T-HEAD 软件开发指南

2020 年 10 月 19 日

平头哥半导体有限公司 T-HEAD Semiconductor Co.,LTD

地址: 杭州市余杭区向往街 1122 号欧美金融城 (EFC) 英国中心西楼 T6

邮编: 311121

网址: www.t-head.cn

版本历史

版本	描述	日期
1.1	修复一些 vdsp intrinsic 接口描述错误的问题	2020.2.11
1.2	添加 RISC-V 系列 CPU 编程章节 修复一些书写的错误	2020.5.7
1.3	添加向量浮点指令说明	2020.5.11
1.4	修改 T-Head 扩展指令的命名规则 新增 c906	2020.6.20
1.5	添加 libcc-rt 说明章节	2020.8.5
1.6	添加 abi 类型 lp64dv	2020.8.9

对应工具版本

类别	版本号
平头哥 800 系列工具	V3.10
平头哥 900 系列工具	V1.10

软件开发指南

第一章　工具链简介

传统的工具链定义通常包括编译器、汇编器、链接器等。所有这些组建共同实现从 C/C++ 源代码到可执行文件的翻译过程，如 图 1.1 编译器对输入的源文件的处理流程，包括：词法分析、语法分析、语义检查、汇编代码生成。当输入的源文件不符合 C/C++ 语言标准或 GNU 扩充语法规范（编译文件选择的语言标准-std=gnu89 或-std=gnu++99）时，编译器将会生成对应的诊断信息，以提示程序开发者对应的源码文件存在语法或语义错误。编译器的最后工作是生成特定目标机器的汇编代码，比如：生成 CSKY 系列 ck810 的汇编代码。然后由汇编器将编译器生成的汇编代码汇编成目标代码。由于现代计算机软件库分离的设计原则，通常需要链接器将多个目标文件和若干静态库、动态共享库链接成一个完整的可执行文件。该三部分共同组成一个统一、有机的工具集供程序开发者使用。

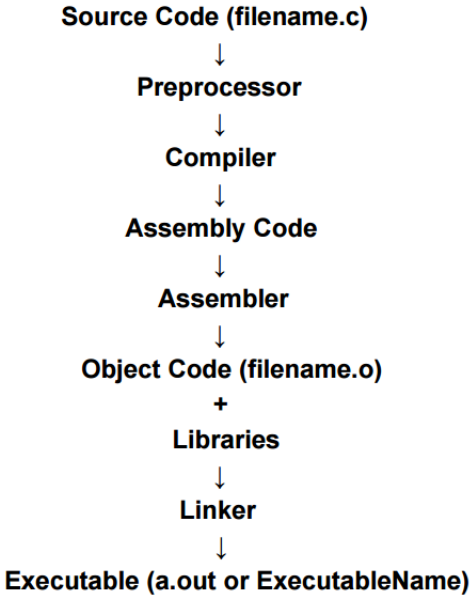

图 1.1: 编译器对输入的源文件的处理流程

当前 T-HEAD 有两套工具链，它们都基于 GNU 工具链开发，其中一套针对 CSKY 系列 CPU，另一套针对 RISC-V 系列 CPU。下文所叙的编译器在本文中特指 GCC/G++，汇编器为 GNU as，链接器为 GNU ld。为了简化起见，在不产生歧义的情况下，以下将交替地使用该类定义。

本章包含如下几个部分：

- 工具链组件版本及语言支持
- 通用选项说明
- 向汇编器、链接器传递选项

1.1 工具链组件版本及语言支持

当前的 CSKY 系列工具链的发布版本基于 GCC 6.3 开发, RISC-V 系列工具链基于 GCC 8.1 开发，它们对 C 语言版本的支持如 表 1.1 所示，对 C++ 语言版本的支持如 表 1.2 所示，可通过选项 -std=[语言标准名称] 选择语言标准。

表 1.1: C 语言标准支持

C 语言标准	CSKY	RISC-V
c90	yes	yes
gnu90	yes	yes
c99	yes	yes
gnu99	yes	yes
c11	yes	yes
gnu11	default	default

表 1.2: C++ 语言标准支持

C++ 语言标准	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

1.2 通用选项说明

为了避免对工具链全部命令行选项的赘述，本节将会介绍对于普通应用开发者使用最为频繁的部分命令行选项，详细的命令行选项介绍请参阅 GCC 命令行在线文档。

1.2.1 编译器命令

1. 控制输出的选项

-E

只执行 C/C++ 的预处理，不执行后续的语法分析，代码生成等过程。

-fsyntax-only

通常用于控制编译器只执行到语义检查，不执行后续过程。该选项通常用于测试输入源文件是否符合 C/C++ 语言标准，不产生任何文件。

-c

只让编译器生成目标文件（通常是后缀为.o 的文件），不执行后续的链接过程。

-S

控制编译器只生成汇编代码。

-v

查看编译器版本，并且打印出编译器编译的命令，汇编器命令和链接器命令。而且也会打印出头文件的搜索目录。

-###

该选项必须仅靠在 gcc 命令之后，功能与-v 选项类似，与-v 的差异是，该选项不会执行实际的编译、汇编、链接过程，只打印执行命令。开发者可以使用该选项获取得到编译器的编译命令，便于对源程序进行调试。注意，通常调试 gcc 并不能进入编译器的流程，gcc 是一个驱动控制程序，用于生成不同的命令来调用编译器 (对于 C 语言程序是 cc1，C++ 程序是 cc1plus)，汇编器 (as) 和链接器 (ld) 来实现所需的目的。

-version

显示所使用的 GCC 的版本号。

2. 语言标准

-std

该选项用于控制编译器所能支持的语言标准，当前平头哥编译器使用的默认语言标准是 gnu99(针对 C 语言) 和 gnu++14(针对 C++ 语言)。当然开发者可以通过该选项强制编译器使用特定的标准。

3. 调试格式支持

-g

控制编译器在目标代码或汇编代码中插入本机系统支持的调试信息 (默认为 dwarf 格式)，供 gdb 调试器使用。该选项通常用于程序开发阶段，便于辅助开发者进行调试与测试。建议在发布版本时关闭该选项。

-ggdb

控制编译器生成符合 gdb 调试器格式的调试信息。

-gdwarf

控制编译器生成符合 dwarf 调试格式的调试信息。

-gcoff

控制编译器生成 coff 格式的调试信息。

-gxcoff

控制编译器生成 gnu 扩充版 coff 格式的调试信息。

4. 控制诊断信息格式

当编译器解析输入的源码文件的时候，如果输入的源文件存在语法或者语义错误，编译器将会生成若干个诊断信息以提示开发者诊断信息发生的源文件位置和诊断类别 (fatal, error, warning)，并给出详细的诊断描述信息。

-fmessage-lengt=n

选项用于控制输出的诊断描述文本的宽度为 n 个字符。

-fdiagnostics-show-location=once

控制编译器是否只输出一次源码位置 (当后续诊断位置在同一文件，同一列时)。

-fdiagnostics-show-location=every-line

控制编译器为每个诊断位置输出完整地位置信息 (文件名: 行号: 列号)。

-fno-diagnostics-color

控制编译器不为诊断信息标注颜色。

-fno-diagnostics-show-option

控制编译器不生成诊断信息描述信息。

-fno-diagnostics-show-caret

控制编译器不产生 ^ 符号来指明产生的位置。

5. 优化选项

该类选项通常用于控制编译器的优化策略，告知编译器是否应该执行某些优化以提高程序的运行速度，如: 执行循环展开 (loop unrolling)，或者是否应该避免某些优化策略以减少目标文件的大小，从而使目标程序能够更加高效率的运行在小内存容量的嵌入式设备中。

-O0

GCC/G++ 默认的优化级别，不执行任何优化以减少编译时间，通常将会生成调试信息。

-O -O1

此两个选项意义相同，通常用于控制编译器执行部分收益较高的优化，同时达到减少代码大小和执行时间的目的。

-O2

该选项将会启用需要更多编译时间的能大幅提高程序执行速度的优化选项，注意，该选项通常会增加目标代码的大小。

-O3

该选项会开启 O2 中所有的选项，同时也会开启若干其他优化，以提高目标代码的性能。

-Os

该优化选项通常用于告知编译器在保持性能的前提下，尽可能的减少目标代码的大小。它将会从 O2 开启的全部选项中去除部分会增加目标代码大小的优化策略。

-Ofast

该选项通常用于告知编译器尽可能生成更快运行速度的目标代码，而不考虑目标代码大小。它将开启 O3 中所有的选项。

1.2.2 汇编器命令

对于大部分开发者来说，很少有直接使用汇编器的机会。所以，本节将介绍比较常用的选项，对于本节未涉及的选项，开发者可自行查询 CSKY 或 RISC-V 汇编器的开发手册。

-g --gen-debug

告知汇编器为目标代码生成可调试信息。

-o

指定输出的目标文件的文件名，默认为 a.out

1.2.2.1 CSKY 汇编器特定选项

-march= 架构

为指定架构的 CPU 生成目标代码，如: *-march=ck803* 将告知编译器生成 ck803 系列 CPU 支持的指令，而不会生成 ck810 系列 CPU 的指令。

-mcpu=CPU 型号

指定为待选择的 CPU 特性生成优化代码，如：*-mcpu=ck803er1* 将启用 dspv2 指令。

-m{no-}ljump

将 jbf、jbt、jbr 指令的目标地址超过指令的偏移范围，将指令转化成 jmpi 指令，默认关闭。

1.2.2.2 RISC-V 汇编器特定选项

-march= 架构

为指定架构的 CPU 生成目标代码，如: *-march=rv32imac* 将告知编译器生成包含基础整型指令集（'I' 指令集）、原子操作指令集（'A' 指令集）、压缩指令集（'C' 指令集）的 32 位 RISC-V 系列 CPU 的目标代码。

1.2.3 示例

本节将 CSKY 为例详细叙述上述每个选项的用法以及其作用，所讲述的方法同样适用于 RISC-V。本节使用 bar.h 和 bar.c 两个文件，bar.h 头文件中声明一个签名为 *int sum(int num)* 的函数，同时在 bar.c 中实现并调用该函数。bar.h 文件内容如下

```
#ifndef BAR_H
#define BAR_H

/**
 * 执行累加运算，计算从 1 累加至 len 的和，如果输入的 len 小于等于 0，
 * 返回 0。
 */
int sum(int len);
#endif
```

bar.c 文件代码内容如下

```
#include "bar.h"

int a = 30;
int b;

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

(下页继续)

(续上页)

```
}

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

通常的编译命令如下

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

上述命令执行之后，会在源文件所在的目录生成一个名字为 bar 的可执行文件，使用 qemu-system-cskyv2 模拟器 (该模拟器用于模拟执行 elf 格式的文件) 就能模拟执行该文件，相应的输出信息将会在标准输出设备 (通常是终端或命令行窗口) 上显示。

在某些开发场景中，如：对某个源码文件进行调试。通常该文件会包括很多个 include 头文件，此时开发者并不知道相应的头文件搜素目录。这种情况下就需要开发者懂得如何生成预处理文件，避免去寻找未知的头文件搜索目录。

要实现上述目的，开发者需要手动的向编译器传递-E 选项告知 gcc 编译器只需要执行预处理即可 (Preprocessing)，同时 gcc 编译器会生成一个与源文件同名但后缀为.i(针对 C 语言文件，如果源文件是 C++，则后缀为.ii)，命令如下

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

输出的预处理文件名为 bar.i，内容如下，很明显，编译器将会把头文件中对应的函数声明抽取至该函数的使用位置，从而实现 C/C++ 语言中一个重要的原则——变量或函数必须先声明再使用。

```
int sum(int len);

int a = 30;
int b;

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

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

(下页继续)

(续上页)

```
  return res;
}
```

在另外一些场景中，如：查看对应生成的汇编代码是否正确。除了使用 csky-elfabiv2-objdump 工具对生成的目标代码反汇编之外，另外一个更为直接的处理方式就是向编译器传递-S 选项，此时 gcc 编译器只会运行到汇编代码生成阶段，而不会继续调用汇编器和链接器来生成可执行代码。可以使用下列命令，生成的汇编问题见图 7。

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

生成的汇编文件 bar.s 中的内容如下

```
# 为了节省篇幅，省略 main 函数的代码并去除无关的调试代码
sum:
        subi    sp, sp, 4
        st.w    l4, (sp, 0)
        mov     l4, sp
        subi    sp, sp, 12
        subi    a3, l4, 12
        st.w    a0, (a3, 0)
        subi    a3, l4, 12
        ld.w    a3, (a3, 0)
        jbhz    a3, .L4
        movi    a3, 0
        jbr     .L5
.L4:
        subi    a3, l4, 4
        movi    a2, 0
        st.w    a2, (a3, 0)
        subi    a3, l4, 8
        movi    a2, 0
        st.w    a2, (a3, 0)
        jbr     .L6
.L7:
        subi    a3, l4, 4
        subi    a1, l4, 4
        subi    a2, l4, 8
        ld.w    a1, (a1, 0)
        ld.w    a2, (a2, 0)
        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
```

(下页继续)

(续上页)

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

在为了提高程序性能的情况下，开发者通常会开启-On(n >= 1) 选项以达到更好的程序运行性能。以上述汇编为例，当打开-O2 时，可以观察如下汇编代码发现程序的整体大小减少的非常明显，同时指令更加精简，如图：

```
# 为了节省篇幅，省略 main 函数的代码并去除无关的调试代码
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
```

同样的话，开启-Os 的时候，会发现与不开启优化对比，在生成的汇编文件中，指令数大幅减少。

类似的，其他选项都可以使用该方法进行测试以观察编译器的输出结果。

1.3 向汇编器、链接器传递选项

GCC 执行时默认包含预编译、编译、汇编、链接的过程，它会自动调用预处理器、汇编器、链接器。在某些情况下，开发者需要向各个组件传递选项，传递方法如 表 1.3 所述：

表 1.3: 各个组件的传递选项

GCC 选项	作用
-Wp [参数 1, 参数 2, …]	向预处理器传递参数
-Wa [参数 1, 参数 2, …]	向汇编器传递参数
-Wl [参数 1, 参数 2, …]	向链接器传递参数
-L [路径]	添加链接器查找库的路径

第二章　工具链错误与警告信息

为了能够更好地提高程序开发者的开发效率，提高程序运行时的稳定性，大多数工具链都会提供良好的程序检查机制，在非法，存在潜在安全或性能隐患的代码片段位置产生诊断信息，打印出易于程序开发者阅读和理解的诊断信息，包括错误信息 (error)、警告信息 (warning) 和部分修改建议 (note)。

本章将会着重介绍 CSKY 系列和 RISC-V 系列工具链输出的诊断信息的分类与格式，并使用若干例子说明如何根据诊断信息对代码片段进行修复 (本章使用 CSKY 系列编译器举例，但规则通用适用于 RISC-V 系列编译器)。

本章包含如下几个部分：

- 编译器错误与警告信息的格式
- 编译器诊断信息的选项
- 其他工具控制错误与警告信息的选项

2.1　编译器错误与警告信息的格式

编译器的错误与警告信息是最常用的且是大部分开发者日常接触的诊断信息类别，该类信息通常是由编译器的前端 (词法分析，语法分析，语义检查) 产生，用于告知开发者被编译的源程序中存在不符合 C 语言标准 (或 GNU 扩充语法) 的数字、标识符、语言结构和针对某个类型变量的非法操作。

2.1.1　错误信息格式

以如下代码片段 (文件名为 bar.c) 为例：

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

使用如下 shell 命令编译，*-fsyntax-only* 选项用于告知编译器只执行前端动作，不执行后端优化和代码生成。

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

上述命令产生的错误信息为：

```
bar.c:2:9: error: initializer element is not constant
int b = a;
        ^
```

错误信息格式将按照如下形式进行显示:

```
bar.c:2:9: error: initializer element is not constant
 |    | |     |                     |
 |    | |     |                     |_____________________ 诊断说明
 |    | |     |_____________________________________________ 表明错误类别的诊断
 |    | |_____________________________________________________ 诊断信息所在的列号
 |    |_______________________________________________________ 诊断信息所在的行号
 |_____________________________________________________________ 诊断发生的源文件名
```

2.1.2 警告诊断格式

```
/* Test function. */
foo(int *ptr)
{
  ptr = ptr + 2;
  return *ptr;
}
```

使用如下 shell 命令编译，*-fsyntax-only* 选项用于告知编译器只执行前端动作，不执行后端优化和代码生成。

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

上述命令产生的错误信息为:

```
foo.c:2:1: warning: return type defaults to 'int' [-Wimplicit-int]
foo(int *ptr)
^~~
```

警告诊断信息格式将按照如下形式进行显示:

```
foo.c:1:1: warning: return type defaults to 'int' [-Wimplicit-int]
 |    | |     |                     |
 |    | |     |                     |_____________________ 诊断说明
 |    | |     |_____________________________________________ 表明该诊断是一个警告信息
 |    | |_____________________________________________________ 该警告发生源文件第 1 列
 |    |_______________________________________________________ 该警告发生源文件第 1 行
 |_____________________________________________________________ 该警告发生在源文件 foo.c
```

2.1.3 其他诊断信息格式

编译器除了报告错误和警告信息之外，通常也会在部分情况下提供一定程度的错误修复建议，告知程序开发者应该如何去修复该错误。此处使用如下代码片段用于说明该情况，在下列代码中，main 函数中调用了 malloc 函数在堆上分配了 10 字节大小的空间，然后给其赋值为 1。注意，代码中并没有 include<stdlib.h>。

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

使用 csky-elfabiv2-gcc 编译得到如下诊断信息

```
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'
```

从上述诊断信息可以看出，除了警告 (warning) 信息之外，还有一个 note 信息，该信息通常建议开发者如何使用 include '<stdlib.h>' 去修复该问题。

2.2 编译器诊断信息的选项

通常在进行嵌入式开发的时候，为了尽可能地降低程序中存在潜在漏洞的风险，都需要确保编译源程序的时候不存在任何警告和错误。但是相当一部分开发者会忽略编译器产生的警告信息，而且更重要的是当编译器产生警告信息的时候，编译过程会照常继续。从而掩盖了将来可能发生的隐患。为了避免这一情况，编译器提供了一系列选项来满足该类需求，其中最重要的一项则是 -Werror，该选项将会将所有的警告诊断信息转换为错误信息显示，从而避免该错误的隐藏。

同时，编译器也提供一些选项控制编译器生成警告信息，提示程序开发者需要注意的地方，此时使用-Wall 选项是比较合适的，该选项将会把所有警告信息开启，而不是默认的关闭 (默认状态是否输出警告信息取决于编译器版本)。不过有时候开发者只需要把特定的警告信息转换为错误信息显示，编译器为每个警告选项都提供了-Wxxx(xxx 为警告选项的名字，如上节提到的 implicit-function-declaration) 选项用来开启特定的警告信息。

某些历史遗留代码为了兼容不同的 GCC 版本，会将某些警告信息显示为错误消息。为了避免较新版本的 GCC 编译失败，可以使用-Wnoxxx(xxx 为警告选项的名字) 来关闭该类警告信息，如：-Wnoimplicit-function-declaration。

2.3 使用 pragma 预处理命令控制错误与警告信息

除了通过命令行选项来控制编译器的诊断信息输出之外，编译器也支持通过 #pragma 预处理选项在源文件中以更加灵活地控制诊断信息的显示。除此之外，还可以通过 #pragma 输出自定义的错误或警告信息，也可以有选择的关闭全部或指定的部分诊断选项。

2.3.1 #pragma GCC error

该选项用于程序开发者在源文件中设置自定义的错误诊断信息，如下列代码片段 (文件名是 pragma-error.c)：

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

使用 csky-elfabiv2-gcc 命令编译，显示的错误信息为：

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

从上述结果可以看出，#pragma GCC error 可以让编译器灵活的输出特定的、自定义的错误信息。

2.3.2 #pragma GCC warning

类似于 #pragma GCC error，warning 选项用于告知编译器生成特定的、自定义的警告信息，显示的格式也基本一样。同样以如下代码片段为例说明该情况。

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

使用 csky-elfabiv2-gcc 命令编译，显示的警告信息为：

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

2.3.3 #pragma message

该选项仅仅用于输出一个编译器注意诊断信息，并不是警告或错误信息。

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

使用 csky-elfabiv2-gcc 命令编译，显示的警告信息为：

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

2.3.4 #pragma GCC diagnostics

上述的三个 #pragma 子类通常用于控制编译器输出自定义的诊断信息，但是该选项则用于告知编译器在编译某个源文件时，遇见该命令就将某个警告选项忽略、显示、或者按照错误显示，如下例子。

```
#pragma GCC diagnostic ignored "-Wimplicit-int"
bar()   // 此处本应出现" 返回类型默认为 'int' "
{
  #pragma GCC diagnostic warning "-Wimplicit-function-declaration"  // 开启-Wimplicit-
↪function-declaration 选项
  int *ptr = (int*)malloc(sizeof(int));   // 显示警告
  *ptr = 1;
  #pragma GCC diagnostic error "-Wimplicit-function-declaration"    // 将-Wimplicit-
↪function-declaration 作为错误显示
  memset(ptr, 0, sizeof(int));  // 将 warning 显示为错误
  return 0;
}
```

使用 csky-elfabiv2-gcc 编译出现如下诊断信息，观察下列信息可以明显地发现上述预处理指令所发挥的作用。

```
implicit.c: 在函数 'bar' 中:
implicit.c:5:20: 警告: 隐式声明函数 'malloc' [-Wimplicit-function-declaration]
   int *ptr = (int*)malloc(sizeof(int));   // 显示警告
                    ^~~~~~
implicit.c:5:20: 警告: 隐式声明与内建函数 'malloc' 不兼容
implicit.c:5:20: 附注: include '<stdlib.h>' or provide a declaration of 'malloc'
implicit.c:8:3: 错误: 隐式声明函数 'memset' [-Werror=implicit-function-declaration]
   memset(ptr, 0, sizeof(int));  // 将 warning 显示为错误
   ^~~~~~
implicit.c:8:3: 警告: 隐式声明与内建函数 'memset' 不兼容
implicit.c:8:3: 附注: include '<string.h>' or provide a declaration of 'memset'
```

2.4　其他工具控制错误与警告信息的选项

2.4.1　汇编器控制诊断信息的选项

开发者在某些情况下在单独使用汇编器的时候，如编译器一样，也需要控制输出诊断信息的输出行为，如：不输出警告信息或将警告信息作为错误显示。

1. -W

隐藏警告信息

2. –warn

不隐藏警告信息

3. –fatal-warnings

把警告作为错误显示

4. -Z

有错误也生成目标文件

第三章　玄铁 800 系列 CPU 编程

玄铁 800 系列 CPU 是基于 CSKY 体系结构开发的处理器。本章主要介绍在 C、C++ 编程过程当中，涉及到与 CSKY 体系结构相关的特殊用法，如 cpu 选择、指令集选择、汇编编程、vdsp 指令 intrinsic 接口、以及 minilibc 等。

本章包含如下几个部分：

3.1　如何选择体系结构、处理器

当前我们 csky 支持多种 ABI 标准，根据不同的 cpu 型号，需要选择不同的 gcc 工具链进行编译/开发，如 表 3.1 所示。

表 3.1: 各种 cpu 对应 gcc 工具链

	abiv1	abiv2
elf	csky-elf-gcc	csky-elfabiv2-gcc
linux/glibc	csky-linux-gnu-gcc	csky-linux-gnuabiv2-gcc
linux/uclibc	csky-linux-uclibc-gcc	csky-linux-uclibcabiv2-gcc

我们可以通过 gcc 工具，查看当前工具所支持的所有 csky 的 cpu 型号，在命令行执行如下命令：

```
csky-elfabiv2-gcc --target-help
```

然后在执行结果中可以查看如下相关 arch、cpu 以及浮点运算相关支持信息：

```
Known CSKY architectures (for use with the -march= option):
ck801 ck802 ck803 ck807 ck810 ck860 native
```

(下页继续)

(续上页)

```
Known CSKY FPUs (for use with the -mfpu= option):
auto fpv2 fpv2_divd fpv2_sf fpv3 fpv3_hf fpv3_hsf fpv3_sdf

Known CSKY CPUs (for use with the -mcpu= options):
ck801 ck801t ck802 ck802j ck802t ck803 ck803e ck803ef ck803efh
ck803efhr1 ck803efhr2 ck803efht ck803efhtr1 ck803efhtr2 ck803efr1
ck803efr2 ck803eft ck803eftr1 ck803eftr2 ck803eh ck803ehr1 ck803ehr2
ck803eht ck803ehtr1 ck803ehtr2 ck803er1 ck803er2 ck803et ck803etr1
ck803etr2 ck803f ck803fh ck803fhr1 ck803fhr2 ck803fr1 ck803fr2 ck803ft
ck803ftr1 ck803ftr2 ck803h ck803hr1 ck803hr2 ck803ht ck803htr1
ck803htr2 ck803r1 ck803r2 ck803s ck803se ck803sef ck803seft ck803sf
ck803st ck803t ck803tr1 ck803tr2 ck807 ck807e ck807ef ck807f ck810
ck810e ck810ef ck810eft ck810et ck810f ck810ft ck810ftv ck810fv ck810t
ck810tv ck810v ck805 ck805e ck805ef ck805eft ck805et ck805f ck805ft ck805t
ck860 ck860f ck860fv ck860v native

Known floating-point ABIs (for use with the -mfloat-abi= option):
hard soft softfp
```

csky cpu 型号的表示，遵循一套基本的命名规则，如：

```
CK810  X4  CEFHMTV   R2
 |     |     |       |
 |     |     |       |
 |     |     |       |_____________________________ R:Revision  2：CPU 的第二个版本
 |     |     |_____________________________________ 增强指令集 A-Z
 |     |___________________________________________ X：多核  4：4 个核
 |                                                  注：X1 单核不需要表示
 |__________________________________________________ 该警告发生在源文件 foo.c
```

增强指令集符号的含义如 表 3.2 所示：

表 3.2: 增强指令集符号的含义

增强指令集符号	说明	说明
C	Crypto enhance	加密增强
E	EDSP	DSP 增强
F	FPU	浮点
H	Shield	物理抗攻击
M	Memory enhance	存储增强
T	TEE	可信执行环境
V	VDSP	向量 DSP

一般情况下，我们编译某些工程，通过编译选项-mcpu 指定相应的 cpu 进行编译：

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

在某些情况下，也可以通过增加一些 csky cpu 特性开关选项来使用，如使用硬件浮点运算功能：

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

3.2　指令集简介

上一章节已经介绍了 CSKY 拥有的体系结构，每个体系结构对应的基本指令集可参见《CSKY CPU 指令实现参考手册》。除了基本指令集之外，CSKY 两套 dsp 指令集、两套 vdsp 指令集、三套浮点指令集，具体参见下面的章节。

3.2.1　dsp 指令集

dsp 指令集有两套，分别为：dsp 1.0 和 dsp 2.0，指令集和 CPU 的对应关系如 表 3.3 所示：

表 3.3: dsp 指令集和 cpu 的对应关系

CPU 型号	dsp 指令集版本
ck803 系列带‘e’标签的 CPU	dsp 1.0
ck803r1 以上（包含 r1）带‘e’标签的 CPU	dsp 2.0
ck804 系列 CPU	dsp 2.0
ck807 系列 CPU	dsp 1.0
ck810 系列 CPU	dsp 1.0

3.2.2　vdsp 指令集

vdsp 指令集有两套，分别为：vdspv1 和 vdspv2，指令集和 CPU 的对应关系如 表 3.4 所示：

表 3.4: vdsp 指令集和 cpu 的对应关系

CPU 型号	vdsp 指令集版本
ck805 系列 CPU	vdspv2
ck810 系列带‘v’标签的 CPU	vdspv1
ck860 系列带‘v’标签的 CPU	vdspv2

3.2.3　浮点指令集

浮点指令集有三套，分别为：fpuv1、fpuv2 和 fpuv3，指令集和 CPU 的对应关系如 表 3.5 所示：

表 3.5: 浮点指令集和 cpu 的对应关系

CPU 型号	浮点指令集版本
ck610 系列带‘f’标签的 CPU	fpuv1
ck803 系列带‘f’标签的 CPU	fpuv2 单精度浮点
ck804 系列带‘f’标签的 CPU	fpuv2 单精度浮点
ck805 系列带‘f’标签的 CPU	fpuv2 单精度浮点
ck807 系列带‘f’标签的 CPU	fpuv2
ck810 系列带‘f’标签的 CPU	fpuv2
ck860 系列带‘f’标签的 CPU	fpuv3

注解： 如果需要编译器编译出硬件浮点指令，除了添加正确的 cpu 型号之外，还需要添加额外的选项，具体见*如何使用硬浮点指令*

3.2.4　CPU 的版本和基础指令集

不同的 CPU 版本有不同的基础指令集，具体如 表 3.6 所示：

表 3.6: CPU 的版本和基础指令集的关系

CPU 型号	基础指令集说明
ck803r1	在 ck803 基础指令集中增加了：mul.u32 mul.s32 mula.u32 mula.s32 mula.32.l mulall.s16.s
ck803r2	在 ck803r1 基础指令集中增加了：bnezad
ck803r3	在 ck803r2 基础指令集中增加了：divul divsl

3.3　如何使用硬浮点指令

当前我们某些 cpu 是支持硬件浮点运算单元的，如何让 gcc 编译器生成带硬件浮点指令的代码呢？可以根据*csky cpu 特性* 来控制编译器生成含有硬件浮点指令的代码：

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

目前我们支持多种浮点运算 ABI 规则，通过**-mfloat-abi** 编译选项进行控制，详见*相关帮助信息*：

soft： 使用软件浮点运算

hard： 使用硬件浮点运算

softfp： 同 hard 选项，但是参数、返回值不使用浮点寄存器

注解： 针对浮点控制的选项，编译器对老版本做了兼容，-msoft-float 等同于-mfloat-abi=soft，-mhard-float 等同于-mfloat-abi=hard。

3.4　使用内联汇编

3.4.1　asm 格式

CSKY GCC 的内联汇编基本格式符合 GNU gcc 的基本语法，使用“**asm**”关键字指出使用汇编语言编写的源代码段落。asm 段的基本格式如下：

asm (“assembly code”);

举例：

```
/* 把 r1 中的值赋给 r0 */
asm("mov r0, r1");

/* 多条内联汇编 */
asm("mov r0, r1\nmov r1, r0");

/* 多条内联汇编，并使用可选的\t 让生成的汇编代码更友好 */
asm("mov r0, r1\n\tmov r1, r0");
```

包含在括号中的汇编代码必须按照特定的格式：

- 指令必须括在引号里
- 如果包含的指令超过一条，那么必须使用新行字符分隔汇编语言代码的每一行。通常，还包含制表符帮助缩进汇编语言代码，使代码行更容易阅读。

需要第二个规则是因为编译器逐字地取得 asm 段中的汇编代码，并且把它们放在为程序生成的汇编代码中去。每条汇编语言指令都必须在单独的一行中——因此需要包含新行字符。

注解： 如果不希望编译器优化内嵌汇编，可添加 volatile 关键字阻止编译器优化，即 asm volatile (“assembly code”)

3.4.2　扩展 asm 格式

基本的 asm 格式提供创建汇编代码的简单方式，但是有其局限性：

- 所有的输入值和输出值都必须使用 c 程序的全局变量。
- 必须极为注意在内联汇编代码中不去改变任何寄存器的值。

gcc 编译器提供 asm 段的扩展格式来帮助解决这些问题。asm 扩展版本格式如下：

asm (“assembly code” : output locations : input operands : changed registers);

这种格式由 4 个部分构成，使用冒号分隔：

- 汇编代码 (assembly code)：使用和基本 asm 格式相同的语法的内联汇编代码

- 输出位置 (output locations)：包含内联汇编代码的输出值的寄存器和内存位置的列表，格式见：指定输入值和输出值
- 输入操作数 (input operands)：包含内联汇编代码的输入值的寄存器和内存位置的列表，格式见：指定输入值和输出值
- 改动的寄存器 (changed registers)：内联代码改变的任何**其他**寄存器的列表

在扩展 asm 格式中，不是所有的这些部分都必须出现。如果汇编代码不生成输出值，这个部分就必须为空，但是必须使用两个冒号把汇编代码和输入操作数分隔开。如果内联汇编代码不改动寄存器的值，那么可以忽略最后的冒号。举例：

```
int a=10, b;
asm ("mov r1, %1\n\t"
 "mov %0, r1"
 : "=r"(b)
 : "r"(a)
 : "r1");
```

3.4.2.1 指定输入值和输出值

输入值和输出值列表的格式是：

“constraint”(variable)

其中 variable 是程序中声明的 c 变量。在扩展 asm 格式中，局部和全局变量都可以使用。constraint 定义把变量存放在哪里（对于输入值）或者从哪里传送变量（对于输出值）。使用它定义把变量存放在寄存器中还是内存位置中。

约束是单一字符的代码，详见：*gcc 约束相关代码*

除了这些约束之外，输出值还包含一个约束修饰符，它指示编译器如何处理输出值，详见：*gcc 输出修饰符*

3.5 汇编语言编程

有些开发者需要手写汇编文件并将其编译成目标文件，一般使用如下基本命令：

```
csky-elfabiv2-gcc -c [输入汇编文件名] -o [输出目标文件文件名]
```

本章包含如下几个部分：

- 汇编指令格式
- 预处理汇编文件
- 汇编伪指令

3.5.1　汇编指令格式

汇编指令的格式分为指令名称和操作数名称两部分，中间用空格分隔，如下:

```
>                                指令名称 操作数 1, 操作数 2, …
```

其中，操作数的类型如 表 3.7 :

表 3.7: 汇编指令中操作数的类型

操作数类型	书写格式	示例
通用寄存器	通用寄存器名称，详见寄存器别名	abs r1
v1 浮点寄存器	fr0-fr31	fabss fr1
v2 浮点寄存器	vr0-vr15	fabss vr0-vr15
v2 向量寄存器	同上，abiv2 中浮点模块和向量模块使用同一组寄存器	vabs.8 vr1
带立即数偏移的内存地址	(rx, offset)	ld.w r1, (r2, 4)
带寄存器索引的内存地址	(rx, ry « n)	ldr.w r1, (r3, r2 « 1)
地址引用	符号名称	bsr functionname
控制寄存器	cr<z, sel>（第 sel 组，第 z 号寄存器）	mtcr r1, cr<0, 0>
通用寄存器序列	rx-ry, rz…	push r4-r11,r15

注解: 一般情况下，目的操作数都书写在源寄存器之前，除了 st.[bhw]、str.[bhw]、mtcr 指令之外。

3.5.2　预处理汇编文件

当汇编文件包含一些 C 语言的宏指令（如 #define、#include、#if 等）和注释时，它必须经过预处理。

GCC 根据汇编文件的后缀名判断它是否需要预处理:

- 当后缀名为（.S）时，表示文件包含宏指令需要被预处理
- 当后缀名为（.s）时，表示文件只包含汇编指令不需要被预处理

比如一个包含宏指令的汇编文件（test.S）如下所示:

```
#define P 2          /* 与 C 语言语法一样，宏定义 */
movi t0,P
```

通过 gcc 添加-E 选项可以得到预处理之后的汇编文件，命令和生成的文件如下所示:

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

文件 test.s:

```
movi t0, 2
```

注解: 不要将 #include、#if 等和.include、.if 等混淆在一起，#include、#if 等是 C 语言宏指令需要被预处理器处理，而.include、.if 等是汇编指令只需要被汇编器处理。

3.5.3 汇编伪指令

汇编源程序中，除了汇编指令，还包含伪指令，伪指令在 CPU 指令集没有对应的指令。汇编伪指令可以扩展成一个或多个的汇编指令，使用伪指令的原因主要分为三种情况：

1. 由于跳转指令的目标地址相对于指令本身的偏移距离不确定，导致使用哪种跳转指令需要汇编器决定；
2. 将一些指令的书写变得更为简洁；
3. C-SKY V2.0 的汇编指令能兼容 C-SKY V1.0 的汇编指令。

汇编伪指令如 表 3.8：

表 3.8: 汇编伪指令

伪指令	扩展后的指令	描述	CPU
clrc	cmpne r0,r0	将 C 位清零	全部
cmplei rd,n	cmplti rd, n+1	立即数有符号的比较 用小于兼容小于等于	全部
cmpls rd,rs	cmphs rs, rd	立即数无符号的比较 用大于等于兼容小于等于	全部
cmpgt rd,rs	cmplt rs, rd	立即数有符号的比较 用小于兼容大于等于	全部
jbsr label	abiv1: bsr label 或 jsri label abiv2: bsr label	跳转到子程序	全部
jbr label	abiv1: br label 或 jmpi label abiv2: br label	无条件跳转	全部

下页继续

表 3.8 – 续上页

伪指令	扩展后的指令	描述	CPU
jbf label	abiv1: bf label 或 bt 1f jmpi label 1:… abiv2: bf label（16/32 位） 或 bt 1f（16 位） br/jmpi label（32 位） 1:…	C 位为 0 跳转	全部
jbt label	abiv1: bt label 或 bf 1f jmpi label 1:… abiv2: bt label（16/32 位） 或 bf 1f（16 位） br/jmpi label（32 位） 1:…	C 位为 1 跳转	全部
rts	jmp r15	从子程序返回	全部
neg rd	abiv1: rsubi rd,0 abiv2: not rd, rd addi rd, 1	取相反数	全部
rotlc rd,1	addc rd,rd	带进位的加法	全部
rotri rd,imm	rotli rd,32-imm	立即数循环左移	全部
setc	cmphs r0,r0	设置 C 位	全部
tstle rd	cmplti rd,1	测试寄存器的值是非正数	全部
tstlt rd	btsti rd,31	测试寄存器的值是负数	全部
tstne rd	cmplnei rd,0	测试寄存器的值是非零数	全部
bgeni rz,imm	movi rz,immpow immpow 为 2 的 imm 次幂	将寄存器的第 imm 位置 1, 其他位置 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

下页继续

表 3.8 – 续上页

伪指令	扩展后的指令	描述	CPU
mov rz,rx	mov rz,rx 或 lsli rz,rx,0	rz=rx 若 rz 和 rx 都为 r0~r15, 为 mov 若 rz 或 rx 为 r16~r31, 为 lsli	V2.0
movf rz,rx	incf rz,rx,0	如果 C 位为 0,rz=rx	V2.0
movt rz,rx	inct rz,rx,0	如果 C 位为 1,rz=rx	V2.0
not rz,rx	nor rz,rx,rx	按位取非	V2.0
rsub rz,rx,ry	subu rz,ry,rx	rz=ry-rx	V2.0
rsubi rx,imm16	movi r1,imm16 subu rx,r1,rx	rz=imm16-rx	V2.0
sextb rz,rx	sext rz,rx,7,0	取 rx 的第一个字节，并有符号扩展给 rz	V2.0
sexth rz,rx	sext rz,rx,15,0	取 rx 的第一个字，并有符号扩展给 rz	V2.0
zextb rz,rx	zext rz,rx,7,0	取 rx 的第一个字节，并无符号扩展给 rz	V2.0
zexth rz,rx	zext rz,rx,15,0	取 rx 的第一个字，并无符号扩展给 rz	V2.0
lrw rz,imm32	movih rz,imm32_hi16 ori rz，rz,imm32_lo16	加载 32 位的立即数到寄存器	V2.0
jbez rx,label	bez rx,label 或 bnez rx,1f br/jmpi label（32 位） 1:…	若 rx 等于零，跳转到子程序	v2.0
jbnez rx,label	bnez rx,label 或 bez rx,1f br/jmpi label（32 位） 1:…	若 rx 不等于零，跳转到子程序	v2.0
jbhz rx,label	bhz rx,label 或 blsz rx,1f br/jmpi label（32 位） 1:…	若 rx 大于零，跳转到子程序	v2.0
jblsz rx,label	blsz rx,label 或 bhz rx,1f br/jmpi label（32 位） 1:…	若 rx 小于等于零，跳转到子程序	v2.0

下页继续

表 3.8 – 续上页

伪指令	扩展后的指令	描述	CPU
jblz rx,label	blz rx,label 或 bhsz rx,1f br/jmpi label（32 位） 1:…	若 rx 小于零，跳转到子程序	v2.0
jbhsz rx,label	bhsz rx,label 或 blz rx,1f br/jmpi label（32 位） 1:…	若 rx 大于等于零，跳转到子程序	v2.0

3.5.4　寄存器别名

许多通用寄存器（r0-r31）被支持别名，这些别名有助于在一定的情况下提高汇编代码的可读性和兼容性，如 表 3.9 和 表 3.10 所示：

表 3.9: CSKY ABI V1 寄存器别名

V1 寄存器名	别名	描述
r2-r3	a0-a1	传参/传递返回值
r4-r7	a2-a5	传参
r8-r13	l0-l5	存储局部变量（使用时需要在函数头尾保存和恢复）
r14	l10/gb	存储局部变量/存储使用 PIC 选项编译时的 GOT 表基地址
r15	lr	存储返回地址
r16-r19	l6-l9	存储局部变量（使用时需要在函数头尾保存和恢复）
r20-r25	t0-t5	存储临时数据（使用时不需要在函数头尾保存和恢复）
r31	tls	TLS 寄存器

表 3.10: CSKY ABI V2 寄存器别名

V2 寄存器名	别名	描述
r0-r1	a0-a1	传参/传递返回值
r2-r3	a2-a3	传参
r4-r11	l0-l7	存储局部变量（使用时需要在函数头尾保存和恢复）
r12-r13	t0-t1	存储临时数据（使用时不需要在函数头尾保存和恢复）
r14	sp	存储栈指针
r15	lr	存储返回地址
r16-r17	l8-l9	存储局部变量（使用时需要在函数头尾保存和恢复）
r18-r25	t2-t9	存储临时数据（使用时不需要在函数头尾保存和恢复）
r28	rgb/rdb	存储 data section 基地址/存储使用 PIC 选项编译时的 GOT 表基地址
r29	rtb	存储 text section 基地址
r30	svbr	存储 handler 基地址
r31	tls	TLS 寄存器

注解: 传参寄存器在传参没有使用到时也可以当作临时寄存器使用，使用时不需要在函数头尾保存和恢复。

3.6　vdsp

目前，CSKY 体系结构支持两个版本的 VDSP 指令集，分别是 vdspv1 和 vdspv2。其中 vdspv1 可配置 64 位和 128 位两种位宽，编译器通过选项-mvdsp-width=<size>（默认 128）控制生成目标代码的位宽；vdspv2 位宽为 128 位。ck810 使用 vdspv1，ck860 和 ck805 使用 vdspv2。

编译器根据 CPU 选项判断生成的目标程序是否支持 vdsp 指令，其中 ck810 支持 vdsp 的 CPU 为:

ck810v，ck810fv，ck810tv，ck810ftv (即 810 包含 v 的 cpu)。

ck860 支持 vdsp 的 CPU 为:

ck860v ck860fv (即 860 包含 v 的 cpu)

ck805 的所有 CPU 都支持 vdsp。

在下面几种形况下，编译器会生成向量指令:

- 向量运算表达式
- 循环优化
- 使用 intrinsic 函数

其中，前两种针对比较基本的场景，而第三种则适用于需要深度优化的场景。详细的说明可参考本章节的下面几个部分:

- 向量数据类型
- 向量类型的参数和返回值的传递规则
- 向量运算表达式
- 循环优化生成向量指令
- *intrinsic* 函数接口命名规则
- *vdspv2* 的 *intrinsic* 接口

3.6.1　向量数据类型

向量数据类型通常建立在普通数据类型的基础之上，例如向量数据类型 int8x8_t 表示元素为 8 位的整型数据类型、由 8 个元素组成的类型，它的总位宽为 64 位。该命名规则如下所示:

```
> [元素类型][元素位宽]x[元素个数]_t
```

其中的元素类型为 int、uint 或 float。使用时需要引用头文件 csky_vdsp.h，vdspv1、vdspv2 支持的向量数据类型如 表 3.11 、表 3.12 所示:

表 3.11: vdspv1 的向量数据类型

vdspv1	64 位	128 位
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

表 3.12: vdspv2 的向量数据类型

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

3.6.2　向量类型的参数和返回值的传递规则

在默认情况下或者开启选项-mfloat-abi=soft/softfp 时，向量类型的参数和返回值仍使用普通寄存器传递。

开启选项-mfloat-abi=hard 时, 向量类型的参数和返回值不再使用普通寄存器传递。向量类型的参数通过寄存器 vr0-vr3 传递，当向量类型参数超出 4 个时，将通过堆栈传递剩余的参数。向量类型的返回值通过寄存器 vr0 传递。

3.6.3　向量运算表达式

编译器支持向量运算表达式，它由向量类型的变量和运算符组成。编译器会根据这些表达式生成相应的向量指令。

3.6.3.1　向量类型变量的定义

向量类型变量的定义有两种方式：

- 第一种方式和数组定义的方式相同，如：

```
#include<csky_vdsp.h>

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

- 第二种方式，先定义一个数组，再将数组地址转化成向量指针类型，如：

```
#include<csky_vdsp.h>

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

3.6.3.2　运算符

C 语言使用运算符来表示算数运算，对于向量类型的变量也是如此。

目前，向量表达式所支持的运算符如下所示：

- 加法：+
- 减法：-
- 乘法：*
- 比较运算符：>，<，!=，>=，<=，==
- 逻辑运算符：&，|，^
- 移位运算符：»，«

下面是一个简单的示例：

```
#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;
}
```

3.6.4　循环优化生成向量指令

编译器支持将部分循环优化生成向量指令。当满足下面几个条件时，编译器会尝试将循环优化成向量指令：

- 当前 CPU 支持向量指令
- 优化等级是-O1 或者-O1 以上，并且添加选项-ftree-loop-vectorize

（-O3 时默认开启此选项）

例如下面的循环：

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

如果当前 CPU 支持 128 位的向量加法指令，在开启循环优化后，上述代码优化后的代码如下面的伪代码所示：

```
void svfun2 (int32x4_t va,int32x4_t vb,int32x4_t vc)
{
    vc = va + vb;    /* 向量运算 */
}
```

3.6.5　intrinsic 函数接口命名规则

intrinsic 接口的函数名称与指令名称基本保持一致，如果指令名称中包含“.”，则在函数名称中会替换成“_”，例如指令 vmfvr.u32 对应的 intrinsic 接口函数名称位 vmfvr_u32。函数的参数和返回值类型由指令操作数的数据类型决定，例如指令 vmfvr.u32 rz,vr[index]，它的功能是将向量寄存器中的第 index 个元素传送到普通寄存器 rz 中，因此函数 vmfvr_u32 的声明如下：

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

其中第一个参数是 uint32x4_t 类型，第二个参数是 int32_t 类型，返回值是 uint32_t 类型。

3.6.6　vdspv2 的 intrinsic 接口

目前，有以下几种 CPU 支持 vdspv2 指令的编译：

- ck860v ck860fv (即 860 包含 v 的 cpu)
- ck805 的所有 CPU

vdspv2 的指令可分为以下几个部分：

- 整型加减法、比较指令
- 整型乘法指令
- 整型倒数、倒数开方、*e* 指数快速运算及逼近指令
- 整型移位指令
- 整型移动 *(MOV)*、元素操作、位操作指令
- 整型立即数生成指令
- *LOAD/STORE* 指令
- 浮点加减法比较指令
- 浮点乘法指令
- 浮点倒数、倒数开方、*e* 指数快速运算及逼近指令
- 浮点转换指令

3.6.6.1　整型加减法、比较指令

vadd.t && vsub.t

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

```
>>> 函数说明：向量加法
    假设参数 Vx,Vy, 返回值 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)

```
>>> 函数说明：向量减法
    假设参数 Vx,Vy, 返回值 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)

```
>>> 函数说明：向量无符号扩展加法
    首先参数零扩展，其次向量加法
    假设参数 Vx,Vy, 返回值 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)

```
>>> 函数说明：向量有符号扩展加法
    首先参数有符号扩展，其次向量加法
    假设参数 Vx,Vy，返回值 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)

```
>>> 函数说明：向量无符号扩展减法
    首先参数零扩展，其次向量减法
    假设参数 Vx,Vy，返回值 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)

```
>>> 函数说明：向量有符号扩展加法
    首先参数有符号扩展，其次向量减法
    假设参数 Vx,Vy，返回值 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)

```
>>> 函数说明：向量高位加法
    加法结果取元素高半部分，按序放入返回值向量的低半部分
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量高位加法
    减法结果取元素高半部分，按序放入返回值向量的低半部分
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量饱和加法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 表示有无符号
    signed=(T==S);  (根据元素 U/S 类型选择)
    Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;
    Min=signed ? -2^(element_size-1) : 0;
    If Vx(i)+Vy(i)>Max    Vz(i)=Max;
    Else if Vx(i)+Vy(i)<Min    Vz(i)=Min;
    Else Vz(i)= Vx(i)+Vy(i);
    End        i=0:(number-1)
```

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

```
>>> 函数说明：向量饱和减法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 表示有无符号
    signed=(T==S);  (根据元素 U/S 类型选择)
    Max=signed ? 2^(element_size-1)-1 : 2^(element_size)-1;
    Min=signed ? -2^(element_size-1) : 0;
    If Vx(i)-Vy(i)>Max    Vz(i)=Max;
    Else if Vx(i)-Vy(i)<Min    Vz(i)=Min;
    Else Vz(i)= Vx(i)-Vy(i);
    End        i=0:(number-1)
```

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

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

```
>>> 函数说明：加法结果带 rounding 取高半部分
    假设 Vx,Vy 是两个参数，Vz 是返回值
    round=1<<(element_size/2-1);
    Tmp(i)=(Vx(i)+Vy(i)+round) [element_size-1:element_size/2];    i=0:(number-1)
    (加法结果带 rounding 取高半部分)
    Vz(i)={Tmp(2i+1), Tmp(2i)};    i=0:number/2-1
    结果按序放至目的寄存器 Vz 的低半部分 (默认)
```

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

```
>>> 函数说明：减法结果带 rounding 取高半部分
    假设 Vx,Vy 是两个参数，Vz 是返回值
    round=1<<(element_size/2-1);
    Tmp(i)=(Vx(i)-Vy(i)+round) [element_size-1:element_size/2];    i=0:(number-1)
    (加法结果带 rounding 取高半部分)
    Vz(i)={Tmp(2i+1), Tmp(2i)};    i=0:number/2-1
    结果按序放至目的寄存器 Vz 的低半部分 (默认)
```

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)

```
>>> 函数说明：加法平均运算
    假设 Vx,Vy 是两个参数，Vz 是返回值,U/S 为符号位
    Vz(i)=(Vx(i)+Vy(i))>>1;  i=0:number-1
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

- 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)

```
>>> 函数说明：减法平均运算
    假设 Vx,Vy 是两个参数，Vz 是返回值,U/S 为符号位
    Vz(i)=(Vx(i)-Vy(i))>>1;  i=0:number-1
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

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)

```
>>> 函数说明：加法平均并舍入运算
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 为符号位
    Vz(i)=(Vx(i)+Vy(i)+1)>>1;  i=0:number-1
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

- 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)

```
>>> 函数说明：减法平均并舍入运算
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 为符号位
    Vz(i)=(Vx(i)-Vy(i)+1)>>1;  i=0:number-1
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

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)

```
>>> 函数说明：扩展加法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 为符号位
    Vz(i) =Vx(i) + extend(Vy(i));  i=0:number-1
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

- 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)

```
>>> 函数说明：扩展减法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 为符号位
    Vz(i) =Vx(i) - extend(Vy(i));  i=0:number-1
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

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)

```
>>> 函数说明：向量临近元素加法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量临近元素饱和加法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    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)

```
>>> 函数说明：向量临近元素扩展加法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(2i+1:2i) =extend(Vx(2i)) +extend(Vx(2i+1));      i=0:(number/2-1)
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

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)

```
>>> 函数说明：向量临近元素扩展累加
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(2i+1:2i) =Vz(2i+1:2i) +extend(Vx(2i))+extend(Vx(2i+1));      i=0:(number/2-
↪1)
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

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)

```
>>> 函数说明：向量错位减加
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    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);
```

(下页继续)

(续上页)

```
    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)

```
>>> 函数说明：向量错位加减
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    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)

```
>>> 函数说明：向量错位减加后取平均值
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
```

(下页继续)

(续上页)

```
    Vz(2i+1)=(Vx(2i+1)-Vy(2i)) >>1;
    Vz(2i)=(Vx(2i)+Vy(2i+1)) >>1;  i=0:(number/2-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

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)

```
>>> 函数说明：向量错位加减后取平均值
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(2i+1)=(Vx(2i+1)+Vy(2i))>>1;
    Vz(2i)=(Vx(2i)-Vy(2i+1))>>1;  i=0:(number/2-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

vabs.t

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

```
>>> 函数说明：向量元素取绝对值
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=abs(Vx(i));   i=0:number-1
```

vabs.t.s

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

```
>>> 函数说明：向量元素饱和绝对值
    假设 Vx 是参数，Vz 是返回值，U/S 是符号位
    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)

```
>>> 函数说明：向量元素减法饱和绝对值
    假设 Vx,Vy 是参数，Vz 是返回值，U/S 是符号位
    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)

```
>>> 函数说明：向量元素拓展减法后取绝对值
    假设 Vx,Vy 是参数，Vz 是返回值，U/S 是符号位
    Vz(i)=abs(extend(Vx(i))-extend(Vy(i)));  i=0:number-1
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

vsabsa.t

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

```
>>> 函数说明：向量元素减法后取绝对值，然后累加
    假设 Vz,Vx,Vy 是参数，Vz 又是返回值，U/S 是符号位
    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, 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)

```
>>> 函数说明：向量元素拓展后减法，取绝对值，然后累加
    假设 Vz,Vx,Vy 是参数，Vz 又是返回值，U/S 是符号位
    Vz(i)= Vz(i)+abs(extend(Vx(i))-extend(Vy(i)));  i=0:number-1
    extend 根据 U/S 将值零扩展或者符号扩展至元素位宽的 2 倍
```

vneg.t

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

```
>>> 函数说明：向量元素取负
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素饱和取负
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素取最大值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=max((Vx(i),Vy(i)) ;  i=0:number-1
    max 取两元素中值较大的一个
```

- 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)

```
>>> 函数说明：向量元素取最小值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=min((Vx(i),Vy(i)) ;  i=0:number-1
    min 取两元素中值较大的一个
```

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)

```
>>> 函数说明：向量临近元素取最大值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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 取两元素中值较大的一个
```

- 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)

```
>>> 函数说明：向量临近元素取最小值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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 取两元素中值较小的一个
```

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)

```
>>> 函数说明：向量元素不等于 0
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素小于等于 0
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素小于 0
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素大于 0
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素大于等于 0
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素小于
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素小于等于
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素大于等于
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素大于
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素不等于
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量裁剪取饱和值
    假设 Vx,imm6 是两个参数，Vz 是返回值，U/S 是符号位
    U: Max=2^(imm6)-1, Min=0;
    S: Max=2^(imm6-1)-1, Min=-2^(imm6-1);
    无论 T 为 U/S，将 Vx(i) 始终看做有符号数 If Vx(i)>Max    Vz(i)=Max;
    else if Vx(i)<Min   Vz(i)=Min;
    else   Vz(i)=Vx(i);
    end        i=0:number-1
    U:imm6 的范围是 0 ~ (element_size-1)
    S:imm6 的范围是 1 ~ (element_size)
```

3.6.6.2　整型乘法指令

vmul.t && vmuli.t

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

```
>>> 函数说明：向量元素乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(i)=Vx(i)*Vy(i);      i=0:number-1
```

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

```
>>> 函数说明：向量元素乘法
    假设 Vx,Vy,index 是三个参数，Vz 是返回值，U/S 是符号位
    Vz(i)=Vx(i)*Vy(index));    i=0:number-1
    index 的范围是 0~(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)

```
>>> 函数说明：向量元素乘法取高半部分
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=(Vx(i)*Vy(i)) [2*element_size-1: element_size];    i=0:number-1
    取乘法结果的高部分
```

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

```
>>> 函数说明：向量元素乘法取高半部分
    假设 Vx,Vy,index 是三个参数，Vz 是返回值
    Vz(i)=(Vx(i)*Vy(index)) [2*element_size-1:element_size];    i=0:number-1
    取乘法结果的高部分
    index 的范围是 0~(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)

```
>>> 函数说明：向量元素扩展乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=(Vx(i)*Vy(i))[2*element_size-1:0];   i=0:(number-1)
    乘法结果取全部精度，即元素位宽的 2 倍
```

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

```
>>> 函数说明：向量元素扩展乘法
    假设 Vx,Vy,index 是三个参数，Vz 是返回值
    Vz(i)=(Vx(i)*Vy(index))[2*element_size-1:0];   i=0:(number-1)
    乘法结果取全部精度，即元素位宽的 2 倍
    index 的范围是 0~(128/element_size -1)
```

vmula.t && vmulai.t

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

```
>>> 函数说明：向量元素乘累加
    假设 Vz,Vx,Vy 是三个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+Vx(i)*Vy(index));     i=0:number-1
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量元素乘累加
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+Vx(i)*Vy(index));     i=0:number-1
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量元素扩展乘累加
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+(Vx(i)*Vy(i))[2*element_size-1:0];   i=0:(number-1)
    乘法结果取全部精度，即元素位宽的 2 倍
```

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

```
>>> 函数说明：向量元素扩展乘累加
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+(Vx(i)*Vy(index))[2*element_size-1:0];   i=0:(number-1)
    乘法结果取全部精度，即元素位宽的 2 倍
    index 的范围是 0~(128/element_size -1)
```

vmuls.t && vmulsi.t

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

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

```
>>> 函数说明：向量元素乘累减
    假设 Vz,Vx,Vy 是三个参数，同时 Vz 是返回值
    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)

```
>>> 函数说明：向量元素乘累减
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)-Vx(i)*Vy(index));     i=0:number-1
    index 的范围是 0~(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)

```
>>> 函数说明：向量元素扩展乘累减
    假设 Vz,Vx,Vy 是 3 个参数，Vz 是返回值
    Vz(i)=Vz(i)-(Vx(i)*Vy(i)) [2*element_size-1:0];    i=0:(number-1)
    乘法结果取全部精度，即元素位宽的 2 倍
```

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

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

```
>>> 函数说明：向量元素扩展乘累减
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(2i+1:2i)=(Vz(2i+1:2i)-(Vx(i)*Vy(index))[2*element_size-1:0];   i=0:(number-
↪1)
    乘法结果取全部精度，即元素位宽的 2 倍
    index 的范围是 0~(128/element_size -1)
```

vmulaca.t && vmulacai.t

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

```
>>> 函数说明：向量链乘累加
    假设 Vx,Vy 是两个参数，Vz 是返回值,U/S 表示符号位
    Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0];     i=0:number-1
    乘法结果取全部精度，即元素位宽的 2 倍
    Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]);     i=number/4-1
    extend 表示将累加结果根据 U/S 扩展至目的元素的位宽，即源操作数元素位宽的 4 倍
```

- 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)

```
>>> 函数说明：向量带索引链乘累加
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值，U/S 表示符号位
    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
    乘法结果取全部精度，即元素位宽的 2 倍
    Vz(4i+3:4i)=extend(Tmp[4i+3]+Tmp[4i+2]+Tmp[4i+1]+Tmp[4i]);     i=number/4-1
    extend 表示将累加结果根据 U/S 扩展至目的元素的位宽，即源操作数元素位宽的 4 倍
```

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)

```
>>> 函数说明：向量链乘累加加
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值,U/S 是符号位
    Tmp(i)=(Vx(i)*Vy(i))[2*element_size-1:0];      i=0:number-1
    乘法结果取全部精度，即元素位宽的 2 倍
    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 表示将累加结果根据 U/S 扩展至目的元素的位宽，即源操作数元素位宽的 4 倍
```

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

```
>>> 函数说明：向量带索引链乘累加加
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值,U/S 是符号位
    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
    乘法结果取全部精度，即元素位宽的 2 倍
    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 表示将累加结果根据 U/S 扩展至目的元素的位宽，即源操作数元素位宽的 4 倍
    index 的范围是 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)

```
>>> 函数说明：向量扩展带饱和小数乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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];
    (乘法结果取全部精度，即元素位宽的 2 倍)
    End     i=0:(number-1)
```

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

```
>>> 函数说明：向量带索引扩展带饱和小数乘法
    假设 Vx,Vy,index 是三个参数，Vz 是返回值
    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];
    (乘法结果取全部精度，即元素位宽的 2 倍)
    End     i=0:(number-1)
    index 的范围是 0~(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)

```
>>> 函数说明：向量带饱和取高半小数乘法
    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];
    (乘法结果取高位)
    End     i=0:(number-1)
```

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

```
>>> 函数说明：向量带索引带饱和取高半小数乘法
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值
    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];
    (乘法结果取高位)
    End     i=0:(number-1)
    index 的范围是 0~(128/element_size -1)
```

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

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

- int32x4_t vrmulh_s32_rs(int32x4_t, int32x4_t)

```
>>> 函数说明：向量带饱和取高半舍入小数乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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];
    (乘法结果取高位)
    End     i=0:(number-1)
```

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

```
>>> 函数说明：向量带索引带饱和取高半舍入小数乘法
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值
    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];
    (乘法结果取高位)
    End     i=0:(number-1)
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量带饱和取高半舍入小数乘累加
    假设 Vx,Vy 是两个参数，Vz 是返回值
    round=1<<(element_size-1);
    Tmp(i)= (Vz(i)<<element_size)+ Vx(i)*Vy(i)*2+round;  i=0:(number-1)
    Tmp(i) 保留运算的全部精度
    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];
    (取乘累加结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在累加后进行)
```

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

```
>>> 函数说明：向量带索引带饱和取高半舍入小数乘累加
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值
    round=1<<(element_size-1);
    Tmp(i)= (Vz(i)<<element_size)+ Vx(i)*Vy(index)*2+round;  i=0:(number-1)
    Tmp(i) 保留运算的全部精度
    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];
    (取乘累加结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在累加后进行)
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量带饱和取高半舍入小数乘累减
    假设 Vx,Vy 是两个参数，Vz 是返回值
    round=1<<(element_size-1);
    Tmp(i)= (Vz(i)<<element_size)-Vx(i)*Vy(i)*2+round;  i=0:(number-1)
    Tmp(i) 保留运算的全部精度
    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];    (取乘累减结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在累加后进行)
```

- 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)

```
>>> 函数说明：向量带索引带饱和取高半舍入小数乘累减
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值
    round=1<<(element_size-1);
    Tmp(i)= (Vz(i)<<element_size)- Vx(i)*Vy(index)*2+round;  i=0:(number-1)
```

(下页继续)

(续上页)

```
Tmp(i) 保留运算的全部精度
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];    (取乘累减结果的高部分)
End   i=0:(number-1)
(注: 饱和操作在累加后进行)
index 的范围是 0~(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)

```
>>> 函数说明: 向量扩展带移位小数乘法
    假设 Vx,Vy,imm4 是 3 个参数, Vz 是返回值
    Vz(i)= (Vx(i)*Vy(i))>>imm4;  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移  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)

```
>>> 函数说明: 向量带索引扩展带移位小数乘法
    假设 Vx,imm4,index 是 4 是参数, Vz 是返回值
    Vz(i)=(Vx(i)*Vx+1(index))>>imm4;  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移  imm4=0~15
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明: 向量扩展带移位小数乘累加
    假设 Vz,Vx,Vy,imm 是 4 个参数, 同时 Vz 是返回值
    Vz(i)=Vz(i) + ((Vx(i)*Vy(i))>>imm);  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移, 再累加  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)

```
>>> 函数说明：向量带索引扩展带移位小数乘累加
    假设 Vz,Vx,imm,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i) + ((Vx(i)*Vx+1(index))>>imm);  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移，再累加  imm=0~15
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量扩展带移位小数乘负累加
    假设 Vz,Vx,Vy,imm 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i) + ((-Vx(i)*Vy(i))>>imm);  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移，再累减  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)

```
>>> 函数说明：向量带索引扩展带移位小数乘负累加
    假设 Vz,Vx,imm,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i) + ((-Vx(i)*Vx+1(index))>>imm);  i=0:(number-1)
    乘法结果保留全部精度后进行算术右移，再累减  imm=0~15
    index 的范围是 0~(128/element_size -1)
```

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

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

```
>>> 函数说明：向量带饱和复数实部虚部交叉相乘累加累加取高半舍入
    假设 Vz,Vx,Vy 是参数，同时 Vz 是返回值
    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) 保留运算的全部精度
    If  Tmp(i)>2^(2*element_size-1)-1Vz(i)= 2^(element_size-1)-1;
    Else if Tmp(i)<-2^(2*element_size-1)
```

(下页继续)

(续上页)

```
    Vz(i)= -2^(element_size-1);
    Else Vz(i)=Tmp(i)[2*element_size-1:element_size];     (取乘累加结果的高部分)
    End    i=0:(number-1)
    (注: 饱和操作在累加后进行)
```

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

```
>>> 函数说明: 向量带索引带饱和复数实部虚部交叉相乘累加累加取高半舍入
    假设 Vz,Vx,Vy,index 是 4 个参数, 同时 Vz 是返回值
    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) 保留运算的全部精度
    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];     (取乘累加结果的高部分)
    End    i=0:(number-1)
    (注: 饱和操作在累加后进行)
    index 的范围是 0 ~ (128/(element_size*2) -1)
```

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

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

```
>>> 函数说明: 向量带饱和复数实部虚部交叉相乘累加累减取高半舍入
    假设 Vz,Vx,Vy 是三个参数, 同时 Vz 是返回值
    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) 保留运算的全部精度
    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];     (取乘累加减结果的高部分)
    End    i=0:(number-1)
    (注: 饱和操作在累加减后进行)
```

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

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

```
>>> 函数说明：向量带索引带饱和复数实部虚部交叉相乘累加累减取高半舍入
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    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) 保留运算的全部精度
    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];    (取乘累加减结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在累加减后进行)
    index 的范围是 0 ~ (128/(element_size*2) -1)
```

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

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

```
>>> 函数说明：向量带饱和复数实部虚部交叉相乘累减累减取高半舍入
    假设 Vz,Vx,Vy 是三个参数，同时 Vz 是返回值
    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) 保留运算的全部精度
    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];    (取乘累加减结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在累加减后进行)
```

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

```
>>> 函数说明：向量带索引带饱和复数实部虚部交叉相乘累减累减取高半舍入
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    round=1<<(element_size-1);
```

(下页继续)

(续上页)

```
    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) 保留运算的全部精度
    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];     (取乘累加减结果的高部分)
    End   i=0:(number-1)
    (注: 饱和操作在累加减后进行)
    index 的范围是 0 ~ (128/(element_size*2) -1)
```

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

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

```
>>> 函数说明: 向量带饱和复数实部虚部交叉相乘累减累加取高半舍入
    假设 Vz,Vx,Vy 是 3 个参数, 同时 Vz 是返回值
    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) 保留运算的全部精度
```

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

```
>>> 函数说明: 向量带索引带饱和复数实部虚部交叉相乘累减累加取高半舍入
    假设 Vz,Vx,Vy,index 是 4 个参数, 同时 Vz 是返回值
    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) 保留运算的全部精度
    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];     (取乘累加减结果的高部分)
    End   i=0:(number-1)
    (注: 饱和操作在累加减后进行)
    index 的范围是 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)

```
>>> 函数说明：复数乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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) 保留运算的全部精度
    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];     (取乘加/减结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在加减后进行)
```

vrcmula.t.e

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

```
>>> 函数说明：复数乘法右移扩位累加
    假设 Vz,Vx,Vy,imm 是 4 个参数，同时 Vz 是返回值
    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)  (虚部)
    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)  (实部)
    复数乘法结果保留全部精度后进行算术右移，再累加  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)

```
>>> 函数说明：复数共轭乘法 conj(x)*y
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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) 保留运算的全部精度
    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];     (取乘加/减结果的高部分)
    End   i=0:(number-1)
    (注：饱和操作在加减后进行)
```

vrcmulca.t.e

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

```
>>> 函数说明：复数共轭乘法右移扩位累加
    假设 Vz,Vx,Vy,imm 是 4 个参数，同时 Vz 是返回值
    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)  (虚部)
    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)  (实部)
    复数乘法结果保留全部精度后进行算术右移，再累加  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)

```
>>> 函数说明：复数取负乘法 (-x)*y
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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;
```

(下页继续)

(续上页)

```
    i=0:(number/2-1)
    Tmp(i) 保留运算的全部精度
    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];     (取乘加/减结果的高部分)
    End   i=0:(number-1)
    (注: 饱和操作在加减后进行)
```

vrcmulna.t.e

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

```
>>> 函数说明: 复数取负乘法右移扩位累加
    假设 Vz,Vx,Vy,imm4 是 4 个参数, 同时 Vz 是返回值
    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)  (虚部)
    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)  (实部)
    复数乘法结果保留全部精度后进行算术右移, 再累加  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)

```
>>> 函数说明: 复数共轭取负乘法 (-conj(x)*y
    假设 Vx,Vy 是两个参数, Vz 是返回值
    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) 保留运算的全部精度
    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);
```

(下页继续)

(续上页)

```
    Else Vz(i)=Tmp(i)[2*element_size-1:element_size];     (取乘加/减结果的高部分)
    End   i=0:(number-1)
    (注: 饱和操作在加减后进行)
```

vrcmulcna.t.e

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

```
>>> 函数说明: 复数共轭取负乘法右移扩位累加
    假设 Vz,Vx,Vy,imm4 是 4 个参数, 同时 Vz 是返回值
    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)  (虚部)
    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)  (实部)
    复数乘法结果保留全部精度后进行算术右移, 再累加  imm=0~15
```

3.6.6.3　整型倒数、倒数开方、e 指数快速运算及逼近指令

vrecpe.t && vrecps.t

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

```
>>> 函数说明: 向量元素取倒数
    假设 Vx 是参数, Vz 是返回值
    Vz(i) ≈ 1/(Vx(i))  i=0:(number-1)
    (快速计算 Vx(i) 的倒数值)
```

- 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)

```
>>> 函数说明：向量倒数逼近
    假设 Vx,Vy 是 2 个两个，Vz 是返回值
    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)

```
>>> 函数说明：向量元素倒数后开方
    假设 Vx 是参数，Vz 是返回值
    Vz(i) ≈     i=0:(number-1)          (快速计算 Vx(i) 的倒数开方值)
```

- 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)

```
>>> 函数说明：
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量元素取 e 的指数值
    假设 Vx 是输入，Vz 是返回值
    Vz(i) ≈ e^(Vx(i))  i=0:(number-1)
    (快速计算 Vx(i) 的 e 指数值)
```

3.6.6.4　整型移位指令

vsht.t

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

```
>>> 函数说明：向量左移
    假设 Vx,Vy 是参数，Vz 是返回值，U/S 是符号位
    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)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为有符号移位索引;
    对于 U，右移为逻辑右移，对于 S，右移为算术右移;
```

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)

```
>>> 函数说明：向量饱和左移
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)   以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为有符号移位索
引;
    sat 根据回写元素位宽判断左移结果是否溢出，并根据 U/S 将溢出结果饱和为相应的最大或最小值;
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

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)

```
>>> 函数说明：向量 round 左移
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为有符号移位索引;
    对于 U，右移为逻辑右移，对于 S，右移为算术右移;
```

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)

```
>>> 函数说明：向量饱和 round
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为有符号移位索引;
    sat 根据回写元素位宽判断左移结果是否溢出，并根据 U/S 将溢出结果饱和为相应的最大或最小值
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

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)

```
>>> 函数说明：向量寄存器左移
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(i)=Vx(i)<<Vy(i)[7:0] ;   i=0:(number-1)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为无符号移位索引;
```

- 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)

```
>>> 函数说明：向量立即数左移
    假设 Vx,imm 是两个参数，Vz 是返回值
    Vz(i)=Vx(i)<<imm;   i=0:(number-1)
    imm 的范围是 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)

```
>>> 函数说明：向量寄存器左移取饱和
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    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)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为无符号移位索引;
```

- 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)

```
>>> 函数说明：向量立即数左移取饱和
    假设 Vx,imm 是两个参数，Vz 是返回值
    signed=(T==S);  (根据元素 U/S 类型选择)
    Max=signed? 2^(element_size-1)-1: 2^(element_size)-1;
    Min=signed? -2^(element_size-1):0;
    If (Vx(i)<<imm)>Max   Vz(i)=Max;
    Else if (Vx(i)<<imm)<Min   Vz(i)=Min;
    Else   Vz(i)= Vx(i)<<imm;        i=0:(number-1)
    imm 的范围是 0 ~ element_size-1
```

vshli.t.e

- 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)

```
>>> 函数说明：向量扩展立即数左移
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(2i+1,2i)=extend(Vx(i))<<imm;  i=0:(number-1)
    extend 将元素根据 U/S 扩展为原始位宽的 2 倍
    imm 的范围是 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)

```
>>> 函数说明：向量寄存器右移
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    Vz(i)=Vx(i)>>Vy(i)[7:0] ;  i=0:(number-1)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为无符号移位索引;
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

- 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)

```
>>> 函数说明：向量立即数右移
    假设 Vx,imm 是两个参数,Vz 是返回值，U/S 是符号位
    Vz(i)=Vx(i)>>imm;   i=0:(number-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 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)

```
>>> 函数说明：向量寄存器右移取 round
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)
    以 Vy 每个元素 Vy(i) 中的低 8-bit 数据 Vy(i)[7:0] 作为无符号移位索引;
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
```

- 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)

```
>>> 函数说明：向量立即数左移取 round
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    round=1<<(imm-1);Vz(i)=(Vx(i)+round)>> imm);     i=0:(number-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 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)

```
>>> 函数说明：向量立即数右移取低半
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    Tmp(i)=(Vx(i)>> imm)[element_size/2-1:0];  i=0:(number-1)    (取结果的低半部分)
    Vz(i)={Tmp(2i+1),Tmp(2i)};  i=0:(number/2-1)
    (结果放于 Vz 的低 64bit)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 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)

```
>>> 函数说明：向量立即数右移 round 取低半
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    round=1<<(imm);
    Tmp(i)=((Vx(i)+round)>> (imm+1))[element_size/2-1:0];  i=0:(number-1)
    (取结果的低半部分)
    Vz(i)={Tmp(2i+1),Tmp(2i)};  i=0:(number/2-1)
    (结果放于 Vz 的低 64bit)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    oimm 的范围是 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)

```
>>> 函数说明：向量立即数右移取低半饱和
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    Max=signed? 2^(element_size/2-1)-1: 2^(element_size/2)-1;
    Min=signed? -2^(element_size/2-1): 0;
    If (Vx(i)>> imm)>Max   Tmp(i)=Max;
    Else if (Vx(i)>> imm)<Min   Tmp(i)=Min;
    Else Tmp(i)=(Vx(i)>> imm)[element_size/2-1:0];    (取结果的低半部分)
    End    i=0:(number-1)
    Vz(i)={Tmp(2i+1),Tmp(2i)};  i=0:(number/2-1)   (结果放于 Vz 的低 64bit)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 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)

```
>>> 函数说明：向量立即数右移 round 取低半饱和
    假设 Vx,imm 是两个参数，Vz 是返回值，U/S 是符号位
    round=1<<(oimm-1);signed=(T==S);  (根据元素 U/S 类型选择)
    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];    (取结果的低半部分)
    End    i=0:(number-1)
    Vz(i)={Tmp(2i+1),Tmp(2i)};  i=0:(number/2-1)   (结果放于 Vz 的低 64bit)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    oimm 的范围是 1 ~ element_size
```

vshria.t

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

```
>>> 函数说明：向量立即数右移累加
    假设 Vz,Vx,imm 是 3 个参数，同时 Vz 是返回值，U/S 是符号位
    Vz(i)=Vz(i)+(Vx(i)>> imm);   i=0:(number-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 1 ~ element_size
```

vshria.t.r

- int8x16_t vshria_s8_r (int8x16_t, int8x16_t, const int)

- int16x8_t vshria_s16_r (int16x8_t, int16x8_t, const int)
- int32x4_t vshria_s32_r (int32x4_t, int32x4_t, const int)
- int64x2_t vshria_s64_r (int64x2_t, int64x2_t, const int)
- uint8x16_t vshria_u8_r (uint8x16_t, uint8x16_t, const int)
- uint16x8_t vshria_u16_r (uint16x8_t, uint16x8_t, const int)
- uint32x4_t vshria_u32_r (uint32x4_t, uint32x4_t, const int)
- uint64x2_t vshria_u64_r (uint64x2_t, uint64x2_t, const int)

```
>>> 函数说明：
    假设 Vz,Vx,imm 是 3 个参数，同时 Vz 是返回值，U/S 是符号位
    round=1<<(imm-1);Vz(i)=Vz(i)+((Vx(i)+round)>> imm) ;    i=0:(number-1)
    对于 U，右移为逻辑右移，对于 S，右移为算术右移
    imm 的范围是 1 ~ element_size
```

vexh.t && vexl.t

- int8x16_t vexh_s8 (int8x16_t, int8x16_t, int)
- int16x8_t vexh_s16 (int16x8_t, int16x8_t, int)
- int32x4_t vexh_s32 (int32x4_t, int32x4_t, int)
- int64x2_t vexh_s64 (int64x2_t, int64x2_t, int)
- uint8x16_t vexh_u8 (uint8x16_t, uint8x16_t, unsigned)
- uint16x8_t vexh_u16 (uint16x8_t, uint16x8_t, unsigned)
- uint32x4_t vexh_u32 (uint32x4_t, uint32x4_t, unsigned)
- uint64x2_t vexh_u64 (uint64x2_t, uint64x2_t, unsigned)

```
>>> 函数说明：向量立即数右移取 round 累加
    假设 Vz,Vx,ry 是参数，同时 Vz 是返回值，U/S 是符号位
    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, int)
- int16x8_t vexl_s16 (int16x8_t, int16x8_t, int)
- int32x4_t vexl_s32 (int32x4_t, int32x4_t, int)
- int64x2_t vexl_s64 (int64x2_t, int64x2_t, int)
- uint8x16_t vexl_u8 (uint8x16_t, uint8x16_t, unsigned)
- uint16x8_t vexl_u16 (uint16x8_t, uint16x8_t, unsigned)
- uint32x4_t vexl_u32 (uint32x4_t, uint32x4_t, unsigned)
- uint64x2_t vexl_u64 (uint64x2_t, uint64x2_t, unsigned)

```
>>> 函数说明：向量高位数据插入
    假设 Vz,Vx,ry 是参数，同时 Vz 是返回值，U/S 是符号位
    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
```

3.6.6.5　整型移动 (MOV)、元素操作、位操作指令

vmtvr.t.1

- int8x16_t vmtvr_s8_1 (int8x16_t, char, const int)
- int16x8_t vmtvr_s16_1 (int16x8_t, short, const int)
- int32x4_t vmtvr_s32_1 (int32x4_t, int, const int)
- uint8x16_t vmtvr_u8_1 (uint8x16_t, unsigned char, const int)
- uint16x8_t vmtvr_u16_1 (uint16x8_t, unsigned short, const int)
- uint32x4_t vmtvr_u32_1 (uint32x4_t, unsigned int, const int)

```
>>> 函数说明：向量单元素写传送
    假设 Vz,rx,index 是三个参数，同时 Vz 是返回值，U/S 是符号位
    Vz(index)=Rx[element_size-1:0]，其余元素不变
    Index 范围为 0~(128/element_size -1)
```

vmtvr.t.2

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

```
>>> 函数说明：向量双元素写传送
    假设 Vz,rx,index 是三个参数，同时 Vz 是返回值，U/S 是符号位
    Vz(index)=Rx[element_size-1:0], Vz(index+1)=Rx[element_size-1+32:32]，其余元素不
变
    Index 范围为 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)

```
>>> 函数说明：向量写传送
    假设 Vx,index 是 3 个参数，Rz 是返回值
    Rz=extend_32(Vx(index));
    extend_32 根据 U/S 将值零扩展或者符号扩展至 32 位
    Index 范围为 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)

```
>>> 函数说明：向量数据符号位读传送
    假设 Vx 是参数，Rz 是返回值，U/S 是符号位
    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;
    (提取 Vx 每个元素的符号位，将其依次放入通用寄存器 Rz 的低位)
```

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)

```
>>> 函数说明：向量扩展传输
    假设 Vx 是参数，Vz 是返回值，U/S 是符号位
    Vz(i)=extend(Vx(i));    i=0:(number/2-1)
    extend 根据 U/S 将值零扩展或者符号扩展为元素位宽的 2 倍
```

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)

```
>>> 函数说明：向量低位传输
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)
    取元素的低半部分
```

- 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)

```
>>> 函数说明：向量高位传输
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    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)
    取元素的高半部分
```

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)

```
>>> 函数说明：向量低位饱和传输
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    signed=(T==S);  (根据元素 U/S 类型选择)
    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];    (取元素的低半部分)
    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];    (取元素的低半部分)
    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)

```
>>> 函数说明：向量高位 round 传输
    假设 Vx,Vy 是两个参数，Vz 是返回值，U/S 是符号位
    round=1<<(elemen_size/2-1)
    Tmp1(i)=(Vx(i)+round)[element_size-1: element_size/2];  i=0:(number-1)    (取元
素的高半部分)
    Tmp2(i)=(Vy(i)+round)[element_size-1: element_size/2];  i=0:(number-1)    (取元
素的高半部分)
    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)

```
>>> 函数说明：向量数据交叉传输
    假设 Vx,Vy 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量数据字节倒序
    假设 Vx 是输入，Vz 是返回值
    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)

```
>>> 函数说明：向量数据半字节倒序
    假设 Vx 是输入，Vz 是返回值
    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)

```
>>> 函数说明：向量数据字倒序
    假设 Vx 是输入，Vz 是返回值
    Vz(number-1:0)= {Vx(0), Vx(1), Vx(2), Vx(3)};
```

- int64x2_t vrevd_s64 (int64x2_t)
- uint64x2_t vrevd_u64 (uint64x2_t)

```
>>> 函数说明：向量数据双字倒序
    假设 Vx 是输入，Vz 是返回值
    Vz(number-1:0)= {Vx(0), Vx(1)};
```

vexti.t && vext.t

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

```
>>> 函数说明：立即数向量数据拼接
    假设 Vx,Vy,imm 是 3 个参数，Vz 是返回值
    If imm[5]==0, Vz (imm[3:0]:0)=Vx(imm[3:0]:0);
```

(下页继续)

(续上页)

```
    (将 Vx 的低位若干个元素拷贝到 Vz 的低位若干个元素)
    Else Vz (imm[3:0]:0)=Vx(15:15-imm[3:0]);
    (将 Vx 的高位若干个元素拷贝到 Vz 的低位若干个元素)
    If imm[4]==0, Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0);
    (将 Vy 的低位若干个元素拷贝到 Vz 的高位若干个元素)
    Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
    (将 Vy 的高位若干个元素拷贝到 Vz 的高位若干个元素)
    其中 imm[3:0] 的范围为 0~14;
```

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

```
>>> 函数说明: 寄存器向量数据拼接
    假设 Vx,Vy,Rk 是 3 个参数, Vz 是返回值
    Imm6 = Rk[5:0];
    If imm6[5]==0, Vz (imm[3:0]:0)=Vx(imm[3:0]:0);
    (将 Vx 的低位若干个元素拷贝到 Vz 的低位若干个元素)
    Else Vz (imm[3:0]:0)=Vx(15:15-imm[3:0]);
    (将 Vx 的高位若干个元素拷贝到 Vz 的低位若干个元素)
    If imm6[4]==0, Vz (15:imm[3:0]+1)=Vy(15-imm[3:0]-1:0);
    (将 Vy 的低位若干个元素拷贝到 Vz 的高位若干个元素)
    Else Vz (15:imm[3:0]+1)=Vy(15:imm[3:0]+1);
    (将 Vy 的高位若干个元素拷贝到 Vz 的高位若干个元素)
    其中 imm[3:0] 的范围为 0~14;
```

vtbl.t && vtbx.t

- int8x16_t vtbl_s8 (int8x16_t, int8x16_t)
- uint8x16_t vtbl_u8 (uint8x16_t, uint8x16_t)

```
>>> 函数说明: 向量数据链接
    假设 Vx,Vy 是两个参数, Vz 是返回值
    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)

```
>>> 函数说明: 向量数据链接
    假设 Vx,Vy 是两个参数, Vz 是返回值
    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)

```
>>> 函数说明：向量按位与运算
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量按位非与运算
    假设 Vx,Vy 是两个参数,Vz 是返回值
    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)

```
>>> 函数说明：向量按位异或运算
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量按位取反运算
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量按位非运算
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量按位或运算
    假设 Vx,Vy 是两个参数,Vz 是返回值
    for j=0:127  Vz[j]=Vx[j] | (! Vy[j])
```

vsel.t

- int8x16_t vsel_s8 (int8x16_t, int8x16_t, int8x16_t)
- int16x8_t vsel_s16 (int16x8_t, int16x8_t, int16x8_t)
- int32x4_t vsel_s32 (int32x4_t, int32x4_t, int32x4_t)
- int64x2_t vsel_s64 (int64x2_t, int64x2_t, int64x2_t)
- uint8x16_t vsel_u8 (uint8x16_t, uint8x16_t, uint8x16_t)
- uint16x8_t vsel_u16 (uint16x8_t, uint16x8_t, uint16x8_t)
- uint32x4_t vsel_u32 (uint32x4_t, uint32x4_t, uint32x4_t)
- uint64x2_t vsel_u64 (uint64x2_t, uint64x2_t, uint64x2_t)

```
>>> 函数说明：向量位选择
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量符号位连续相同
    假设 Vx 是参数，Vz 是返回值
    从 MSB 开始与元素符号位相同的连续位数，符号位不计入计数
    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)

```
>>> 函数说明：向量最高位连续 0 个数
    假设 Vx 是参数，Vz 是返回值
    从 MSB 开始连续为 0 的位数
```

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)

```
>>> 函数说明：向量数据 1 个数
    假设 Vx 是参数，Vz 是返回值
    count_one 计算元素中 1 的位数
    Vz(i)=count_one(Vx(i)) ;    i=0:(number-1)
```

vtst.t

- int8x16_t vtst_s8 (int8x16_t, int8x16_t)
- int16x8_t vtst_s16 (int16x8_t, int16x8_t)
- int32x4_t vtst_s32 (int32x4_t, int32x4_t)
- int64x2_t vtst_s64 (int64x2_t, int64x2_t)
- uint8x16_t vtst_u8 (uint8x16_t, uint8x16_t)
- uint16x8_t vtst_u16 (uint16x8_t, uint16x8_t)
- uint32x4_t vtst_u32 (uint32x4_t, uint32x4_t)
- uint64x2_t vtst_u64 (uint64x2_t, uint64x2_t)

```
>>> 函数说明：向量按位与置位
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量目的计算器与通用源之间的拷贝
    假设 Rx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：一元向量源目的寄存器之间的拷贝
    假设 Vx,index 是输入，Vz 是返回值
    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)

```
>>> 函数说明：二元向量源目的寄存器之间的拷贝
    假设 Vx,index 是输入，Vz 是返回值
    Vz(i)=Rx(index);  i=0:number-1
    Vz(i)=Rx(index+1);  i=number:2*number-1
    index=0 ~ (128/element_size -1)
    number = 128/element_size
```

vins.t.1 && vins.t.2

- int8x16_t vins_s8_1 (int8x16_t, int8x16_t, const int, const int)
- int16x8_t vins_s16_1 (int16x8_t, int16x8_t, const int, const int)
- int32x4_t vins_s32_1 (int32x4_t, int32x4_t, const int, const int)
- uint8x16_t vins_u8_1 (uint8x16_t, uint8x16_t, const int, const int)
- uint16x8_t vins_u16_1 (uint16x8_t, uint16x8_t, const int, const int)
- uint32x4_t vins_u32_1 (uint32x4_t, uint32x4_t, const int, const int)

```
>>> 函数说明：一元向量插入
    假设 Vz,Vx,index,index2 是 4 个参数，同时 Vz 是返回值
    Vz(index2)=Vx(index);   Vz 其余元素值不变
    index=0 ~ (128/element_size -1);
    index2=0 ~ (128/element_size -1)
```

- int8x32_t vins_s8_2 (int8x32_t, int8x32_t, const int, const int)
- int16x16_t vins_s16_2 (int16x16_t, int16x16_t, const int, const int)
- int32x8_t vins_s32_2 (int32x8_t, int32x8_t, const int, const int)
- uint8x32_t vins_u8_2 (uint8x32_t, uint8x32_t, const int, const int)
- uint16x16_t vins_u16_2 (uint16x16_t, uint16x16_t, const int, const int)
- uint32x8_t vins_u32_2 (uint32x8_t, uint32x8_t, const int, const int)

```
>>> 函数说明：二元向量插入
    假设 Vz,Vx,index,index2 是 4 个参数，同时 Vz 是返回值
    Vz(index2)=Vx(index);
    Vz(index2+number) = Vx(index+1)
    Vz 其余元素值不变
    index=0 ~ (128/element_size -1);
    index2=0 ~ (128/element_size -1)
    number = 128/element_size
```

vpkg.t.2

- int8x32_t vpkg_s8_2 (int8x32_t, int8x32_t, const int, const int)
- int16x16_t vpkg_s16_2 (int16x16_t, int16x16_t, const int, const int)
- int32x8_t vpkg_s32_2 (int32x8_t, int32x8_t, const int, const int)
- uint8x32_t vpkg_u8_2 (uint8x32_t, uint8x32_t, const int, const int)
- uint16x16_t vpkg_u16_2 (uint16x16_t, uint16x16_t, const int, const int)
- uint32x8_t vpkg_u32_2 (uint32x8_t, uint32x8_t, const int, const int)

```
>>> 函数说明：二元封装
    假设 Vz,Vx,index,index2 是 4 个参数，同时 Vz 是返回值
    bound=number; bound 用来判断是否跨寄存器，number 即元素个数
    Vz(index2)=Vx(index);
    if index2+1<bound
    Vz(index2+1)=Vx(index+bound);
    else Vz(index2+1)=Vx(index+bound);
    Vz,Vz+1 中其余元素不变
    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)

```
>>> 函数说明：二元交织
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：二元去交织
    假设 Vx 是参数，Vz 是返回值
    Vz(i) = Vx(2i);Vz(i+number) = Vx(2i+1);
    i=0:(number-1)
```

3.6.6.6　整型立即数生成指令

vmovi.8

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

```
>>> 函数说明：向量立即数传输
    假设 imm8 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量立即数传输
    假设 imm8,index 是两个参数，Vz 是返回值，U/S 是符号位
    IMM16= {8'b0, imm8[7:0]}<<(index*8)    (index= 0~1)
    If Type=U, Vz(i)= IMM16;    i=0:(number-1)
    If Type=S, Vz(i)= ~IMM16;   i=0:(number-1)
```

vmovi.t32

- uint32x4_t vmovi_u32 (const signed char, const int)
- int32x4_t vmovi_s32 (const signed char, const int)

```
>>> 函数说明：向量立即数传输
    假设 imm8,index 是两个参数，Vz 是返回值，U/S 是符号位
    IMM32= {24'b0, imm8[7:0]}<<(index*8)    (index= 0~3)
    If Type=U, Vz(i)= IMM32;    i=0:(number-1)
    If Type=S, Vz(i)= ~IMM32;   i=0:(number-1)
```

vmaski.8.l && vmaski.8.h

- int8x16_t vmaski_s8_l (const signed char)
- uint8x16_t vmaski_u8_l (const signed char)

```
>>> 函数说明：向量立即数扩展传输
    假设 imm8 是参数，Vz 是返回值
    Vz(i)= {8{imm8[i]}}; i=0:(number/2-1); Vz 其余元素置 0
```

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

```
>>> 函数说明：向量立即数扩展传输
    假设 imm8 是参数，Vz 是返回值
    Vz(i+8)= {8{imm8[i]}}; i=0:(number/2-1); Vz 其余元素保持不变
```

vmaski.16

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

```
>>> 函数说明：向量立即数扩展传输
    假设 imm8 是参数，Vz 是返回值
    Vz(i)= {16{imm8[i]}}; i=0:(number-1);
```

3.6.6.7 LOAD/STORE 指令

vld.t.n(n=1/2/3/4) && vst.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)

```
>>> 函数说明：固定长度向量加载
    假设 Rx,offset 是两个参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    size=00/01/10/11 对应 byte/half word/word/double word
    Offset=imm7<<size(offset 在 (0-127)<<size 范围内)
    for j=0:N-1(VLD.T.N, N = 1/2/3/4)
    Vz(j)=MEM(Rx+offset+j*(2^(size))) ;
    end 其余元素置 0
```

- void vst_8_1 (int8x16_t*, const int, int8x16_t)
- void vst_16_1 (int16x8_t*, const int, int16x8_t)
- void vst_32_1 (int32x4_t*, const int, int32x4_t)
- void vst_8_2 (int8x16_t*, const int, int8x16_t)
- void vst_16_2 (int16x8_t*, const int, int16x8_t)
- void vst_32_2 (int32x4_t*, const int, int32x4_t)

- void vst_8_3 (int8x16_t*, const int, int8x16_t)
- void vst_16_3 (int16x8_t*, const int, int16x8_t)
- void vst_32_3 (int32x4_t*, const int, int32x4_t)
- void vst_8_4 (int8x16_t*, const int, int8x16_t)
- void vst_16_4 (int16x8_t*, const int, int16x8_t)
- void vst_32_4 (int32x4_t*, const int, int32x4_t)

```
>>> 函数说明：固定长度向量存储
    假设 Rx,offset,Vz 是参数
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    size=00/01/10/11 对应 byte/half word/word/double word
    offset=imm7<<size(offset 范围在 (0-127)<<size) 范围内
    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)

```
>>> 函数说明：向量加载基址跳跃更新
    假设 Rx,Ry 是两个参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常
    size=00/01/10 对应 byte/half word/word
    for j=0:N-1  Vz(j)=MEM(Rx+j*(2^(size)));end 其余元素置 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)

```
>>> 函数说明：向量存储基址跳跃更新
    假设 Rx,Ry,Vz 是三个参数
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常
    size=00/01/10 对应 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*)

```
>>> 函数说明：向量加载基址更新
    假设 Rx 是参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    size=00/01/10 对应 byte/half word/word
    for j=0:N-1  Vz(j)=MEM(Rx+j*(2^(size)));  end 其余元素置 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)

```
>>> 函数说明：向量存储基址更新
    假设 Vz,Rx 是输入
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    size=00/01/10 对应 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*)

```
>>> 函数说明：连续向量加载
    假设 Rx 是参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    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)

```
>>> 函数说明：连续向量存储
    假设 Rx,Vz 是两个参数
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    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*)

```
>>> 函数说明：连续向量加载基址更新
    假设 Rx 是参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    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)

```
>>> 函数说明：  连续向量存储基址更新
    假设 Rx,Vz 是两个参数
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    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)

```
>>> 函数说明：跳跃向量加载基址更新
    假设 Rx，Ry 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：跳跃向量存储基址更新
    假设 Rx,Ry,Vz 是 3 个参数
    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)

```
>>> 函数说明：可变长度向量加载
    假设 Vx,Vy 是两个参数，Vz 是返回值
    当 rx 所对应的地址为非 element_size 对齐，则置非对齐异常。
    size=00/01/10/对应 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 其余元素置 0
    若 N=0，则结果不可预期
```

vlrw.t.n

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

```
>>> 函数说明：向量存储器读入
    假设 imm1,imm2,imm3,imm4 是 4 个参数，Vz 是返回值
    Vz[0] = imm1; Vz[1] = imm2; Vz[2] = imm3; Vz[3] = imm4
```

3.6.6.8　浮点加减法比较指令

vadd.t && vsub.t

- float32x4_t vadd_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：浮点向量加法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=Vx(i)+Vy(i);     i=0:(number-1)
```

- float32x4_t vsub_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：浮点向量减法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=Vx(i)-Vy(i);     i=0:(number-1)
```

vpadd.t

- float32x4_t vpadd_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点耦合加法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点交叉加减法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点交叉减加法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点绝对值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=abs(Vx(i));    i=0:number-1
```

- float32x4_t vsabs_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点减法绝对值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=abs(Vx(i)-Vy(i));    i=0:number-1
```

vneg.t

- float32x4_t vneg_f32 (float32x4_t)

```
>>> 函数说明：向量浮点取反
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=-Vx(i) ;  i=0:number-1
```

vmax.t && vmin.t

- float32x4_t vmax_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点最大值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=max((Vx(i),Vy(i)) ;  i=0:number-1
    max 取两元素中值较大的一个
```

- float32x4_t vmin_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点最小值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=min((Vx(i),Vy(i)) ;  i=0:number-1
    min 取两元素中值较小的一个
```

vmaxnm.t && vminnm.t

- float32x4_t vmaxnm_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点规范最大值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=max((Vx(i),Vy(i)) ;  i=0:number-1
    max 取两元素中值较大的一个;
    与 VMAX 不同的是，两个元素中一个为 quite NaN 而另一个为规范数时，取规范数的值作为输出。
```

- float32x4_t vminnm_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点规范最小值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=min((Vx(i),Vy(i)) ;  i=0:number-1
    min 取两元素中值较小的一个;
    与 VMIN 不同的是，两个元素中一个为 quite NaN 而另一个为规范数时，取规范数的值作为输出。
```

vpmax.t && vpmin.t

- `float32x4_t vpmax_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点相邻最大值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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 取两元素中值较大的一个
```

- `float32x4_t vpmin_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点相邻最小值
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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 取两元素中值较小的一个
```

vcmpnez.t && vcmpne.t

- `float32x4_t vcmpnez_f32 (float32x4_t)`

```
>>> 函数说明：向量浮点不等于零比较
    假设 Vx 是参数，Vz 是返回值
    If Vx(i)!=0 Vz(i)=11…111;   Else Vz(i)=00…000;  i=0:number-1
```

- `float32x4_t vcmpne_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点等于零比较
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)`

```
>>> 函数说明：向量浮点大于等于零比较
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点大于零比较
    假设 Vx,Vy 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点小于等于零比较
    假设 Vx 是参数，Vz 是返回值
    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)

```
>>> 函数说明：向量浮点小于零比较
    假设 Vx,Vy 是参数，Vz 是返回值
    If Vx(i)<Vy(i) Vz(i)=11…111;   Else Vz(i)=00…000;  i=0:number-1
```

vcmphz.t && vcmplsz.t

- float32x4_t vcmphz_f32 (float32x4_t)

```
>>> 函数说明：向量浮点大于零比较
    假设 Vx 是参数，Vz 是返回值
    If Vx(i)>0 Vz(i)=11…111;   Else Vz(i)=00…000;  i=0:number-1
```

- float32x4_t vcmplsz_f32 (float32x4_t)

```
>>> 函数说明：向量浮点小于等于零比较
    假设 Vx 是参数，Vz 是返回值
    If Vx(i)≤0 Vz(i)=11…111;   Else Vz(i)=00…000;  i=0:number-1
```

3.6.6.9　浮点乘法指令

vmult.t && vmuli.t

- float32x4_t vmul_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量单精度乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i)=Vx(i)*Vy(i);      i=0:number-1
```

- float32x4_t vmuli_f32 (float32x4_t, float32x4_t, const int)

```
>>> 函数说明：向量单精度索引乘法
    假设 Vx,Vy,index 是 3 个参数，Vz 是返回值
    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)

```
>>> 函数说明：向量单精度乘累加
    假设 Vz,Vx,Vy 是三个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+Vx(i)*Vy(i);    i=0:number-1
    注：乘法结果舍入后再累加
```

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

```
>>> 函数说明：向量单精度索引乘累加
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+Vx(i)*Vy(index);     i=0:number-1
    注：乘法结果舍入后再累加
    index=0 ~ (128/element_size -1);
```

vmuls.t && vmulsi.t

- float32x4_t vmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量单精度乘累减
    假设 Vz,Vx,Vy 是三个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)-Vx(i)*Vy(i);   i=0:number-1
    注：乘法结果舍入后再累减
```

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

```
>>> 函数说明：向量单精度索引乘累减
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)-Vx(i)*Vy(index));    i=0:number-1
    注：乘法结果舍入后再累减
    index=0 ~ (128/element_size -1);
```

vfmula.t && vfmuls.t

- float32x4_t vfmula_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量单精度融合乘累加
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)+Vx(i)*Vy(i);    i=0:number-1
    注：乘法结果保留全部精度参与累加
```

- float32x4_t vfmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量单精度融合乘累减
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    Vz(i)=Vz(i)-Vx(i)*Vy(i);    i=0:number-1
    注：乘法结果保留全部精度参与累减
```

vfnmula.t && vfnmuls.t

- float32x4_t vfnmula_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点融合乘取负累减
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    Vz(i)=-Vz(i)-Vx(i)*Vy(i);    i=0:number-1
    注：乘法结果保留全部精度参与累加
```

- float32x4_t vfnmuls_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点乘累减
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    Vz(i)=-Vz(i)+Vx(i)*Vy(i);    i=0:number-1
    注：乘法结果保留全部精度参与累减
```

vfmulxaa.t && vfmulxaai.t

- float32x4_t vfmulxaa_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
```

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

```
>>> 函数说明：向量浮点索引复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
    index=0 ~ (128/(element_size*2) -1);
```

vfmulxas.t && vfmulxasi.t

- float32x4_t vfmulxas_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
```

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

```
>>> 函数说明：向量浮点索引复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
    index=0 ~ (128/(element_size*2) -1);
```

vfmulxss.t && vfmulxssi.t

- float32x4_t vfmulxss_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
```

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

```
>>> 函数说明：向量浮点索引复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
    index=0 ~ (128/(element_size*2) -1);
```

vfmulxsa.t && vfmulxsai.t

- float32x4_t vfmulxsa_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy 是 3 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
```

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

```
>>> 函数说明：向量浮点索引复数乘累加实部虚部部分计算
    假设 Vz,Vx,Vy,index 是 4 个参数，同时 Vz 是返回值
    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)
    注：乘法结果保留全部精度参与累加/减
    index=0 ~ (128/(element_size*2) -1);
```

vfcmul.t && vfcmula.t

- float32x4_t vfcmul_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：  向量浮点复数乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘法结果作一次舍入饱和操作，Vz(i) 中乘累加结果再作一次舍入饱和操作
```

- float32x4_t vfcmula_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数乘累加
    假设 Vz,Vx,Vy 是 3 个参数，Vz 是返回值
    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);
    注：Tmp(i) 乘累加作一次舍入饱和操作，Vz(i) 乘累加结果再作一次舍入饱和操作
```

vfcmulc.t && vfcmulca.t

- float32x4_t vfcmulc_f32 (float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数共轭乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘法结果作一次舍入饱和操作，Vz(i) 中乘累加结果再作一次舍入饱和操作
```

- float32x4_t vfcmulca_f32 (float32x4_t, float32x4_t, float32x4_t)

```
>>> 函数说明：向量浮点复数共轭乘累加
    假设 Vz,Vx,Vy 是 3 个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘累加作一次舍入饱和操作，Vz(i) 乘累加结果再作一次舍入饱和操作
```

vfcmuln.t && vfcmulna.t

- `float32x4_t vfcmuln_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点复数取负乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘法结果作一次舍入饱和操作，Vz(i) 中乘累加结果再作一次舍入饱和操作
```

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

```
>>> 函数说明：向量浮点复数取负乘累加
    假设 Vz,Vx,Vy 是 3 个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘累加作一次舍入饱和操作，Vz(i) 乘累加结果再作一次舍入饱和操作
```

vfcmulcn.t && vfcmulcna.t

- `float32x4_t vfcmulcn_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点复数共轭取负乘法
    假设 Vx,Vy 是两个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘法结果作一次舍入饱和操作，Vz(i) 中乘累加结果再作一次舍入饱和操作
```

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

```
>>> 函数说明：向量浮点复数共轭取负乘累加
    假设 Vz,Vx,Vy 是 3 个参数，Vz 是返回值
    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)
    注：Tmp(i) 乘累加作一次舍入饱和操作，Vz(i) 乘累加结果再作一次舍入饱和操作
```

3.6.6.10　浮点倒数、倒数开方、e 指数快速运算及逼近指令

vrecpe.t && vrecps.t

- `float32x4_t vrecpe_f32 (float32x4_t)`

```
>>> 函数说明：向量浮点倒数指令
    假设 Vx 是参数，Vz 是返回值
    Vz(i) ≈1/(Vx(i))  i=0:(number-1)  (快速计算 Vx(i) 的倒数值)
```

- `float32x4_t vrecps_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点倒数逼近
    假设 Vx,Vy 是两个参数，Vz 是返回值
    Vz(i) = 2 - Vx(i) * Vy(i)   i=0:(number-1)
```

vrsqrte.t && vrsqrts.t

- `float32x4_t vrsqrte_f32 (float32x4_t)`

```
>>> 函数说明：向量浮点倒数开方
    假设 Vx 是参数，Vz 是返回值
    Vz(i) ≈1/((Vx(i))^(1/2))   i=0:(number-1)  (快速计算 Vx(i) 的倒数开方值)
```

- `float32x4_t vrsqrts_f32 (float32x4_t, float32x4_t)`

```
>>> 函数说明：向量浮点开方逼近
    假设 Vx,Vy 是参数，Vz 是返回值
    Vz(i) = (3-Vx(i)*Vy(i))/2   i=0:(number-1)
```

vexpe.t

- `float32x4_t vexpe_f32 (float32x4_t)`

```
>>> 函数说明：向量浮点快速 e 指令计算
    假设 Vx 是参数，Vz 是返回值
    Vz(i) ≈ e^(Vx(i))  i=0:(number-1)      (快速计算 Vx(i) 的 e 指数值)
```

3.6.6.11 浮点转换指令

vdtos.t

- float32x4_t vdtos_f64 (float64x2_t)

```
>>> 函数说明：向量双精度浮点单精度浮点转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)={double_to_single(Vx(2i+1)), double_to_single(Vx(2i))}; i=0:(number/2-
↪1);
    64-bit 双精度浮点数转换为 32-bit 单精度浮点数
```

vftox.t1.t2 && vxtof.t1.t2 (位宽不变)

- int32x4_t vftox_f32_s32 (float32x4_t)
- uint32x4_t vftox_f32_u32 (float32x4_t)

```
>>> 函数说明：向量浮点定点转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=float_to_fix(Vx(i)); i=0:(number-1);
    将 (16-bit/32-bit) 浮点数转换为相同位宽的 U/S 定点数，
    FCR[20:16].frpos[4:0] 指定定点数的小数点位置
    16bit 定点数采用 frpos[3:0] 指示 1~16 位的小数部分
    32bit 定点数采用 frpos[4:0] 指示 1~32 位的小数部分
```

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

```
>>> 函数说明：向量定点浮点转换
    Vx 是参数，Vz 是返回值
    Vz(i)=fix_to_float(Vx(i)); i=0:(number-1);
    将 (16-bit/32-bit)U/S 定点数转换为相同位宽的浮点数，
    FCR[20:16].frpos[4:0] 指定定点数的小数点位置
    16bit 定点数采用 frpos[3:0] 指示 1~16 位的小数部分
    32bit 定点数采用 frpos[4:0] 指示 1~32 位的小数部分
```

vftox.t1.t2 && vxtof.t1.t2 (位宽扩展)

- float32x8_t vxtof_s16_f32 (int16x8_t)
- float32x8_t vxtof_u16_f32 (uint16x8_t)

```
>>> 函数说明：向量定点浮点转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=fix16_to_single(Vx(i)); i=0:(number-1);
    将 16-bit 的 U/S 定点数转换为 32-bit 单精度浮点数，
```

(下页继续)

(续上页)

```
    FCR[20:16].frpos[4:0] 指定定点数的小数点位置
    16bit 定点数采用 frpos[3:0] 指示 1~16 位的小数部分
```

vftox.t1.t2 && vxtof.t1.t2 (位宽减半)

- int16x8_t vftox_f32_s16 (float32x4_t)
- uint16x8_t vftox_f32_u16 (float32x4_t)

```
>>> 函数说明: 向量浮点定点转换
    假设 Vx 是参数, Vz 是返回值
    Vz(i)={single_to_fix16(Vx(2i+1)), single_to_fix16(Vx(2i))} ; i=0:(number/2-1);
    将 32-bit 单浮点数转换为 16-bit U/S 定点数,
    FCR[20:16].frpos[4:0] 指定定点数的小数点位置
    16bit 定点数采用 frpos[3:0] 指示 1~16 位的小数部分
```

vftoi.t1.t2 (位宽不变)

- int32x4_t vftoi_f32_s32 (float32x4_t)
- uint32x4_t vftoi_f32_u32 (float32x4_t)

```
>>> 函数说明: 向量浮点整数转换
    假设 Vx 是参数, Vz 是返回值
    Vz(i)=float_to_int(Vx(i)); i=0:(number-1);
    将 (16-bit/32-bit) 浮点数转换为相同位宽的 U/S 整型数
```

vftoi.t1.t2 (位宽减半)

- int16x8_t vftoi_f32_s16 (float32x4_t)
- uint16x8_t vftoi_f32_u16 (float32x4_t)

```
>>> 函数说明: 向量浮点整数转换
    假设 Vx 是参数, Vz 是返回值
    Vz(i)={single_to_int16(Vx(2i+1)), single_to_fix16(Vx(2i))} ; i=0:(number/2-1);
    将 32-bit 单浮点数转换为 16-bit U/S 整型数
```

vitof.t1.t2 (位宽相同)

- float32x4_t vitof_s32_f32 (int32x4_t)
- float32x4_t vitof_u32_f32 (uint32x4_t)

```
>>> 函数说明：向量整数浮点转换
    假设 Vx 是参数,Vz 是返回值
    Vz(i)=int_to_float(Vx(i)); i=0:(number-1);
    将 (16-bit/32-bit)U/S 整型数转换为相同位宽的浮点数
```

vitof.t1.t2 (位宽扩展)

- `float32x8_t vitof_s16_f32 (int16x8_t)`
- `float32x8_t vitof_u16_f32 (uint16x8_t)`

```
>>> 函数说明：向量整数浮点转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=int16_to_single(Vx(i)); i=0:(number-1);
    将 16-bit 的 U/S 整型数转换为 32-bit 单精度浮点数
```

vftoi.t1.t2.rn （round to nearest)

- `int32x4_t vftoi_f32_s32_rn (float32x4_t)`
- `uint32x4_t vftoi_f32_u32_rn (float32x4_t)`

```
>>> 函数说明：带舍入向量浮点整数转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);
    将 32-bit 单精度浮点数转换为 32-bit 的 U/S 整型数
```

vftoi.t1.t2.rz （round to zero)

- `int32x4_t vftoi_f32_s32_rz (float32x4_t)`
- `uint32x4_t vftoi_f32_u32_rz (float32x4_t)`

```
>>> 函数说明：带舍入向量浮点整数转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);
    将 32-bit 单精度浮点数转换为 32-bit 的 U/S 整型数
```

vftoi.t1.t2.rpi （round to +inf)

- `int32x4_t vftoi_f32_s32_rpi (float32x4_t)`
- `uint32x4_t vftoi_f32_u32_rpi (float32x4_t)`

```
>>> 函数说明：带舍入向量浮点整数转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);
    将 32-bit 单精度浮点数转换为 32-bit 的 U/S 整型数
```

vftoi.t1.t2.rni （round to -inf)

- `int32x4_t vftoi_f32_s32_rni (float32x4_t)`
- `uint32x4_t vftoi_f32_u32_rni (float32x4_t)`

```
>>> 函数说明：带舍入向量浮点整数转换
    假设 Vx 是参数，Vz 是返回值
    Vz(i)=single_to_int32(Vx(i)); i=0:(number-1);
    将 32-bit 单精度浮点数转换为 32-bit 的 U/S 整型数
```

3.7　minilibc

3.7.1　math

3.7.1.1　基本运算

- **fabs, fabsf**

Defined in header <math.h>

```
float       fabsf( float arg );          (1)      (since C99)
double      fabs( double arg );          (2)
```

1-2) 计算参数 *arg* 的绝对值

参数

arg - 浮点参数

返回值

返回参数 *arg* 的绝对值 (|*arg*|)

- **fmod, fmodf**

Defined in header <math.h>

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

1-2) 计算参数 *x/y* 的余数，由公式 *x* - *n* * *y* 得到，n 采用**朝 0 方向舍入**，返回值符号位与 *x* 一样。

参数

x, y - 浮点参数

返回值

返回参数 *x/y* 的余数

- **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) 计算参数 *x/y* 的余数，由公式 *x* - *n* * *y* 得到，n 采用**舍入到最接近**，返回值不能保证符号位与 *x* 一样。

参数

x, y - 浮点参数

返回值

返回参数 *x/y* 的余数

- **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) 类似*remainder()* 计算参数 *x/y* 的余数，并且把商存储在参数 *quo* 中。

参数

x, y - 浮点参数

quo - 商

返回值

返回参数 *x*/*y* 的余数，并且把商存储在参数 *quo* 中。

- **fma, fmaf**

Defined in header <math.h>

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

1-2) 计算参数 *(x * y) + z* 的结果。

参数

x, y, z - 浮点参数

返回值

返回参数 *(x * y) + z* 的结果。

- **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) 返回两个参数中较大的值。

参数

x, y - 浮点参数

返回值

返回两个参数中较大的值。

- **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) 返回两个参数中较小的值。

参数

x, y - 浮点参数

返回值

返回两个参数中较小的值。

- **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) 返回两个参数的正差值，假如 x > y, 返回 x - y，否则返回 +0。

参数

x, y - 浮点参数

返回值

返回两个参数的正差值。

- **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) 返回 not-a-number 值

参数

unused - 常量字符指针参数

返回值

返回 not-a-number 值

3.7.1.2 指数运算

- **exp, expf**

Defined in header <math.h>

```
float       expf( float arg );   (1)      (since C99)
double      exp( double arg );   (2)
```

1-2) 计算 e 的 arg 次方

参数

arg - 浮点参数

返回值

返回 e 的 arg 次方结果。

- **exp2, exp2f**

Defined in header <math.h>

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

1-2) 计算 2 的 n 次方

参数

n - 浮点参数

返回值

返回 2 的 n 次方结果。

- **expm1, expm1f**

Defined in header <math.h>

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

1-2) 计算 e 的 arg 次方 - 1

参数

arg - 浮点参数

返回值

返回 e 的 arg 次方-1 的结果。

- **log, logf**

Defined in header <math.h>

```
float       logf( float arg );   (1)      (since C99)
double      log( double arg );   (2)
```

1-2) 计算以 e 为底数，arg 为真数的对数

参数

arg - 浮点参数

返回值

返回以 e 为底数，arg 为真数的对数

- **log10, log10f**

Defined in header <math.h>

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

1-2) 计算以 10 为底数，arg 为真数的对数

参数

arg - 浮点参数

返回值

返回以 10 为底数，arg 为真数的对数

- **log1p, log1pf**

Defined in header <math.h>

```
float       log1pf( float arg );         (1)      (since C99)
double      log1p( double arg );         (2)      (since C99)
```

1-2) 计算以 e 为底数，1+arg 为真数的对数

参数

arg - 浮点参数

返回值

返回以 e 为底数，1+arg 为真数的对数

- **log2, log2f**

Defined in header <math.h>

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

1-2) 计算以 2 为底数，arg 为真数的对数

参数

arg - 浮点参数

返回值

返回以 2 为底数，arg 为真数的对数

3.7.1.3 乘方运算

- **sqrt, sqrtf**

Defined in header <math.h>

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

1-2) 计算 arg 的平方根

参数

arg - 浮点参数

返回值

返回 arg 的平方根。

- **cbrt, cbrtf**

Defined in header <math.h>

```
float       cbrtf( float arg );          (1)      (since C99)
double      cbrt( double arg );          (2)      (since C99)
```

1-2) 计算 arg 立方根

参数

arg - 浮点参数

返回值

返回 arg 的立方根。

- **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) 计算 x，y 平方和的平方根

参数

x - 浮点参数
y - 浮点参数

返回值

返回 x，y 平方和的平方根。

- **pow, powf**

Defined in header <math.h>

```
float powf( float base, float exponent );        (1)      (since C99)
double pow( double base, double exponent );      (2)
```

1-2) 计算 base 的 exponent 次方

参数

base - 浮点底数参数
exponent - 浮点指数参数

返回值

返回 base 的 exponent 次方。

3.7.1.4　三角及双曲线运算

- **sin, sinf**

Defined in header <math.h>

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

1-2) 计算 arg(弧度) 的正弦

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的正弦值。

- **cos, cosf**

Defined in header <math.h>

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

1-2) 计算 arg(弧度) 的余弦

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的余弦值。

- **tan, tanf**

Defined in header <math.h>

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

1-2) 计算 arg(弧度) 的正切

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的正切值。

- **asin, asinf**

Defined in header <math.h>

```
float       asinf( float arg );          (1)      (since C99)
double      asin( double arg );          (2)
```

1-2) 计算 arg(弧度) 的反正弦

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的反正弦值。

- **acos, acosf**

Defined in header <math.h>

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

1-2) 计算 arg(弧度) 的反余弦

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的反余弦值。

- **atan, atanf**

Defined in header <math.h>

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

1-2) 计算 arg(弧度) 的反正切

参数

arg - 角度的浮点表示

返回值

返回 arg(弧度) 的反正切值。

- **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) 计算 y/x(弧度) 的反正切

参数

y - 角度的浮点表示
x - 角度的浮点表示

返回值

返回 y/x(弧度) 的反正切值。

- **sinh, sinhf**

Defined in header <math.h>

```
float       sinhf( float arg );          (1)      (since C99)
double      sinh( double arg );          (2)
```

1-2) 计算 arg 的双曲正弦值

参数

arg - 角度的浮点表示

返回值

返回 arg 的双曲正弦值。

- **cosh, coshf**

Defined in header <math.h>

```
float       coshf( float arg );          (1)      (since C99)
double      cosh( double arg );          (2)
```

1-2) 计算 arg 的双曲余弦值

参数

arg - 角度的浮点表示

返回值

返回 arg 的双曲余弦值。

- **tanh, tanhf**

Defined in header <math.h>

```
float       tanhf( float arg );          (1)      (since C99)
double      tanh( double arg );          (2)
```

1-2) 计算 arg 的双曲正切值

参数

arg - 角度的浮点表示

返回值

返回 arg 的双曲正切值。

- **asinh, asinhf**

Defined in header <math.h>

```
float       asinhf( float arg );         (1)      (since C99)
double      asinh( double arg );         (2)      (since C99)
```

1-2) 计算 arg 的反双曲正弦值

参数

arg - 角度的浮点表示

返回值

返回 arg 的反双曲正弦值。

- **acosh, acoshf**

Defined in header <math.h>

```
float       acoshf( float arg );          (1)      (since C99)
double      acosh( double arg );          (2)      (since C99)
```

1-2) 计算 arg 的反双曲余弦值

参数

arg - 角度的浮点表示

返回值

返回 arg 的反双曲余弦值。

- **atanh, atanhf**

Defined in header <math.h>

```
float       atanhf( float arg );          (1)      (since C99)
double      atanh( double arg );          (2)      (since C99)
```

1-2) 计算 arg 的反双曲正切值

参数

arg - 角度的浮点表示

返回值

返回 arg 的反双曲正切值。

3.7.1.5 错误和伽马运算

- **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)
```

3.7.1.6　浮点运算

- **ldexp, ldexpf**

Defined in header <math.h>

```
float       ldexpf( float arg, int exp );        (1)      (since C99)
double      ldexp( double arg, int exp );        (2)
```

1-2) 计算 arg * 2 的 exp 次方

参数

arg - 浮点参数
exp - 整形参数

返回值

返回 arg * 2 的 exp 次方值。

- **scalbn, scalbnf**

Defined in header <math.h>

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

1-2, 5-6) 计算 arg * FLT_RADIX 的 exp 次方

参数

arg - 浮点参数

exp - 整形参数

返回值

返回 arg * FLT_RADIX 的 exp 次方值。

- **ilogb, ilogbf**

Defined in header <math.h>

```
int ilogbf( float arg );          (1)      (since C99)
int ilogb( double arg );          (2)      (since C99)
```

1-2) 从浮点参数 arg 中提取无偏指数的值，并将其作为有符号整数值返回。

参数

arg - 浮点参数

返回值

从浮点参数 arg 中提取无偏指数的值，并将其作为有符号整数值返回。

- **logb, logbf**

Defined in header <math.h>

```
float       logbf( float arg );          (1)      (since C99)
double      logb( double arg );          (2)      (since C99)
```

1-2) 从浮点参数 arg 中提取无偏基数独立指数的值，并将其作为浮点值返回。

参数

arg - 浮点参数

返回值

从浮点参数 arg 中提取无偏基数独立指数的值，并将其作为浮点值返回。

- **frexp, frexpf**

Defined in header <math.h>

```
float       frexpf( float arg, int* exp );      (1)     (since C99)
double      frexp( double arg, int* exp );      (2)
```

1-2) 将给定的浮点值 x 分解为归一化分数和 2 的整数幂。

参数

arg - 浮点参数
exp - 整形参数

返回值

将给定的浮点值 x 分解为归一化分数和 2 的整数幂。

- **modf, modff**

Defined in header <math.h>

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

1-2) 将给定的浮点值 arg 分解为整数和小数部分，每个部分具有与 arg 相同的类型和符号。

参数

arg - 浮点参数
iptr - 指向浮点值的指针，用于存储整数部分

返回值

将给定的浮点值 arg 分解为整数和小数部分，每个部分具有与 arg 相同的类型和符号。

- **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) 首先，将两个参数转换为函数的类型，然后在 to 的方向上返回 from 的下一个可表示值。如果 from 等于 to，则返回 to。

参数

from - 浮点参数
to - 浮点参数

返回值

首先，将两个参数转换为函数的类型，然后在 to 的方向上返回 from 的下一个可表示值。如果 from 等于 to，则返回 to。

- **copysign, copysignf**

Defined in header <math.h>

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

1-2) 用 x 的大小和 y 的符号组成浮点值。

参数

x - 浮点参数
y - 浮点参数

返回值

用 x 的大小和 y 的符号组成浮点值。

3.7.1.7　近似运算

- **ceil, ceilf**

Defined in header <math.h>

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

1-2) 计算不小于 arg 的最小整形值

参数

arg - 浮点参数

返回值

返回不小于 arg 的最小整形值。

- **floor, floorf**

Defined in header <math.h>

```
float       floorf( float arg );        (1)     (since C99)
double      floor( double arg );        (2)
```

1-2) 计算不大于 arg 的最大整形值

参数

arg - 浮点参数

返回值

返回不大于 arg 的最大整形值。

- **round, roundf**

Defined in header <math.h>

```
float       roundf( float arg );        (1)             (since C99)
double      round( double arg );        (2)             (since C99)
long            lroundf( float arg );   (5)             (since C99)
long            lround( double arg );   (6)             (since C99)
long long       llroundf( float arg );  (9)             (since C99)
long long       llround( double arg );  (10)    (since C99)
```

1-2) 计算最接近 arg 的浮点数

5-6, 9-10) 计算最接近 arg 的整数

参数

arg - 浮点参数

返回值

返回最接近 arg 的值。

- **trunc, truncf**

Defined in header <math.h>

```
float       truncf( float arg );        (1)     (since C99)
double      trunc( double arg );        (2)     (since C99)
```

1-2) 计算最大的整数，其大小不大于 arg。

参数

arg - 浮点参数

返回值

返回不大于 arg 的最大整形值。

- **nearbyint, nearbyintf**

Defined in header <math.h>

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

1-2) 将浮点参数 arg 以浮点格式舍入为整数值。

参数

arg - 浮点参数

返回值

返回参数 arg 的整形值。

- **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 arg );          (9)             (since C99)
long long llrint( double arg );          (10)    (since C99)
```

1-2) 将浮点参数 arg 以浮点格式舍入为整数值。

5-6, 9-10) 将浮点参数 arg 以整形格式舍入为整数值。

参数

arg - 浮点参数

返回值

返回参数 arg 的整形值。

第四章　玄铁 900 系列 CPU 编程

玄铁 900 系列 CPU 是基于 RISC-V 体系结构开发的处理器。本章主要介绍在 C、C++ 编程过程当中，涉及到与 RISC-V 体系结构相关的特殊用法，如指令集选择、汇编编程、以及指令 intrinsic 接口等。

本章包含如下几个部分：

- 如何选择体系结构、处理器
- *T-HEAD* 小体积运行时库 *libcc-rt*

4.1　如何选择体系结构、处理器

当前平头哥开发了多款 RISC-V 体系架构的处理器，它们统一使用平头哥 RISC-V 工具链编译开发，并通过不同的编译选项生成指定处理器的目标代码，具体的对应关系如 表 4.1 所示。

表 4.1: CPU 与选项的对应关系

	CPU	-march	-mabi	-mtune
E902 系列	E902	rv32ecxtheadse	ilp32e	e902
	E902M	rv32emcxtheadse	ilp32e	e902
	E902T	rv32ecxtheadse	ilp32e	e902
	E902MT	rv32emcxtheadse	ilp32e	e902
E906 系列	E906	rv32imacxtheade	ilp32	e906
	E906F	rv32imafcxtheade	ilp32f	e906
	E906FD	rv32imafdcxtheade	ilp32d	e906
C906 系列	C906	rv64imacxtheadc	lp64	c906
	C906FD	rv64imafdcvxtheadc	lp64d	c906
	C906V	rv64imafdcvxtheadc	lp64dv	c906
C910 系列	C910	rv64imafdcxtheadc	lp64d	c910
	C910V	rv64imafdcvxtheadc	lp64dv	c910

其中-march、-mabi 和-mtune 选项的具体作用和用法见下述章节：

- *-march* 选项
- *-mabi* 选项
- *-mtune* 选项

4.1.1　-march 选项

-march 选项用于指定生成目标文件所使用的指令集，不同的字符对应不同的指令集，对应关系如 表 4.2 所示。比如 rv32imacxthead 表示当前目标文件使用到了 32 位整型基础指令集、整型乘除指令集、原子操作指令集、压缩指令集和平头哥性能增强指令集。除扩展指令集外，其余的指令集均为 RISC-V 标准指令集，更详细的描述可见 RISC-V ISA SPEC。

表 4.2: 字符串对应的指令集

字符名称	指令集
rv32i	32 位整型基础指令集
rv32e	嵌入式 32 位整型基础指令集（与 rv32i 基本相同但只使用 16 个寄存器）
rv64i	64 位整型基础指令集
m	整型乘、除指令集
a	原子操作指令集
f	单精度浮点指令集
d	双精度浮点指令集
c	压缩指令集（即 16 位长度的指令集）
p	Packed-SIMD 指令集
v	向量指令集
xtheadc	平头哥 C 系列性能增强指令集
xtheade	平头哥 E 系列性能增强指令集
xtheadse	平头哥 Small E 系列性能增强指令集
rv32g	rv32imafd 的简写
rv64g	rv64imafd 的简写

注解： 老版本工具中，平头哥的性能增强指令集，不论是 C 系列的扩展还是 E 系列的扩展都使用 xthead 表示。目前工具仍兼容这种方式，但未来可能不兼容，不建议使用。

4.1.2　-mabi 选项

-mabi 选项用于指定生成目标文件的 ABI（Application Binary Interface）规则，简要的说明如 表 4.3 所示，更详细的描述可见 RISC-V ABI SPEC。

表 4.3: ABI 规则说明

名称	说明
ilp32e	rv32e 时 16 个寄存器的传参规则
ilp32	rv32i 时，所有类型都使用参数整型寄存器的传参规则
ilp32f	rv32i 时，单精度浮点类型参数使用浮点寄存器的传参规则
ilp32d	rv32i 时，单精度和双精度浮点类型参数使用浮点寄存器的传参规则
lp64	rv64i 时，所有类型都使用参数整型寄存器的传参规则
lp64f	rv64i 时，单精度浮点类型参数使用浮点寄存器的传参规则
lp64d	rv64i 时，单精度和双精度浮点类型参数使用浮点寄存器的传参规则
lp64v	rv64i 时，单精度和双精度浮点类型参数使用整型寄存器，向量类型参数使用向量寄存器的传参规则
lp64dv	rv64i 时，单精度和双精度浮点类型参数使用浮点寄存器，向量类型参数使用向量寄存器的传参规则

4.1.3　-mtune 选项

-mtune 选项不会影响程序执行的正确性，但指定对应的 tune 可以生成针对特定 CPU 做过代价和流水线优化的目标程序。目前平头哥所有 CPU 的 mtune 的选项指定可参考 表 4.1 。

4.2　T-HEAD 小体积运行时库 libcc-rt

运行时库（runtime library）是指一种被编译器用来是实现编程语言内置操作和内置函数以提供该语言程序运行时支持的一种程序库。这种库一般包括一些基本的数学运算，比如当指令集不包含浮点指令时，编程语言中的浮点计算就需要调用库中的软浮点函数。

在平头哥 GCC 系列编译器中，libgcc 是默认的运行时库，它的浮点计算遵循 IEEE754 的标准。libcc-rt 是平头哥为对代码大小有深度需求的嵌入式领域客户开发的另一种运行时库。它与 libgcc 的功能基本相似。同时，为了得到最优的代码尺寸，它的浮点计算函数中部分会与 libgcc 接口不兼容，具体可参考*libcc-rt 与 libgcc 浮点计算部分的差异* 章节。

4.2.1　libcc-rt 使用方法

在链接时添加选项-mccrt，即可使用 libcc-rt 替换 libgcc。

注解: 目前，该选项只在 E902 和 E906 系列下有效。

4.2.2　libcc-rt 与 libgcc 浮点计算部分的差异

为了达到更好的 code size 优化效果，libcc-rt 中对于浮点接口的实现相较于 libgcc 做了差异化处理。具体差异点见下表:

表 4.4: libcc-rt 浮点接口差异

	libcc-rt 实现标准	libgcc 实现标准
1. 对于结果精度下溢的值，不返回非规格化数，而是返回 0	a. 浮点算术运算（加、减、乘、除）对于精度下溢的值，返回 0	返回非规格化数
	b. 浮点精度转换且精度减少时（double→float），若发生下溢，返回 0	返回非规格化数
2. 对于结果精度上溢的值，返回一个不可预期的溢出值	a. 浮点算术运算（加、减、乘、除），对于精度上溢的值，返回一个难以预期的溢出值	发生上溢时，返回对应符号无穷值
	b. 浮点精度转换且精度减少时（double→float），对于精度上溢的值，返回一个难以预期的溢出值	发生上溢时，返回对应符号无穷值
	c. 浮点转为整型，当浮点值大于将要表示的整型的最值时，返回实际值对应整型二进制的低位截取	对于溢出值，返回对应符号的该整型类型的饱和值
3. 对于运算过程和结果中的非数、无穷，当作具有对应指数的规格化数处理	a. 浮点算术运算（加、减、乘、除），当操作数中存在非数、无穷时，当作具有对应指数的规格化数，继续运算	无穷与其它数加减,还是该无穷值,而无穷之间发生相减时或非数参与运算时，返回 Nan
	b. 浮点精度转换且精度增加时（float→double），对于低精度值的无穷、非数和非规格化数，高精度度当作规格化数处理其指数	无穷和非数会转换为对应的高精度的无穷和非数
	c. 浮点比较指令，将非数认作是对应的有序浮点进行比较	当参与比较的数中存在非数 (Nan) 时，比较结果认为是无序的：
		这个结果不同于大于、小于、等于，因此在判断等于、大于、大于等于、小于、小于等于的浮点比较函数中，都认为比较结果为失败；
		相对的,在判断是否为无序的函数 unord 中，认为比较结果为成功
4. 浮点加减运算，当操作数为相反数，而结果为 0 时，0 的符号位与第一个操作数相同		当两个负 0 相减（负 0 加负 0 与其等价），则返回负 0，其余时候返回正 0
5. 浮点除法，若除数或被除数中有 ±0，则返回合适符号的 0，且不记录除 0 异常		当非 0 值除 0 时，得到合适符号的无穷值

4.2.3　libcc-rt 与 libgcc 浮点计算部分的差异举例

本小节使用 C 语言举例说明 libcc-rt 与 libgcc 浮点计算部分的差异，帮助读者进行更好的理解。

差异 1: 对于结果精度下溢的值，不返回非规格化数，而是返回 0

```
#include<stdio.h>

/*
```

(下页继续)

(续上页)

```
  |s|    e     |           t            |
   0 00000001 00000000000000000000000
*/
int minest_float_value = 0x00800000;

/*
  |s|     e     |     t     |
   0 01101111111 1111...111
*/
long long double_minor_float_value = 0x37ffffffffffff;

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

  //测试浮点算术运算中的下溢
  res = fa * 0.5;
  printf("算术运算浮点下溢时结果为: %f\n", res);
  printf("    结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);


  double da = *(double*)&double_minor_float_value;
  res = (float)da;

  //测试浮点转换时的下溢
  printf("浮点转换发生下溢时结果为: %f\n", res);
  printf("    结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);

  return 0;
}
```

运算结果展示:

```
$riscv32-unknown-elf-gcc underflow.c -march=rv32imac -mabi=ilp32 -mccrt
$qemu-riscv32 a.out
算术运算浮点下溢时结果为: 0.000000
    结果用十六进制表示为: 0x00000000

浮点转换发生下溢时结果为: 0.000000
    结果用十六进制表示为: 0x00000000

$riscv32-unknown-elf-gcc underflow.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out
算术运算浮点下溢时结果为: 0.000000
```

(下页继续)

(续上页)

```
    结果用十六进制表示为: 0x00400000

浮点转换发生下溢时结果为: 0.000000
    结果用十六进制表示为: 0x00400000
```

差异 2: 对于结果精度上溢的值，返回不可难以预期的溢出值

```
#include<stdio.h>

/*
  |s|   e    |          t            |
   0 11111110 11111111111111111111111
*/
int big_float_value = 0x7f7fffff;

/*
  |s|     e     |     t     |
   0 10011111111 0000...001
*/
long long double_greater_float_value = 0x4ff0000000000001;

/*
  |s|   e    |          t            |
   0 10100000 00000000000000000000001
*/
int float_greater_int_value = 0x50000001;

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

  //测试浮点算术运算中的上溢
  res = fa * 2;
  printf("算术运算浮点上溢时结果为: %f\n", res);
  printf("    结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);


  double da = *(double*)&double_greater_float_value;
  res = (float)da;

  //测试浮点转换时的上溢
  printf("浮点转换发生上溢时结果为: %f\n", res);
  printf("    结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);
```

(下页继续)

(续上页)

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

  //测试浮点转整型时的上溢
  printf("浮点转整型发生上溢时结果为: %d\n", ires);
  printf("      结果用十六进制表示为: 0x%08x\n\n", ires);

  return 0;
}
```

运算结果展示:

```
$riscv32-unknown-elf-gcc overflow.c -march=rv32imac -mabi=ilp32 -mccrt
$qemu-riscv32 a.out
算术运算浮点上溢时结果为: 680564693277057719623408366969033850880.000000
     结果用十六进制表示为: 0x7fffffff

浮点转换发生上溢时结果为: -1.000000
     结果用十六进制表示为: 0xbf800000

浮点转整型发生上溢时结果为: 1024
      结果用十六进制表示为: 0x00000400

$riscv32-unknown-elf-gcc overflow.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out
算术运算浮点上溢时结果为: inf
     结果用十六进制表示为: 0x7f800000

浮点转换发生上溢时结果为: inf
     结果用十六进制表示为: 0x7f800000

浮点转整型发生上溢时结果为: 2147483647
      结果用十六进制表示为: 0x7fffffff
```

差异 3: 对于运算过程和结果中的非数、无穷，当作具有对应指数的规格化数处理

```
#include<stdio.h>

/*
  |s|    e     |           t             |
   0 11111101 00000000000000000000000
*/
```

(下页继续)

(续上页)

```
int quarter_inf_float_value = 0x7e800000;

/*
  |s|   e    |          t            |
   0 11111111 00000000000000000000000
*/
int inf_float_value = 0x7f800000;

/*
  |s|   e    |          t            |
   0 11111111 00000000000000000000001
*/
int nan_float_value = 0x7f800001;


int main() {
  float res;
  float fqinf = *(float*)&quarter_inf_float_value;
  float finf = *(float*)&inf_float_value;
  float fnan = *(float*)&nan_float_value;


  //测试浮点算术运算中的非数、无穷参与的运算
  res = finf - fqinf;
  printf("无穷减一个大值结果为: %f\n", res);
  printf("结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);
  res = fnan / finf;
  printf("非数参与算术运算结果为: %f\n", res);
  printf("  结果用十六进制表示为: 0x%08x\n\n", *(int*)&res);


  double dres;
  //测试浮点转换高精度时无穷和非数的处理
  dres = finf;
  printf("低精度无穷转为高精度时结果为: %f\n", dres);
  printf("       结果用十六进制表示为: 0x%08x\n\n", *(long long*)&dres);
  dres = fnan;
  printf("低精度非数转为高精度时结果为: %f\n", dres);
  printf("       结果用十六进制表示为: 0x%08x\n\n", *(long long*)&dres);


  //测试浮点无序比较
  char *cmp_res = finf < fnan ? "浮点正无穷小于浮点正非数"
```

(下页继续)

(续上页)

```
                                : "浮点正无穷不小于浮点正非数";
  printf(cmp_res);
  printf("\n");
  cmp_res = finf > fnan ? "浮点正无穷大于浮点正非数"
                        : "浮点正无穷不大于浮点正非数";
  printf(cmp_res);
  printf("\n\n");

  return 0;
}
```

运算结果展示:

```
$riscv32-unknown-elf-gcc input_inf_or_nan.c -march=rv32imac -mabi=ilp32 -mccrt
$qemu-riscv32 a.out
无穷减一个大值结果为: 25521177519070384759753095557382615859 2.000000
结果用十六进制表示为: 0x7f400000

非数参与算术运算结果为: 1.000000
  结果用十六进制表示为: 0x3f800002

低精度无穷转为高精度时结果为: 340282366920938463463374607431768211456.000000
        结果用十六进制表示为: 0x0001d008

低精度非数转为高精度时结果为: 340282407485757670766715455326270783488.000000
        结果用十六进制表示为: 0x0001d008

浮点正无穷小于浮点正非数
浮点正无穷不大于浮点正非数

$riscv32-unknown-elf-gcc input_inf_or_nan.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out
无穷减一个大值结果为: inf
结果用十六进制表示为: 0x7f800000

非数参与算术运算结果为: nan
  结果用十六进制表示为: 0x7fc00000

低精度无穷转为高精度时结果为: inf
        结果用十六进制表示为: 0x0001e008

低精度非数转为高精度时结果为: nan
        结果用十六进制表示为: 0x0001e008
```

(下页继续)

(续上页)

```
浮点正无穷不小于浮点正非数
浮点正无穷不大于浮点正非数
```

差异 4: 浮点加减运算，当操作数为相反数，而结果为 0 时，0 的符号位与被加数（被减数相同）

```
#include<stdio.h>

/*
  |s|   e    |          t            |
   0 10000000 00000000000000000000000
*/
int float_p2_value = 0x40000000;

/*
  |s|   e    |          t            |
   1 10000000 00000000000000000000000
*/
int float_n2_value = 0xc0000000;

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

  //测试加减结果为 0 时, 0 的符号
  res = p2 - p2;
  printf("+2 - +2 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = n2 - n2;
  printf("-2 - -2 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = p2 + n2;
  printf("+2 + -2 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = n2 + p2;
  printf("-2 + +2 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);

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

运算结果展示:

```
$riscv32-unknown-elf-gcc cancel_to_zero.c -march=rv32imac -mabi=ilp32 -mccrt
$qemu-riscv32 a.out
+2 - +2 = 0.000000,     结果 16 进制表示为: 0x00000000
-2 - -2 = -0.000000,    结果 16 进制表示为: 0x80000000
```

(下页继续)

(续上页)

```
+2 + -2 = 0.000000,      结果 16 进制表示为: 0x00000000
-2 + +2 = -0.000000,     结果 16 进制表示为: 0x80000000

$riscv32-unknown-elf-gcc cancel_to_zero.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out
+2 - +2 = 0.000000,      结果 16 进制表示为: 0x00000000
-2 - -2 = 0.000000,      结果 16 进制表示为: 0x00000000
+2 + -2 = 0.000000,      结果 16 进制表示为: 0x00000000
-2 + +2 = 0.000000,      结果 16 进制表示为: 0x00000000
```

差异 5: 浮点除法，若除数或被除数中有 ±0，则返回合适符号的 0，且不记录除 0 异常

```c
#include<stdio.h>

/*
  |s|   e    |          t            |
   0 00000000 00000000000000000000000
*/
int float_p0_value = 0x00000000;

/*
  |s|   e    |          t            |
   1 00000000 00000000000000000000000
*/
int float_n0_value = 0x80000000;

int main() {
  float res;
  float p0 = *(float*)&float_p0_value;
  float n0 = *(float*)&float_n0_value;

  //测试除数为 0 时结果
  res = 1.0 / p0;
  printf("+value / +0 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = -1.0 / n0;
  printf("-value / -0 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = 1.0 / n0;
  printf("+value / -0 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);
  res = -1.0 / p0;
  printf("-value / +0 = %f,\t结果 16 进制表示为: 0x%08x\n", res, *(int*)&res);

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

运算结果展示：

```
$riscv32-unknown-elf-gcc input_with_zero.c -march=rv32imac -mabi=ilp32 -mccrt
$qemu-riscv32 a.out
+value / +0 = 0.000000, 结果 16 进制表示为：0x00000000
-value / -0 = 0.000000, 结果 16 进制表示为：0x00000000
+value / -0 = -0.000000,        结果 16 进制表示为：0x80000000
-value / +0 = -0.000000,        结果 16 进制表示为：0x80000000

$riscv32-unknown-elf-gcc input_with_zero.c -march=rv32imac -mabi=ilp32
$qemu-riscv32 a.out
+value / +0 = inf,      结果 16 进制表示为：0x7f800000
-value / -0 = inf,      结果 16 进制表示为：0x7f800000
+value / -0 = -inf,     结果 16 进制表示为：0xff800000
-value / +0 = -inf,     结果 16 进制表示为：0xff800000
```

第五章　链接 object 文件生成可执行文件

链接器将各个 object 文件组合生成最后的可执行文件，object 文件中的内容可以通过链接描述文件调整排列顺序和位置。它的基本命令是

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

其中：

options 链接选项

input-file-list 所有的输入 object 文件

本章包含如下几个部分：

5.1　如何链接库

库是包装应用程序编程接口 (API, Application Programming Interface) 最常用的手段，它分为静态库和动态库。

- 静态库: 一组 object 文件的集合，即很多目标文件打包形成的一个文件，一般以”.a”作为文件的扩展名。
- 动态库: 也称为动态共相对象 (DSO, Dynamic Shared Objects)，简称共相对象，一般以”.so”作为文件的扩展名。

5.1.1　库文件的生成

- 静态库的生成：先使用编译器生成各个 object 文件，再使用 csky-elfabiv2-ar 打包生成库文件

```
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
```

- 动态库的生成：使用编译器，添加选项-shared -fPIC 生成库文件

```
csky-elfabiv2-gcc -shared -fPIC csky.c -o libcsky.so
```

5.1.2　链接库

不论是静态库还是动态库，它们的命名方式都遵照统一的规则，即 lib[name].[a/so]。链接库的方法是添加选项-l[name]和-L [libpath]。其中，libpath 指 lib[name].[a/so] 文件所在的路径。

例如，如果需要链接 libcsky.a，则需要添加链接选项-lcsky -L [libcsky.a 的路径]

5.2　代码段、数据段在目标文件中的内存布局

每个 object 文件都是由代码段、数据段等多个段组成，链接的过程其实就是将各个输入 object 文件的相似段合并加工后生成最终的可执行文件的过程，如 图 5.1 所示。

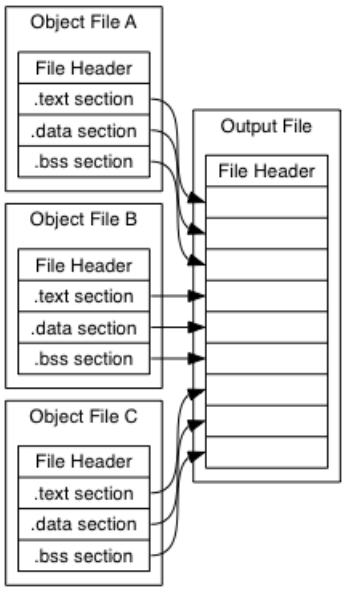

图 5.1: 链接的过程

为了精确地控制输入文件段在输出文件中的布局，链接器设计了链接脚本（Linker Script）来完成这项艰巨的任务，它通过链接器选项—T [linksript] 指定。如果不指定链接脚本，链接器会使用默认的链接脚本控制链接过程，它会将相似段合并后放在固定的地址上，一般情况下这很难满足嵌入式软件开发者的需求。

一个简单的链接描述文件如下所示：

```
ENTRY(__start)

MEMORY
{
    INST    : ORIGIN = 0x00000000 , LENGTH = 0x00020000     /* ROM */
```

(下页继续)

(续上页)

```
    DATA   : ORIGIN = 0x00400000 , LENGTH = 0x00004000    /* RAM */
    EEPROM : ORIGIN = 0x00600000 , LENGTH = 0x00010000
}
PROVIDE (__stack =  0x00404000 - 0x8);
SECTIONS
{
    .text : {
            . = ALIGN(0x4) ;
            *crt0.o(.exp_table)
            *(.text*)
            . = ALIGN (0x10) ;
    } > INST
    .rodata : {
            . = ALIGN(0x4) ;
            *(.rdata*)
            . = ALIGN(0x10) ;
    } > DATA
    .data : {
            . = ALIGN(0x4) ;
            *(.data*)
            . = ALIGN(0x10) ;
    } > DATA
    .bss : {
            . = ALIGN(0x4) ;
            *(.bss*)
            *(COMMON)
            . = ALIGN(0x10) ;
    } > DATA
}
```

链接脚本包含很多复杂的语法来控制可执行文件的生成，常用的是几个控制段在目标文件中的内存布局的语法。段的内存布局即段在目标文件中存放的地址，地址分为虚拟地址和加载地址两种：

- 虚拟地址：VMA，Virtual Memory Address，表示代码或数据在运行时的地址
- 加载地址：LMA，Load Memory Address

大多数情况下，VMA 和 LMA 都是一样的，但是在一些嵌入式系统中，特别是程序放在 ROM 的系统中，LMA 和 VMA 是不同的。指定目标文件输出段的方法如上述代码所示。

1. 使用 MEMORY 表达式定义内存区域，格式如下

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

(下页继续)

(续上页)

```
    ...
  }
```

2. 在 section 表达式中使用 “> [memory region]”，指定输出段的 VMA，格式如下

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

3. 如果不特殊指定，输出段的 VMA 和 LMA 相同，如果想指定 LMA，可在 section 表达式中使用 “AT>[memory region]”

5.3　通过 ckmap 查看生成目标文件的内存布局

如果想要查看最终生成的目标文件的内存布局和其他一些详细信息，可以通过添加链接选项-ckmap=[输出文件名]，生成 ckmap 文件查看。ckmap 主要包含以下五部分的内容：

1. Section Cross References

罗列所有的 section 之间的调用关系，格式如下：

```
([文件 A 名])([段名]) refers to ([文件 B 名])([段名]) for [符号名]
```

这个部分就是有多个上述的语句组成，表示 A 文件的某一段引用了 B 文件中的某一段定义的符号。

2. Removing Unused input sections from the image

罗列所有开启链接选项–gc-sections 后删除的段，格式如下：

```
Removing [段名]([文件名]), ([大小] bytes).
...
[number] unused seciton(s) (total [大小] bytes) removed from the image.
```

3. Image Symbol Table

分别罗列所有的本地符号和全局符号，并现实符号的地址、属性、大小、所在段名称，格式如下：

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

其中，Type 的值有以下几种：

- w：Weak，弱符号
- d：Debug，一些调试需要用到的辅助符号，如文件名、段名
- F：Function，函数名
- f：filename，文件名
- O：zero，bss 段的符号名

4. Memory Map of the image

显示目标文件的入口地址，输入文件段在输出文件中的内存布局，格式如下：

```
Image Entry point : [入口地址]
Region [内存区域名称] (Base: [起始地址], Size: [实际大小], Max: [内存区域的最大值])
Base Addr    Size         Type   Attr      Idx    Section Name       Object
[起始地址]  [大小]        [类型] [属性]    [段标号]   [段标名称]         [文件名]
...
```

每个链接到某个内存区域（Region）的输入段都会现实在上述表格中，其中 Type 有以下几种：

- Code：代码段
- Data：数据段
- PAD：为了对齐填补的区域
- LD_GEN：链接器生成的代码

Attr 有以下几种：

- RO：Read Only，只读
- RW：Read Write，可读写

5. Image component sizes

统计每个输入文件的数据在目标文件中所占用的大小，格式如下：

```
Code            RO Data         RW Data              ZI Data      Debug       Object␣
↪Name
[代码段大小] [只读数据段大小]  [可读写数据段大小]  [bss 段大小] [调试信息段大小] [输入文件名]
```

第六章　优化

本章主要介绍如何使用平头哥编译器工具来优化代码大小或者性能，以及优化级别对调试功能使用的影响。

本章包含如下几个部分：

6.1　代码大小或性能优化

编译器以及相关工具实现了多种优化技术，这些优化技术有些能够改进代码性能，有些能够减少代码大小。

使用这些优化技术需要注意它们使用之间的一些冲突，如当你使用优化代码性能技术的时候，有时候代码会增大；而当你使用那些优化代码大小的技术时，性能可能就会降低。比如，编译器优化循环性能的时候，会展开循环逻辑代码，导致代码量会增加。

默认情况下，平头哥编译器不使用任何优化，也就是说，在默认情况下，优化级别是**-O0**。

下面是一些平头哥编译器帮助选项中与优化性能有关的选项：

-O0 | -O1 | -O2 | -O3

指定编译优化级别，-O0 是最低优化，-O3 是最高优化。

-Ofast

在-O3 优化的基础上，增加一些有可能违反严格语言标准的更激进的优化。

下面是一些平头哥编译器帮助选项中与优化代码大小有关的选项：

-Os

在-O2 优化的基础上，关闭某些会增大代码的优化项，尽可能减小代码大小，有可能造成性能的损失。

下面是一些平头哥编译器帮助选项中与优化代码大小和性能都有关的选项：

-flto

使用链接时优化，该选项能够让链接器通过多个源代码文件进行额外的优化。

6.2 链接时优化

Link Time Optimization(LTO) 使得 gcc 能够把生成的内部数据结构 (GIMPLE) 存储到磁盘上，这样就能够把所有的编译单元当做一个整体来进行优化。gcc 会把 GIMPLE 输出到.o 文件的一个特殊的 section 中，当这些.o 文件链接在一起的时候，链接器会搜集所有这些特殊的 section 中的信息，通过这些信息，优化器能够判断这些模块之间的依赖关系，从而实现更多的优化。举例，以下是两个 C 代码文件 foo.c 和 bar.c 的源代码，foo.c 中 main 函数调用了 foo 函数，foo 函数调用了 bar.c 中的 bar 函数，而 bar 函数只是简单的加法运算:

```
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;
}
```

平头哥编译器通过**-flto** 选项来使用链接时优化功能，需要注意的是，我们必须在程序编译时和链接时都使用该选项，如:

```
csky-elfabiv2-gcc -c -O2 -flto foo.c
csky-elfabiv2-gcc -c -O2 -flto bar.c
csky-elfabiv2-gcc -o myprog -flto -O2 foo.o bar.o
```

另外一种比较常见的方式是:

```
csky-elfabiv2-gcc -o myprog -flto -O2 foo.c bar.c
```

在这个例子中，两个文件中函数之间相互调用，在没有使用 LTO 的情况下，foo 函数调用 bar 函数的反汇编代码:

```
foo:
    push            r15
    bsr             bar
    pop             r15

main:
    push            r15
    movi            r1, 3
    movi            r0, 12
```

(下页继续)

(续上页)

```
    bsr             foo
    pop             r15
```

而在开启 LTO 优化的情况下，main 函数不需要调用 foo 函数再调用 bar 函数，而是直接返回了加法的计算结果。显然，在开启 LTO 的情况下指令优化的更好：

```
bar.constprop.0:
    movi            r0, 15
    rts

main:
    push            r15
    bsr             bar.constprop.0
    pop             r15
```

6.3 优化选项对调试信息的影响

优化后的代码对调试信息有一定的影响，所以我们在特定情况下，需要平衡两者的功能。而且选择优化代码大小，还是选择优化性能对优化的最后结果也不一样。

一般来说，编译选项**-O0** 所编译出来的代码与调试信息的关系是最准确的，所有生成的代码结构能够直接对应到相关的源代码上。随着优化级别的提高，编译生成的代码与源代码的对应程度会越来越低，因为被编译器优化后的代码，有些很难用调试信息来表示。当然用户也可以根据自身的需要，在不想使用-O0 的情况下，可以使用优化选项**-Og** 来优化代码，尽量保证代码与调试信息的准确性。

6.4 代码优化建议

本节主要介绍一些优秀的编程经验及相关技术，用于提高 C、C++ 代码代码的可移植性、效率以及健壮性。

本节包含如下几个部分：

- 循环迭代条件优化
- 循环展开优化
- 减少函数参数传递

6.4.1 循环迭代条件优化

循环结构是程序中比较常见的一类运算，大量运算时间会耗费在这些循环上，因此那些对时间比较敏感的环境下，需要非常注意这些地方。

写循环结束条件判断，优先参考如下编码准则：

- 使用简单的条件判断

- 循环使用计数到 0
- 使用类型为 **unsigned int** 的计数器
- 测试是否等于 0

下面两个对比例子介绍计算 n！的循环运算，对比生成的汇编代码，可以看出，使用自减运算的代码能够使用一条 **jbnez** 汇编指令来达到比较跳转的功能，而在自增运算下需要两条指令，先比较 (cmp)，然后再跳转 (jbf)：

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

- 自增运算：

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

```
fact1:
    movi    a3, 1
    mov     a2, a3
.L2:
    cmplt   a0, a2
    jbf     .L3
    mov     a0, a3
    rts
.L3:
    mult    a3, a3, a2
    addi    a2, a2, 1
    jbr     .L2
```

- 自减运算：

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

```
fact2:
    mov     a3, a0
```

(下页继续)

(续上页)

```
    movi    a0, 1
.L2:
    jbnez   a3, .L3
    rts
.L3:
    mult    a0, a0, a3
    subi    a3, a3, 1
    jbr     .L2
```

6.4.2 循环展开优化

有些短循环被展开后，可以提高性能，但是代码会相应的增大一些。循环展开后，会减少循环的次数，当然也就减少了跳转的分支执行。一些小的短循环，会被完全展开，循环的代价彻底消失。在优化级别**-O3** 情况下，循环展开特性被自动启用，其他情况下，需要在代码中手动展开循环。

下面两个对比例子介绍数据拷贝的循环运算，在循环展开情况下，能够减少跳转带来的性能损耗，一般情况下，运行性能会比循环不展开的好，但是缺点就是会增加代码的大小：

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

- 循环不展开：

```
int countbit1(unsigned int n, char *d, char *s)
{
    int bits = 0;
    while (n != 0)
    {
        d[bits] = s[bits];
        bits++;
        n -= 1;
    }
    return bits;
}
```

```
countbit1:
    addu    a3, a2, a0
.L2:
    cmpne   a2, a3
    jbt     .L3
    rts
.L3:
    ld.b    t0, (a2, 0)
    st.b    t0, (a1, 0)
```

(下页继续)

(续上页)

```
    addi    a2, a2, 1
    addi    a1, a1, 1
    jbr     .L2
```

- 循环展开:

```
int countbit2(unsigned int n, char *d, char *s)
{
    int bits = 0;
    while (n != 0)
    {
        d[bits+0] = s[bits+0];
        d[bits+1] = s[bits+1];
        d[bits+2] = s[bits+2];
        d[bits+3] = s[bits+3];
        bits += 4;
        n -= 4;
    }
    return bits;
}
```

```
countbit2:
    movi    a3, 0
.L2:
    cmpne   a0, a3
    jbt     .L3
    rts
.L3:
    ld.b    t0, (a2, 0)
    st.b    t0, (a1, 0)
    ld.b    t0, (a2, 1)
    st.b    t0, (a1, 1)
    ld.b    t0, (a2, 2)
    st.b    t0, (a1, 2)
    ld.b    t0, (a2, 3)
    st.b    t0, (a1, 3)
    addi    a3, a3, 4
    addi    a2, a2, 4
    addi    a1, a1, 4
    jbr     .L2
```

6.4.3 减少函数参数传递

对于函数参数的传递需要注意以下几点：

- 在 abiv2 情况下，函数有 4 个整形参数寄存器，若使用了硬浮点功能的情况下，会有 4 个浮点寄存器，因此，在函数调用的情况下，尽可能减少参数的个数，能让其个数在寄存器个数范围内，提高效率。
- 在 C++ 情况下，非静态函数隐含的 this 指针是通过 r0 传递的，因此参数寄存器的个数会相应的减少一个。
- 把相关的参数放到一个结构体当中，然后函数调用时，通过传递该结构体指针来进行参数的传递。这样也就减少了寄存器使用的个数。
- 减少使用 long long 类型的参数，它会占用 2 个寄存器。
- 在使用软浮点的情况下，要减少使用 double 类型的参数。

第七章　编程要点

本章介绍开发人员开发过程中经常会碰到的几个问题，主要包含以下内容：

- 外设寄存器
- *Volatile* 对编译优化的影响
- 函数栈的使用
- *inline* 函数
- 内存屏障（*Memory Barriers*）
- 变量和函数 *Section* 的指定
- 将函数、数据指定到绝对地址
- 延时操作
- 自定义 *C* 语言标准输入输出流
- 基本的 *ABI* 描述
- 变量同步
- 自修改代码的注意事项

7.1　外设寄存器

外设寄存器的操作是嵌入式软件开发中频繁遇到的场景，由于它的一些特殊性（比如容易被编译器优化），本节专门介绍外设寄存器相关的常用开发方式，以帮助开发者避免一些不必要的麻烦。

7.1.1　外设寄存器描述

本小节阐述如何使用 C 语言来描述外设寄存器，使代码在编译器开启优化条件下，依然能被编译成正确而又高效的指令序列，同时保持良好的阅读性。方法如下：

1. 定义外设寄存器标志宏，把外设定义为 volatile 类型，并给输入型的外设寄存器修饰上 const 属性，使得该寄存器只能被读取，当往里面写入值时，编译器会报警告信息。

```
#define __I  volatile const
#define __O  volatile
#define __IO  volatile
```

2. 定义和外设寄存器编程模型相应的数据结构，并使用相应的 IO 修饰这些寄存器。例如：

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

3. 定义外设寄存器的操作宏，例如：

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

4. 定义好上述宏之后，就可以在程序中引用这些宏读写外设寄存器了。例如：

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

当然，用户可以进一步把 SOC_UART0->TXD 定义为 Uart0_TXD，以方便用户在代码中对外设寄存器的操作。

7.1.2　外设位域操作

外设寄存器通常会被分拆成多个表示不同功能的部分，因此为了便于控制外设寄存器，可通过结构的位域来表示外设寄存器的每个部分，每个域会有一个域名，在程序中按域名进行操作。下面是通过结构体位域操作外设寄存器的一个例子：

```
//---------------  SPI 控制寄存器 0
typedef volatile union {
  unsigned int  Word;
  struct {
    unsigned DSS          :4;
    unsigned FRF          :2;
    unsigned SPO          :1;
    unsigned SPH          :1;
    unsigned SCR          :8;
    unsigned              :16;
  } Bits;
} SPI_CR0_STR;
```

(下页继续)

(续上页)

```
#define HMS_SPI_BASE        0x40003800
#define _SPI_CR0            *(SPI_CR0_STR *)(HMS_SPI_BASE +  0x000) //SPI Control␣
↪Register 0
#define SPI_CR0              (_SPI_CR0).Word
#define SPI_CR0_DSS          (_SPI_CR0).Bits.DSS
#define SPI_CR0_FRF          (_SPI_CR0).Bits.FRF
#define SPI_CR0_SPO          (_SPI_CR0).Bits.SPO
#define SPI_CR0_SPH          (_SPI_CR0).Bits.SPH
#define SPI_CR0_SCR          (_SPI_CR0).Bits.SCR

void test ()
{
    SPI_CR0_DSS = 7;         //8-bit data size
    SPI_CR0_FRF = 0;         //SPI frame mode
    SPI_CR0_SPO = 0;
    SPI_CR0_SPH = 0;         //SPI mode 00
    SPI_CR0_SCR = 0;         //clock post-scaler=1
}
```

7.2 Volatile 对编译优化的影响

（1）不会将重复被使用的变量进行缓存优化

如果变量前不加关键词 Volatile，编译器发现该变量被连续使用了两次以上，便会通过寄存器将变量的值进行缓存，而非每次都从初始的内存位置中读取。Volatile 表明变量的值可能会在外部被改变，每次使用都需要重新存取，因此编译器不会进行缓存优化。

（2）不做常量合并、常量传播等优化。

编译器的数据流分析会分析变量的赋值、使用以便进行常量合并、常量传播等优化，进一步消除死代码。当程序不需要这些优化时，可通过 Volatile 关键词禁止做此类优化，例如下面的代码，if 条件不会一直为真。

```
Volatile int i = 1;
if (i)
...
```

7.3 函数栈的使用

在 c/c++ 中，栈被频繁使用，栈可以保存以下的内容：

1. 局部变量。由于寄存器数量有限，有些局部变量不能存储在寄存器中，栈会给这些变量分配空间。
2. 溢出的参数。由于被调用函数的参数数量大于传参寄存器的数量或者由于参数位宽较大，一个传参寄存器不足以存储整个参数，从而导致传参寄存器只能存储部分参数，其余参数或参数的一部分被溢出，其值被存储在栈中。

3. 不能被寄存器传递的返回值。如果返回值的位宽大于存储返回值的寄存器的位宽和，该返回值存储在栈中，例如位宽大于 8 字节的结构体类型的参数。
4. 寄存器原来的值。部分寄存器原来的值需要被保护，因此使用这部分寄存器来存储函数的局部变量需要将寄存器原来的值入栈，在被调用函数返回主函数之前再将栈的值存入寄存器。
5. 函数的返回地址。
6. 除上述几点之外，如果使用了 alloc() 函数，也会在栈中占用一部分内存。

一般情况下，很难估计栈的使用情况，因为代码的依赖性会随着程序的执行路径的不同而产生变化。以下是估计栈的使用程度的方法：

1. 编译是使用-fstack-usage，会产生后缀为.su 的文件，可查看栈的大小。
2. 链接时使用选项–callgraph，会产生一个 html 文件，可查看栈的大小。
3. 用调试器设定一个栈位置的观察点（watchpoint），观察命中情况。

为了尽可能的不使用栈，可以让程序降低对栈的需求，一般通过以下方法来解决：

1. 避免局部变量是结构体类型或数组。
2. 避免递归算法。
3. 在函数内不要写过多的变量。
4. 使用代码块作用域且在需要的作用域内定义变量。

7.4 inline 函数

inline 函数是一种能权衡代码规模和性能的方法。GCC 可以通过“inline”等关键词指导编译器去内联需要的函数, 但是否内联由编译器决定; 另外，也可以通过属性的关键词将 inline 函数强制内联。

inline 函数的定义通常会放在头文件中，因为编译器在优化内联函数时，需要知道函数定义的内容，因此 inline 函数的内容和对该函数的调用必须在同一个文件中，在使用该 inline 函数时需将头文件包含在内。

7.4.1 内联

声明 inline 函数的关键词为“inline”，如果是 C90，则使用“__inline__”，具体定义如下：

```
inline int fun1(int x,int y)
{
  return x+y;
}

int fun2 (int xx,int yy)
{
  return 2*fun1(2,6);
}
```

7.4.2 强制内联

当 GCC 不内联任何函数的时候，可以使用属性关键词“always_inline”进行强制内联，具体声明如下：

```
inline void foo (const char) __attribute__((always_inline));
```

7.4.3 inline 函数与外部调用的混合使用

当 C 语言中存在同一函数的 inline 函数定义和外部函数定义的时候，编译器只选择 inline 函数的代码。

7.5 内存屏障（Memory Barriers）

内存屏障指令，是 CPU 或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。

在 CSKY 中，内存栅栏的定义比较简单，它其实就是一段内嵌汇编指令，具体定义如下：

```
#define MEMORY_BARRIER __asm ("":::"memory");
```

7.6 变量和函数 Section 的指定

变量和函数都可以通过 gcc 的 section 属性将其指定到特定的 section，它的使用方法是在变量和函数的定义或声明中添加 __attribute__ ((section (“< 段名 >”))），可参考如下实例。

指定函数到特定 section

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

指定变量到特定 section

```
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
```

7.7 将函数、数据指定到绝对地址

将函数和数据指定到绝对地址的方法是：

1. 先将函数或者数据指定到特殊的 section，方法见*变量和函数 Section 的指定*
2. 然后在链接时，修改链接脚本，将 section 指定到相应的地址，方法见*代码段、数据段在目标文件中的内存布局*

下面一个实例，它将变量 stack 指定到地址 0x500000。

- 变量的定义

```
char stack[10000]  __attribute__  ((section ("STACK"))) = { 0 };
```

- 链接脚本修改，在 MEMORY 中添加内存区域 STACKR，并在 SECTIONS 中将 STACK 段指定到 STACKR 区域中

```
ENTRY(__start)

MEMORY
{
    INST   : ORIGIN = 0x00000000 , LENGTH = 0x00020000    /* ROM */
    DATA   : ORIGIN = 0x00400000 , LENGTH = 0x00004000    /* RAM */
    STACKR : ORIGIN = 0x00500000 , LENGTH = 0x00010000
    EEPROM : ORIGIN = 0x00600000 , LENGTH = 0x00010000
}
PROVIDE (__stack =  0x00404000 - 0x8);
SECTIONS
{
    .text : {
            . = ALIGN(0x4) ;
            *crt0.o(.exp_table)
            *(.text*)
            . = ALIGN (0x10) ;
    } > INST
    .stack : {
        . = ALIGN(0x4) ;
        *(STACK)
    } > STACKR
    .rodata : {
            . = ALIGN(0x4) ;
            *(.rdata*)
            . = ALIGN(0x10) ;
    } > DATA
    .data : {
            . = ALIGN(0x4) ;
            *(.data*)
            . = ALIGN(0x10) ;
    } > DATA
    .bss : {
            . = ALIGN(0x4) ;
            *(.bss*)
            *(COMMON)
            . = ALIGN(0x10) ;
    } > DATA
}
```

7.8 延时操作

在一些 MCU 的应用中，两个操作之间需要间隔一段时间。一些开发者比较喜欢使用一段空的循环体来达到一定的延时（具体延时时间根据指令执行的条数来计算），这种风格代码有以下缺点：

1. 操作比较危险，一段无用的空操作，往往会被编译器直接删除掉，从而没有延时效果
2. 延时时间会随编译的优化选项不同而发生变化
3. 不利于不同频率下的应用移植性

建议做采用以下方法：

抽象一个延时函数，如 delay_us(int val)，该函数使用汇编或者内嵌汇编的方式编写，以防止编译器优化对其的影响。延时函数如 delay_us 函数中可以根据当前系统的工作频率来调整执行的指令数，用于适应不同的系统工作频率。

假设系统的工作频率是 20Mhz，801 的延迟函数可按如下方式实现：

```
.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
    bf .L1
.L2:
    rts
.size delay_us, .-delay_us
```

7.9 自定义 C 语言标准输入输出流

由于嵌入式平台开发的特殊性，在当前的 CSky 平台提供的 C 语言标准库中，标准输入 scanf 和输出函数 printf 会使用钩子函数 fgetc 和 fputc 来实现相应的输入输出的功能，从而提高开发的灵活性。用户在开发的时候，如果必要时，则需要提供用户自定义的 fgetc 和 fputc 函数。

7.10 基本的 ABI 描述

当用户混合使用 C 与汇编代码进行开发的时候，此时需要关心应用程序二进制接口，该接口说明了参数该如何传递，返回值如何存放。本节将简要介绍第二版 CSKY ABI 中基本数据类型变量的传递与返回的机制。

7.10.1 函数参数传递

传递的参数类型分为两种，一种是基本数据类型 (char, short, int, long 等)，另外一种是聚合类型 (数组，结构体，类等)。对于聚合类型的参数传递方式请参阅 *CSKY 应用程序二进制接口规范 (第二版)*。

首先需要注意的是，csky 系列编译器对大小低于 32 位的类型传递会执行扩充操作，使其大小可以存放在一个 32 位的寄存器中。当前 CSKY ABI 规范规定前四大小低于 4 字节的参数能够使用寄存器 r0-r3 传递，剩下的其他参数使用栈槽 (stack slot) 进行传递。以如下代码为例，

```
void bar(char ch, short sh, int i, long l, int rem1)
{
  int res = ch + sh + i + l + rem1;
}
```

ch, sh, i, l 将依次存放在 r0, r1, r2, r3(或别名 a0, a1, a2, a3) 寄存器中，rem1 将会存放在 sp-8 所在位置的栈槽中。

7.10.2 函数返回值传递

ABI 除了规定如何往被调函数 (callee) 传入数据之外，也规定了如何从被调函数 (callee) 中读取数据，该功能需要主调函数和被调函数协调一致地实现。被调函数按照 ABI 规范将某个类型的数据放置在既定位置 (r0/r1) 或者栈槽 (stack slot) 中。如下例子：

```
int bar(int a, int b)
{
  return a + b;
}
```

上述代码中的返回值将会放置在 r0 寄存器中，待主调函数使用。

当被调函数返回聚合类型的值时，如，一个较大的结构体数组，此时 csky 系列 CPU 中的寄存器无法提供足够大的空间供返回值传递使用。如下例子：

```
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;
```

(下页继续)

(续上页)

```
  st.c = a + b;
  st.d = a - b;
  st.e = a * b;
  return st;
}
```

上述代码中展示了聚合类型返回值的传递规则，根据 CSKY ABI 规范第二版，聚合类型的返回值将会使用间接传递的方式实现。首先，主调函数为该返回值在主调函数的调用栈中分配一段空间，然后将该段空间的入口地址作为函数调用的第一个参数传入被调函数，被调函数则使用该地址来实现各项操作。变换之后的等价 C 语言代码类似如下的形式：

```
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;
  st->c = a + b;
  st->d = a - b;
  st->e = a * b;
}
```

7.11 变量同步

变量同步是应用开发中的常见问题，可使用 volatile 声明实现变量的同步，在多任务编程中，CSKY 体系结构为用户提供了 idly4 指令屏蔽中断。

7.11.1 使用 volatile 同步变量

在开发应用的过程中，经常会碰到变量同步的问题。例如在某应用场景下，接收数据的中断服务程序每次收到数据后，把数据放入接收 buffer 中，并更改全局变量 Received_flag 通知主程序去处理；主程序则不断地读取该变量，当该变量被置位的时候，则调用处理函数去处理接收 buffer 中的数据。用户经常忽略了该全局变量的特殊性，直接采用普通的申明的方式，这将可能导致应用程序直接陷入死循环。为什么会这样?

我们来简单分析下原因，一个全局变量在编译链接完成之后，编译工具将为该变量分配一块内存空间，作为其值的存放空间。当用户在 C 语言中访问（读取）该变量时，编译工具会生成相应的访问内存指令，去获取变量内存空间的值。但我们在 C 语言中重复写同样的语句，不断地去读取同一个变量中的值时，编译工具将如何处理这种情况呢？不开启优

化，会为每一条 C 语句生成相应的访问内存指令，依次去获取内存的值；开启优化的情况下，编译器只产生一条访问内存的指令，后面的将会一直使用原来的值用于处理和计算，因为编译器认为当前作用域下，认为内存的值是不会变化的。

如何正确应对这种情况呢？最简单的方法就是使用 volatile 了修饰此类全局变量（Received_flag），编译器将为每次的变量访问操作产生访问内存指令，去获取内存中最新的值用于计算。

7.11.2 多任务编程中的变量同步

在不同任务（或者线程）中，如何去访问并更改同一个变量，是在多任务编程中比较关键的问题。在 CSKY 体系结构中，为用户提供了特殊的指令 (idly4)，用于此类场景。

idly4 的指令功能定义如下，该指令执行后，其后面四条指令执行的过程将屏蔽中断，若执行过程有异常产生，则置高标志位（C 位），通知用户有异常发生。应用实例如下：

```
bmaski r1, 32            ; 获取 all 1's 常量
lrw r2, Semaphore        ; 获取内存中信号量指针
idly4                    ; 开始不可中断 4 号指令队列
ld.w   r3,(r2,0)         ; 从内存中读取信号量
bt Sequence_failed       ; 检查异常 (可选)
or r1,r1                 ; No-Op
st.w r1,(r2,0)           ; (id) Set semaphore to all 1's
bt Semaphore_corrupted   ; 检查异常是否发生 (可选)
cmpnei r3,0              ; 信号量测试
```

7.12 自修改代码的注意事项

在 BootLoader 中，通常需要把一段代码从存储地址搬运到其运行地址，并跳转到地址；在一些场景下，需要动态地改变指令码，并跳转执行，这些都可以认为是自修改代码的行为。

在 CSKY 体系结构中，由于是冯诺依曼和哈弗可配，所以做此类操作时，需要考虑 CPU Cache 和内存内容一致问题。即修改完成之后需要把 D Cache 中的内容 Clear 到内存，同时 Invalid I Cache 中的内容，才能做最后的跳转操作。

第八章　二进制工具的使用

除了编译器、汇编器、链接器、调试器之外，平头哥工具链还包含很多二进制工具：

- *-*-addr2line - 根据程序地址得到其所在的源代码文件名和行号。
- *-*-ar - 创建、修改和提取静态库（archive）。
- *-*-c++filt - 获得被重载的 C++ 符号的原符号名。
- *-*-gprof - 显示程序分析（profiling）信息。
- *-*-nm - 列出给定目标文件中的符号。
- *-*-objcopy - 复制和转化目标文件。
- *-*-objdump - 显示目标文件的信息。
- *-*-ranlib - 为静态库（archive）生成内容的索引。
- *-*-readelf - 现实 ELF 格式的目标文件的信息。
- *-*-size - 罗列目标文件或静态库的段大小。
- *-*-strings - 列出文件中所有的可打印字符串。
- *-*-strip - 移除目标文件中的符号信息，减小文件大小。

这些工具中的一些工具会在开发中经常用到，下面介绍的两个小节内容只使用了其中的两个工具：

8.1　ELF 文件常用信息的查看和分析

ELF 文件即 ELF（Executable and Linkable Format）格式的目标文件，可以通过 *-*-readelf 命令查看相关信息，通过不同的选项可以查看不同的信息。

1. -S

显示程序的段信息，格式如下：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
```

(下页继续)

(续上页)

```
  [ 1] .text             PROGBITS        00000000 001000 0022c0 00  AX  0   0 1024
  [ 2] .rodata           PROGBITS        00400000 004000 000550 00   A  0   0  4
  [ 3] .data             PROGBITS        00400550 004550 000230 00  WA  0   0  4
  [ 4] .bss              NOBITS          00400780 004780 000090 00  WA  0   0  4
```

其中，

- Name：段名称
- Type：段类型，它的常见值如下所示
  - NOBITS：文件中不需要存储的程序数据，一般是.bss 段的类型
  - PROGBITS：与 NOBITS 相对应，在文件中存有程序数据，除.bss 段之外的代码段和数据段一般都是这个类型
  - SYMTAB：符号表，一般是.symtab 段的类型
  - STRTAB：字符串表，一般是.strtab 段的类型
- Addr：段的起始运行地址
- Off：段在文件中的偏移
- Size：段大小，单位为 byte
- Flg：段属性，它的常见值如下所示
  - W：Write，可写的
  - A：Alloc，执行时需要加载到内容中的
  - X：Execute，可执行的
  - M：Merge，链接器认为可以合并的，并且链接器会尝试合并压缩段
  - S：Strings，段内容是字符串
- Al：段的对齐要求，单位为 byte

2. -l

显示程序头（program header）信息，格式如下：

```
 Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 LOAD           0x001000 0x00000000 0x00000000 0x022c0 0x022c0 R E 0x1000
 LOAD           0x004000 0x00400000 0x00400000 0x00780 0x00810 RW  0x1000

Section to Segment mapping:
 Segment Sections...
  00     .text
  01     .rodata .data .bss
```

所谓的 program 是多个 section 拼成的，属于同一个 program 有相同的属性。在程序执行时，文件是以 program 为单位加载到内存中的。Program 各个字段的含义如下：

- Type：program 类型，一般都为 LOAD，表示 program 需要加载到内存
- Offset：program 在文件中的偏移
- VirtAddr：program 的运行地址

- PhysAddr：program 的加载地址
- FileSiz：program 在文件中的大小
- MemSiz：program 加载到内存的大小
- Flg：program 属性，有以下几种值
  - R：Readable，可读的
  - E：Executable，可执行的
  - W：Writable，可写的
- Align：program 的对其要求

3. -s

显示程序的符号表，格式如下：

```
Symbol table '.symtab' contains 170 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     ...
     96: 00001a38   386 FUNC    GLOBAL DEFAULT    1 printf
     ...
```

这个选项会列出程序的所有符号和符号的相关信息，它的每个字段的含义如下：

- Value：符号的地址
- Size：符号的大小（比如函数或者变量的大小）
- Type：符号类型，有以下几种常见的值
  - FUNC：函数名
  - OBJECT：变量名
  - FILE：文件名
  - NOTYPE：没有声明类型的符号
- Bind：符号的作用域范围，有以下几种常见的值
  - LOCAL：本地符号
  - GLOBAL：全局符号，即其他文件可以访问
  - WEAK：弱符号
- Name：符号名称

8.2　bin 和 hex 文件生成方式

Bin 文件就是二进制文件，内部没有地址标记。一般用编程器烧写时从零地址开始，而如果下载运行，则下载到链接时的运行地址即可。Bin 文件可以通过转化 ELF 文件得到，命令如下：

```
*-*-objcopy -O binary [输入 ELF 文件] [输出 bin 文件]
```

Hex 文件经常被用于将程序或数据传输存储到 ROM、EPROM，可以通过转化 bin 文件得到，命令如下：

```
*-*-objcopy -I binary -O [输出 bin 文件] [输入 hex 文件]
```

第九章　图表

9.1　gcc 约束相关代码

9.1.1　CSKY 体系结构相关约束

约束	描述
a	使用 r0 - r7 寄存器
b	使用 r0 - r15 寄存器
c	使用 c 寄存器
y	使用 hi 或者 lo 寄存器
l	使用 lo 寄存器
h	使用 hi 寄存器
v	使用 vector 寄存器
z	使用 sp 寄存器

9.1.2　gcc 公共约束代码

约束	描述
m	使用变量的内存位置
r	使用任何可用的通用寄存器
i	使用立即整数值
g	使用任何可用的寄存器或者内存位置

9.1.3　gcc 输出修饰符

输出修饰符	描述
+	可以读取和写入操作数
=	只能写入操作数
%	如果必要，操作数之间可以交换顺序
&	在内联函数完成之前，可以删除或者重新使用操作数

索引

H

S