Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
(Copy the higher numbered elements of Vy to the higher
numbered elements of Vz )
where imm[3:0] ranges from 0 to 14.
```

- int8x16_t vext_s8 (int8x16_t, int8x16_t,)
- uint8x16_t vext_u8 (uint8x16_t, uint8x16_t, int)

```
>>> Function Description: Register Vector Data Splice
Suppose Vx,Vy,Rk 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 numbered elements
of Vx to the lower numbered elements
of Vz) Else Vz (imm[3:0]:0)=Vx(15:15-
imm[3:0]);
(Copy the high elements of Vx to the low elements of Vz )
If imm6[4]==0 ,Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0);
(Copy the lower number of elements of Vy to the upper number of
elements of Vz )
Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
(Copy the higher elements of Vy to the higher elements of Vz )
where imm[3:0] ranges from 0 to 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
Suppose Vx,Vy are two parameters and 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
    Suppose Vx,Vy are two parameters and 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: Vectors by bits and operations
    Suppose Vx,Vy are two parameters and 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: Vectors by bitwise non-sum operation
    Suppose Vx,Vy are two parameters and 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 Or Arithmetic
    Suppose Vx,Vy are two parameters and 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: Invert vectors by bits
    Suppose Vx,Vy are two parameters and 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 by Bitwise Non Arithmetic
    Suppose Vx,Vy are two parameters and 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: Vectors by bitwise or operation
    Suppose Vx,Vy are two parameters and 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,)
- 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
    Suppose Vx,Vy are two parameters and 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:
    Vector symbols are
    consecutively the same
    assuming that Vx is the
    parameter and Vz is the
    return value.
    Consecutive digits from the MSB that are identical to the element's
    sign digit, the sign digit is not counted
    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:
    Vector with 0
    consecutive digits in the
    highest bit assumes that
    Vx is the parameter and Vz
    is the return value.
    Number of consecutive 0's 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 number
    Assuming Vx is the
    parameter, Vz is the
    return value count_one
    calculates the number
    of bits of 1 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)

```
    t       4          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 Positioning and Setting
    Suppose Vx,Vy are two parameters and 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 registers and
    general purpose source registers
    assuming 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 the source and
    destination registers of a unary
    vector assuming Vx,index are the
    inputs 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
    binary vectors between source
    and destination registers
    assuming Vx,index are the inputs
    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 , const int)
- int16x8_t vins_s16_1 (int16x8_t, int16x8_t, const , const int)
- int32x4_t vins_s32_1 (int32x4_t, int32x4_t, const , 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: Monadic vector insertion
    Suppose Vz,Vx,index,index2 4 parameters and Vz is the return value.
    Vz(index2)=Vx(index);     The values of the remaining elements of
    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 , const int)
- int16x16_t vins_s16_2 (int16x16_t, int16x16_t, const , const int)
- int32x8_t vins_s32_2 (int32x8_t, int32x8_t, const , 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
    Suppose Vz,Vx,index,index2 4 parameters and Vz is the return value.
    Vz(index2)=Vx(index);
    Vz(index2+number)=
    Vx(index+1)
    The remaining elements of Vz have the same value
    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 , const int)
- int16x16_t vpkg_s16_2 (int16x16_t, int16x16_t, const , const int)
- int32x8_t vpkg_s32_2 (int32x8_t, int32x8_t, const , 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
    Suppose Vz,Vx,index,index2 4 parameters and Vz is the return value.
    bound=number; bound is used to determine if a register is crossed, and
    number is the number of elements.
    Vz(index2)=Vx(inde
    x); if
    index2+1<bound
    Vz(index2+1)=Vx(index+bound);
    else
    Vz(index2+1)=Vx(index+bound);
    The remaining elements of Vz,Vz+1 are 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
    Suppose Vx is a parameter and Vz is a 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 de-interleaving
    Suppose Vx is a parameter and Vz is a return value.
    Vz(i)= Vx(2i);Vz(i+number)=
    Vx(2i+1); i=0:(number-1)
```

4.5.6.6 Integer Immediate Number Generation Instruction

vmo vi.8

- `int8x16_t vmovi_s8 (const signed char)`
- `uint8x16_t vmovi_u8 (const signed char)`

```
>>> Function
    Description: Vector
    Immediate Number
    Transfer Suppose 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 Number Transfer
    Suppose imm8,index are the two parameters, Vz is the return value,
    and 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)
    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 Number Transfer
    Suppose imm8,index are the two parameters, Vz is the return value,
    and 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)
    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
    Number Expansion
    Transfer Assuming imm8
    is the parameter and Vz
    is the return value.
    Vz(i)= {8{imm8[i]}}; i=0:(number/2-1); the rest of the elements of
    Vz are set to 0
```

- int8x16_t vmaski_s8_h (const signed char)
- uint8x16_t vmaski_u8_h (const signed char)

```
>>> Function Description:
    Vector Immediate
    Number Expansion
    Transfer Assuming 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
    of Vz remain unchanged
```

vmaski.16

- int16x8_t vmaski_s16 (const signed char)
- uint16x8_t vmaski_u16 (const signed char)

```
>>> Function Description:
    Vector Immediate
    Number Expansion
    Transfer Assuming 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 Command

vld.t.n(n=1/2/3/4) && vs t.t.n(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 loading
    Suppose Rx,offset are the two parameters and Vz is the return value.
    When the address corresponding to rx is not
    element_size aligned, a non-alignment
    exception is thrown. 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.T.N, N= 1/2/3/4)
    Vz(j)=MEM(Rx+offset+j*(2^(size))
    ) ; end The rest of the
    elements are set to zero.
```

- 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
   assuming Rx, offset,
   Vz are the parameters
   When the address corresponding to rx is not
   element_size aligned, a non-alignment
   exception is thrown. size=00/01/10/11
   corresponds to byte/half word/word/double
   word offset=imm7<<size (offset is in the range
   (0-127)<<size).
   for j=0:N-1 MEM(Rx+offset+j*(2^(size)))=Vz(j); end
```

vldru.t.n(n=1/2/3/4) && vstru.t.n(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 assuming
    Rx,Ry are the two
    parameters and Vz is the
    return value.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    size=00/01/10 corresponds to byte/half word/word
    for j=0:N-1 Vz(j)=MEM(Rx+j*(2^(size)));end rest of elements set 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
    assuming Rx,Ry,Vz are the
    three parameters.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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.t.n(n=1/2/3/4) && vstu.t.n(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 assuming Rx is
    a parameter and Vz is a
    return value.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    size=00/01/10 corresponds to byte/half word/word
    for j=0:N-1 Vz(j)=MEM(Rx+j*(2^(size))); end the rest of the
    elements set 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 assuming
    Vz,Rx are inputs.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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+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 Suppose Rx
    is the parameter and
    Vz is the return
    value.
    When the address corresponding to rx is not element_size aligned,
    a non-alignment exception is thrown.
    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 assuming
    Rx,Vz are two
    parameters.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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
    assuming Rx is the
    parameter and Vz is the
    return value.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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 assuming that
    Rx,Vz are two parameters.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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 Suppose
    Rx,Ry are the 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
    assuming 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 Load
    Suppose Vx,Vy are two parameters and Vz is the return value.
    When the address corresponding to rx is not element_size aligned,
    a non-aligned exception is thrown.
    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 The remaining
    elements are set to zero.
    If N = 0, then the result is not expected
```

vlrw.t.n

- int32x4_t vlrw_s32_4 (const int, const int, const int, const)
- uint32x4_t vlrw_u32_4 (const int, const int, const int, const)

```
>>> Function Description: Read from vector memory
    Suppose imm1,imm2,imm3,imm4 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 Add/Subtract Compare Instruction

vadd.t && vsu b.t

- `float32x4_t vadd_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description: Floating Point Vector Addition
    Suppose Vx,Vy are two parameters and 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
    Suppose Vx,Vy are two parameters and 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
    Suppose Vx,Vy are two parameters and 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 Suppose
    Vx,Vy are the two
    parameters and 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 Suppose Vx,Vy
    are the two parameters and
    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
    Suppose Vx,Vy are two parameters and 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
    Subtract Absolute Value
    Suppose Vx,Vy are the two
    parameters and 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
    Inversion Suppose
    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
    Suppose Vx,Vy are two parameters and 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
    Suppose Vx,Vy are two parameters and Vz is the return value.
    Vz(i)=min((Vx(i),Vy(i)) ;
    i=0:number-1 min takes the
    smaller of the values of two
    elements
```

vmaxnm.t && vminnm.t

- `float32x4_t vmaxnm_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector Floating Point
    Specification Maximum
    Assume Vx,Vy are the two
    parameters and 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 ;
    Unlike VMAX, one of the two elements is a quote.␣
↪NaN and the other is a norm, the value of the norm is taken as the
output.
```

- float32x4_t vminnm_f32 (float32x4_t, float32x4_t)

```
>>> Function Description:
    Vector Floating Point
    Specification Minimum
    Assume Vx,Vy are the two
    parameters and 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 ;
    Unlike VMIN, one of the two elements is a quote.␣
↪NaN and the other is a norm, the value of the norm is taken as the
output.
```

vpmax.t && vpmin.t

- float32x4_t vpmax_f32 (float32x4_t, float32x4_t)

```
>>> Function Description:
    Vector Floating Point
    Neighborhood Maximum
    Assume Vx,Vy are the two
    parameters and 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)).
    i=0:(number/2-1) max takes the greater of two
    elements
```

- float32x4_t vpmin_f32 (float32x4_t, float32x4_t)

```
>>> Function Description:
    Vector Floating Point
    Neighborhood Minimum
    Assume Vx,Vy are the 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 of the
    two elements
```

vcmpnez.t && vcmpne.t

- `float32x4_t vcmpnez_f32 (float32x4_t)`

```
>>> Function description:
    Vector floating point not
    equal to zero
    comparison assumes Vx
    is the parameter, Vz is the
    return value
    If Vx(i)! = 0 V…; i=0:number-1      Else Vz(i)=00… 000;
    i=0:number-1
```

- `float32x4_t vcmpne_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating point is
    equal to zero comparison
    Suppose Vx,Vy are the 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 is
    greater than or equal to
    zero comparison assuming
    Vx is the parameter, Vz is
    the return value
    If Vx(i)≥0 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 assuming
    Vx,Vy are the
    parameters, Vz is the
    return value
    If Vx(i)≥Vy(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 assuming Vx is
    the parameter, 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 assuming
    that Vx,Vy are
    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
```

- float32x4_t vcmphz_f32 (float32x4_t)

```
>>> Function Description:
    Vector floating-point
    greater than zero
    comparison assumes 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 less
    than or equal to zero
    comparison assuming Vx is
    the parameter, Vz is the
    return value
    If Vx(i)≤0 Vz(i)=11··· 111;      Else Vz(i)=00··· 000; i=0:number-1
```

4.5.6.9 floating-point multiply instruction

vmult.t && vmu li.t

- float32x4_t vmul_f32 (float32x4_t, float32x4_t)

```
>>> Function Description: Multiply vectors with single precision
    Suppose Vx,Vy are two parameters and 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: Single precision indexed multiplication of
vectors
    Suppose Vx,Vy,index 3 parameters and 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: Accumulate vectors with single-precision
multiplication.
    Suppose Vz,Vx,Vy are three
    parameters and Vz is the return
    value Vz(i)=Vz(i)+Vx(i)*Vy(i);
    i=0:number-1Note: The result of
    the multiplication is rounded up
    and then added.
```

- float32x4_t vmulai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

```
>>> Function Description: Vector single precision indexed multiply-
accumulate
    Suppose Vz,Vx,Vy,index 4 parameters,
    and Vz is the return value
    Vz(i)=Vz(i)+Vx(i)*Vy(index);
    i=0:number-1Note: The result of the
    multiplication is rounded up and
    then added.
    index=0~ (128/element_size -1);
```

vmuls.t && vmulsi.t

- float32x4_t vmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> Function Description: Decrease vector multiplication with single
precision.
    Suppose Vz,Vx,Vy are three parameters and Vz is the return value.
    Vz(i)=Vz(i)-Vx(i)*Vy(i);  i=0:number-1
    Note: Multiplication results are rounded up and down.
```

- float32x4_t vmulsi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)

```
>>> Function Description: Vector single precision indexed multiply-accumulate
    Suppose Vz,Vx,Vy,index 4 parameters and Vz is the return value.
    Vz(i)=Vz(i)-Vx(i)*Vy(index)).   i=0:number-1
    Note: Multiplication results are rounded up and down.
    index=0~ (128/element_size -1);
```

vfmula.t && vfmuls.t

- `float32x4_t vfmula_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description: Fusion multiply accumulate vectors with
single precision.
    Suppose Vz,Vx,Vy are 3
    parameters and Vz is the return
    value Vz(i)=Vz(i)+Vx(i)*Vy(i);
    i=0:number-1Note: The result of
    multiplication retains all
    precision in the totalization.
```

- `float32x4_t vfmuls_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description: Vector single precision fusion
multiplication and decimation
    Suppose Vz,Vx,Vy are 3
    parameters and Vz is the return
    value Vz(i)=Vz(i)-Vx(i)*Vy(i);
    i=0:number-1Note: The result of
    the multiplication retains all
    the precision in the accumulation.
```

vfnmula.t && vfnmuls.t

- `float32x4_t vfnmula_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description: Vector Floating Point Fusion Multiplication
with Negative Decrementation
    Suppose Vz,Vx,Vy 3 parameters
    and Vz is the return value Vz(i)=-
    Vz(i)-Vx(i)*Vy(i);          Vz(i)=-
    Vz(i)-Vx(i)*Vy(i); i=0:number-
    1Note: The result of the
    multiplication retains the full
    precision of the sum.
```

- `float32x4_t vfnmuls_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description: Vector Floating Point Multiply Decrement
    Suppose Vz,Vx,Vy 3 parameters
    and Vz is the return value Vz(i)=-
    Vz(i)+Vx(i)*Vy(i);
    i=0:number-1Note: The result of
    the multiplication retains all the
    precision in the accumulation.
```

vfmulxaa.t && vfmulxaai.t

- `float32x4_t vfmulxaa_f32 (float32x4_t, float32x4_t, float32x4_t)`

>>> Function Description: Vector floating-point complex multiply-accumulate real part of the imaginary part of the calculation assuming that Vz, Vx, Vy are parameters, while Vz is the return value of the
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 for addition/subtraction.

- `float32x4_t vfmulxaai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)`

>>> Function Description: Vector Floating Point Index Complex Multiply Accumulate Real Imaginary Part Calculation Assuming Vz,Vx,Vy,index are 4 parameters, and 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 for addition/subtraction.
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 multiply-accumulate real part of the imaginary part of the calculation assuming that Vz, Vx, Vy are parameters, while Vz is the return value of the
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 for addition/subtraction.
```

- `float32x4_t vfmulxasi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)`

```
>>> Function Description: Vector Floating Point Index Complex Multiply Accumulate Real Imaginary Part Calculation Assuming 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 for addition/subtraction.
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 multiply-accumulate real part of the imaginary part of the calculation assuming that Vz, Vx, Vy are parameters, while Vz is the return value of the

```
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 for addition/subtraction.

- `float32x4_t vfmulxssi_f32 (float32x4_t, float32x4_t, float32x4_t, const int)`

>>> Function Description: Vector Floating Point Index Complex Multiply Accumulate Real Imaginary Part Calculation Assuming 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 for addition/subtraction.

```
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 multiply-
    accumulate real part of the
    imaginary part of the calculation
    assuming that Vz, Vx, Vy are
    parameters, while Vz is the return
    value of the
    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 for
    addition/subtraction.
```

- `float32x4_t vfmulxsai_f32 (float32x4_t, float32x4_t, float32x4_t, const int)`

```
>>> Function Description: Vector Floating
    Point Index Complex Multiply
    Accumulate Real Imaginary Part
    Calculation Assuming 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 for
    addition/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

Suppose Vx,Vy are two parameters and 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: Tmp(i) multiplication is saturated with a rounding operation, and Vz(i) multiplication accumulation is saturated with a rounding operation.

- `float32x4_t vfcmula_f32 (float32x4_t, float32x4_t, float32x4_t)`

>>> Function Description: Vector Floating Point Complex Multiplier Accumulator

Suppose Vz,Vx,Vy are 3 parameters and 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) multiply and add for one rounding saturation operation, Vz(i) multiply and add for another rounding saturation operation.

vfcmulc.t && vfcmulca.t

- `float32x4_t vfcmulc_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating-point
    complex conjugate
    multiplication assuming
    Vx,Vy are the two
    parameters and 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: Tmp(i) multiplication is saturated with a rounding operation,
    and Vz(i) multiplication accumulation is saturated with a
    rounding operation.
```

- `float32x4_t vfcmulca_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating point
    complex conjugate
    multiply accumulate
    assuming Vz,Vx,Vy are 3
    parameters and 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) multiply and add for one rounding saturation
    operation, Vz(i) multiply and add for another rounding
    saturation operation.
```

vfcmuln.t && vfcmulna.t

- `float32x4_t vfcmuln_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating-point
    complex multiplication
    assuming that Vx,Vy are
    the two parameters and 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: Tmp(i) multiplication is saturated with a rounding operation,
    and Vz(i) multiplication accumulation is saturated with a
    rounding operation.
```

- `float32x4_t vfcmulna_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating-point
    complex multiply-
    accumulate assuming Vz,
    Vx, Vy are three 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) multiply and add for one rounding saturation
    operation, Vz(i) multiply and add for another rounding
    saturation operation.
```

vfcmulcn.t && vfcmulcna.t

- `float32x4_t vfcmulcn_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector Floating Point
    Complex Conjugate
    Negative Multiplication
    Assuming Vx,Vy are the two
    parameters and 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: Tmp(i) multiplication is saturated with a rounding operation,
    and Vz(i) multiplication accumulation is saturated with a
    rounding operation.
```

- `float32x4_t vfcmulcna_f32 (float32x4_t, float32x4_t, float32x4_t)`

```
>>> Function Description: Vector
    Floating Point Complex
    Conjugate Multiply
    Accumulate Assume Vz,Vx,Vy
    are 3 parameters and 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) multiply and add for one rounding saturation
    operation, Vz(i) multiply and add for another rounding
    saturation operation.
```

4.5.6.10 Floating-point inverse, inverse square, e exponential fast arithmetic and approximation instructions

vrecpe.t && vrec ps.t

- `float32x4_t vrecpe_f32 (float32x4_t)`

```
>>> Function
    Description: The
    vector floating-point
    inverse instruction
    assumes that Vx is a
    parameter and Vz is a
    return value.
    Vz(i) ≈ 1/(Vx(i)) i=0:(number-1) (quick calculation of the inverse
    value of Vx(i) )
```

- `float32x4_t vrecps_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description: Vector Floating Point Cepstrum
Approximation
    Suppose Vx,Vy are two parameters and 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
    Cepstrum Assuming Vx
    is the parameter and Vz
    is the return value.
    Vz(i) ≈ 1/((Vx(i))^(1/2))       i=0:(number-1) ( quick calculation
    of the inverse square of Vx(i))
```

- `float32x4_t vrsqrts_f32 (float32x4_t, float32x4_t)`

```
>>> Function Description:
    Vector floating-point
    square approximation
    assuming Vx,Vy are the
    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-
computation                                    (Quick calculation of
    Suppose Vx is a parameter                  the e-index value of
    and Vz is a return value.                  Vx(i))
    Vz(i)≈ e^(Vx(i))
    i=0:(number-1)
```

4.5.6.11 floating-point conversion instruction

vd tos.t

- `float32x4_t vdtos_f64 (float64x2_t)`

```
>>> Function Description: Vector
    double-precision floating-
    point single-precision floating-
    point conversion assumes that
    Vx is a parameter and Vz is a
    return value.
    Vz(i)={double_to_single(Vx(2i+1)), double_to_single(Vx(2i))};
    i=0:(number/2-
↪1);
    Converting 64-bit double-precision floating-point numbers to 32-
    bit single-precision floating-point numbers
```

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 that Vx is a
    parameter and Vz is a
    return value.
    Vz(i)=float_to_fix(Vx(i)); i=0:(number-1).
    Converts (16-bit/32-bit) floating point numbers to U/S fixed point
    numbers of the same bit width.
    FCR[20:16].frpos[4:0] Specify the
    decimal point position for fixed point
    numbers 16bit fixed point numbers
    use frpos[3:0] to indicate the decimal
    portion of 1~16 bits
    The 32bit fixed point number uses frpos[4:0] to indicate the
    fractional part of 1~32 bits.
```

- `float32x4_t vxtof_s32_f32 (int32x4_t)`
- `float32x4_t vxtof_u32_f32 (uint32x4_t)`

```
>>> Function Description: Vector Fixed to 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)

```
    Converts a (16-bit/32-bit) U/S fixed-point number to a floating-
    point number of the same bit width.
    FCR[20:16].frpos[4:0] Specify the
    decimal point position for fixed point
    numbers 16bit fixed point numbers
    use frpos[3:0] to indicate the decimal
    portion of 1~16 bits
    The 32bit fixed point number uses frpos[4:0] to indicate the
    fractional part of 1~32 bits.
```

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
    assuming 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 position of the fixed-point
    number
    The 16bit fixed point number uses frpos[3:0] to indicate the
    fractional part of 1~16 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
    assumes that Vx is a
    parameter and Vz is a
    return value.
    Vz(i)={single_to_fix16(Vx(2i+1)), single_to_fix16(Vx(2i))} ;
    i=0:(number/2-1);
    Convert 32-bit single float to 16-bit
    U/S fixed point, FCR[20:16].frpos[4:0]
    specifies the decimal point position
    of the fixed point 16-bit fixed point
    uses frpos[3:0] to indicate the
    fractional part of the 1~16 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 assumes
    that Vx is a parameter
    and Vz is a return
    value.
    Vz(i)=float_to_int(Vx(i)); i=0:(number-1).
    Converts (16-bit/32-bit) floating point numbers to U/S integers of
    the same bit width.
```

vftoi.t1.t2 (halve bit width)

- `int16x8_t vftoi_f32_s16 (float32x4_t)`

- uint16x8_t vftoi_f32_u16 (float32x4_t)

```
>>> Function
    Description: Vector
    floating-point integer
    conversion assumes
    that Vx is a parameter
    and Vz is a return
    value.
    Vz(i)={single_to_int16(Vx(2i+1)), single_to_fix16(Vx(2i))} ;
    i=0:(number/2-1);
    Converts 32-bit single floating point numbers to 16-bit U/S
    integers
```

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 assumes
    that Vx is a parameter
    and Vz is the return
    value.
    Vz(i)=int_to_float(Vx(i)); i=0:(number-1).
    Convert (16-bit/32-bit) U/S integer numbers to floating point
    numbers 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
    that Vx is a parameter
    and Vz is a return
    value.
    Vz(i)=int16_to_single(Vx(i)); i=0:(number-1).
    Converts 16-bit U/S integer numbers to 32-bit single-precision
    floating-point numbers
```

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:
    Rounded vector floating-
    point integer conversion
    assuming Vx is the argument
    and Vz is the return value.
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1).
    Convert 32-bit single-precision floating-point numbers to 32-bit
    U/S integers
```

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:
    Rounded vector floating-
    point integer conversion
    assuming Vx is the argument
    and Vz is the return value.
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1).
    Convert 32-bit single-precision floating-point numbers to 32-bit
    U/S integers
```

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:
    Rounded vector floating-
    point integer conversion
    assuming Vx is the argument
    and Vz is the return value.
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1).
    Convert 32-bit single-precision floating-point numbers to 32-bit
    U/S integers
```

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:
    Rounded vector floating-
    point integer conversion
    assuming Vx is the argument
    and Vz is the return value.
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1).
    Convert 32-bit single-precision floating-point numbers to 32-bit
    U/S integers
```

4.6 dsp

Currently, the CSKY architecture supports two versions of the DSP instruction set, dspv1

and dspv2, where DSPV2 contains vectorized instructions and DSPV1 contains DL and LO registers as operands. the DSP instructions are 32-bit wide, and the compiler determines whether the generated target program supports the dsp instructions or not according to the CPU options. The compiler determines whether the generated target program supports the dsp instruction according to the CPU options. Table 4.2.For details of the support, please refer to

In the following cases, the compiler generates DSP-type vector instructions:

- vector operator expression
- circular optimization
- Using the intrinsic function

Of these, the first two are for more basic scenarios, while the third applies to scenarios that require deep optimization. Detailed descriptions can be found in the following sections of this chapter:

- vector data type (computing)
- Passing rules for parameters and return values of vector types
- vector operator expression

- Loop-optimized generation of vector instructions *(*currently only supported in the *GNU* worker具chain*)*
- *intrinsic* function interface naming convention
- *The intrinsic* interface to *dspv2*

4.6.1 vector data type (computing)

Vector datatypes are usually based on common datatypes, e.g., the vector datatype int8x4_t represents an integer datatype with 8-bit elements, consisting of 4 elements, which has a total bit width of 32 bits. The naming convention is as follows.

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

The element types are int and uint. You need to refer to the header file csky_vdsp.h. The vector data types supported by dspv2 are shown Table 4.12:

Table 4.12: Vector Data Types for dspv2

dspv2	32-bit
int	int8x4_t
	int16x2_t
uint	uint8x4_t
	uint16x2_t

4.6.2 Passing rules for parameters and return values of vector types

Parameters and return values for DSP vector types are passed using ordinary registers.

4.6.3 vector operator expression

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

4.6.3.1 Definition of Vector Type Variables

Vector type variables are defined in two ways:

- The first way is the same as an array definition, e.g.:

```
#include<csky_vdsp.h>

int8x4_t a= {1,2,3,4};
```

- In the second way, an array is defined, and then the array address is converted to a vector pointer type, e.g:

```
#include<csky_vdsp.h>

int a[ ]= {1,2,3,4}.
int8x4_t *ap= (int8x4_t *)a;
```

4.6.3.2 operator (computing)

C uses operators to represent arithmetic operations,

and this is also true for variables of vector type.

Currently, the operators supported by vector

expressions are shown below:

- Addition:+
- Subtraction: -
- Multiplication:*
- Comparison operators:> ,< , ! =,>= ,<= .==
- Logical operators: &,| .^
- Shift operator: ","

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 partial loops to generate vector instructions. The compiler tries to optimize loops into vector instructions when the following conditions are met:

- Current CPU support for vector instructions
- Optimization level is -O1 or above and且 add option -ftree-loop-vectorize

(This option is
turned on by default
for -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 operations */
}
```

If the current CPU supports 32-bit vector addition instructions, with loop optimization turned on, the above code is optimized as shown in the pseudo-code below:

```
#include<csky_vdsp.h>

int8x4_t svfun2 (int8x4_t va,int8x4_t vb,int8x4_t vc)
{
    int8x4_t vc=  va+  vb;    /* Vector operations */
    return vc.
}
```

4.6.5 intrinsic function interface naming convention

The intrinsic interface function name is basically the same as the instruction name, if the instruction name contains ".", it will be replaced by "_" in the function name, e.g. padd.8 corresponds to the intrinsic interface function name. If the name of the instruction contains ".", it will be replaced by "_" in the function name, for example, the name of the intrinsic interface function corresponding to the instruction padd.8 is padd_8. The types of the function's parameters and return values are determined by the data types of the instruction's operands, for example, the instruction padd.8 Rz, Rx, Ry, which adds the result of an 8-bit wide, 4-element shaping vector in two general registers into the general register rz. For example, padd.8 Rz, Rx, Ry is an instruction that adds two 8-bit wide, 4-element shaping vectors from two general registers into general register rz. The function padd_8 is therefore declared as follows:

```
int8x4_t padd_8(int8x4_t__ a, int8x4_t__ b)
```

where 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 The intrinsic interface to dspv2

The commands of dspv2 can be divided into the following sections:

- Integer addition and subtraction, comparison instructions

- Integer shift instruction
- Other Arithmetic Instructions

4.6.6.1 Integer addition and subtraction, comparison instructions

padd.t && psu b.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.t.s && psub.t.s

- 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
    Suppose Vx,Vy are the two parameters, Vz is the return value, and U/S
    denotes with or without sign
    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); Vz(i)=Max; Else if
    Vx(i)+Vy(i)<Min
    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
    Suppose Vx,Vy are the two parameters, Vz is the return value, and U/S
    denotes with or without sign
    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); If Vx(i)-Vy(i)>Max; Else
    if Vx(i)-Vy(i)<Min
    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 averaging
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S
    is the sign bit.
    Vz(i)=(Vx(i)+Vy(i))>>1; i=0:number-1
    For U, the right shift is a logical right shift, and for S, the right shift
    is an 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: Subtractive averaging
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S
    is the sign bit.
    Vz(i)=(Vx(i)-Vy(i))>>1; i=0:number-1
    For U, the right shift is a logical right shift, and for S, the right shift
    is an 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: The elements of the vector are not equal to
    Suppose Vx,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 greater than or equal to
    Suppose Vx,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 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: The vector element is less than
    Suppose Vx,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
```

4.6.6.2 Integer shift instruction

pas ri.t

- int16x2_t pasri_s16 (int16x2_t, const int)

```
>>> Function description:
    vector immediate number
    of arithmetic right shift
    Suppose Vx, imm is the
    two parameters, Vz is the
    return value of the
    Vz(i)=Vx(i)>>imm;
    i=0:(number-1) imm ranges
    from 1 ~ element_size
```

pasr.t

- int16x2_t pasr_s16 (int16x2_t, int)

```
>>> Function Description:
    Vector register
    arithmetic right shift
    Suppose 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 Suppose Vx,imm
    are the two parameters
    and Vz is the return value.
    Vz(i)=Vx(i)>>imm;
    i=0:(number-1) imm ranges
    from 1 ~ element_size
```

plsr.t

- uint16x2_t plsr_u16 (uint16x2_t, int)

```
>>> Function Description:
    Vector Immediate
    Logical Right Shift
    Suppose Vx,Rx are the
    two parameters and 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: Immediate vector left shift
    Suppose Vx,imm are two parameters and Vz is the return value.
    Vz(i)=Vx(i)<<imm.
    i=0:(number-1) imm ranges
    from 1 ~ element_size
```

plsl.t

- int16x2_t plsl_s16 (int16x2_t, int)

```
>>> Function Description: Immediate vector left shift
    Suppose Vx,Rx are two parameters and Vz is the return value.
    imm= Rx[4:0]
    Vz(i)=Vx(i)<<imm.  i=0:(number-1)
```

pasri.t.r

- int16x2_t pasri_s16_r (int16x2_t, const int)

```
>>> Function Description: Vector
    immediate number of
    arithmetic right shift results
    take round assumptions Vx, imm
    is the two parameters, Vz is the
    return value
    round=1<<(imm-1)
    Vz(i)=(Vx(i)+round)>>imm;
    i=0:(number-1) imm has range 1 ~
    element_size
```

pasr.t.r

- int16x2_t pasr_s16_r (int16x2_t, int)

```
>>> Function Description: Vector
    immediate number of
    arithmetic right shift results
    take round assumptions Vx, Rx is
    the two parameters, Vz is the
    return value of the
    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.t.r

- uint16x2_t plsri_u16_r (uint16x2_t, const int)

```
>>> Function Description: Vector
    immediate number of logical
    right shift results take round
    assumption Vx, imm is the two
    parameters, Vz is the return
    value of the
    round=1<<(imm-1)
    Vz(i)=(Vx(i)+round)>>imm;
    i=0:(number-1) imm has range 1 ~
    element_size
```

plsr.t.r

- uint16x2_t plsr_u16_r (uint16x2_t, int)

```
>>> Function Description: Vector
    immediate number of logical
    right shift results take round
    assumptions Vx, Rx is the two
    parameters, Vz is the return
    value of the
    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.t.s

- int16x2_t plsli_s16_s (int16x2_t, const int)
- uint16x2_t plsli_u16_s (uint16x2_t, const int)

```
>>> Function Description:
    Immediate vector left
    shift saturation assuming
    that Vx,imm are two
    parameters, Vz is the return
    value of the
    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)<<imm)>Max
    Vz(i)=Max; Else if
    (Vx(i)<<imm)<Min)   Vz(i)=Min;
    Else if (Vx(i)<<imm)<Min)
    Else   Vz(i)= Vx(i)<<imm.
    i=0:(number-1) imm is in the range 1 ~
    element_size
```

plsl.t.s

- int16x2_t plsl_s16_s (int16x2_t, int)
- uint16x2_t plsl_u16_s (uint16x2_t, int)

```
>>> Function Description:
    Immediate vector left
    shift saturation assuming
    that Vx, Rx are two
    parameters, Vz is the return
    value of the
    imm= Rx[4:0]
    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)<<imm)>Max
    Vz(i)=Max; Else if
    (Vx(i)<<imm)<Min)   Vz(i)=Min;
    Else if (Vx(i)<<imm)<Min)
    Else   Vz(i)= Vx(i)<<imm.
    i=0:(number-1) imm ranges from 0 ~
    element_size-1
```

4.6.6.3 Other Arithmetic Instructions

pas x.t

- int16x2_t pasx_16 (int16x2_t, int16x2_t)

```
>>> Function Description: Add or subtract vectors in the wrong place
    Suppose Vx,Vy are two parameters and 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.t.s

- int16x2_t pasx_s16_s (int16x2_t, int16x2_t)
- uint16x2_t pasx_u16_s (uint16x2_t, uint16x2_t)

```
>>> Function Description: Add or subtract vectors in the wrong place
    Suppose Vx,Vy are two parameters, Vz is the return value, and U/S is
    the sign bit.
    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(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 mismatch subtraction and addition
    Suppose 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);
    Vz(2i)= Vx(2i)+Vy(2i+1); i=0:(number/2-1)
```

psax.t.s

- int16x2_t psax_s16_s (int16x2_t, int16x2_t)
- uint16x2_t psax_u16_s (uint16x2_t, uint16x2_t)

```
>>> Function description: vector mismatch subtraction and addition
    Suppose Vx,Vy are two parameters, Vz
    is the return value, U/S is the sign
    bit 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(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: Average the vectors after adding and subtracting the mismatches.

Suppose 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, the right shift is a logical right shift, and for S, the right shift is an arithmetic right shift.

psaxh.t

- `int16x2_t vsaxh_s16(int16x2_t, int16x2_t)`
- `uint16x2_t vsaxh_u16(uint16x2_t, uint16x2_t)`

```
>>> Function Description: Average the vector after subtracting and adding the mismatches.
    Suppose 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, the right shift is a logical right shift, and for S, the right shift is an 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: Maximize vector elements
    Suppose Vx,Vy are two parameters and 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: Minimize vector elements
    Suppose Vx,Vy are two parameters and Vz is the return value.
    Vz(i)=min((Vx(i),Vy(i)) ;
    i=0:number-1 min takes the
    smaller of the values of two
    elements
```

pext.t.e

- `int16x4_t pext_s8_e (int8x4_t)`
- `uint16x4_t pext_u8_e (uint8x4_t)`

```
>>> Function Description: Vector Expansion
    Assume that Vx is the argument, Vz is the return value, and U/S is the
    sign bit
    Vz(i)=extend(Vx(i)); i=0:(number/2-1)
    extend extends the value zero or sign to two timesthe element bit
    width according to U/S
```

pextx.t.e

- `int16x4_t pextx_s8_e (int8x4_t)`
- `uint16x4_t pextx_u8_e (uint8x4_t)`

```
>>> Function Description: Vector Interleaved Expansion
    Assume that Vx is the argument, Vz is the return value, and 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 extends the value zero or sign to two timesthe element bit
    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 cropping
to get saturation value.
    Suppose Vx,imm4 are two parameters, Vz
    is the return value, and U/S is the sign bit.
    U: Max=2^(imm4)-1, Min=0.                          Vz(i)=Ma
    S: Max=2^(imm4-1)-1, Min=-2^(imm4-1).              x.
    else    Vz(i)=Vx(i);
    Regardless of whether T is U/S,treat
    end         i=0:number-1
    Vx(i) as always signed If Vx(i)>Max
    The range of U:imm4 is 0~
    else if Vx(i)<Min Vz(i)=Min.
    (element_size-1)
    The range of S:imm4 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 cropping to take saturation values
    Assuming Vx, Ry are the two parameters, Vz is the return value, and
    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).
    If Vx(i) > Max, Vz(i) = Max; else if Vx(i) <
    Min(s), Vz(i) = Max; else if Vx(i) < Min(s), Vz(i)
    = Min(s)          Vz(i)=Max; else if Vx(i)<Min
    Vz(i)=Min; else if Vx(i)<Min
    else   Vz(i)=Vx(i);
    end
    i=0:number-1
    The range of U:imm4 is 0~ (element_size-1)
    The range of S:imm4 is 1~ (element_size)
```

pabs.t.s

- `int8x4_t pabs_s8_s(int8x4_t)`

- int16x2_t pabs_s16_s(int16x2_t)

```
>>> Function Description: Absolute value of vector element saturation
    Assume that Vx is the argument, Vz is the return value, and 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.t.s

- int8x4_t pneg_s8_s (int8x4_t)
- int16x2_t pneg_s16_s (int16x2_t)

```
>>> Function
    description: Vector
    elements saturated
    with negative
    assumptions 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: Extended multiplication of vector elements
    Suppose Vx,Vy are two parameters and Vz is the return value.
    Vz(i)=(Vx(i)*Vy(i))[2*element_size-1:0];
    The result is multiplied to full precision, i.e., 2 timesthebit width
    of the element.
```

pmulx.t

- int32x2_t pmulx_s16 (int16x2_t, int16x2_t)
- uint32x2_t pmulx_u16 (uint16x2_t, uint16x2_t)

```
>>> Function Description:
    Vector element
    interleaved extended
    multiplication Suppose
    Vx,Vy are the two
    parameters and Vz is the
    return value.
    Vz(1)= Vx(1) X Vy(0)
    Vz(0)= Vx(0) X Vy(1)
    The result is multiplied to full precision, i.e., 2 timesthebit width
    of the element.
```

prmul.t

- `int32x2_t prmul_s16 (int16x2_t, int16x2_t)`

```
>>> Function Description:
      Vector expansion with
      saturated decimal
      multiplication assuming
      Vx,Vy are the two
      parameters and 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]; (
      Multiplication takes full precision,
      i.e., 2 times the element bit-width)
      End      i=0:(number-1)
```

prmulx.t

- int32x2_t prmulx_s16 (int16x2_t, int16x2_t)

```
>>> Function Description: Vector
      interleaved expansion with
      saturated decimal
      multiplication assuming
      Vx,Vy are the two parameters
      and 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.t.h

- int16x2_t prmul_s16_h (int16x2_t, int16x2_t)

```
>>> Function Description: Vector
    expansion with saturated
    fractional multiplication to take
    the high part of the assumption
    that Vx, Vy are two parameters, Vz
    is the return value of the
    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
    saturated decimal multiplication with
    rounding to take the high part of the
    assumption that Vx, Vy are two parameters, Vz
    is the return value of the
    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.t.h

- int16x2_t prmulx_s16_h (int16x2_t, int16x2_t)

```
>>> Function Description: Vector
    interleaved expansion with
    saturated decimal
    multiplication assuming
    Vx,Vy are the 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 expansion with
    saturated decimal
    multiplication assuming
    Vx,Vy are the two parameters
    and 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: Subtract the
    vector elements to get the absolute
    value, then add the assumption that
    Vx,Vy are the 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: Subtract the
    vector elements to get the absolute
    value, and then add the assumption
    that Rz, Vx, Vy are the parameters,
    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 Base本 operations

- **fabs, fabsf**

Defined in header <math.h>

```
float    fabsf( float arg ).       (1)    (sinc C99
                                          e     )
double   fabs( double arg ).       (2)
```

1-2) Calculate the absolute value of the parameter *arg*.

parameters

arg - floating point parameter

return value

Returns the absolute value of the argument *arg* (|arg|).

- **fmod, fmodf**

Defined in header <math.h>

float double	fmodf(float x, float y); fmod(double x, double y).	(1) (2)	(since C99)

1-2) Calculate the remainder of the parameter *x/y*, which is obtained by the formula *x* - *n* * *y*, n is **rounded up 0**, and the sign of the return value is the as that *of x*.

parameters

x, y - floating point parameters

return value

Returns the remainder of the argument *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*, which is obtained by the formula *x* - *n* * *y*. n is **rounded to the nearest** integer, and the return value is not guaranteed to have the sign as *x*.

parameters

x, y - floating point parameters

return value

Returns the remainder of the argument *x/y*

- **remquo, remquof**

Defined in header <math.h>

```
float      remquof( float x, float y, int *quo );     (1) (since␣
→C99)
double     remquo( double x, double y, int *quo       (2) (since␣
           );
→C99)
```

1-2) Similarly *remainder()* computes the remainder of the argument *x/y* and且 stores the quotient in the argument *quo*.

parameters

x, y - floating point parameters
quo - business

return value

Returns the remainder of the argument *x/y* and且 stores the quotient in the argument *quo*.

- **fma, fmaf**

Defined in header <math.h>

```
float   fmaf( float x, float y, float )          (1)    (since C99)
                                      z ;
double       fma( double x, double y, z );       (2)    (since C99)
                                  double
```

1-2) Calculate the result for the parameter *(x * y)* + z.

parameters

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 the two arguments.

parameters

x, y - floating point parameters

return value

Returns the larger of the 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 the two arguments.

parameters

x, y - floating point parameters

return value

Returns the smaller of the 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 between the two arguments, if x > y, x - y, otherwise +0.

parameters

x, y - floating point parameters

return value

Returns the positive difference between the 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

parameters

unused - constant character pointer parameter

return value

Returns the 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 the arg power of e

parameters

arg - floating point parameter

return value

Returns the arg-squared result of e.

- **exp2, exp2f**

Defined in header <math.h>

```
float   exp2f( float n ).   (1)    (since C99)
double  exp2( double n ).   (2)    (since C99)
```

1-2) Calculate the nth power of 2

parameters

n - floating point parameter

return value

Returns the result to the nth of 2.

- **expm1, expm1f**

Defined in header <math.h>

```
float   expm1f( float arg );       (1)    (since C99)
double  expm1( double arg );       (2)    (since C99)
```

1-2) Calculate the arg power of e - 1

parameters

arg - floating point parameter

return value

Returns the arg power of e - 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 with e as the base and arg as the true number.

parameters

arg - floating point parameter

return value

Returns the logarithm with e as the base and arg as the true number.

- **log10, log10f**

Defined in header <math.h>

```
float   log10f( float arg );       (1)    (sinc C99
                                          e     )
double  log10( double arg );       (2)
```

1-2) Calculate the logarithm with base 10 and arg as the true number.

parameters

arg - floating point parameter

return value

Returns the logarithm with base 10 and arg as the true number.

- **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 with e as the base and 1+arg as the true number.

parameters

arg - floating point parameter

return value

Returns the logarithm with e as the base and 1+arg as the true number.

Defined in header <math.h>

```
float   log2f( float arg );        (1)    (since C99)
double  log2( double arg );        (2)    (since C99)
```

1-2) Calculate the logarithm with 2 as the base and arg as the true number.

parameters

arg - floating point parameter

return value

Returns the logarithm with base 2 and arg as the true number.

4.7.1.3 the multiplication operation (math.)

- **sqrt, sqrtf**

Defined in header <math.h>

```
float    sqrtf( float arg );       (1)    (since C99
                                          e     )
double   sqrt( double arg );       (2)
```

1-2) Calculate the square root of arg

parameters

arg - floating point parameter

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 arg cubic root

parameters

arg - floating point parameter

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.

parameters

x - 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 ); float powf( float base, float exponent ); float powf( float base, float exponent )
double pow( double base, double exponent ); double pow(double base, double exponent ).
```

(1) (since C99)
(2)

1-2) Calculate exponent base

parameters

base - floating point base parameter
exponent - floating point exponent parameter

return value

Returns exponent of base.

- **sin, sinf**

Defined in header <math.h>

```
float    sinf( float arg );  (1)    (since C99)
double   sin( double arg );  (2)
```

1-2) Calculate the sine of arg(radian).

parameters

arg - floating point representation of the angle

return value

Returns the sine of arg(radians).

- **cos, cosf**

Defined in header <math.h>

```
float    cosf( float arg );  (1)    (since C99)
double   cos( double arg );  (2)
```

1-2) Calculate the cosine of arg(radian).

parameters

arg - floating point representation of the angle

return value

Returns the cosine of arg(radians).

- **tan, tanf**

Defined in header <math.h>

```
float    tanf( float arg );  (1)    (since C99)
double   tan( double arg );  (2)
```

1-2) Calculate the tangent of arg(radian)

parameters

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 arcsine of arg(radian).

parameters

arg - floating point representation of the angle

return value

Returns the arcsine of arg(radians).

- **acos, acosf**

Defined in header <math.h>

```
float    acosf( float arg );        (1)    (since C99)
double   acos( double arg );        (2)
```

1-2) Calculate the inverse cosine of arg(radians).

parameters

arg - floating point representation of the angle

return value

Returns the inverse cosine of arg(radians).

- **atan, atanf**

Defined in header <math.h>

```
float    atanf( float arg );        (1)    (sinc C99
                                           e     )
double   atan( double arg );        (2)
```

1-2) Calculate arg(radian) arctangent

parameters

arg - floating point representation of the angle

return value

Returns the arg(radians) arg-tangent.

- **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 arctangent of y/x (radians).

parameters

y - floating point representation of the angle
x - 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

parameters

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

parameters

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)     (sinc C99
                                             e     )
double     tanh( double arg ).      (2)
```

1-2) Calculate the hyperbolic tangent of arg

parameters

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) Calculate the inverse hyperbolic sine of arg

parameters

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) Calculate the inverse hyperbolic cosine of arg

parameters

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

parameters

arg - floating point representation of the angle

return value

Returns the inverse hyperbolic tangent of arg.

4.7.1.5 Errors and Gamma Arithmetic

- **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 );      (1)    (since C99)
double  tgamma( double arg );      (2)    (since C99)
```

4.7.1.6 floating point operation

- **ldexp, ldexpf**

Defined in header <math.h>

```
float   ldexpf( float arg, int exp ).     (1)    (sinc C99
                                                 e     )
double  ldexp( double arg, int exp ).     (2)
```

1-2) Calculate arg * 2 the exp power.

parameters

arg - floating point parameter
exp - plasticity parameter

return value

Returns the of arg * 2.

- **scalbn, scalbnf**

Defined in header <math.h>

```
float      scalbnf( float arg, int exp    (1) (sinc C99)
           );                                   e
double     scalbn( double arg, int exp    (2) (sinc C99)
           );                                   e
float      scalblnf( float arg, long      (5) (sinc C99)
           exp );                               e
double     scalbln( double arg, long      (6) (sinc C99)
           exp );                               e
```

1-2, 5-6) Calculate arg * FLT_RADIX the exp

power.

parameters

arg - floating point parameter

exp - plasticity parameter

return value

Returns the of arg * FLT_RADIX.

- **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 floating-point argument arg and returns it as a signed integer value.

parameters

arg - floating point parameter

return value

Takes the value of the unbiased exponent from 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 index from floating-point parameter arg and returns it as a floating-point value.

parameters

arg - floating point parameter

return value

Takes the value of the unbiased base-independent exponent from 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)    (sinc C99
                                                e     )
double  frexp( double arg, int* exp );   (2)
```

1-2) Decompose a given floating-point value x into normalized fractions and integer powers of 2.

parameters

arg - floating point parameter
exp - plasticity parameter

return value

Decomposes a given floating-point value x into normalized fractions and integer powers of two.

- **modf, modff**

Defined in header <math.h>

```
float       ( float arg, float* iptr );    (1)    (since C99)
double      modf( double arg, double*      (2)
            iptr );
```

1-2) Decompose a given floating-point value arg into integer and decimal parts, each with the same type and sign as arg.

parameters

arg - floating point parameter
iptr - pointer to a floating point value, used to store integer parts

return value

Decomposes a given floating-point value, arg, into integer and decimal parts, each with 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, then return the next representable value of from in the direction of to. If from is equal to to, then to is returned.

parameters

from - floating point parameter
to - floating point parameter

return value

First, the two arguments are converted to the type of the function, and then the next representable value of from is returned in the direction of to. If from is equal to to, then to is returned.

- **copysign, copysignf**

Defined in header <math.h>

```
float      copysignf( float float y ).    (1)    (since C99)
                          x,
double     copysign( double double y ).   (2)    (since C99)
                          x,
```

1-2) Compose a floating-point value with the magnitude of x and the sign of y.

parameters

x - floating point parameter
y - floating point parameter

return value

Compose a floating-point value using the size of x and the sign of y.

4.7.1.7 approximate operation

- **ceil, ceilf**

Defined in header <math.h>

```
float   ceilf( float arg );        (1)    (sinc C99
                                          e     )
double  ceil( double arg );        (2)
```

1-2) Calculate the smallest shaping value that is not less than arg.

parameters

arg - floating point parameter

return value

Returns the smallest plastic value not less than arg.

- **floor, floorf**

Defined in header <math.h>

```
float    floorf( float arg ).      (1)    (sinc C99
                                          e     )
```

```
double   floor( double arg ).      (2)
```

1-2) Calculate the maximum shaping value that is not greater than arg.

parameters

arg - floating point parameter

return value

Returns the largest plastic 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)   (sinc C99)
                                           e
```

1-2) Calculate the closest floating point number to arg.

5-6, 9-10) Calculate closest integer to arg.

parameters

arg - floating point parameter

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) Calculate the largest integer whose size is not greater than arg.

parameters

arg - floating point parameter

return value

Returns the largest plastic value not greater than arg.

Defined in header <math.h>

```
float     nearbyintf( float arg    (1)    (since C99)
                                ).
double    nearbyint( double arg    (2)    (since C99)
                                ).
```

1-2) Rounds the floating-point parameter arg to an integer value in floating-point format.

parameters

arg - floating point parameter

return value

Returns the shaped value of the argument arg.

- **rint, rintf**

Defined in header <math.h>

```
float rintf( float arg                    (1)          (since C99)
).
double rint( double arg                   (2)          (since C99)
).
long lrintf( float arg                    (5)          (since C99)
).
long lrint( double arg                    (6)          (since C99)
).
long long llrintf( float );        (9)          (sinc C99)
arg                                             e
long long llrint( double );        (10)   (sinc C99)
arg                                            e
```

1-2) Rounds the floating-point parameter arg to an integer value in floating-point format.

5-6, 9-10) Rounds the floating-point argument arg to an integer value in a shaping format.

parameters

arg - floating point parameter

return value

Returns the shaped value of the argument arg.

Chapter 5 Programming the Gentei 900 Series CPU

The Gentei 900 series CPUs are processors developed based on the RISC-V architecture. This chapter mainly introduces the programming process of C/C++ , which involves the RISC-V Architecture-related special usages such as instruction set selection, assembly programming, and the instruction intrinsic interface.

This chapter contains the following sections:

- How to add the corresponding option to the processor
- Xantei small size runtime library *libcc-rt*
- *pthread* multithreading *(*currently only supported in the *GNU* worker具 chain*)*
- *C/C++* Language Extensions
- Dynamic linker name
- Generalized coprocessor extensions *C intrinsic* interface
- Instructions for using *RISC-V Vector*

5.1 How to add the corresponding option to the processor

Currently, Gentech has developed a number of processors for the RISC-V architecture, which are compiled and developed using the Gentech RISC-V toolchain in a unified way, and the target code for a given processor is generated through different compilation options.

The Gentei LLVM compiler supports generating target files for the corresponding CPU model by specifying the -mcpu=<CPU model> option, and is also compatible with specifying -march separately,
The -mabi and -mtune options, where -march can be obtained with "-mcpu=<CPU model > -v", -

mabi and -mtune are shown in Table 5.1. The -mcpu option is recommended.

The Gentei GNU compiler supports the -mcpu=<CPU model> selection of all CPUfeatures starting with version V2.6.0. It is also compatible with specifying -march, -mabi and -mtune separately. The -mcpu option is recommended.

For the selection of -march and -mabi, since the implementation of RVV Instrinsic has been upgraded from SIMD to VECTOR V2.2.0(GNU), the corresponding options are different, and且 has been updated from V3.0.0(GNU) to 20191213 for the default ISA-SPEC, and part of the arch has been adjusted, so it is shown in three tables below. The correspondences are shown Table 5.1 after V3.0.0(GNU), 5.2 from V2.2.0(GNU) to V2.10.2(GNU) (c908 is supported since V2.4(GNU)), and Table 5.3 before V2.2.0(GNU).

Table 5.1: CPU to Option Correspondence (After V3.0.0 (GNU) Version (Inclusive))

	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	lp64d	c906v
		tor		
c907 Series	c907	rv64imac_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei_z	lp64	c907
		ihintntl_zihintpause_zihpm_zawrs_zca_zcb_zba_zbb_zbc_zbs_ssc		
		ofpmf_sstc_svinval_svnapot_svpbmt_xtheadc		
	c907fd	rv64imafdc_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifencei	lp64d	c907
		_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_z		
		cd_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot_svpbmt_xthe		
		adc		
	c907fdv	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	lp64d	c907
		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		
	c907fdvm	rv64imafdcv_zicbom_zicbop_zicboz_zicntr_zicond_zicsr_zifence	lp64d	c907
		i_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb		

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_zife	lp64d	c910
		ncei_zihintntl_zihintpause_zihpm_zawrs_zfa_zfbfmin_zfh_zca_zcb_z		
		cd_zba_zbb_zbc_zbs_sscofpmf_sstc_svinval_svnapot_svpbm		

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		

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	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_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		

Remarks.

1. **The default -misa-spec for the Gentei GNU compiler for V3.0.0 (and above)本 is 20191213.**
2. **Extends XTheadVector to be intrinsic 0p10 compatible with the -mrvv-v0p10-compatible option.**

Table 5.2: CPU to Option Correlation (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_	lp64d	c908
		zbs_svinval_svnapot_svpbmt_xtheadc		
	c908i	rv64imac_zicbom_zicbop_zicboz_zihintpause_zba_zbb_zbc_zbs_sv	lp64	c908
		inval_svnapot_svpbmt_xtheadc		
	c908v	rv64imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc	lp64d	c908
		_zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot		
	c908-rv32	rv32imafdc_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc_	ilp32d	c908
		zbs_svinval_svnapot_svpbmt_xtheadc		
	c908v-rv32	rv32imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc	ilp32d	c908
		_zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot		
c910v2 series	c910v2	rv64imafdc_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause	lp64d	c910v2
		_zawrs_zfa_zfbfmin_zfh_zca_zcb_zcd_zba_zbb_zbc_zbs_svinval_s		
		vnapot_svpbmt_xtheadc		
	c920v2	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus	lp64d	c920v2
		e_zawrs_zfa_zfbfmin_zfh_zca_zcb_zcd_zba_zbb_zbc_zbs_zvfbfmin		
		_zvfbfwma_svinval_svpbmt_xtheadc_xtheadvdot		
c920v2.c908v Series	c920v2.c908v	rv64imafdcv_zicbom_zicbop_zicboz_zihintpause_zfh_zba_zbb_zbc	lp64d	c920v2.c908v
		_zbs_svinval_svnapot_svpbmt_xtheadc_xtheadvdot		
c907 Series	c907	rv64imac_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause_z	lp64	c907
		awrs_zcb_zba_zbb_zbc_zbs_svinval_svnapot_svpbmt_xtheadc		
	c907fd	rv64imafdc_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpause	lp64d	c907
		_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_svinval_svnapot_s		
		vpbmt_xtheadc		
	c907fdv	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihintpaus	lp64d	c907
		e_zawrs_zfa_zfbfmin_zfh_zcb_zba_zbb_zbc_zbs_zvfbfmin_zvfbfwm		
		a_svinval_svnapot_svpbmt_xtheadc_xtheadvdot		
	c907fdvm	rv64imafdcv_zicbom_zicbop_zicboz_zicond_zihintntl_zihin	lp64d	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		

Remarks.

1. The -mcpu option is recommended for both the Xinti LLVM compiler and the Xinti GNU compiler for V2.6.0 and above本 and本 for ease of use and compatibility.

2. The Xantei LLVM compiler follows the community rule that the vector instruction set v does not contain the half-precision vector instruction set zvfh by default; to use it you need to additionally add zvfh0p1 to the -march option, as well as add -menable-experimental-extensions; p extends the representation in -march to p0p94.

Table 5.3: CPU to Option Correlation (Before .2.0(GNU))

	CPU	-march (before version V2.0.3)	-march (after version V2.0.3 (inclusive))	-mabi	-mtune
e902 Series	e902	rv32ecxtheadse	ditto	ilp32e	e902
	e902m	rv32emcxtheadse	ditto	ilp32e	e902
	e902t	rv32ecxtheadse	ditto	ilp32e	e902
	e902mt	rv32emcxtheadse	ditto	ilp32e	e902
e906 Series	e906	rv32imacxtheade	ditto	ilp32	e906
	e906f	rv32imafcxtheade	ditto	ilp32f	e906
	e906fd	rv32imafdcxtheade	ditto	ilp32d	e906
	e906p	rv32imacpzp64_xtheade	rv32imacpzpsfoperand_xtheade	ilp32	e906
	e906fp	rv32imafcpzp64_xtheade	rv32imafcpzpsfoperand_xtheade	ilp32f	e906
	e906fdp	rv32imafdcpzp64_xtheade	rv32imafdcpzpsfoperand_xtheade	ilp32d	e906
e907	e907	rv32imacxtheade	ditto	ilp32	e907

Series	e907f	rv32imafcxtheade	ditto	ilp32f	e907
	e907fd	rv32imafdcxtheade	ditto	ilp32d	e907
	e907p	rv32imacpzp64_xtheade	rv32imacpzpsfoperand_xtheade	ilp32	e907
	e907fp	rv32imafcpzp64_xtheade	rv32imafcpzpsfoperand_xtheade	ilp32f	e907
	e907fdp	rv32imafdcpzp64_xtheade	rv32imafdcpzpsfoperand_xtheade	ilp32d	e907
c906 Series	c906	rv64imacxtheadc	ditto	lp64	c906
	c906fd	rv64imafdcxtheadc	ditto	lp64d	c906
	c906fdv	rv64imafdcvxtheadc	ditto	lp64dv	c906
c910 Series	c910	rv64imafdcxtheadc	ditto	lp64d	c910
	c920	rv64imafdcvxtheadc	ditto	lp64dv	c920
	r910	rv64imafdcxtheadc	ditto	lp64d	r910

continued on next page

Table 5.3 - continued from previous page

	CPU	-march (before version V2.0.3)	-march (after version V2.0.3 (inclusive))	-mabi	-mtune
	r920	rv64imafdcvxtheadc	ditto	lp64dv	r920

Remarks: In XuanTie V2.0.3(GNU)本 , the P extension has been upgraded to 0.9.4本 , the zp64 instruction set has been changed to zpsfoperand, and everything else remains the same.

The -mcpu, -march, -mabi, and -mtune options are described in the following sections:

- *-mcpu* option (supported since Gentei *V2.6 (GNU)*)
- *-march* option
- *-mabi* option
- *-mtune* option

5.1.1 -mcpu option (supported since Gentei V2.6 (GNU) version本)

The -mcpu option selects arch, tune, and the default abi according to the correspondences in Table 5.1 and Table 5.2, where abi can be selected from other non-default abi with the -mabi option.

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, as shown in Table 5.4. For example, rv32imacxthead indicates that the 32-bit integer base, integer multiply-divide, atomic operation, compression, and metaphysical performance enhancement instruction sets are used in the current target file. Except for the extended instruction set, the rest of the instruction sets are RISC-V standard instruction sets, which are described in more detail in the RISC-V ISA SPEC.

Table 5.4: Instruction Sets for Strings

character name	instruction set
rv32i	32-bit Integer Base Instruction Set
rv32e	Embedded 32-bit integer base instruction set (essentially the same as rv32i but using only 16 registers)

rv64i	64-bit Integer Base Instruction Set
m	Integer multiply and divide 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 length instruction set)
p	Packed-SIMD Instruction Set
zba	Address Calculation Instruction Set (B Extended Instruction Subset)
zbb	Basic bit manipulation instruction set (B extended instruction subset)
zbc	carry-less multiplication instruction set (B extended instruction subset)
zbs	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 (computing)
zvfh	Half-Precision Vector Instruction Set
v0p7	Vector instruction set with version number v0.7.1
xtheadcmo	Instruction set for Xantium cache management
xtheadsync	Instruction set for the synchronization of the Gentech multiprocessor
xtheadba	Instruction set for basalt address calculation
xtheadbb	Instruction set for basic bit manipulation of basalt
xtheadbs	Gentian Bit command set
xtheadcondmov	Instruction set for basalt conditional movement
xtheadmemidx	Xantei Integer Memory Operation Instruction Set
xtheadmempair	Xantei Double Integer Register Memory Operation Instruction Set
xtheadfmemidx	Gentian floating-point memory operation instruction set
xtheadmac	Genji Multiplication Cumulative Instruction Set
xtheadfmv	Gentry Dual Floating Point High Bit Data Transfer Instruction Set
xtheadint	Xenon Accelerated Interrupt Instruction Set
xtheadvdot	Gantry Vector dot Operational Enhancement Instruction Set
xtheadvector	Instruction set for basalt vectors
xtheadmatrix	Matrix Arithmetic Enhancement Instruction Set
xtheadfpp	Gentech Fast Peripheral Command Set
xxtccei	Xenon Coprocessor Integer Instruction Set
xxtccef	Gentech coprocessor floating-point instruction set
xxtccev	Xenon Coprocessor Vector Instruction Set
xtheadc	Xthead C-series performance-enhancing instruction set, equivalent to xtheadba_xtheadbb_xtheadbs_xtheadcmo_xtheadc
	ondmov_xtheadfmemidx_xtheadfmv_xtheadmac_xtheadmemidx_xthead
	mempair_xtheadsync
xtheade	Xthead E-Series Performance Enhancement Instruction Set, Equivalent to xtheadba_xtheadbb_xtheadbs_xtheadcmo_xtheadc
	ondmov_xtheadfmv_xtheadint_xtheadmac_xtheadmemidx_xtheadsync
xtheadse	Gentech Small E-Series Performance Enhancement Instruction Set, Equivalent to xtheadcmo
rv32g	Abbreviations for rv32imafd_zicsr_zifencei
rv64g	Abbreviations for rv64imafd_zicsr_zifencei

Note: The T-HEAD T-HEAD Extension Specification Extension specification splits the set of T-HEAD extensions such as xtheadc into sub-extensions, see the for details.

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 of which is shown in Table 5.5, and a more detailed description can be found in the RISC-V ABI SPEC.

Table 5.5: ABI Rule Descriptions

name (of a thing)	clarification
ilp32e	Passing rules for 16 registers at rv32e
ilp32	When rv32i, all types use the parameter integer register passing rule
ilp32f	Single-precision floating-point type parameter passing rules using floating-point registers at rv32i
ilp32d	Passing rules for single- and double-precision floating-point type parameters using floating-point registers at rv32i
lp64	When rv64i, all types use the parameter integer register passing rule
lp64f	Single-precision floating-point type parameter passing rules using floating-point registers at rv64i
lp64d	Passing rules for single and double precision floating-point type parameters using floating-point registers at rv64i

5.1.4 -mtune option

The -mtune option does not affect the correctness of program execution, but specifying the corresponding tune can generate a target program optimized for a particular CPU with cost modeling and pipelining. The current mtune options for all CPUs in Xuan Tie can be found in Table 5.1 and Table 5.2.

5.2 Xantei small size runtime library libcc-rt

A runtime library is a library used by a compiler to implement built-in operations and functions of a programming language to provide runtime support for programs in that language. Such libraries generally include 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 the library's soft floating-point functions.

In Gentei GNU series compilers, libgcc is the default runtime library, which follows IEEE754 standard floating-point computation. libcc-rt is another runtime library developed by Gentei for embedded customers who have in-depth requirements on code size. It basically similar to libgcc. At the same time, in order to get the optimal code size, some of its floating-point computation functions are incompatible with the libgcc interface. For details, please refer to the chapter on the differences between *libcc-rt* and *libgcc* floating-

5.2.1 libcc-rt Usage

Add the option -mccrt at link time to replace libgcc with libcc-rt.

NOTE: Currently, this option is only available under the E902, E906, and E907 series.

5.2.2 Differences between libcc-rt and libgcc floating-point computation sections

In order to achieve better code size optimization, the implementation of the floating-point interface in libcc-rt is differentiated from libgcc. The differences are summarized below
Table:

Table 5.6: libcc-rt Floating-Point Interface Differences

	libcc-rt implementation standard	**libgcc implementation standard**
1. return 0 instead of an unspecified number for values that result in precision overflow	a. Floating-point arithmetic operations (addition, subtraction, multiplication, and division) on the Returns 0 if the value is underflowed by precision.	Return Unspecified Number
	b. Floating-point precision conversion且 When precision is reduced (dou- ble→float), if underflow occurs, return 0	Return Unspecified Number
2. return an unanticipated overflow value for values that result in precision overflow	a. Floating-point arithmetic operations (addition, subtraction, multiplication, and division) that return an unpredictable value for precision overflow. Overflow value for the period	When overflow occurs, return the corresponding symbolic infinity value.
	b. Floating-point precision conversion且 On precision reduction (dou- ble→float), for values with precision overflow, returns An unanticipated overflow value	When overflow occurs, return the corresponding symbolic infinity value.
	c. Floating-point to integer conversion, which returns the actual value of the pair when the floating-point value is greater than the most significant value of the integer to be represented. Low-bit interception in integer binary	For overflow values, returns the saturation value for that integer type with the corresponding sign
3. Treat non-numbers and infinities in the process and result of operations as normalized numbers with corresponding indices.	a. Floating-point arithmetic operations (addition, subtraction, multiplication, and division), when there are nonnumeric, infinite numbers in the operands, are treated as if they were	An infinity is added to or subtracted from another number, or the value of that infinity, while a subtraction between infinities occurs when or if a non-number is involved in the operation.

	Specification of numbers with corresponding indices, continue arithmetic	Returns Nan when
	b. Floating-point precision conversion且 When precision is increased (float→double) , for low-precision values of infinities, nonnumbers, and unspecified numbers, and high-precision numbers when normalize the number of its exponents	Infinity and non-infinity numbers are converted to their corresponding high-precision infinity and non-infinity numbers.
	c. Floating-point comparison instructions, which recognize non-numbers as their ordered floating-point counterparts for comparison.	there is a non-numeric (Nan) number among the numbers involved in the comparison, the Comparison results are considered to be disordered:
		This result is different from greater than, less than, or equal to, so in floating-point comparison functions that determine equal to, greater than, greater than or equal to, less than, or less than or equal to, all of the Consider the comparison result a failure;
		Relatively, the function un- in determining whether it is unordered or not In ord, the results of the comparison are considered to be successful
4. 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 sign bit of the first operation. operative number (i.e. same number as above)		When two negative 0's are subtracted (negative 0 plus negative 0 and its equivalent) 价），则返回负 0，其余时候返回正 0
5. Floating-point division returns a 0 of the appropriate sign if ±0 is present in the divisor or divisor, and且 does not record division by 0. constant		a non-zero value is divided by zero, the appropriate sign of the no poor value

5.2.3 Examples of differences between libcc-rt and libgcc floating-point computation sections

This subsection uses C to illustrate the differences between libcc-rt and libgcc floating-point computation to help the reader understand better.

Variance 1.

Instead of returning an unspecified number for values that result in precision overflow, 0 is returned.

```
#include<stdio.h>
```

(Continued on next page)

```
/*
  |s|       |         t           |
  e
  0 00000001 00000000000000000000000000
*/
int minest_float_value= 0x00800000;


/*
  |s|     e    |      t   |
   0 01101111111 1111... .111
*/
long long double_minor_float_value=
0x37ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffff;


int main() {
  float res.
  float fa= *(float*)&minest_float_value;


  //Testing for underflow in floating-point
  arithmetic operations
  res= fa * 0.5;
  double da=*(double*)&double_minor_float_value;
  res=(float)da;
  printf("The result of arithmetic operation with
  floating-point underflow is: %f\n", res);
  printf("   The result in hexadecimal is: 0x%08x\n\n",
  *(int*)&res);
  // Test for underflow on floating point conversions
  printf("The result when underflow occurs for
  floating-point conversions: %f\n", res);
  printf("   The result in hexadecimal is: 0x%08x\n\n",
  *(int*)&res);
}
  return 0;
```

Arithmetic results are shown:

```
$qemu-riscv32 a.out
The result of an arithmetic
    operation with floating
    point underflow is:
    0.000000 The result in
    hexadecimal is: 0x00000000

The result of a floating-point
    conversion with underflow
    is: 0.000000 The result in
    hexadecimal is: 0x00000000
```

(Continued on next page)

```
$riscv64-unknown-elf-gcc underflow.c -march= rv32imac -mabi= ilp32
$qemu-riscv32 a.out
```

(continued from previous page)

```
The result of an arithmetic
    operation with floating
    point underflow is:
    0.000000 The result in
    hexadecimal is: 0x00400000

The result of a floating-point
    conversion with underflow
    is: 0.000000 The result in
    hexadecimal is: 0x00400000
```

Variance 2

For values that result in precision overflow, return the value of overflow that is not unpredictable

```
  double da =
  *(double*)&double_greater_float_value;
  res = fa + da;
  printf("The result of arithmetic operation with
  floating point overflow is: %f\n", res);
  //Test for overflow on floating point
  conversions
  printf("The result in hexadecimal is: 0x%08x\n\n",
  (int*)&res);
```

(Continued on next page)

(continued from previous page)

```
    printf("The result when an overflow occurs for a floating-point
    conversion is: %f\n", res);
    printf("    The result in hexadecimal is: 0x%08x\n\n", *(int*)&res);


    float fb= *(float*)&float_greater_int_value;
    int ires= fb;

    // Test for overflow when converting float to integer
    printf(" The result when floating point to
    integer overflow occurs is:%d\n", ires);
    printf("    The result in hexadecimal is:
    0x%08x\n\n", ires);

    return 0;
}
```

Arithmetic results are shown:

(Continued on next page)

```
$riscv64-unknown-elf-gcc overflow.c -march= rv32imac -mabi= ilp32 -mccrt
$qemu-riscv32 a.out
The result of the arithmetic operation floating point overflow is:
6805646932770577196234083669690338508 80.000000
    The result in hexadecimal is: 0x7fffffff

Floating-point conversion
    results in an overflow: -
    1.000000 The result in
    hexadecimal is: 0xbf800000

Floating point to integer overflow results in: 1024
     The result in hexadecimal is: 0x00000400

$riscv64-unknown-elf-gcc overflow.c -march= rv32imac -mabi= ilp32
$qemu-riscv32 a.out
The result of an arithmetic operation with floating-point overflow is: inf
    The result in hexadecimal is: 0x7f800000

The result when overflow occurs for floating-point conversions is: inf
    The result in hexadecimal is: 0x7f800000

Floating point to integer
     overflow results in:
     2147483647 results in
     hexadecimal: 0x7fffffff
```

Variance 3.

Treat non-numbers and infinities in the process and result of an operation as normalized numbers with corresponding indices.

```
#include<stdio.h>

/*
```

(Continued on next page)

(continued from previous page)

```
  |s|    e   |          t          |
   0 11111101 00000000000000000000000000
*/
int quarter_inf_float_value= 0x7e800000;

/*
  |s|    e   |          t          |
   0 11111111 00000000000000000000000000
*/
int inf_float_value= 0x7f800000;

/*
  |s|    e   |          t          |
   0 11111111 00000000000000000000000001
*/
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;


  //Testing non-numeric, infinitely participating operations in floating-point
  arithmetic.
  res= finf - fqinf;
  printf("Infinitely subtracting a large value results in: %f\n", res);
  printf("The result in hexadecimal is: 0x%08x\n\n", *(int*)&res);
  res= fnan / finf.
  printf("The result of the participation of non-numbers in the arithmetic
  operation is: %f\n", res);
  printf(" The result in hexadecimal is: 0x%08x\n\n", *(int*)&res);


  double dres.
  //Testing the handling of infinities and non-numbers when converting floating
  point to high precision
  dres= finf.
  printf("The result when converting low-precision infinity to high-precision is:
```

(Continued on next page)

```
  %f\n", dres);
  printf("      The result in hexadecimal is: 0x%08x\n\n", *(long long*)&dres);
  dres= fnan.
  printf("The result when converting low-precision non-numbers to high-
  precision is: %f\n", dres);
  printf("      The result in hexadecimal is: 0x%08x\n\n", *(long long*)&dres);
```

(Continued on next page)

(continued from previous page)

```
  // Test floating-point unordered comparisons
  char *cmp_res=  finf<  fnan ? "Floating-point positive infinity is
  smaller than floating-point positive non-numeric "
                :: "Floating-point positive infinity is not less
                   than a floating-point positive integer".
  printf(cmp_res);
  printf("\n").
  cmp_res=  finf>  fnan ? "Floating-point positive infinity over floating-
  point positive nonce "
                    :: "Floating-point positive infinity is not greater
                    than floating-point positive irrationality".
  printf(cmp_res);
  printf("\n\n").


  return 0;
}
```

Arithmetic results are shown:

(Continued on next page)

```
$riscv64-unknown-elf-gcc input_inf_or_nan.c -march= rv32imac -mabi= ilp32 -mccrt
$qemu-riscv32 a.out
Infinity minus one large value results in:
25521177519070384759753095557382615859 2.000000
The result in hexadecimal is: 0x7f400000

The result of the
  participation of a non-
  number in an arithmetic
  operation is: 1.000000 The
  result in hexadecimal is:
  0x3f800002

The result when low precision infinity is converted to high precision is:
340282366920938463463374607431768211456.000000
      The result in hexadecimal is: 0x0001d008

The result when converting low-precision non-numbers to high-precision is:
340282407485857576707667154553262707834 88.000000
      The result in hexadecimal is: 0x0001d008

Floating-point
positive infinity is
less than floating-
point positive non-
numeric Floating-
point positive
infinity is not greater
than floating-point
positive non-numeric

$riscv64-unknown-elf-gcc input_inf_or_nan.c -march= rv32imac -mabi= ilp32
$qemu-riscv32 a.out
Infinity minus a large value results in: inf
The result in hexadecimal is: 0x7f800000

The participation of a nonnumber in an arithmetic operation results in: nan
  The result in hexadecimal is: 0x7fc00000

The result when the low precision infinity is converted to high precision is:
inf
      The result in hexadecimal is: 0x0001e008
```

(Continued on next page)

(continued from previous page)

```
The result when converting low-precision non-numbers to high-precision is:
nan
       The result in hexadecimal is: 0x0001e008

Floating-point
positive infinity is
not less than
floating-point
positive non-numeric
floating-point
positive infinity is
not greater than
floating-point
positive non-numeric
```

Variance 4.

Floating point addition and subtraction operations, the operands are opposite and the result is 0, the sign bit of 0 is the same as that of the number being added (subtracted)

```
#include<stdio
.h>

/*                    t           |
  |s|100e000|0 00000000000000000000000000
*/
int float_p2_value= 0x40000000;

/*
  |s|    e   |           t          |
   1 10000000 00000000000000000000000000
*/
int float_n2_value= 0xc0000000;

int main() {
  float res.
  float p2= *(float*)&float_p2_value;
  float n2= *(float*)&float_n2_value;

  //Test for the sign of 0 when adding or subtracting 0.
  res= p2 - p2.
  printf("+2 - +2 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res); res=  n2 - n2;
  printf("-2 - -2 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res);
  res= p2+ n2.
  printf("+2 + -2 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res); res=  n2+  p2;
  printf("-2 +  +2 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res);

  printf("\n").
  return 0;
}
```

Arithmetic results are shown:

```
+2 - +2 = 0.000000.   The result in hexadecimal
                                  is: 0x00000000
-2 - -2 = -0.000000, The result in is: 0x80000000
+2 + -2 = 0.000000.  The result in is: 0x00000000
-2 + +2 = -0.000000, The result in is: 0x80000000
$qemu-riscv32 a.out
```

```
$riscv64-unknown-elf-gcc cancel_to_zero.c -march= rv32imac -
mabi=ilp32 -mccrt
```

```
+2 - +2 = 0.000000.   The result in hexadecimal
                                  is: 0x00000000
$riscv64-unknown-elf-gcc cancel_to_zero.c -march= rv32imac
-mabi= ilp32
+2 + -2 = 0.000000.   The result in hexadecimal
                                  is: 0x00000000
$qemu-riscv32 a.out
-2 + +2 = 0.000000.   The result in hexadecimal
                                  is: 0x00000000
```

Variance 5.

Floating-point division, if there is ±0 in the divisor or divisor, then return 0 with the appropriate sign,且 does not record a divide by 0 exception

```
  float res.
  float p0= *(float*)&float_p0_value;
  float p0= *(float*)&float_n0_value;

  // Test for division by 0
  res= 1.0 / p0;
  printf("+value / +0 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res); res=  -1.0 / n0;
  printf("-value / -0 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res);
  res= 1.0 / n0.
  printf("+value / -0 =  %f,\t The result in representation is:
  0x%08x\n", res, *(int*)&res); res=  -1.0 / p0;
```

(Continued on next page)

(continued from previous page)

```
  printf("-value / +0 =  %f,\t The result in representation is: 0x%08x\n",
  res, *(int*)&res);

  printf("\n").
  return 0;
}
```

Arithmetic results are shown:

```
$riscv64-unknown-elf-gcc input_with_zero.c -march= rv32imac -mabi= ilp32 -
mccrt
$qemu-riscv32 a.out
+value / +0=  0.000000, result in : 0x00000000
-value / -0=  0.000000, result in : 0x00000000
+value / -0=  -0.000000,     The result in hexadecimal is: 0x80000000
-value / +0=  -0.000000, -value / +0 -0.000000, -value / +0 -0.000000
The result in hexadecimal is: 0x80000000

$riscv64-unknown-elf-gcc input_with_zero.c -march= rv32imac -mabi= ilp32
$qemu-riscv32 a.out
+value / +0=  inf,    The result in hexadecimal is: 0x7f800000
-value / -0=  inf, -value / -0 inf, -value / -0 inf  The result in
hexadecimal is: 0x7f800000
+value / -0=  -inf,   The result in hexadecimal is: 0xff800000
-value / +0=  -inf, -value / +0 -inf, -value / +0   The result in
hexadecimal is: 0xff800000
```

5.3 pthread multithreading (currently only supported in the GNU toolchain)

The implementation of multithreading function is related to specific platform/OS. Various OSes support different multithreading implementations, such as Linux, rtems, vxWorks, zephyr, FreeRTOS, RT-Thread, and so on.

The underlying implementation of certain features of the compiler tool, such as the standard multithreading support for the C++11 language: std::thread, also calls the platform/OS-specific interface to the multithreading feature, which mainly involves the relevant interface in the pthread.h header file. Therefore, in order to ensure the correctness of the multi-threading functionality on the compiler tool, it is necessary to ensure the consistency of the size of the data structures on both sides of the compiler tool and the platform/OS.

```
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 section describes the C/C++ language extensions of the Xantei extension, such as function attributes, variable types, and intrinsic interfaces.

5.4.1 Nested Interrupt Function Properties

Nested interrupt functions are enabled by adding attribute ((interrupt("THead-interrupt-nesting"))) when declaring or defining a function, which is used to generate interrupt handler functions that support nested interrupts in machine mode. Compared to normal interrupt handler functions, it saves macuse, mepc and mstatus at the beginning of the function in addition to the general purpose registers and re-enables the interrupt; at the end of the function, it restores macuse, mepc and mstatus in addition to the general purpose registers and returns them with mret.

When the target CPU contains the xtheadint (included in xtheade, ipush/ipop instructions) instruction set, attribute ((interrupt("THead-interrupt- nesting"))) is the same as __ attribute__ ((interrupt))

or__ attribute__ ((interrupt("machine"))) have the same effect, and both of them will give priority to the ipush/ipop instruction; when the target CPU does not contain the xtheadint instruction set, take the E902 as an example. When the target CPU does not contain the xtheadint instruction set, as in the case of the E902, 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
lw      t0,16(sp)
csrw    mcause,t0
lw      t0,20(sp)
csrw    mepc,t0
lw      t0,24(sp)
lw      a4,8(sp)
lw      a5,4(sp)
addi
sp,sp,28 mret
```

5.4.2 __bf16 Data type

The bf16 datatype has been supported since the Gentei GNU compiler V2.8.0 and the Gentei LLVM compiler V1.0.0-beta. The bf16 datatype has now been incorporated by the community into the ABI documentation, see RISC-V Calling Conventions.

Under rv32 and rv64, it is 2bytes in length and alignment, following the arithmetic rules for regular floating-point types (e.g. float types).

5.5 Dynamic linker name

In versions of the Gentei GNU compiler prior to V2.8.0 and the Gentei LLVM compiler prior to V1.0.0-beta, in order to handle multilib scenarios more conveniently, the name of the dynamic linker would include thead when the target CPU contained the XThead extension instruction, which resulted in incompatibility between the community version of the program and the dynamic libraries, and our SDK. This practice would result in community versions of programs and dynamic libraries that are not compatible with our SDK. To solve this problem, from Gentei GNU Compiler V2.8.0 and Gentei LLVM Compiler V1.0.0-beta, the dynamic linker will not contain special characters such as xthead, which is in line with the community rules, as shown Table 5.7.

Table 5.7: Comparison of Dynamic Linker Names

	Before GCC V2.8.0 and LLVM V1.0.0-beta.	After GCC V2.8.0 and LLVM V1.0.0-beta (inclusive).
Include xtheadc in march	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

Remarks: where [xlen] is the bit-width of the target CPU, rv64 is 64, rv32 is 32; [abi] is the ABI specified by the -mabi option, such as lp64d, lp64, etc.

5.5.1 compatibility issue

Often, when the name of a dynamic linker changes, the method of simply generating a soft link pointing to the new dynamic linker with the name of the old dynamic linker name will result in an error when both the program and the dynamic libraries it relies on contain dynamic linker information with different名 names. We have solved this problem in the Dynamic Linker in Gentei GNU Compiler V2.8.0 and Gentei LLVM Compiler V1.0.0-beta, so that when you need to be compatible with programs or dynamic libraries compiled by an older version of the compiler, you can solve the problem by creating a soft link.

Taking the C920 as an example, its dynamic linker is /lib/ld-linux-riscv64-lp64d.so.1. First, use the ls -l command to see the actual file it points to. Assuming the file it points to is /lib64/lp64d/ld-2.33.so, use ln -s to create a new soft link with the following command:

```
ln -s /lib64/lp64d/ld-2.33.so /lib/ld-linux-riscv64v0p7_xthead-
lp64d.so.1
```

5.6 Generalized coprocessor extensions C intrinsic interface

This section describes the intrinsic interface of the Xentei general-purpose coprocessor extensions (including Xxtccei, Xxtccev, and Xxtccef), including general naming rules and C intrinsic
The full set of interfaces.

These interfaces are declared in the `riscv_xt_cce.h` header file.

Remarks: The coding domains of the Xtheadc extensions (Xxtccei, Xxtccev, Xxtccef) conflict with other xtheadc extensions except xtheadcmo (see Instruction Sets of Strings for information on the extensions), and cannot be used at the same time. If you are using a CPU that includes the Xentei general-purpose coprocessor extensions, the CPU does not include any xtheadc extensions other than xtheadcmo, and if you use them, you will experience unexpected behavioral errors.

5.6.1 naming convention

The Gentry Generalized Coprocessor Extension C intrinsic interface code is as follows:

```
__
riscv_xt_{INSTRUCTION_MNEMONIC}_{BASE_TYPE}_{ROUND_MODE}_{POLICY}{(
```

- INSTRUCTION_MNEMONIC is an instruction mnemonic as specified in the General Purpose Coprocessor Interface documentation, such as cpx0 and vcpx4.
- BASE_TYPE indicates the type of the operand. xxtccev, which uses only the EEW part since it contains only register-wide operations, is one of i8| i16| i32| i64| u8| u16| u32| u64| bf16| f16| f32| f64. xxtccef is of type bf16| f16 | One of f32| f64. Only the intrinsic of Xxtccev and Xxtccef contains this part.
- ROUND_MODE applies only to Xxtccev containing floating-point vector operands.且 explicitly specifies the intrinsic, i.e., rm, of the floating-point rounding mode.
- POLICY is null or mu for Xxtccev only.
  - No suffix: indicates a vector operation without mask (vm=1);
  - _mu suffix: indicates vector operation with mask (vm=0),且 is mask-undisturbed.

Similar to the RISC-V Standard Vector Extension C intrinsic interface, the Xxtccev intrinsic interface provides an implicit (overloaded) naming scheme that omits the BASE_TYPE, ROUND_MODE, and POLICY parts.

5.6.2 interface parameter

`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 Base本 set

```
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,    mask,          vs2)
vbool32_t                                     vuint32m1_t    .
void__ riscv_xt_vcpx0_u64(unsigned idx,      vs2).
vuint64m1_t
void__ riscv_xt_vcpx0_u64_mu(unsigned idx,    mask,          vs2).
vbool64_t                                     vuint64m1_t
void__ riscv_xt_vcpx0_f32(unsigned idx,       vs2).
vfloat32m1_t
void__ riscv_xt_vcpx0_f32_rm(unsigned idx,      vs2, unsigned frm);
vfloat32m1_t
void__ riscv_xt_vcpx0_f32_mu(unsigned idx,    mask,          vs2).
vbool32_t                                     vfloat32m1_t
void__ riscv_xt_vcpx0_f32_rm_mu(unsigned idx,   mask,          vs2.
vbool32_t                                       vfloat32m1_t
                            unsigned frm);
void__ riscv_xt_vcpx0_f64(unsigned idx,       vs2).
vfloat64m1_t
void__ riscv_xt_vcpx0_f64_rm(unsigned idx,      vs2, unsigned frm);
vfloat64m1_t
void__ riscv_xt_vcpx0_f64_mu(unsigned idx,    mask,          vs2).
vbool64_t                                     vfloat64m1_t
void__ riscv_xt_vcpx0_f64_rm_mu(unsigned idx,   mask,          vs2.
vbool64_t                                       vfloat64m1_t
                            unsigned frm);
vint8m1  __riscv_xt_vcpx1_i8(unsigned idx,     vs2).
_t       vint8m1_t
vint8m1  __riscv_xt_vcpx1_i8_mu(unsigned idx,   mask,          vd.
_t       vbool8_t                               vint8m1_t
                             vint8m1_t vs2).
vint16m1 __riscv_xt_vcpx1_i16(unsigned idx,     vs2).
_t       vint16m1_t
vint16m1 __riscv_xt_vcpx1_i16_mu(unsigned idx,    mask,          vd.
_t       vbool16_t                                vint16m1_t
                              vint16m1_t vs2).
vint32m1 __riscv_xt_vcpx1_i32(unsigned idx,     vs2).
_t       vint32m1_t
vint32m1 __riscv_xt_vcpx1_i32_mu(unsigned idx,    mask,          vd.
_t       vbool32_t                                vint32m1_t
                              vint32m1_t vs2).
vint64m1 __riscv_xt_vcpx1_i64(unsigned idx,     vs2).
_t       vint64m1_t
vint64m1 __riscv_xt_vcpx1_i64_mu(unsigned idx,    mask,          vd.
_t       vbool64_t                                vint64m1_t
                              vint64m1_t vs2).
vuint8m1    _t vuint8m1_t                                __              riscv_xt_vcpx1_u8(u
```

```
nsigned      vs2).
idx,          mask, vuint8m1_t                      vd.
vuint8m1_t

__
riscv_xt_v
cpx1_u8_mu
(unsigned
idx,
vbool8_t
                    v
                    u
                    i
                    n
                    t
                    8
                    m
                    1
                    _
                    t

                    v
                    s
                    2
                    )
                    .
  vuint16m1  __riscv_xt_vcpx1_u16(unsigned idx,       vs2).
  _t         vuint16m1_t
  vuint16m1  __riscv_xt_vcpx1_u16_mu(unsigned idx,    mask,
  _t         vbool16_t
                                vuint16m1_t vd, vuint16m1_t vs2).
  vuint32m1  __riscv_xt_vcpx1_u32(unsigned idx,       vs2).
  _t         vuint32m1_t
  vuint32m1  __riscv_xt_vcpx1_u32_mu(unsigned idx,    mask,
  _t         vbool32_t
                                vuint32m1_t vd, vuint32m1_t vs2).
  vuint64m1  __riscv_xt_vcpx1_u64(unsigned idx,       vs2).
  _t         vuint64m1_t
  vuint64m1  __riscv_xt_vcpx1_u64_mu(unsigned idx,    mask,
  _t         vbool64_t
                                vuint64m1_t vd, vuint64m1_t vs2).
  vfloat32m1 __riscv_xt_vcpx1_f32(unsigned idx,       vs2).
  _t          vfloat32m1_t
  vfloat32m1 __riscv_xt_vcpx1_f32_rm(unsigned idx,      vs2.
  _t          vfloat32m1_t
                                unsigned frm);
  vfloat32m1 __riscv_xt_vcpx1_f32_mu(unsigned idx,    mask,
  _t          vbool32_t
                                vfloat32m1_t vd, vfloat32m1_t vs2).
```

```
vfloat32m1_t __riscv_xt_vcpx1_f32_rm_mu(unsigned idx, vbool32_t mask,
```

(Continued on next page)

```
                                    vfloat32m1_t vd,
                                    vfloat32m1_t vs2,
                                    unsigned frm);
vfloat64m1_t __riscv_xt_vcpx1_f64(unsigned idx,       vs2).
vfloat64m1_t __riscv_xt_vcpx1_f64_rm(unsigned idx,       vs2.
                                 unsigned frm);
vfloat64m1_t __riscv_xt_vcpx1_f64_mu(unsigned idx,    mask,
                                 vfloat64m1_t vd, vfloat64m1_t vs2).
vfloat64m1_t __riscv_xt_vcpx1_f64_rm_mu(unsigned idx,    mask,
                                    vfloat64m1_t vd, vfloat64m1_t vs2,
                                    unsigned frm);
void__ riscv_xt_vcpx2_i8(unsigned idx,    vs2, unsigned imm5).
vint8m1_t
void__ riscv_xt_vcpx2_i8_mu(unsigned idx,   mask,         vs2.
vbool8_t                                    vint8m1_t
                         unsigned imm5).
void__ riscv_xt_vcpx2_i16(unsigned idx,    vs2, unsigned imm5).
vint16m1_t
void__ riscv_xt_vcpx2_i16_mu(unsigned idx,   mask,          vs2.
vbool16_t                                    vint16m1_t
                          unsigned imm5).
void__ riscv_xt_vcpx2_i32(unsigned idx,    vs2, unsigned imm5).
vint32m1_t
void__ riscv_xt_vcpx2_i32_mu(unsigned idx,   mask,          vs2.
vbool32_t                                    vint32m1_t
                          unsigned imm5).
void__ riscv_xt_vcpx2_i64(unsigned idx,    vs2, unsigned imm5).
vint64m1_t
void__ riscv_xt_vcpx2_i64_mu(unsigned idx,   mask,          vs2.
vbool64_t                                    vint64m1_t
                          unsigned imm5).
void__ riscv_xt_vcpx2_u8(unsigned idx,    vs2, unsigned imm5).
vuint8m1_t
void__ riscv_xt_vcpx2_u8_mu(unsigned idx,   mask,          vs2.
vbool8_t                                    vuint8m1_t
                         unsigned imm5).
void__ riscv_xt_vcpx2_u16(unsigned idx,     vs2, unsigned imm5).
vuint16m1_t
void__ riscv_xt_vcpx2_u16_mu(unsigned idx,                          u
vbool16_t                                                           n
```

```
                        ned imm5).                    mask,          vs2.
                                                      vuint16m1_t
void__ riscv_xt_vcpx2_u32(unsigned idx,      vs2, unsigned imm5).
vuint32m1_t
void__ riscv_xt_vcpx2_u32_mu(unsigned idx,    mask,          vs2.
vbool32_t                                     vuint32m1_t
                        unsigned imm5).
void__ riscv_xt_vcpx2_u64(unsigned idx,      vs2, unsigned imm5).
vuint64m1_t
void__ riscv_xt_vcpx2_u64_mu(unsigned idx,    mask,          vs2.
vbool64_t                                     vuint64m1_t
                        unsigned imm5).
void__ riscv_xt_vcpx2_f32(unsigned idx,       vs2, unsigned imm5).
vfloat32m1_t
void__ riscv_xt_vcpx2_f32_rm(unsigned idx,      vs2, unsigned imm5.
vfloat32m1_t
                        unsigned frm);
void__ riscv_xt_vcpx2_f32_mu(unsigned idx,    mask,           vs2.
vbool32_t                                     vfloat32m1_t
                        unsigned imm5).
void__ riscv_xt_vcpx2_f32_rm_mu(unsigned idx,   mask,           vs2.
vbool32_t                                       vfloat32m1_t
                         unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx2_f64(unsigned idx,       vs2, unsigned imm5).
vfloat64m1_t
void__ riscv_xt_vcpx2_f64_rm(unsigned idx,      vs2, unsigned imm5.
vfloat64m1_t
```

(Continued on next page)

(continued from previous page)

```
                        unsigned frm);
void__ riscv_xt_vcpx2_f64_mu(unsigned idx,    mask,           vs2
vbool64_t                                     vfloat64m1_t    .
                        unsigned imm5).
void__ riscv_xt_vcpx2_f64_rm_mu(unsigned idx,   mask,           vs2.
vbool64_t                                       vfloat64m1_t
                        unsigned imm5, unsigned frm);
vint8m1  __riscv_xt_vcpx3_i8(unsigned idx,     vs2, unsigned imm5).
_t       vint8m1_t
vint8m1  __riscv_xt_vcpx3_i8_mu(unsigned idx,    mask,         vd.
_t       vbool8_t                                vint8m1_t
                         vint8m1_t vs2, unsigned imm5).
vint16m1  __riscv_xt_vcpx3_i16(unsigned idx,      vs2, unsigned imm5).
_t        vint16m1_t
vint16m1  __riscv_xt_vcpx3_i16_mu(unsigned idx,     mask,          vd.
_t        vbool16_t                                 vint16m1_t
                            vint16m1_t vs2, unsigned imm5).
vint32m1  __riscv_xt_vcpx3_i32(unsigned idx,      vs2, unsigned imm5).
_t        vint32m1_t
vint32m1  __riscv_xt_vcpx3_i32_mu(unsigned idx,     mask,          vd.
_t        vbool32_t                                 vint32m1_t
                            vint32m1_t vs2, unsigned imm5).
vint64m1  __riscv_xt_vcpx3_i64(unsigned idx,      vs2, unsigned imm5).
_t        vint64m1_t
vint64m1  __riscv_xt_vcpx3_i64_mu(unsigned idx,     mask,          vd.
_t        vbool64_t                                 vint64m1_t
                            vint64m1_t vs2, unsigned imm5).
vuint8m1  __riscv_xt_vcpx3_u8(unsigned idx,      vs2, unsigned imm5).
_t        vuint8m1_t
vuint8m1  __riscv_xt_vcpx3_u8_mu(unsigned idx,    mask,          vd.
_t        vbool8_t                                vuint8m1_t
                           vuint8m1_t vs2, unsigned imm5).
vuint16m1  __riscv_xt_vcpx3_u16(unsigned idx,       vs2, unsigned imm5).
_t         vuint16m1_t
vuint16m1  __riscv_xt_vcpx3_u16_mu(unsigned idx,     mask,
_t         vbool16_t
                            vuint16m1_t vd, vuint16m1_t vs2,
                            unsigned imm5).
vuint32m1  __riscv_xt_vcpx3_u32(unsigned idx,       vs2, unsigned imm5).
_t         vuint32m1_t
vuint32m1  __riscv_xt_vcpx3_u32_mu(unsigned idx,     mask,
_t         vbool32_t
                            vuint32m1_t vd, vuint32m1_t vs2,
                            unsigned imm5).
```

```
vuint64m1_t __riscv_xt_vcpx3_u64(unsigned idx,       vs2, unsigned imm5).
vuint64m1_t
vuint64m1_t __riscv_xt_vcpx3_u64_mu(unsigned idx,    mask,
vbool64_t
                            vuint64m1_t vd, vuint64m1_t vs2,
                            unsigned imm5).
vfloat32m1_t __riscv_xt_vcpx3_f32(unsigned idx,       vs2.
vfloat32m1_t
                           unsigned imm5).
vfloat32m1_t __riscv_xt_vcpx3_f32_rm(unsigned idx,       vs2.
vfloat32m1_t
                            unsigned imm5, unsigned frm);
vfloat32m1_t __riscv_xt_vcpx3_f32_mu(unsigned idx,    mask,
vbool32_t
                            vfloat32m1_t vd, vfloat32m1_t vs2,
                            unsigned imm5).
vfloat32m1_t __riscv_xt_vcpx3_f32_rm_mu(unsigned idx,    mask,
vbool32_t
                               vfloat32m1_t vd, vfloat32m1_t vs2,
                               unsigned imm5, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx3_f64(unsigned idx,       vs2.
vfloat64m1_t
```

(Continued on next 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                    v
```

```
                      uint16m1_t vs1).      vs3,          vs2.
                                            vuint16m1_t
void__ riscv_xt_vcpx4_u16_mu(unsigned idx,   mask,          vs3.
vbool16_t                                   vuint16m1_t
                      vuint16m1_t vs2, vuint16m1_t vs1).
void__ riscv_xt_vcpx4_u32(unsigned idx,     vs3,          vs2.
vuint32m1_t                                 vuint32m1_t
                      vuint32m1_t vs1).
void__ riscv_xt_vcpx4_u32_mu(unsigned idx,   mask,          vs3.
vbool32_t                                   vuint32m1_t
                      vuint32m1_t vs2, vuint32m1_t vs1).
void__ riscv_xt_vcpx4_u64(unsigned idx,     vs3,          vs2.
vuint64m1_t                                 vuint64m1_t
                      vuint64m1_t vs1).
void__ riscv_xt_vcpx4_u64_mu(unsigned idx,   mask,          vs3.
vbool64_t                                   vuint64m1_t
                      vuint64m1_t vs2, vuint64m1_t vs1).
void__ riscv_xt_vcpx4_f32(unsigned idx,     vs3,          vs2.
vfloat32m1_t                                vfloat32m1_t
                      vfloat32m1_t vs1).
```

(Continued on next page)

(continued from previous page)

```
void__ riscv_xt_vcpx4_f32_rm(unsigned idx,      vs3,           vs2
vfloat32m1_t                                    vfloat32m1_t    .
                        vfloat32m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4_f32_mu(unsigned idx,   mask,          vs3.
vbool32_t                                       vfloat32m1_t
                      vfloat32m1_t vs2, vfloat32m1_t vs1).
void__ riscv_xt_vcpx4_f32_rm_mu(unsigned idx,   mask,           vs3.
vbool32_t                                       vfloat32m1_t
                          vfloat32m1_t vs2, vfloat32m1_t vs1,
                          unsigned frm);
void__ riscv_xt_vcpx4_f64(unsigned idx,      vs3,          vs2.
vfloat64m1_t                                    vfloat64m1_t
                      vfloat64m1_t vs1).
void__ riscv_xt_vcpx4_f64_rm(unsigned idx,      vs3,           vs2.
vfloat64m1_t                                    vfloat64m1_t
                        vfloat64m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4_f64_mu(unsigned idx,   mask,          vs3.
vbool64_t                                       vfloat64m1_t
                      vfloat64m1_t vs2, vfloat64m1_t vs1).
void__ riscv_xt_vcpx4_f64_rm_mu(unsigned idx,   mask,           vs3.
vbool64_t                                       vfloat64m1_t
                          vfloat64m1_t vs2, vfloat64m1_t vs1,
                          unsigned frm);
vint8m1  __riscv_xt_vcpx5_i8(unsigned idx,     vs2,         vs1).
_t       vint8m1_t                              vint8m1_t
vint8m1  __riscv_xt_vcpx5_i8_mu(unsigned idx,   mask,        vd.
_t       vbool8_t                               vint8m1_t
                       vint8m1_t vs2, vint8m1_t vs1).
vint16m1 __riscv_xt_vcpx5_i16(unsigned idx,     vs2,         vs1).
_t       vint16m1_t                             vint16m1_t
vint16m1 __riscv_xt_vcpx5_i16_mu(unsigned idx,    mask,        vd.
_t       vbool16_t                              vint16m1_t
                          vint16m1_t vs2, vint16m1_t vs1).
vint32m1 __riscv_xt_vcpx5_i32(unsigned idx,     vs2,         vs1).
_t       vint32m1_t                             vint32m1_t
vint32m1 __riscv_xt_vcpx5_i32_mu(unsigned idx,    mask,        vd.
_t       vbool32_t                              vint32m1_t
                          vint32m1_t vs2, vint32m1_t vs1).
vint64m1 __riscv_xt_vcpx5_i64(unsigned idx,     vs2,         vs1).
_t       vint64m1_t                             vint64m1_t
vint64m1 __riscv_xt_vcpx5_i64_mu(unsigned idx,    mask,        vd.
_t       vbool64_t                              vint64m1_t
                          vint64m1_t vs2, vint64m1_t vs1).
vuint8m1    _t                                  __          riscv_xt_vcpx5_u8(uns
```

```
igned idx,  vs2, vuint8m1_t                          vs1).
vuint8m1_t
  vuint8m1  __riscv_xt_vcpx5_u8_mu(unsigned idx,    mask,          vd.
  _t        vbool8_t                                 vuint8m1_t
                              vuint8m1_t vs2, vuint8m1_t vs1).
  vuint16m1 __riscv_xt_vcpx5_u16(unsigned idx,       vs2.
  _t        vuint16m1_t
                             vuint16m1_t vs1).
  vuint16m1 __riscv_xt_vcpx5_u16_mu(unsigned idx,    mask,
  _t        vbool16_t
                               vuint16m1_t vd, vuint16m1_t
                               vs2, vuint16m1_t vs1);
  vuint32m1 __riscv_xt_vcpx5_u32(unsigned idx,       vs2.
  _t        vuint32m1_t
                             vuint32m1_t vs1).
  vuint32m1 __riscv_xt_vcpx5_u32_mu(unsigned idx,    mask,
  _t        vbool32_t
                               vuint32m1_t vd, vuint32m1_t
                               vs2, vuint32m1_t vs1);
  vuint64m1 __riscv_xt_vcpx5_u64(unsigned idx,       vs2.
  _t        vuint64m1_t
                             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,    mask,          vs3.
vbool16_t                                     vint16m1_t
                         vint16m1_t vs2, int16_t rs1).
void__ riscv_xt_vcpx6_i32(unsigned idx,     vs3,          vs2.
vint32m1_t                                    vint32m1_t
                       int32_t rs1).
void__ riscv_xt_vcpx6_i32_mu(unsigned idx,    mask,          vs3.
vbool32_t                                     vint32m1_t
                         vint32m1_t vs2, int32_t rs1).
void__ riscv_xt_vcpx6_u8(unsigned idx,     vs3,          vs2.
vuint8m1_t                                    vuint8m1_t
                     uint8_t rs1).
void__ riscv_xt_vcpx6_u8_mu(unsigned idx,   mask,          vs3.
vbool8_t                                      vuint8m1_t
                        vuint8m1_t vs2, uint8_t rs1).
void__ riscv_xt_vcpx6_u16(unsigned idx,      vs3,          vs2.
vuint16m1_t                                   vuint16m1_t
                       uint16_t rs1).
void__ riscv_xt_vcpx6_u16_mu(unsigned idx,    mask,          vs3.
vbool16_t                                     vuint16m1_t
                         vuint16m1_t vs2, uint16_t rs1).
```

(Continued on next page)

(continued from previous page)

```
void__ riscv_xt_vcpx6_u32(unsigned idx,      vs3,          vs2
vuint32m1_t                                  vuint32m1_t    .
                      uint32_t rs1).
void__ riscv_xt_vcpx6_u32_mu(unsigned idx,    mask,          vs3.
vbool32_t                                    vuint32m1_t
                        vuint32m1_t vs2, uint32_t rs1).
vint8m1  __riscv_xt_vcpx7_i8(unsigned idx,     vs2,       rs1).
_t       vint8m1_t                           int8_t
vint8m1  __riscv_xt_vcpx7_i8_mu(unsigned idx,    mask,         vd.
_t       vbool8_t                            vint8m1_t
                        vint8m1_t vs2, int8_t rs1).
vint16m1 __riscv_xt_vcpx7_i16(unsigned idx,      vs2,       rs1).
_t       vint16m1_t                          int16_t
vint16m1 __riscv_xt_vcpx7_i16_mu(unsigned idx,    mask,          vd.
_t       vbool16_t                           vint16m1_t
                       vint16m1_t vs2, int16_t rs1).
vint32m1 __riscv_xt_vcpx7_i32(unsigned idx,      vs2,       rs1).
_t       vint32m1_t                          int32_t
vint32m1 __riscv_xt_vcpx7_i32_mu(unsigned idx,    mask,          vd.
_t       vbool32_t                           vint32m1_t
                       vint32m1_t vs2, int32_t rs1).
vuint8m1 __riscv_xt_vcpx7_u8(unsigned idx,      vs2,       rs1).
_t       vuint8m1_t                          uint8_t
vuint8m1 __riscv_xt_vcpx7_u8_mu(unsigned idx,    mask,          vd.
_t       vbool8_t                            vuint8m1_t
                        vuint8m1_t vs2, uint8_t rs1).
vuint16m1 __riscv_xt_vcpx7_u16(unsigned idx,      vs2,       rs1).
_t        vuint16m1_t                         uint16_t
vuint16m1 __riscv_xt_vcpx7_u16_mu(unsigned idx,    mask,
_t        vbool16_t
                        vuint16m1_t vd, vuint16m1_t vs2,
                        uint16_t rs1).
vuint32m1 __riscv_xt_vcpx7_u32(unsigned idx,      vs2,       rs1).
_t        vuint32m1_t                         uint32_t
vuint32m1 __riscv_xt_vcpx7_u32_mu(unsigned idx,    mask,
_t        vbool32_t
                        vuint32m1_t vd, vuint32m1_t vs2,
                        uint32_t rs1).
void__ riscv_xt_vcpx8_i8(unsigned idx,    vs3,         vs2.
vint8m1_t                                 vint8m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_i8_mu(unsigned idx,   mask,         vs3.
vbool8_t                                  vint8m1_t
                       vint8m1_t vs2, unsigned imm5).
```

```
void__ riscv_xt_vcpx8_i16(unsigned idx,     vs3,          vs2.
vint16m1_t                                  vint16m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_i16_mu(unsigned idx,    mask,          vs3.
vbool16_t                                    vint16m1_t
                         vint16m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_i32(unsigned idx,     vs3,          vs2.
vint32m1_t                                  vint32m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_i32_mu(unsigned idx,    mask,          vs3.
vbool32_t                                    vint32m1_t
                         vint32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_i64(unsigned idx,     vs3,          vs2.
vint64m1_t                                  vint64m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_i64_mu(unsigned idx,    mask,          vs3.
vbool64_t                                    vint64m1_t
                         vint64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_u8(unsigned idx,     vs3,          vs2.
vuint8m1_t                                  vuint8m1_t
                     unsigned imm5).
void__ riscv_xt_vcpx8_u8_mu(unsigned idx,   mask,          vs3.
vbool8_t                                     vuint8m1_t
```

(Continued on next page)

```
                        vuint8m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_u16(unsigned idx,      vs3,          vs2.
vuint16m1_t                                  vuint16m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_u16_mu(unsigned idx,    mask,          vs3.
vbool16_t                                    vuint16m1_t
                        vuint16m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_u32(unsigned idx,      vs3,          vs2.
vuint32m1_t                                  vuint32m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_u32_mu(unsigned idx,    mask,          vs3.
vbool32_t                                    vuint32m1_t
                        vuint32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_u64(unsigned idx,      vs3,          vs2.
vuint64m1_t                                  vuint64m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_u64_mu(unsigned idx,    mask,          vs3.
vbool64_t                                    vuint64m1_t
                        vuint64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_f32(unsigned idx,      vs3,          vs2.
vfloat32m1_t                                 vfloat32m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_f32_rm(unsigned idx,      vs3,          vs2.
vfloat32m1_t                                    vfloat32m1_t
                        unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8_f32_mu(unsigned idx,    mask,          vs3.
vbool32_t                                    vfloat32m1_t
                      vfloat32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_f32_rm_mu(unsigned idx,   mask,          vs3.
vbool32_t                                       vfloat32m1_t
                          vfloat32m1_t vs2, unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8_f64(unsigned idx,      vs3,          vs2.
vfloat64m1_t                                 vfloat64m1_t
                      unsigned imm5).
void__ riscv_xt_vcpx8_f64_rm(unsigned idx,      vs3,          vs2.
vfloat64m1_t                                    vfloat64m1_t
                        unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8_f64_mu(unsigned idx,    mask,          vs3.
vbool64_t                                    vfloat64m1_t
                      vfloat64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8_f64_rm_mu(unsigned idx,   mask,          vs3.
vbool64_t                                       vfloat64m1_t
                          vfloat64m1_t vs2, unsigned imm5, unsigned frm);
```

```
void__ riscv_xt_vcpx9_f32(unsigned idx,       vs3,           vs2.
vfloat32m1_t                                  vfloat32m1_t
                      float fs1).
void__ riscv_xt_vcpx9_f32_rm(unsigned idx,      vs3,           vs2.
vfloat32m1_t                                    vfloat32m1_t
                        float fs1, unsigned
                        frm);
void__ riscv_xt_vcpx9_f32_mu(unsigned idx,    mask,           vs3.
vbool32_t                                     vfloat32m1_t
                        vfloat32m1_t vs2, float fs1).
void__ riscv_xt_vcpx9_f32_rm_mu(unsigned idx,   mask,           vs3.
vbool32_t                                       vfloat32m1_t
                          vfloat32m1_t vs2, float fs1, unsigned frm);
void__ riscv_xt_vcpx9_f64(unsigned idx,       vs3,           vs2.
vfloat64m1_t                                  vfloat64m1_t
                      double fs1).
void__ riscv_xt_vcpx9_f64_rm(unsigned idx,      vs3,           vs2.
vfloat64m1_t                                    vfloat64m1_t
                        double fs1, unsigned
                        frm);
void__ riscv_xt_vcpx9_f64_mu(unsigned idx,    mask,           vs3.
vbool64_t                                     vfloat64m1_t
                        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 Part RV64

The following intrinsic interfaces are only available on the RV64 platform.

```
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 only available when the Zvfh or Zvfhmin extension is supported.

```
void__ riscv_xt_vcpx0_f16(unsigned idx,       vs2).
vfloat16m1_t
void__ riscv_xt_vcpx0_f16_rm(unsigned idx,      vs2, unsigned frm);
vfloat16m1_t
void__ riscv_xt_vcpx0_f16_mu(unsigned idx,    mask,           vs2).
vbool16_t                                     vfloat16m1_t
void__ riscv_xt_vcpx0_f16_rm_mu(unsigned idx,   mask,           vs2.
vbool16_t                                       vfloat16m1_t
                            unsigned frm);
vfloat16m1 __riscv_xt_vcpx1_f16(unsigned idx,       vs2).
_t         vfloat16m1_t
vfloat16m1 __riscv_xt_vcpx1_f16_rm(unsigned idx,       vs2.
_t         vfloat16m1_t
                                unsigned frm);
vfloat16m1 __riscv_xt_vcpx1_f16_mu(unsigned idx,    mask,
_t         vbool16_t
                                vfloat16m1_t vd, vfloat16m1_t vs2).
vfloat16m1 __riscv_xt_vcpx1_f16_rm_mu(unsigned idx,    mask,
_t         vbool16_t
                                   vfloat16m1_t vd, vfloat16m1_t vs2,
                                   unsigned frm);
void__ riscv_xt_vcpx2_f16(unsigned idx,       vs2, unsigned imm5).
vfloat16m1_t
void__ riscv_xt_vcpx2_f16_rm(unsigned idx,      vs2, unsigned imm5.
vfloat16m1_t
                           unsigned frm);
void__ riscv_xt_vcpx2_f16_mu(unsigned idx,    mask,           vs2.
vbool16_t                                     vfloat16m1_t
                           unsigned imm5).
void__ riscv_xt_vcpx2_f16_rm_mu(unsigned idx,   mask,           vs2.
vbool16_t                                       vfloat16m1_t
                            unsigned imm5, unsigned frm);
vfloat16m1 __riscv_xt_vcpx3_f16(unsigned idx,       vs2.
_t         vfloat16m1_t
                              unsigned imm5).
vfloat16m1 __riscv_xt_vcpx3_f16_rm(unsigned idx,       vs2.
_t         vfloat16m1_t
                                unsigned imm5, unsigned frm);
vfloat16m1 __riscv_xt_vcpx3_f16_mu(unsigned idx,    mask,
_t         vbool16_t
                                vfloat16m1_t vd, vfloat16m1_t vs2,
```

```
                                unsigned imm5).
vfloat16m1 __riscv_xt_vcpx3_f16_rm_mu(unsigned idx,    mask,
_t         vbool16_t
                                   vfloat16m1_t vd, vfloat16m1_t vs2,
                                   unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx4_f16(unsigned idx,      vs3,           vs2.
vfloat16m1_t                                 vfloat16m1_t
                      vfloat16m1_t vs1).
void__ riscv_xt_vcpx4_f16_rm(unsigned idx,      vs3,           vs2.
vfloat16m1_t                                   vfloat16m1_t
                        vfloat16m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4_f16_mu(unsigned idx,    mask,          vs3.
vbool16_t                                    vfloat16m1_t
                      vfloat16m1_t vs2, vfloat16m1_t vs1).
void__ riscv_xt_vcpx4_f16_rm_mu(unsigned idx,   mask,          vs3.
vbool16_t                                      vfloat16m1_t
                           vfloat16m1_t vs2, vfloat16m1_t vs1,
                           unsigned frm);
vfloat16m1 __riscv_xt_vcpx5_f16(unsigned idx,       vs2
_t         vfloat16m1_t                             .
                             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 __riscv_xt_vcpx10_f16_rm_mu(unsigned idx,    mask,
  _t         vbool16_t
                                      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)

```
                                vbfloat16m1_t vs2, unsigned frm).
vbfloat16m1 __riscv_xt_vcpx1_bf16(unsigned idx,        vs2).
_t          vbfloat16m1_t
vbfloat16m1 __riscv_xt_vcpx1_bf16_rm(unsigned idx,       vs2.
_t          vbfloat16m1_t
                                unsigned frm);
vbfloat16m1 __riscv_xt_vcpx1_bf16_mu(unsigned idx,    mask,
_t          vbool16_t
                                vbfloat16m1_t vd, vbfloat16m1_t vs2).
vbfloat16m1 __riscv_xt_vcpx1_bf16_rm_mu(unsigned idx,    mask,
_t          vbool16_t
                                   vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                   unsigned frm);
void__ riscv_xt_vcpx2_bf16(unsigned idx,       vs2, unsigned imm5).
vbfloat16m1_t
void__ riscv_xt_vcpx2_bf16_rm(unsigned idx,       vs2, unsigned imm5.
vbfloat16m1_t
                           unsigned frm);
void__ riscv_xt_vcpx2_bf16_mu(unsigned idx,    mask,            vs2.
vbool16_t                                  vbfloat16m1_t
                           unsigned imm5).
void__ riscv_xt_vcpx2_bf16_rm_mu(unsigned idx,   mask,
vbool16_t
                           vbfloat16m1_t vs2, unsigned imm5,
                           unsigned frm);
vbfloat16m1 __riscv_xt_vcpx3_bf16(unsigned idx,        vs2.
_t          vbfloat16m1_t
                              unsigned imm5).
vbfloat16m1 __riscv_xt_vcpx3_bf16_rm(unsigned idx,       vs2.
_t          vbfloat16m1_t
                                unsigned imm5, unsigned frm);
vbfloat16m1 __riscv_xt_vcpx3_bf16_mu(unsigned idx,    mask,
_t          vbool16_t
                                vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                unsigned imm5).
vbfloat16m1 __riscv_xt_vcpx3_bf16_rm_mu(unsigned idx,    mask,
_t          vbool16_t
                                   vbfloat16m1_t vd, vbfloat16m1_t vs2,
                                   unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx4_bf16(unsigned idx,       vs3,            vs2.
vbfloat16m1_t                              vbfloat16m1_t
                        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,    mask,            vs3.
vbool16_t                                      vbfloat16m1_t
                          vbfloat16m1_t vs2, vbfloat16m1_t vs1).
void__ riscv_xt_vcpx4_bf16_rm_mu(unsigned idx,   mask,
vbool16_t
                            vbfloat16m1_t vs3, vbfloat16m1_t
                            vs2, vbfloat16m1_t vs1, unsigned
                            frm);
vbfloat16m1 __riscv_xt_vcpx5_bf16(unsigned idx,        vs2.
_t          vbfloat16m1_t
                              vbfloat16m1_t vs1).
vbfloat16m1 __riscv_xt_vcpx5_bf16_rm(unsigned idx,       vs2.
_t          vbfloat16m1_t
                                vbfloat16m1_t vs1, unsigned frm).
vbfloat16m1 __riscv_xt_vcpx5_bf16_mu(unsigned idx,    mask,
_t          vbool16_t
                                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 Base本 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)

```
void__ riscv_xt_vcpx0(unsigned idx,     mask,          vs2)
vbool32_t                               vint32m1_t     .
void__ riscv_xt_vcpx0(unsigned idx,      vs2).
vint64m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,          vs2).
vbool64_t                               vint64m1_t
void__ riscv_xt_vcpx0(unsigned idx,      vs2).
vuint8m1_t
void__ riscv_xt_vcpx0(unsigned idx,    mask,          vs2).
vbool8_t                                vuint8m1_t
void__ riscv_xt_vcpx0(unsigned idx,      vs2).
vuint16m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2).
vbool16_t                               vuint16m1_t
void__ riscv_xt_vcpx0(unsigned idx,      vs2).
vuint32m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2).
vbool32_t                               vuint32m1_t
void__ riscv_xt_vcpx0(unsigned idx,      vs2).
vuint64m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2).
vbool64_t                               vuint64m1_t
void__ riscv_xt_vcpx0(unsigned idx,       vs2).
vfloat32m1_t                            vs2, unsigned frm);
void__ riscv_xt_vcpx0(unsigned idx,
vfloat32m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2)
vbool32_t                               vfloat32m1_t    ;
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2.
vbool32_t                               vfloat32m1_t
                  unsigned frm);
void__ riscv_xt_vcpx0(unsigned idx,       vs2).
vfloat64m1_t                            vs2, unsigned frm);
void__ riscv_xt_vcpx0(unsigned idx,
vfloat64m1_t
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2)
vbool64_t                               vfloat64m1_t    ;
void__ riscv_xt_vcpx0(unsigned idx,     mask,           vs2.
vbool64_t                               vfloat64m1_t
                  unsigned frm);
vint8m1  __riscv_xt_vcpx1(unsigned idx,     vs2).
_t       vint8m1_t                         mask,         vd.
vint8m1  __riscv_xt_vcpx1(unsigned idx,    vint8m1_t
_t       vbool8_t
                     vint8m1_t vs2).
```

```
vint16m1_t __riscv_xt_vcpx1(unsigned idx,      vs2).
           vint16m1_t
vint16m1_t __riscv_xt_vcpx1(unsigned idx,     mask,          vd.
           vbool16_t                          vint16m1_t
                        vint16m1_t vs2).
vint32m1_t __riscv_xt_vcpx1(unsigned idx,      vs2).
           vint32m1_t
vint32m1_t __riscv_xt_vcpx1(unsigned idx,     mask,          vd.
           vbool32_t                          vint32m1_t
                        vint32m1_t vs2).
vint64m1_t __riscv_xt_vcpx1(unsigned idx,      vs2).
           vint64m1_t
vint64m1_t __riscv_xt_vcpx1(unsigned idx,     mask,          vd.
           vbool64_t                          vint64m1_t
                        vint64m1_t vs2).
vuint8m1_t __riscv_xt_vcpx1(unsigned idx,      vs2).
           vuint8m1_t
vuint8m1_t __riscv_xt_vcpx1(unsigned idx,    mask,          vd.
           vbool8_t                           vuint8m1_t
                        vuint8m1_t vs2).
vuint16m1_t __riscv_xt_vcpx1(unsigned idx,       vs2).
            vuint16m1_t
vuint16m1_t __riscv_xt_vcpx1(unsigned idx,     mask,           vd.
            vbool16_t                          vuint16m1_t
                         vuint16m1_t vs2).
vuint32m1_t __riscv_xt_vcpx1(unsigned idx,       vs2).
            vuint32m1_t
vuint32m1_t __riscv_xt_vcpx1(unsigned idx,     mask,           vd.
            vbool32_t                          vuint32m1_t
                         vuint32m1_t vs2).
vuint64m1_t __riscv_xt_vcpx1(unsigned idx,       vs2).
            vuint64m1_t
```

(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,      vs2, unsigned imm5).
vuint8m1_t
void__ riscv_xt_vcpx2(unsigned idx,    mask,          vs2.
vbool8_t                               vuint8m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx2(unsigned idx,      vs2, unsigned imm5).
vuint16m1_t
void__ riscv_xt_vcpx2(unsigned idx,     mask,          vs2.
vbool16_t                              vuint16m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx2(unsigned idx,      vs2, unsigned imm5).
vuint32m1_t
void__ riscv_xt_vcpx2(unsigned idx,     mask,          vs2.
vbool32_t                              vuint32m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx2(unsigned idx,      vs2, unsigned imm5).
vuint64m1_t
void__ riscv_xt_vcpx2(unsigned idx,     mask,          vs2.
vbool64_t                              vuint64m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx2(unsigned idx,       vs2, unsigned imm5);
vfloat32m1_t                           vs2, unsigned imm5,
void__ riscv_xt_vcpx2(unsigned idx,
vfloat32m1_t
                    unsigned frm);
void__ riscv_xt_vcpx2(unsigned idx,     mask,          vs2.
vbool32_t                              vfloat32m1_t
                    unsigned imm5).
```

(Continued on next page)

(continued from previous page)

```
void__ riscv_xt_vcpx2(unsigned idx,     mask,           vs2
vbool32_t                               vfloat32m1_t    .
                  unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx2(unsigned idx,       vs2, unsigned imm5);
vfloat64m1_t                              vs2, unsigned imm5,
void__ riscv_xt_vcpx2(unsigned idx,
vfloat64m1_t
                  unsigned frm);
void__ riscv_xt_vcpx2(unsigned idx,     mask,           vs2.
vbool64_t                               vfloat64m1_t
                  unsigned imm5).                       vs2.
void__ riscv_xt_vcpx2(unsigned idx,     mask,
vbool64_t                               vfloat64m1_t
                  unsigned imm5, unsigned frm);
vint8m1  __riscv_xt_vcpx3(unsigned idx,     vs2, unsigned imm5).
_t       vint8m1_t
vint8m1  __riscv_xt_vcpx3(unsigned idx,     mask,         vd.
_t       vbool8_t                           vint8m1_t
                      vint8m1_t vs2, unsigned imm5).
vint16m1 __riscv_xt_vcpx3(unsigned idx,      vs2, unsigned imm5).
_t       vint16m1_t
vint16m1 __riscv_xt_vcpx3(unsigned idx,     mask,         vd.
_t       vbool16_t                          vint16m1_t
                      vint16m1_t vs2, unsigned imm5).
vint32m1 __riscv_xt_vcpx3(unsigned idx,      vs2, unsigned imm5).
_t       vint32m1_t
vint32m1 __riscv_xt_vcpx3(unsigned idx,     mask,         vd.
_t       vbool32_t                          vint32m1_t
                      vint32m1_t vs2, unsigned imm5).
vint64m1 __riscv_xt_vcpx3(unsigned idx,      vs2, unsigned imm5).
_t       vint64m1_t
vint64m1 __riscv_xt_vcpx3(unsigned idx,     mask,         vd.
_t       vbool64_t                          vint64m1_t
                      vint64m1_t vs2, unsigned imm5).
vuint8m1 __riscv_xt_vcpx3(unsigned idx,      vs2, unsigned imm5).
_t       vuint8m1_t
vuint8m1 __riscv_xt_vcpx3(unsigned idx,     mask,         vd.
_t       vbool8_t                           vuint8m1_t
                      vuint8m1_t vs2, unsigned imm5).
vuint16m1 __riscv_xt_vcpx3(unsigned idx,       vs2, unsigned imm5).
_t        vuint16m1_t
vuint16m1 __riscv_xt_vcpx3(unsigned idx,     mask,           vd.
_t        vbool16_t                          vuint16m1_t
                      vuint16m1_t vs2, unsigned imm5).
```

```
vuint32m1_t __riscv_xt_vcpx3(unsigned idx,       vs2, unsigned imm5).
vuint32m1_t
vuint32m1_t __riscv_xt_vcpx3(unsigned idx,     mask,          vd.
vbool32_t                                      vuint32m1_t
                        vuint32m1_t vs2, unsigned imm5).
vuint64m1_t __riscv_xt_vcpx3(unsigned idx,       vs2, unsigned imm5).
vuint64m1_t
vuint64m1_t __riscv_xt_vcpx3(unsigned idx,     mask,          vd.
vbool64_t                                      vuint64m1_t
                        vuint64m1_t vs2, unsigned imm5).
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx,        vs2, unsigned imm5);
vfloat32m1_t                                       vs2, unsigned imm5,
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx,
vfloat32m1_t
                        unsigned frm);
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx,     mask,           vd.
vbool32_t                                      vfloat32m1_t
                      vfloat32m1_t vs2, unsigned imm5).
vfloat32m1_t __riscv_xt_vcpx3(unsigned idx,     mask,           vd.
vbool32_t                                      vfloat32m1_t
                        vfloat32m1_t vs2, unsigned imm5, unsigned frm);
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx,        vs2, unsigned imm5);
vfloat64m1_t                                       vs2, unsigned imm5,
vfloat64m1_t __riscv_xt_vcpx3(unsigned idx,
vfloat64m1_t
                        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).
```

```
                  vs1).                                vs3,          vs2.
                                                       vuint32m1_t
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool32_t                               vuint32m1_t
                  vuint32m1_t vs2, vuint32m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,       vs3,          vs2.
vuint64m1_t                             vuint64m1_t
                  vuint64m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool64_t                               vuint64m1_t
                  vuint64m1_t vs2, vuint64m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,       vs3,           vs2.
vfloat32m1_t                            vfloat32m1_t
                  vfloat32m1_t vs1).                              vs2.
void__ riscv_xt_vcpx4(unsigned idx,       vs3,
vfloat32m1_t                            vfloat32m1_t
                  vfloat32m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool32_t                               vfloat32m1_t
                  vfloat32m1_t vs2, vfloat32m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool32_t                               vfloat32m1_t
```

(Continued on next page)

```
                vfloat32m1_t vs2, vfloat32m1_t vs1, unsigned frm);
void__ riscv_xt_vcpx4(unsigned idx,       vs3,           vs2.
vfloat64m1_t                              vfloat64m1_t
                    vfloat64m1_t vs1).                   vs2.
void__ riscv_xt_vcpx4(unsigned idx,       vs3,
vfloat64m1_t                              vfloat64m1_t
                    vfloat64m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool64_t                                 vfloat64m1_t
                    vfloat64m1_t vs2, vfloat64m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,     mask,          vs3.
vbool64_t                                 vfloat64m1_t
                vfloat64m1_t vs2, vfloat64m1_t vs1, unsigned frm);
vint8m1  __riscv_xt_vcpx5(unsigned idx,     vs2,         vs1)
_t       vint8m1_t                          vint8m1_t    ;
vint8m1  __riscv_xt_vcpx5(unsigned idx,     mask,        vd.
         vbool8_t
_t                                          vint8m1_t
                      vint8m1_t vs2, vint8m1_t vs1).
vint16m1 __riscv_xt_vcpx5(unsigned idx,      vs2,          vs1).
_t       vint16m1_t                         vint16m1_t
vint16m1 __riscv_xt_vcpx5(unsigned idx,     mask,         vd.
_t       vbool16_t                          vint16m1_t
                     vint16m1_t vs2, vint16m1_t vs1).
vint32m1 __riscv_xt_vcpx5(unsigned idx,      vs2,          vs1).
_t       vint32m1_t                         vint32m1_t
vint32m1 __riscv_xt_vcpx5(unsigned idx,     mask,         vd.
_t       vbool32_t                          vint32m1_t
                     vint32m1_t vs2, vint32m1_t vs1).
vint64m1 __riscv_xt_vcpx5(unsigned idx,      vs2,          vs1).
_t       vint64m1_t                         vint64m1_t
vint64m1 __riscv_xt_vcpx5(unsigned idx,     mask,         vd.
_t       vbool64_t                          vint64m1_t
                     vint64m1_t vs2, vint64m1_t vs1).
vuint8m1 __riscv_xt_vcpx5(unsigned idx,      vs2,          vs1).
_t       vuint8m1_t                         vuint8m1_t
vuint8m1 __riscv_xt_vcpx5(unsigned idx,    mask,          vd.
_t       vbool8_t                           vuint8m1_t
                     vuint8m1_t vs2, vuint8m1_t vs1).
vuint16m1 __riscv_xt_vcpx5(unsigned idx,       vs2,           vs1).
_t        vuint16m1_t                          vuint16m1_t
vuint16m1 __riscv_xt_vcpx5(unsigned idx,     mask,            vd.
_t        vbool16_t                          vuint16m1_t
                    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 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 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 vfloat32m1_t unsigned frm);
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vfloat32m1_t vfloat32m1_t vs2, vfloat32m1_t vs1).
vfloat32m1_t __riscv_xt_vcpx5(unsigned idx, vbool32_t mask, vfloat32m1_t 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 vfloat64m1_t 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,       mask, vint8m1_t                    vd.
vbool8_t
                        vint8m1_t vs2, int8_t rs1).
  vint16m1  __riscv_xt_vcpx7(unsigned idx,      vs2,       rs1).
  _t        vint16m1_t                          int16_t
  vint16m1  __riscv_xt_vcpx7(unsigned idx,     mask,         vd.
  _t        vbool16_t                          vint16m1_t
                        vint16m1_t vs2, int16_t rs1).
  vint32m1  __riscv_xt_vcpx7(unsigned idx,      vs2,       rs1).
  _t        vint32m1_t                          int32_t
  vint32m1  __riscv_xt_vcpx7(unsigned idx,     mask,         vd.
  _t        vbool32_t                          vint32m1_t
                        vint32m1_t vs2, int32_t rs1).
  vuint8m1  __riscv_xt_vcpx7(unsigned idx,      vs2,       rs1).
  _t        vuint8m1_t                          uint8_t
  vuint8m1  __riscv_xt_vcpx7(unsigned idx,    mask,         vd.
  _t        vbool8_t                          vuint8m1_t
                        vuint8m1_t vs2, uint8_t rs1).
  vuint16m1 __riscv_xt_vcpx7(unsigned idx,      vs2,       rs1).
  _t         vuint16m1_t                        uint16_t
  vuint16m1 __riscv_xt_vcpx7(unsigned idx,     mask,         vd.
  _t         vbool16_t                         vuint16m1_t
                        vuint16m1_t vs2, uint16_t rs1).
  vuint32m1 __riscv_xt_vcpx7(unsigned idx,      vs2,       rs1).
  _t         vuint32m1_t                        uint32_t
```

(Continued on next page)

(continued from previous page)

```
vuint32m1 __riscv_xt_vcpx7(unsigned idx,     mask,          vd
_t        vbool32_t                          vuint32m1_t    .
                        vuint32m1_t vs2, uint32_t rs1).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,         vs2.
vint8m1_t                               vint8m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,    mask,         vs3,         vs2.
vbool8_t                                vint8m1_t     vint8m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vint16m1_t                              vint16m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,         vs3.
vbool16_t                               vint16m1_t
                    vint16m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vint32m1_t                              vint32m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,         vs3.
vbool32_t                               vint32m1_t
                    vint32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vint64m1_t                              vint64m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,         vs3.
vbool64_t                               vint64m1_t
                    vint64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vuint8m1_t                              vuint8m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,    mask,         vs3.
vbool8_t                               vuint8m1_t
                    vuint8m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vuint16m1_t                             vuint16m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,         vs3.
vbool16_t                               vuint16m1_t
                    vuint16m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     vs3,          vs2.
vuint32m1_t                             vuint32m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,         vs3.
vbool32_t                               vuint32m1_t
```

```
                    vuint32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,       vs3,           vs2.
vuint64m1_t                               vuint64m1_t
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,          vs3.
vbool64_t                                 vuint64m1_t
                    vuint64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,       vs3,           vs2.
vfloat32m1_t                              vfloat32m1_t
                    unsigned imm5).                          vs2.
void__ riscv_xt_vcpx8(unsigned idx,       vs3,
vfloat32m1_t                              vfloat32m1_t
                    unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8(unsigned idx,     mask,          vs3.
vbool32_t                                 vfloat32m1_t
                    vfloat32m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,          vs3.
vbool32_t                                 vfloat32m1_t
                vfloat32m1_t vs2, unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8(unsigned idx,       vs3,           vs2.
vfloat64m1_t                              vfloat64m1_t
```

(Continued on next page)

(continued from previous page)

```
                    unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,         vs3,            vs2
vfloat64m1_t                                vfloat64m1_t    .
                    unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8(unsigned idx,     mask,           vs3.
vbool64_t                                   vfloat64m1_t
                    vfloat64m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,     mask,           vs3.
vbool64_t                                   vfloat64m1_t
                vfloat64m1_t vs2, unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx9(unsigned idx,         vs3,            vs2
vfloat32m1_t
                    float fs1).             vfloat32m1_t    ,
void__ riscv_xt_vcpx9(unsigned idx,
vfloat32m1_t
                                            vs3,            vs2
                    float fs1, unsigned
                    frm);
                                            vfloat32m1_t    .


void__ riscv_xt_vcpx9(unsigned idx,     mask,           vs3.
vbool32_t                                   vfloat32m1_t
                    vfloat32m1_t vs2, float fs1).
void__ riscv_xt_vcpx9(unsigned idx,     mask,           vs3.
vbool32_t                                   vfloat32m1_t
                    vfloat32m1_t vs2, float fs1, unsigned frm);
void__ riscv_xt_vcpx9(unsigned idx,         vs3,            vs2
vfloat64m1_t
                    double fs1).            vfloat64m1_t    ,
void__ riscv_xt_vcpx9(unsigned idx,
vfloat64m1_t
                                            vs3,            vs2
                    double fs1, unsigned
                    frm);
                                            vfloat64m1_t    .


void__ riscv_xt_vcpx9(unsigned idx,     mask,           vs3.
vbool64_t                                   vfloat64m1_t
                    vfloat64m1_t vs2, double fs1).
void__ riscv_xt_vcpx9(unsigned idx,     mask,           vs3.
vbool64_t                                   vfloat64m1_t
                    vfloat64m1_t vs2, double fs1, unsigned frm);
vfloat32m1    _t                                    __riscv_xt_vcpx10(unsigned idx,
_t                                                  vfloat32m1_t
vfloat32m1                                          __riscv_xt_vcpx10(unsigned idx,
                                                    vfloat32m1_t
```

```
            vs2, float fs1);
            vs2, float fs1,
              i
              g
              n
              e
              d

              f
              r
              m
              )
              ;
vfloat32m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
_t         vbool32_t                           vfloat32m1_t
                          vfloat32m1_t vs2, float fs1).
vfloat32m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
_t         vbool32_t                           vfloat32m1_t
                            vfloat32m1_t vs2, float fs1, unsigned frm);
vfloat64m1 __riscv_xt_vcpx10(unsigned idx,        vs2, double
_t         vfloat64m1_t                           fs1); vs2,
vfloat64m1 __riscv_xt_vcpx10(unsigned idx,        double fs1,
           vfloat64m1_t
_t
                            unsigned frm);
vfloat64m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
_t         vbool64_t                           vfloat64m1_t
                          vfloat64m1_t vs2, double fs1).
vfloat64m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
_t         vbool64_t                           vfloat64m1_t
                            vfloat64m1_t vs2, double fs1, unsigned frm);
```

5.6.5.2 Part RV64

The following intrinsic interfaces are only available on the RV64 platform.

```
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 only available when the Zvfh or Zvfhmin extension is supported.

```
void__ riscv_xt_vcpx0(unsigned idx,       vs2).
vfloat16m1_t                              vs2, unsigned frm);
void__ riscv_xt_vcpx0(unsigned idx,
vfloat16m1_t
void__ riscv_xt_vcpx0(unsigned idx,    mask,          vs2)
vbool16_t                              vfloat16m1_t   ;
void__ riscv_xt_vcpx0(unsigned idx,    mask,          vs2.
vbool16_t                              vfloat16m1_t
                  unsigned frm);
vfloat16m1 __riscv_xt_vcpx1(unsigned idx,       vs2).
_t         vfloat16m1_t                         vs2, unsigned frm);
vfloat16m1 __riscv_xt_vcpx1(unsigned idx,
_t         vfloat16m1_t
vfloat16m1 __riscv_xt_vcpx1(unsigned idx,    mask,          vd.
_t         vbool16_t                         vfloat16m1_t
                       vfloat16m1_t vs2).                   vd.
vfloat16m1 __riscv_xt_vcpx1(unsigned idx,    mask,
_t         vbool16_t                         vfloat16m1_t
                    vfloat16m1_t vs2, unsigned frm).
void__ riscv_xt_vcpx2(unsigned idx,            vfloat16m1_t
```

```
void__ riscv_xt_vcpx2(unsigned idx,          vs2, unsigned imm5);
vfloat16m1_t                                 vs2, unsigned imm5,
                  unsigned frm);
void__ riscv_xt_vcpx2(unsigned idx,     mask,           vs2.
vbool16_t                               vfloat16m1_t
                  unsigned imm5).                       vs2.
void__ riscv_xt_vcpx2(unsigned idx,     mask,
vbool16_t                               vfloat16m1_t
                  unsigned imm5, unsigned frm);
vfloat16m1 __riscv_xt_vcpx3(unsigned idx,       vs2, unsigned imm5);
_t         vfloat16m1_t                         vs2, unsigned imm5,
vfloat16m1 __riscv_xt_vcpx3(unsigned idx,
_t         vfloat16m1_t
                       unsigned frm);
vfloat16m1 __riscv_xt_vcpx3(unsigned idx,     mask,           vd.
_t         vbool16_t                          vfloat16m1_t
                    vfloat16m1_t vs2, unsigned imm5).
vfloat16m1 __riscv_xt_vcpx3(unsigned idx,     mask,           vd.
_t         vbool16_t                          vfloat16m1_t
                       vfloat16m1_t vs2, unsigned imm5, unsigned frm);
```

(Continued on next page)

(continued from previous page)

```
void__ riscv_xt_vcpx4(unsigned idx,          vs3,          vs2
vfloat16m1_t
                  vfloat16m1_t vs1).     vfloat16m1_t   ,
void__ riscv_xt_vcpx4(unsigned idx,
vfloat16m1_t                             vs3,          vs2

                                         vfloat16m1_t   .
                  vfloat16m1_t vs1, unsigned frm).
void__ riscv_xt_vcpx4(unsigned idx,    mask,          vs3.
vbool16_t                                vfloat16m1_t
                  vfloat16m1_t vs2, vfloat16m1_t vs1).
void__ riscv_xt_vcpx4(unsigned idx,    mask,          vs3.
vbool16_t                                vfloat16m1_t
               vfloat16m1_t vs2, vfloat16m1_t vs1, unsigned frm);
vfloat16m1 __riscv_xt_vcpx5(unsigned idx,       vs2,          vs1)
_t         vfloat16m1_t                          vfloat16m1_t   ;
vfloat16m1 __riscv_xt_vcpx5(unsigned idx,       vs2,          vs1.
_t         vfloat16m1_t                          vfloat16m1_t
                        unsigned frm);
vfloat16m1 __riscv_xt_vcpx5(unsigned idx,    mask,          vd.
_t         vbool16_t                             vfloat16m1_t
                     vfloat16m1_t vs2, vfloat16m1_t vs1).
vfloat16m1 __riscv_xt_vcpx5(unsigned idx,    mask,          vd.
_t         vbool16_t                             vfloat16m1_t
                        vfloat16m1_t vs2, vfloat16m1_t vs1, unsigned frm);
void__ riscv_xt_vcpx8(unsigned idx,       vs3,          vs2.
vfloat16m1_t                             vfloat16m1_t
                  unsigned imm5).                          vs2.
void__ riscv_xt_vcpx8(unsigned idx,       vs3,
vfloat16m1_t                             vfloat16m1_t
                  unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx8(unsigned idx,    mask,          vs3.
vbool16_t                                vfloat16m1_t
                  vfloat16m1_t vs2, unsigned imm5).
void__ riscv_xt_vcpx8(unsigned idx,    mask,          vs3.
vbool16_t                                vfloat16m1_t
               vfloat16m1_t vs2, unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx9(unsigned idx,       vs3,          vs2.
vfloat16m1_t                             vfloat16m1_t
                  _Float16 fs1).                           vs2.
void__ riscv_xt_vcpx9(unsigned idx,       vs3,
vfloat16m1_t                             vfloat16m1_t
                  _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,     mask,           vs3.
  vbool16_t                               vfloat16m1_t
                    vfloat16m1_t vs2, _Float16 fs1, unsigned frm);
  vfloat16m1 __riscv_xt_vcpx10(unsigned idx,        vs2 _Float  fs1)
  _t         vfloat16m1_t                             , 16      ;
  vfloat16m1 __riscv_xt_vcpx10(unsigned idx,        vs2 _Float  fs1,
  _t         vfloat16m1_t                             16
                           unsigned frm);            .
  vfloat16m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
  _t         vbool16_t                           vfloat16m1_t
                        vfloat16m1_t vs2, _Float16 fs1).
  vfloat16m1 __riscv_xt_vcpx10(unsigned idx,     mask,           vd.
  _t         vbool16_t                           vfloat16m1_t
                           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,        vs2).
vbfloat16m1_t                              vs2, unsigned frm);
void__ riscv_xt_vcpx0(unsigned idx,
vbfloat16m1_t
void__ riscv_xt_vcpx0(unsigned idx,    mask,           vs2)
vbool16_t                              vbfloat16m1_t   ;
void__ riscv_xt_vcpx0(unsigned idx,    mask,           vs2.
vbool16_t                              vbfloat16m1_t
                  unsigned frm);
vbfloat16m1 __riscv_xt_vcpx1(unsigned idx,        vs2).
_t          vbfloat16m1_t                          vs2, unsigned frm);
vbfloat16m1 __riscv_xt_vcpx1(unsigned idx,
_t          vbfloat16m1_t
vbfloat16m1 __riscv_xt_vcpx1(unsigned idx,    mask,           vd.
_t          vbool16_t                          vbfloat16m1_t
                       vbfloat16m1_t vs2).                    vd.
vbfloat16m1 __riscv_xt_vcpx1(unsigned idx,    mask,
_t          vbool16_t                          vbfloat16m1_t
                    vbfloat16m1_t vs2, unsigned frm).
void__ riscv_xt_vcpx2(unsigned idx,        vs2, unsigned imm5);
vbfloat16m1_t                              vs2, unsigned imm5,
void__ riscv_xt_vcpx2(unsigned idx,
vbfloat16m1_t
                  unsigned frm);
void__ riscv_xt_vcpx2(unsigned idx,    mask,           vs2.
vbool16_t                              vbfloat16m1_t
                  unsigned imm5).                      vs2.
void__ riscv_xt_vcpx2(unsigned idx,    mask,
vbool16_t                              vbfloat16m1_t
                  unsigned imm5, unsigned frm);
vbfloat16m1 __riscv_xt_vcpx3(unsigned idx,        vs2, unsigned imm5);
_t          vbfloat16m1_t                          vs2, unsigned imm5,
vbfloat16m1 __riscv_xt_vcpx3(unsigned idx,
_t          vbfloat16m1_t
                       unsigned frm);
vbfloat16m1 __riscv_xt_vcpx3(unsigned idx,    mask,           vd.
_t          vbool16_t                          vbfloat16m1_t
                    vbfloat16m1_t vs2, unsigned imm5).
vbfloat16m1 __riscv_xt_vcpx3(unsigned idx,    mask,           vd.
_t          vbool16_t                          vbfloat16m1_t
                       vbfloat16m1_t vs2, unsigned imm5, unsigned frm);
void__ riscv_xt_vcpx4(unsigned idx,          void__                vbfloat16m1_t
vbfloat16m1_t                                riscv_xt_vcpx
                  vbfloat16m1_t vs1).        4(unsigned
                                             idx,
```

```
vs3, vbfloat16m1_t                          vs2.

vs3, vbfloat16m1_t                          vs2.
                    vbfloat16m1_t vs1, unsigned frm).
  void__ riscv_xt_vcpx4(unsigned idx,     mask,            vs3.
  vbool16_t                               vbfloat16m1_t
                    vbfloat16m1_t vs2, vbfloat16m1_t vs1).
  void__ riscv_xt_vcpx4(unsigned idx,     mask,            vs3.
  vbool16_t                               vbfloat16m1_t
                 vbfloat16m1_t vs2, vbfloat16m1_t vs1, unsigned frm);
  vbfloat16m1 __riscv_xt_vcpx5(unsigned idx,         vs2.
  _t          vbfloat16m1_t
                           vbfloat16m1_t vs1).      vs2.
  vbfloat16m1 __riscv_xt_vcpx5(unsigned idx,
  _t          vbfloat16m1_t
                        vbfloat16m1_t vs1, unsigned frm).
  vbfloat16m1 __riscv_xt_vcpx5(unsigned idx,      mask,            vd.
  _t          vbool16_t                           vbfloat16m1_t
                           vbfloat16m1_t vs2, vbfloat16m1_t vs1).
  vbfloat16m1 __riscv_xt_vcpx5(unsigned idx,      mask,            vd.
  _t          vbool16_t                           vbfloat16m1_t
                           vbfloat16m1_t vs2, vbfloat16m1_t vs1,
                           unsigned frm);
  void__ riscv_xt_vcpx8(unsigned idx,         vs3,             vs2
  vbfloat16m1_t                               vbfloat16m1_t    .
                    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 Xxtceei interface

5.6.6.1 Base本 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); float

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 extensions

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 example

The following C code example shows how the Xxtccef extension intrinsic interface is used.

```
#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 begins with the header file `riscv_xt_cce.h`, which declares the intrinsic function of the generic coprocessor extension. This code calls the `riscv_xt_fcpx3_f64`

interface, which is the double-precision floating-point interface for the fcpx3 instruction. idx parameter is 1, which means that the interface to the number 1
The `fs1` parameter is `a` and the `fs2` parameter is `b`. These parameters are the two source operands of the fcpx3 instruction. fcpx3 instruction destination register is the return value of this function.

Take the C920V3 (`c920v3-cp`), which supports the General Purpose Co-Processor 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 to compile the target file `test.o`.

5.7 Instructions for using RISC-V Vector

Gentei GNU Compiler V3.0.0 onwards (inclusive) supports RISC-V Vector Intrinsic V1.0-RC2 and support for switching RVV variable types to fixed-length mode, as well as compatibility with RISC-V Vector Intrinsic V0.10, which was used in older versions. support for the RISC-V Vector V1.0 Automatic Vectorization

RISC-V Vector Intrinsic V1.0-RC2 is supported in Xantei LLVM Compiler V2.0.0 onwards (included). RISC-V Vector V1.0 auto-vectorization is supported.

Note: The list of Intrinsic interfaces for RISC-V Vector Intrinsic V0.10 includes "Xuantie 900 Series RVV-1.0 Intrinsic Manual.pdf","Xuantie 900 Series RVV-0.7.1 Intrinsic Manual.pdf".

5.7.1 Usage of RISC-V Vector V1.0 Intrinsic

RISC-V Vector V1.0 can be directly referenced to RISC-V Vector Intrinsic V1.0-RC2 for Intrinsic programming in Gentei GNU/LLVM compiler, as shown in example one. RISC-V Vector Intrinsic V0.10 can also be supported by adding the option -mrvv-v0p10-compatible, as shown in example two.

Example one:

```
#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
```

Generated instruction sequences:

```
add_vectorized.
        a3,zero a3,zero,.L8
.L3.
        vsetvli
        a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        sh2add a3,a3,a5
        sh2add a1,a5,a1
        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 two:

```
#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
```

Generated instruction sequences:

```
add_vectorized.
        a3,zero a3,zero,.L8
.L3.
        vsetvli
        a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        sh2add a3,a3,a5
        sh2add a1,a5,a1
        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 Usage of RISC-V Vector V1.0 Automatic Vectorization

XuanTie GNU/LLVM compiler both support automatic vectorization, XuanTie GNU compiler need to enable the compile option -mrvv-auto-vectorize, XuanTie LLVM compiler is not enabled by default on XuanTie series CPUs, you can enable vectorization by option "-mllvm -xt-enable-vectorization". vectorization", as shown in example 3.

Example Three:

```
#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];
}
```

Gentei GNU compiler compilation options:

```
-mcpu= c908v -O2 -mrvv-auto-vectorize
```

Xantei LLVM compiler compilation options:

```
-mcpu= c908v -O2 -mllvm -xt-enable-vectorization
```

Generated instruction sequences:

```
add_vectorized.
        a3,zero a3,zero,.L8
.L3.
        vsetvli
        a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        sh2add a3,a3,a5
        sh2add a1,a5,a1
        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 Usage of RISC-V Vector V0.7.1 Intrinsic

In XuanTie GNU compiler, v0p7 and xtheadvector are equivalent, you can refer to XuanTie ISA extension specification for XTheadvector to do the intrinsic programming, as in example four. You can also add the option -mrvv-v0p10-compatible to support RISC-V Vector Intrinsic V0.10, as shown in example 5.

Example Four:

```
{
  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
```

Generated instruction sequences:

```
add_vectorized.
        a3,zero a3,zero,.L8
.L3.
        vsetvli
        a5,a3,e32,m1,ta,ma
        vle32.v v1,0(a1)
        sh2add a3,a3,a5
        sh2add a1,a5,a1
        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 Five:

```
add_vectorized (int *c, int *a, int *b, int N)
{
  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
```

Generated instruction sequences:

```
add_vectorized.
        a3,zero 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
        a1,a1,a4 th.vadd.vvv
                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: XThead LLVM compiler does not support XTheadvector, only v0p7.

5.7.4 RISC-V Vector V1.0/V0.7.1 Intrinsic Fixed Length Usage

RISC-V Vector V0.7.1 and RISC-V Vector V1.0 in XuanTie GNU compiler V3.0.0 or later (inclusive) support fixed-length mode programming, the length is specified by zvl in arch, e.g., if the length is specified to be 128, the arch will be zvl128b, Example VI:

```
#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;
  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
```

Remarks: The Xantei LLVM compiler does not support fixed-length mode programming.

Chapter 6 Linking object files to generate executables

The linker combines the object files to produce the final executable file. The contents of the object files can be reordered and repositioned by linking the description file. Its basic commands are

```
csky-elfabiv2-ld options input-file-list
```

Among them:

options Link options
input-file-list all input object files

This chapter contains the following sections:

6.1 How to link libraries

Libraries are the most common means of packaging Application Programming Interface (API, Application Programming Interface), which is divided into static and dynamic libraries.

- Static library: a collection of object files, i.e., many object files are packaged to form a file, usually with ".a" as the file extension.
- Dynamic Libraries: Also known as Dynamic Shared Objects (DSO, Dynamic Shared Objects), short for Shared Objects, usually with ".so" as the file extension.

6.1.1 Generation of library files

- Generation of static libraries: first use the compiler to generate each object file, then use the csky-elfabiv2-ar package to generate the library file.

```
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
```

- Generation of dynamic libraries: use the compiler, add the option -shared -fPIC to generate library files (only supported by linux tools)

```
csky-linux-gnuabiv2-gcc -shared -fPIC csky.c -o libcsky.so
```

6.1.2 link library

Whether static or dynamic libraries, their naming convention follows a uniform rule, namely lib[name]. [a/so]. Linking libraries is done by adding the option -l[name].
and -L [libpath]. where libpath is the path to the lib[name]. [a/so] file.

For example, if you need to link libcsky.a, you need to add the linking option -lcsky -L [path to libcsky.a]

6.2 Memory layout of code segments and data segments in the target file

Each object file is composed of code segments, data segments and other segments, the linking process is actually the process of combining similar segments of each input object file to generate the final executable file, as shown in Figure 6.1.

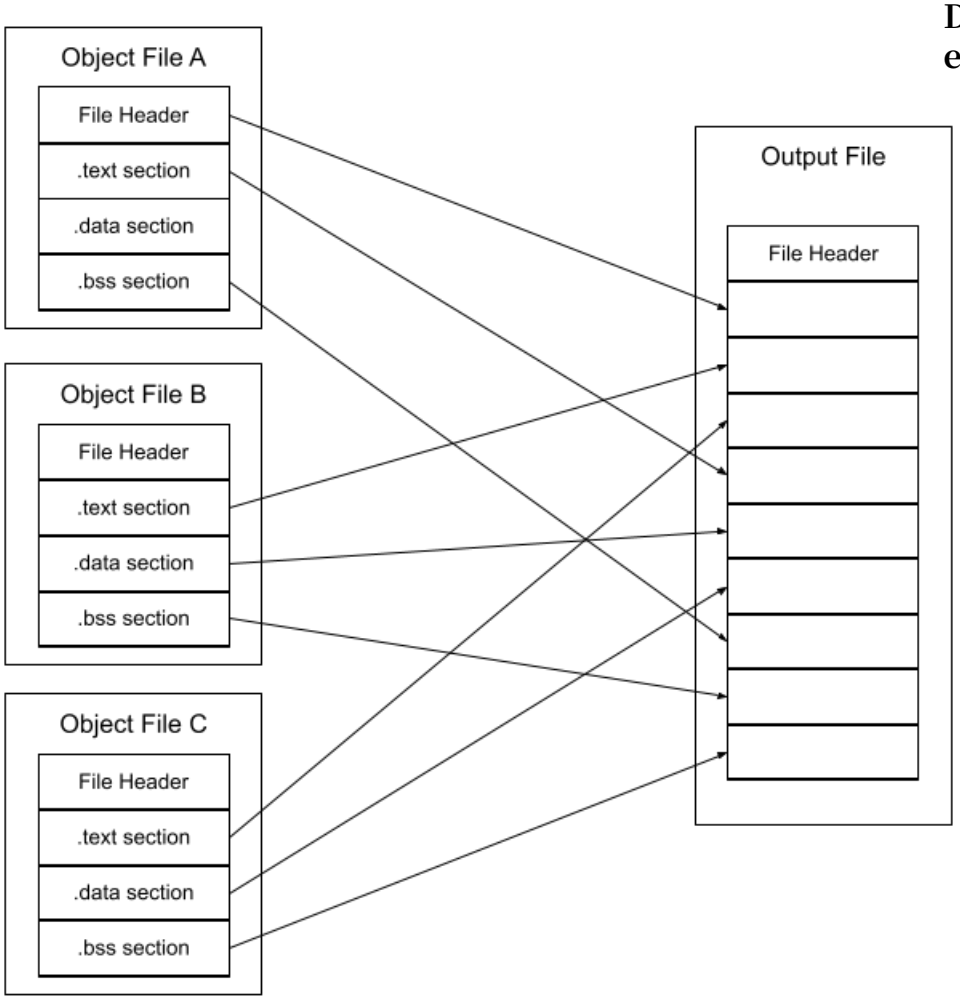


Figure 6.1: The process of linking

In order to precisely control the layout of the input file segments in the output file, the linker has designed the Linker Script to accomplish this difficult task, which

Specified via linker option -T [linksript]. If you do not specify a linkscript, the linker controls the linking process using the default linkscript, which merges similar segments and places them at a fixed address, which is generally difficult to meet the needs of embedded software developers.

A simple link description file is shown below:

```
ENTRY(__ start)

MEMORY
{
    INST   : ORIGIN= 0x00000000 , LENGTH=          /* ROM
    0x00020000 DATA      : origin= 0x00400000 ,    /* 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
}
```

Linking scripts contain many complex syntaxes to control the generation of executables,

commonly used are several syntaxes that control the memory layout of segments in the target file. The memory layout of a segment is the address at which the segment is stored in the target file, which is categorized as either a virtual address or a loaded address:

- Virtual Address: VMA, Virtual Memory Address, indicates the address of code or data at runtime
- Load Address: LMA, Load Memory Address

For the most part, VMA and LMA are the same, but in some embedded systems, especially those where the program is in ROM, LMA and

VMA is different. The method of specifying the target file output segment is shown in the above code.

1. Use the MEMORY expression to define a memory region in the following format

```
MEMORY
  {
    name [(attr)] : ORIGIN= origin, LENGTH= len
    ...
  }
```

2. Use "> [memory region]" in the section expression to specify the VMA of the output section in the following format

```
Section.
  {
    output-section-
    command output-
    section-command
    ...
  } [>region] [AT>lma_region]
```

3. If not specified, the VMA and LMA of the output section are the same, but if you want to specify the LMA, use "AT>[memory region]" in the section expression.

6.3 View the memory layout of the generated target file via ckmap

---

Note: This feature is currently not supported by the Xuan Tie 900 series toolchain.

If you want to see the memory layout and some other details of the final generated target file, you can generate a ckmap file by adding the link option -ckmap=[output filename]. ckmap contains the following five main sections:

1. Section Cross References

List all the call relationships between sections in the following format:

```
([ file A name])([ paragraph name]) refers to ([ file B name])([ paragraph
name]) for [ symbol name ])
```

This section is made up of several of the above statements, which indicate that a section of file A references a symbol defined in a section of file B.

2. Removing Unused input sections from the image

List all segments deleted with the link option -gc-sections turned on, in the following format:

```
Removing [ segment name] ([ filename]), ([ size] bytes).
...
[number] unused seciton(s) (total [ size] bytes) removed from the image.
```

3. Image Symbol Table

List all local and global symbols separately, with realistic symbol addresses, attributes, sizes, and segment names in the following format:

```
Local Symbols
Symbol Name                          Value       Type   Size  Section
...
Global Symbols
Symbol Name                          Value       Type   Size  Section
...
```

Among them, the values of Type are the following:

- w: Weak, weak symbol
- d: Debug, some debugging needs to use the auxiliary symbols, such as file name, paragraph name
- F: Function, function name
- f: filename
- 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 [ name of memory region ] (Base: [ start address ], Size: [ actual size ], Max: [ maximum value of memory region ])
Base Addr Size                      Type            Attr    Idx Name            Section                  Object
[Starting address ] [ Size ]        [Type ] [ Attributes ]  [Paragraph number]  [Name of paragraph]      [File name ]
...
```

Each input segment linked to a certain memory region (Region) is represented in the above table, where the Type is one of the following:

- Code: Code segment
- Data: data segment
- PAD: To align the filled area
- LD_GEN: The code Attr

generated by the linker

is of the following

types:

- RO: Read Only
- RW: Read Write.

5. Image component sizes

Counts the size of each input file's data occupied in the target file in the following format:

```
Code          RO            RW                  ZI          Debu      Object
 →Name        Data          Data                Data        g         ␣
[Code Segment Size ] [ Read-Only Data Segment Size ] [Read-
Write Data Segment Size ] [ bss Segment Size ] [ Debug Info
Segment Size ]␣
 →[ enter filename ]
```

Chapter 7 Optimization

This chapter focuses on how to use the Gentry compiler tool to optimize code size or performance, and the impact of optimization level on the use of debugging features. The chapter contains the following sections:

- Link-time optimization
- Impact of Optimization Options on Debugging Information
- Code Optimization Recommendations

7.1 Link-time optimization

Link Time Optimization (LTO) enables the compiler to the generated internal data structures (GIMPLE or LLVM IR) to disk so that all compilation units can be optimized as a whole. The compiler outputs the internal data structures to special section of the .o file. When these .o files are linked together, the linker collects information from all these special sections, and with this information, the optimizer is able to determine the dependencies between these modules, which allows for more optimization. For example, here 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, which 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 Gentry compiler uses link-time optimization via the **-flto** option, and it is important to note that we must use this option at both compile time and link time of the program, as in:

```
csky-elfabiv2-gcc -c -O2 -flto
foo.c 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, and the foo function calls the disassembled code of the bar function without using LTO:

```
foo.
    pus             r15
    h
    bsr             bar
    pop             r15
main.
    pus             r15
    h
    mov             r0, 3
    i
    mov             r0,
    i               12
    bsr             foo
    pop             r15
```

With LTO optimization turned on, the main function doesn't need to call the foo function and then the bar function, but instead returns the result of the addition. Obviously, the instructions are better optimized with LTO turned on:

```
bar.constprop.
0.
    mov             r0,
    i               15
    rts
main.
    pus             r15
    h
    bsr             bar.constprop
                    .0 r15
    pop
```

7.2 Impact of Optimization Options on

Optimized code has some impact on debugging information, so we need to balance the functionality of both in a given situation. And且 choosing to optimize code size or choosing to optimize performance has a different effect on the final result of the optimization.

In general, the code compiled with the **-O0** compilation option has the most accurate relationship with debugging information, and all generated code structures can directly correspond to the relevant source code. As the optimization level increases, the degree of correspondence between the compiled code and the source code will be lower and lower, because some of the code optimized by the compiler is difficult to be represented by debugging information. Of course, you can use the optimization option **-Og** to optimize the code according to your own needs, if don't want to use -O0, to ensure the accuracy of the code and the debugging information as much as possible.

7.3 Code Optimization Recommendations

This section introduces some good programming experiences and related techniques for improving the portability, efficiency, and robustness of C and C++ code. This section contains the following sections:

- Loop Iteration Conditional Optimization
- loop unrolling optimization
- Reduced function parameter passing

7.3.1 Loop Iteration Conditional Optimization

Loop structures are one of the more common types of operations in programs, and a lot of computing time is spent on these loops, so those in more time-sensitive environments need to pay a lot of attention to these areas.

To write an end-of-loop condition judgment, refer to the following coding guidelines in preference:

- Using Simple Conditional Judgments
- Loop iteration variable decremented to zero
- Use a counter of type **unsigned int**
- Using an iteration variable not equal to zero as a loop exit condition

The following two comparative examples present the loop operations for computing n! Comparing the generated assembly code, it can be seen that the code using the self-subtracting operations is able to use a single
The jbnez assembly instruction is used to perform a comparison jump, whereas with self-additions two instructions are required, first a comparison (cmp) and then a jump (jbf):

```
csky-elfabiv2-gcc -Os -S loop.c
```

- Self-incrementing arithmetic:

```
int fact1(int n)
{
    int i, fact= 1;
    for (i= 1; i<= n; i++ )
        fact *= i;
    return (fact);
}
```

```
fact1.
    mov     a3, 1
    i       a2,
            a3
    mov
            a0,
.L2.        a2
    cmpl    .L3
    t       a0,
    jbf     a3
    mov
    rts
```

(Continued on next page)

(continued from previous page)

```
.L3.
    mul     a3, a3, a2
    t
    add     a2, a2, 1
    i
    jbr     .L2
```

- Self-subtracting arithmetic:

```
int fact2(int n)
{
    unsigned int i, fact= 1;
    for (i= n; i ! = 0; i--
        ) fact *= i.
    return (fact);
}
```

```
fact2.
    mov     a3, a0
    mov     a0, 1
    i
            a3, .L3
.L2.
    jbne
    z
            a0, a0,
    rts     a3 a3,
.L3.        a3, 1
    mul     .L2
    t
    sub
    i
    jbr
```

7.3.2 loop unrolling optimization

Some short loops can be expanded to improve performance, but the code will be a bit larger accordingly. When loops are expanded, the number of loops is reduced, and of course the number of branch jump instructions executed is reduced. Some small short loops will be fully expanded, and the cost of loops will disappear completely. Loop unrolling is automatically enabled at optimization level **-O3**, otherwise you need to manually unroll loops in your code.

The following two comparative examples introduce the circular operation of data

copying. In the case of loop unrolling, it can reduce the performance loss caused by jumping, and in general, the running performance will be better than that of looping without unrolling, but the disadvantage is that it will increase the size of the code:

```
csky-elfabiv2-gcc -Os -S loop.c
```

- The cycle does not unfold:

```
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.
}
```

```
countbit
1.          a3, a2,
    addu    a0
.L2.        a2, a3
    cmpn    .L3
    e
    jbt
    rts     t0, (a2,
            0)
.L3.        t0, (a1,
    ld.     0)
    b
    st.     a2, a2, 1
    b       a1, a1, 1
    add     .L2
    i
    add
    i
    jbr
```

- The cycle unfolds:

```
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.
}
```

```
countbit
2.          a3,
    movi    0
.L2.        a0,
    cmpn    a3
    e       .L3
    jbt
```

(Continued on next page)

(continued from previous page)

```
    rt
    s
.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 Reduced function parameter passing

For passing function parameters you need to pay attention to the following points:

- In the abiv2 case, the function has 4 shaping parameter registers, and if the hard floating point function is used, there will 4 floating point registers, so minimizing the number of parameters in the case of a function call will keep the number within the number of registers and improve efficiency.
- In the C++ case, the implicit this pointer for non-static functions is through r0, so the number of argument registers is reduced by one.
- Put the relevant parameters into a structure, and then when the function is called, pass the parameters by passing the pointer to the structure. This also reduces the number of registers used.
- Reduce the use of long long type parameters, which take up 2 registers.
- In the case of soft floating point, minimize the use of arguments of type double.

7.4 KO file size optimization

Currently the mainstream RISCV compilers, including XuanTie compiler, will enable relax by default to optimize the program during linking. KO file is a special kind of ELF file, many of its symbols are external symbols, which can't be determined when linking, enabling relax function not only can't optimize the KO file, but also increase the size of the KO file.

In addition, Gentei compiler optimizes the size of KO file for scenarios where relax is turned off, and removes some redundant information. Therefore, you can add the -mno-relax option to reduce the size of the KO file when using the Gentei compiler.

Chapter 8 Programming Essentials

This chapter introduces several issues that developers often encounter during the development process, and contains the following topics:

- Peripheral Registers
- Impact of *Volatile* on Compilation Optimization
- Use of function stacks
- *inline* functions
- *Memory Barriers*
- Specifying Variables and Function *Sections*
- Assign functions and data to absolute addresses
- delay operation
- Customizing *C* Standard Input and Output Streams
- Basic *ABI* Description
- Variable synchronization
- Notes on self-modifying code
- Using inline assemblies
- *newlib* implements reentrant

8.1 Peripheral Registers

The operation of peripheral registers is a frequently encountered scenario in embedded software development. Due to some of its special characteristics (such as easy to be optimized by compiler), this section is dedicated to introduce the common development methods related to peripheral registers to help developers avoid some unnecessary troubles.

8.1.1 Peripheral Register Description

This subsection describes how to use C to describe peripheral registers in such a way that

the code can be compiled into a correct and efficient sequence of instructions while maintaining good readability despite the optimization conditions turned on by the compiler. The methodology is as follows:

1. Define the peripheral register flag macro, define the peripheral as volatile, and modify the input peripheral register with the const attribute, so that the register can only be read, and the compiler will report a warning message when a value is written to it.

```
#define I volatile
const #define O
volatile #define IO
volatile
```

2. Define data structures corresponding to the peripheral register programming model and qualify these registers with the appropriate IOs. Example:

```
typedef struct {
...
       __I uint32_t RXD.
       __O uint32_t TXD.
       __IO uint32_t STATUS.
       __I uint32 RESERVERD[5];
...
}Device_Uart_Type.
```

3. Define operation macros for peripheral registers, for example:

```
#define                         ((Device_Uart_Type *)
SOC_UART0                       SOC_UART0_BASE)
#define                         ((Device_Uart_Type *)
SOC_UART1                       SOC_UART1_BASE)
#define                         ((Device_Uart_Type *)
SOC_UART2                       SOC_UART2_BASE)
```

4. After defining the above macros, you can refer to them in your program to read and write peripheral registers. Example:

```
Receive_buf[0]= SOC_UART0->
RXD; SOC_UART0->TXD=
Receive_buf[0].
While (! (SOC_UART0->STATUS & UART_SENT_BIT));
```

Of course, the user can further SOC_UART0->TXD as Uart0_TXD to facilitate the user's operation of peripheral registers in the code.

8.1.2 peripheral bitfield operation

Peripheral registers are usually split into multiple parts that represent different functions, so to make it easier to control the peripheral registers, each part of the peripheral register can be represented by a bit field of the structure, and each field will have a domain name, which will be operated by the domain name in the program. The following is an example of manipulating a peripheral register through a bit field of a structure:

```
//-------------- 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            :8.
    SCR                 :16
    unsigned            .
  } Bits.
}
spi_cr0_str.
#define                 0x40003800
HMS_SPI_BASE            *(SPI_CR0_STR *)(HMS_SPI_BASE+ 0x000) //SPI
#define _SPI_CR0        Control␣
↪Register 0
                         (_SPI_CR0).Word
#define SPI_CR0          (_SPI_CR0).Bits.DSS
#define                  (_SPI_CR0).Bits.FRF
SPI_CR0_DSS              (_SPI_CR0).Bits.SPO
#define                  (_SPI_CR0).Bits.SPH
SPI_CR0_FRF              (_SPI_CR0).Bits.SCR
void test ()
#define
{
SPI_CR0_SPO
    SPI_CR0_DSS=         //8-bit data
#define                  size
    7;
SPI_CR0_SPH              //SPI frame
    SPI_CR0_FRF=         mode
    0;                   //SPI mode 00
#define
SPI_CR0_SCR              //clock post-
    SPI_CR0_SPO=         scaler=1
    0;.
    SPI_CR0_SPH=
    0;.
    SPI_CR0_SCR=
    0;.
}
```

8.1.3 -fstrict-volatile-bitfields option

In addition to adding volatile to type definitions, in most cases it is necessary to take care to add the -fstrict-volatile-bitfileds option at compile time to ensure that the resulting instructions manipulate the bitfields according to the word as the smallest read/write unit.

8.2 Impact of Volatile on Compilation Optimization

(1) Does not cache optimized variables that are used repeatedly

If a variable is not preceded by the keyword Volatile, and the compiler finds that the

variable has been used more than twice in a row, it will cache the value of the variable in a register instead of reading it from the initial memory location each time.Volatile indicates that the value of the variable may be changed externally and will need to be re-accessed each time it is used, and therefore the compiler will not optimize for caching.

（2） No optimizations for constant merging, constant propagation, etc.

The compiler's data flow analysis analyzes the assignment and use of variables in order to perform optimizations such as constant merging, constant propagation, etc. to further eliminate dead code. When a program does not need these optimizations, they can be disabled by the Volatile keyword, as in the following code, where the if condition will not always be true.

```
Volatile int i= 1;
if (i)
...
```

8.3 Use of function stacks

c/c++, the stack is used frequently, and the stack can hold the following:

1. Local Variables. Due to the limited number of registers, some local variables cannot be stored in registers and the stack allocates space for these variables.
2. Overflowed parameters. Because the number of arguments of the called function is larger than the number of passing parameter registers or because of a large parameter bit width, one passing parameter register is not enough to store the entire parameter, which results in the passing parameter registers being able to store only a part of the parameter, and the rest of the parameter or a part of the parameter is overflowed and its value is stored in the stack.
3. A return value that cannot be passed by a register. If the bit-width of the return value is greater than the sum of the bit-widths of the registers in which the return value is stored, the return value is stored on the stack, for example, a parameter of a structure type with a bit-width greater than 8 bytes.
4. The original value of the register. Some of the original values of the registers need to be protected, so using these registers to store local variables of a function requires that the original values of the registers be put on the stack, and then the value of the stack be deposited into the registers before the called function returns to the main function.
5. The return address of the function.
6. In addition to the above, if the alloc() function is used, it will also take up some memory on the stack.

In general, it is difficult to estimate stack usage because code dependencies vary with the execution path of the program. The following are ways to estimate the extent of stack usage:

1. Compilation is done with -fstack-usage, which produces a file with the suffix .su, which allows you to see the size of the stack.
2. Linking with the option -callgraph produces an html file that allows you to view the size of the stack.
3. Use the debugger to set a watchpoint at a stack position to observe the hit.

In order to not use the stack as much as possible, programs can be made to reduce the need for the stack, generally by doing the following:

1. Avoid local variables that are struct types or arrays.

2. Avoid recursive algorithms.
3. Don't write too many variables within a function.
4. Use code block scoping且 to define variables in the desired scope.

8.4 inline functions

The inline function is a trade-off between code size and performance; GCC can direct the compiler to inline the required function with keywords such as "inline", but it is up to the compiler to decide whether or not to inline the function; alternatively, the inline function can be forced to be inlined by using 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, so the content of the inline function and the call to the function must be in the same file, and the header file needs to be included in the header file when using the inline function.

8.4.1 inline

The keyword for declaring the inline function is "inline", or in the case of C90, "inline ", as defined below:

```
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 inline connection (computing)

When GCC does not inline any function, you can force it to be inlined by using the attribute keyword "always_inline", which is declared as follows:

```
inline void foo (const char)__ attribute__ ((always_inline));
```

8.4.3 Mixing inline functions with external calls

When there is an inline function definition and an external function definition for the same function in C, the compiler selects only the code for the inline function.

8.5 Memory Barriers

A memory barrier instruction is a synchronization point in the CPU or compiler's operation of random access to memory such that all read and write operations prior to this point are executed before the execution of operations after this point can begin.

CSKY, the definition of a memory fence is relatively simple; it is actually an inline assembly instruction, defined as follows:

```
#define MEMORY_BARRIER asm (""::: "memory");.
```

8.6 Specifying Variables and Function Sections

Variables and functions can be assigned to a specific section through gcc's section attribute, which is used by adding the attribute__ ((section ("< section name> "))) to the definition or declaration of the variable and function, see the following example.

Specify a function to a specific section

```
extern void foobar (void)__ attribute___ ((section (".bar"))).
```

Assigning variables to a specific section

```
char stack[10000]__ attribute_____   ((section (".STACK")))=  { 0 };
```

8.7 Assign functions and data to absolute addresses

The way to assign functions and data to absolute addresses is:

1. To assign a function or data to a special section, see Assigning Variables and Functions *Sections*.
2. Then when linking, modify the linking script to assign the section to the appropriate address, as shown in the memory layout of code and data segments in the target file.

The following example the variable stack to address 0x500000.

- Definition of variables

```
char stack[10000]__ attribute_____   ((section (".STACK")))=  { 0 };
```

- Link script modification to add memory region STACKR MEMORY and assign the STACK segment to the STACKR region SECTIONS

```
 INST    ORIGI = 0x0000000 , LENGTH = 0x00020000  /* ROM */
  INST      N          0
 DATA    ORIGI = 0x0040000 , LENGTH = 0x00004000  /* RAM */
    :      N          0
STACKR. ORIGI = 0x0050000 , LENGTH = 0x00010000
          N          0
EEPROM. ORIGI = 0x0060000 , LENGTH = 0x00010000
          N          0
```

```
    .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 delay operation

In some MCU applications, a period of time is needed between two operations. Some developers prefer to use an empty loop body to achieve a certain delay (the specific delay time is calculated according to the number of instructions executed), this style of code has the following disadvantages:

1. The operation is more dangerous, a useless null operation, which is often simply removed by the compiler, thus having no delay effect
2. The delay time varies with the optimization options of the compilation
3. The portability of the

function at different

frequencies is proposed

to do using the

following method:

Abstract a delay function, such as delay_us(int val), which is written in assembly or inline assembly to prevent compiler optimizations from affecting it.

Delay functions such as delay_us function can be based on the current system operating frequency to adjust the number of instructions executed, used to adapt to different system operating frequency.

Assuming that the operating frequency of the system is 20Mhz, the delay function of 801 can be

realized 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 Standard Input and Output Streams

Due to the specificity of embedded platform development, in the current CSKY platform to provide the C language standard library, the standard input scanf and output function printf will use the hook function fgetc and fputc to realize the corresponding input and output functions, so as to improve the flexibility of development. Users need to provide user-defined fgetc and fputc functions if necessary.

8.10 ABI Description for本

When users develop in a mix of C and assembly code, they need to be concerned with the application binary interface, which describes how parameters should be passed and how return values should be stored. This section briefly describes the mechanism for passing and returning variables of basic data types in the second edition of the CSKY ABI.

8.10.1 function parameter passing

There are two types of parameters passed, basic data types (char, short, int, long, etc.) and aggregate types (array, structure, class, etc.). See *the CSKY* Application Binary Interface Specification *(*Second Edition*)* for more information on how to pass parameters of aggregated types.

The first thing to note is that the csky family of compilers performs an expansion operation on type passes of less than 32 bits in size so that the size can be stored in a 32-bit register. The current CSKY ABI specification specifies that the first four parameters that are less than 4 bytes in size can be passed using registers r0-r3, while the rest of the parameters are passed using stack slots. As an example, the following code.

```
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 r0, r1, r2, r3 (or alias a0, a1, a2, a3) registers in that order, and rem1 will be stored in the stack slot at the location of sp-8.

8.10.2 Function Return Value Passing

The ABI specifies how to read data a callee in addition to how to pass data to the callee, which requires a coordinated implementation between the calling and callee functions. The callee function places a certain type of data in a given location (r0/r1) or stack slot according to the ABI specification. The following is an 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 main call function.

When the called function returns an aggregated value, e.g., a large array of structures, the registers in the csky series CPUs are not large enough for the return value to be passed. The following is an 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 an aggregate type, which according to the CSKY ABI specification version 2 will be implemented using indirect passing. First, the calling function allocates a space on the call stack of the calling function

for the return value, and then passes the entry address of this space to the callee function as the first parameter of the function call, and the callee function uses this address to implement the operations. The equivalent C 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

Synchronization of variables is a common problem in application development, and can be achieved using volatile declarations. In multitasking programming, the CSKY architecture provides the user with the idly4 instruction to mask interrupts.

8.11.1 Synchronizing Variables with Volatile

In the process of developing applications, the problem of variable synchronization is often encountered. For example, in an application scenario, the interrupt service program that receives data puts the data into the receive buffer every time it receives the data, and changes the global variable Received_flag to notify the main program to process it; the main program keeps reading the variable, and calls the processing function to process the data in the receive buffer when the variable is set. Users often ignore the specificity of this global variable and use the normal declaration method, which can cause the application to fall into a dead loop. Why does this happen?

Let's briefly analyze the reason. After a global variable is compiled and linked, the compiler tool will allocate a memory space for the variable as a storage space for its value. When the user accesses (reads) the variable in C, the compiler tool will generate the corresponding access instructions to get the value of the variable's memory space. But how does the compiler handle the situation when we write the same statement over and over again in C to read the value of the same variable? Without optimization, the compiler generates a memory access instruction for each C statement to fetch the value of the memory in turn; with optimization turned on, the compiler generates only one memory access instruction, and the subsequent ones will always use the original value for processing and computation, because the compiler believes that the value of the memory will not change under the current scope.

How to deal with this situation correctly? The easiest way is to modify such global variables (received_flag) with volatile, and the compiler will generate memory access instructions for each variable access operation to get the latest value in memory for computation.

8.11.2 Variable synchronization in multitasking programming

How to access and change the same variable in different tasks (or threads) is a critical

issue in multitasking programming. In the CSKY architecture, the user is provided with special instructions (idly4) for such scenarios.

The instruction function of idly4 is defined as follows. After the execution of this instruction, the process of executing the next four instructions will be shielded from interrupts, and if an exception is generated during the execution process, the high flag bit (C bit) will be set to notify the user that an exception has occurred. Application examples are as follows:

```
bmaski r1, 32          Get all 1 's constants
lrw r2,                ;Get the pointer to the
Semaphore idly4        semaphore in memory
ld.w                   Starts a non-interruptible
r3,(r2,0) bt           queue of 4 instructions.
Sequence_failed        ;Read a semaphore from
                       memory
bt Semaphore_corrupted ;Check for exceptions
st.w r1,(r2,0)         (optional)
cmpnei r3,0            ;signaling test
                       ;No-Op
                       ; (id) Set semaphore to all
                       1 's
```

8.12 Notes on self-modifying code

BootLoader, it is usually necessary to carry a piece of code from the storage address to its runtime 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 regarded as self-modifying code behavior.

In CSKY architecture, since it is Von Neumann and Haver mismatchable, when doing this kind of operation, we need to consider the problem of CPU Cache and memory content consistency. That is to say, after modification, you need to clear the contents of D Cache to memory and Invalid the contents of I Cache before you can do the final jump operation.

8.13 Using inline assemblies

The basic format of the Gentei CPU toolchain's embedded assembly conforms to the basic syntax of GNU gcc. The following is an example of a Gentei 800 series instruction if not noted, but the same syntax rules apply to the Gentei 900 series.

8.13.1 asm format

Use the "**asm**" keyword to indicate source code passages 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 compilations */
asm volatile ("mov r0, r1\nmov r1, r0");

/* Multiple inline assemblies, with optional \t to make the generated
assembly code more user-friendly */
asm volatile ("mov r0, r1\n\tmov r1, r0");
```

Assembly code included in parentheses must follow a specific format:

- Instructions must be enclosed in quotation marks
- If more than one instruction is included, then a newline character must be used to separate each line of assembly language code. Often, tabs are also included to help indent the assembly language code and make the lines of code easier to read.

The second rule is needed because the compiler takes the assembly code in the asm segment verbatim and且 puts it in the assembly code generated for the program. Each assembly language instruction must be on a separate line - hence the need to include newline characters.

Note: If you do not want the compiler to optimize the inline assembly, you can add the volatile keyword to prevent the compiler from optimizing, i.e. asm volatile ("assembly code"). It is not required, but in most cases it will be added, so volatile will be added to all 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 the global variables of the c program.
- Great care must be taken not to change the value of any registers in the embedded assembly code.

The gcc compiler provides an extended format for asm segments

to help with these problems. the extended version of asm has

the following format: **asm ("assembly code": output locations :**

input operands : changed registers). This format consists of 4

sections separated by colons:

- assembly code: Embedded assembly code that uses the same syntax as the basic asm format.
- output locations: a list of registers and memory locations that contain the output values of the embedded assembly code, in the format: Specify input and output values.
- input operands: a list of registers and memory locations containing the input values of the embedded assembly code, in the form of: specified input values and output values
- changed : a list of any **other** registers changed by the inline code

Not all of these sections must appear in the extended asm format. If the assembly code does not generate output values, this section 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 alter register values, then the final 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 list of

input and output values

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. constraint defines where to store the variable (for input values) or where to transmit it from (for output values) Use it to define whether to store the variable in a register or in a memory location.

Constraints are composed of a single string, see: *gcc* constraints related code

In addition to these constraints, the output value contains a constraint modifier that instructs the compiler how to treat the output value, see: *gcc* output modifiers for details

Notice! "Constraints" can be used to specify the registers used for variable storage, but do not implicitly type convert at the C 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 of the two shorts. The author wants a and b to be sign-extended to the register width in the embedded assembly, but in fact the compiler only guarantees that the lower 16 bits of a and b are valid, while the higher bits (16 bits for 32-bit systems and 48 bits for 64-bit systems) are undefined. To fix the problem caused by the above code, you can try to explicitly convert a and b at the C level as follows:

```
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 reentrant

The newlib implementation has some global variables, chains or queues. When the system is multi-threaded, these variables, chains or queues can cause thread contention problems. Therefore, newlib provides some locks and interfaces that allow the user to interface to specific operating systems, so that the data structures used by newlib are thread-safe and are reentrant.

```
#include <sys/lock.h>
struct lock {
  char unused;
};

struct lock lock___
sinit_recursive_mutex; struct lock lock
_____
sfp_recursive_mutex; struct lock lock
_____
sfp_recursive_mutex; struct lock lock
sinit_recursive_mutex; struct lock
```

(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
struct lock__ lock__ dd_hash_mutex.
struct lock__ lock__ arc4random_mutex.


void
__                  (_LOCK  *lock)
retarget_lock_ini _T
t
{
}


void
__                                  *lock)
retarget_lock_init_recursive(_LO
CK_T
{
}


void
__                          lock)
retarget_lock_close(_LOCK
_T
{
}


void
__                                  lock)
retarget_lock_close_recursive(_LO
CK_T
{
}


void                    }
__                            (_LOCK_T
retarget_lock_acqui
re
{
```

```
                    lock)


void
__                              (_LOCK  lock)
retarget_lock_acquire_recurs  _T
ive
{
}

int
__                               lock)
retarget_lock_try_acquire(_LOC
K_T
{
  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 realize the file as above, perform the following steps:

1. Define struct_ lock structure according to the system's lock definition
2. Define the above mutex locks
3. real Realization of retarget_lock_acquire, retarget_lock_release, retarget_lock_acquire_recursive, retarget_lock_release_recursive, retarget_lock_close_recursive, retarget_lock_init_recursive, and a few other interfaces are not called by the C library, so they can be left empty.

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 compilers, assemblers, linkers, and debuggers, the Gentei toolchain contains many binary tools:

- *-addr2line - Get the name and line number of the source code file where the program is located based on its address.
- *-ar - Create, modify and extract static libraries (archive).
- *-c++filt - Get the original symbol name of the C++ symbol being overloaded.
- *-gprof - Displays program analysis (profiling) information.
- *-nm - Lists the symbols in the given target file.
- *-objcopy - Copy and transform target files.
- *-objdump - Displays information about the target file.
- *-ranlib - Generates an index of content for static libraries (archive).
- *-readelf - Displays information about the target file in ELF format.
- *-size - List the segment size of the target file or static library.
- *-strings - Lists all printable strings in the file.
- *-strip - Removes symbolic information from the target file, reducing the file size.

The above tools are supported by both the Gentei GNU toolchain, which has a prefix (Table 2.1) plus a component name, e.g., riscv64-unknown-linux-addr2line, and the Gentei LLVM toolchain, which has an llvm-plus-component-name, e.g., llvm-addr2line. Some of these tools will be used frequently in development, and only two of them are used in the two subsections presented below:

- Viewing and analyzing common information *in ELF* files
- How *bin* and *hex* files are generated

9.1 Viewing and analyzing common

information in ELF files

ELF files, i.e., target files in ELF (Executable and Linkable Format) format, can be viewed through the *-readelf command, and different information can be viewed through different options.

1. -S

Displays segment information about the program in the following format:

```
Section Headers.
  [Nr] Name              Type            Addr    Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000 00000 00000 00      0   0  0
                                         0       0     0
  [ 1] .text             PROGBITS        0000000 00100 0022c 00  AX  0   0 1024
                                         0       0     0
  [ 2] .rodata           PROGBITS        0040000 00400 00055 00   A  0   0  4
                                         0       0     0
  [ 3] .data             PROGBITS        0040055 00455 00023 00  WA  0   0  4
                                         0       0     0
  [ 4] .bss              NOBITS          0040078 00478 00009 00  WA  0   0  4
                                         0       0     0
```

Among them.

- Name: Segment name
- Type: the type of the segment, its common values are as follows
  - NOBITS: program data that does not need to be stored in the file, in general, with the NOBITS attribute is the .bss segment
  - PROGBITS: corresponds to NOBITS, contains program data in the file, except for the .bss segment, code segments and data segments are generally of this type.
  - SYMTAB: Symbol table, typically of type .symtab segment
  - STRTAB: a table of strings, generally of the type of .strtab segments
- Addr: starting running address of the segment
- Off: Offset of the paragraph in the document
- Size: segment size in byte
- Flg: the segment attribute, which has the following common values
  - W: Write, writable
  - A: Alloc, which needs to be loaded into the content when executed
  - X: Execute
  - M: Merge, which the linker thinks can be merged, and且 The linker will try to merge the compressed segments
  - S: Strings, segment contents are strings
- Al: Alignment requirement of the segment in byte

2. -l

Displays program header information in the following format:

```
 Type           Offset   VirtAddr  PhysAddr  FileSiz MemSiz Flg
 e              Align
 LOA            0x001000 0x00000000 0x00000000 0x022c0 0x022c0 R
 D              E 0x1000
 LOA            0x004000 0x00400000 0x00400000 0x00780 0x00810
 D              RW 0x1000
Section to Segment
mapping.
 Segment
  Sections... 00
         .text
  01     .rodata .data
  .bss
```

The so-called program is a of sections that belong to the same program with the same attributes. When the program is executed, the file is loaded into the memory in the unit of program.The meaning of each field of program is as follows:

- Type: program type, usually LOAD, means the program needs to be loaded into memory.

- Offset: the offset of the program in the file.
- VirtAddr: the program's runtime address.
- PhysAddr: the address where the program is loaded.
- FileSiz: program's size in the file
- MemSiz: size of program loaded into memory
- Flg: the program attribute, which has the following values
  - R: Readable
  - E: Executable
  - W: Writable
- Align: the program's requirements for it.

3. -s

Displays the symbol table for the program in the following format:

```
Symbol table '.symtab' contains 170
entries.
   Num.    ValueSize Type   Bind  Bind    Ndx
     Name
     0:
     00000000      0 NOTYPE LOCAL DEFAULT UND

     ...          386       GLOBAL          1
     96:          FUNC      DEFAULT         printf
     00001a38
     ...
```

This option lists all the symbols of the program and information about the symbols, and the meaning of each of its fields is as follows:

- Value: address of the symbol
- Size: the size of the symbol (e.g. the size of a function or variable)
- Type: symbolic type with the following common values
  - FUNC: Function name
  - OBJECT: variable name
  - FILE: File name
  - NOTYPE: symbols with no declared type
- Bind: the scope of the symbol, with the following common values
  - LOCAL: local symbol
  - GLOBAL: global symbol, i.e. accessible to other files
  - WEAK: weak symbol
- Name: Symbol name

9.2 How bin and hex files are generated

A Bin file is a binary file with no internal address tag. Generally, the programmer burns the program from zero address, and if it is loaded into the memory, it is loaded to the address of the link when it is run.Bin file can be obtained by converting the ELF file, the command is as follows:

```
*-*-objcopy -O binary [ Input ELF file ] [ Output bin file ]
```

Hex files are often used to store programs or data transfers to ROM, EPROM, and can be obtained by converting bin files with the following commands:

```
*-*-objcopy -I binary -O ihex [Input bin file ] [ Output hex file ]
```

Chapter 10 Charts

10.1 gcc constraints related code

10.1.1 CSKY architecture related constraints

austerity	descriptive
a	Using the r0 - r7 registers
b	Using the r0 - r15 registers
c	Using the c register
y	Use hi or lo registers
l	Using the lo register
h	Using the hi register
v	Using the vector register
z	Using the sp register

10.1.2 RISC-V Architecture Related Constraints

austerity	descriptive
f	Using Floating Point Registers
I	12-bit Signed Immediate
J	integer zero
K	5-bit unsigned immediate number

A	Memory addresses held in general purpose registers

10.1.3 gcc public constraint code

austerity	descriptive
m	Memory locations of used variables
r	Use any available general purpose registers
i	Using immediate integer values
g	Use any available registers or memory locations

10.1.4 gcc output modifiers

output modifier	descriptive
+	Can read and write operands
=	Write operands only
%	If necessary, the order of the operands can be swapped
&	Operands can be deleted or reused before the inline function is completed