XuanTieCustomExtension

发行版本 *V0.9.0*

XuanTie

2025 年 06 月 16 日

杭州中天微系统有限公司 Hangzhou C-SKY MicroSystems Co., LTD

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

Website: www.xrvm.cn

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。杭州中天微系统有限公司不对任何第三方使用本文档产生的损失承担任何法律责任。

杭州中天微系统有限公司 Hangzhou C-SKY MicroSystems Co., LTD

地址：中国浙江省杭州市网商路 699 号 5 号楼 2 楼 201 室

网址：www.xrvm.cn

版本历史

版本	描述	日期
0.9.0	初版	2025-06-13

XuanTieCustomExtension

第一章　简介

物联网和家庭自动化设备等智能设备市场的迅猛增长要求 SoC 设计以最低成本和低功耗提供高性能。为了帮助设计人员应对这一挑战，XuanTie Custom Extension™ (XCE) 应运而生，客户可以将自己的指令添加到 XuanTieCore™ 处理器中。通过将 XuanTieCore™ 的高性能高效基线功能与为客户应用量身定制的 XCE 指令相结合，SoC 可以从两方面获得优势：处理器可编程性的完全灵活性和硬连线引擎的绝对效率。

用户首先通过在模拟器或者性能分析工具上分析 C 程序，识别那些占执行时间很大比例的代码段，根据这些热点代码段的语义来定义 XCE 指令。这些热点代码段可以成为 XCE 描述文件中指令语义的一部分。该文件还指定了指令助记符、操作数等。使用自动生成的 XCE 内部函数替换原始代码以表示相应的 XCE 汇编指令，可以再次分析应用软件以获得估计的性能改进。

自定义指令开发工具 (XuanTie IGN™) 解析 XCE 描述文件并生成工具链识别 XCE 指令所需的共享库。然后，配备自定义指令的新处理器即可进行 SoC 集成和软件开发。XCE 和 IGN™ 环境的简单性使用户可以专注于 XCE 指令本身的功能，由于扩展组件都是在本地生成的，因此户可以轻松尝试所需的各种的实验。

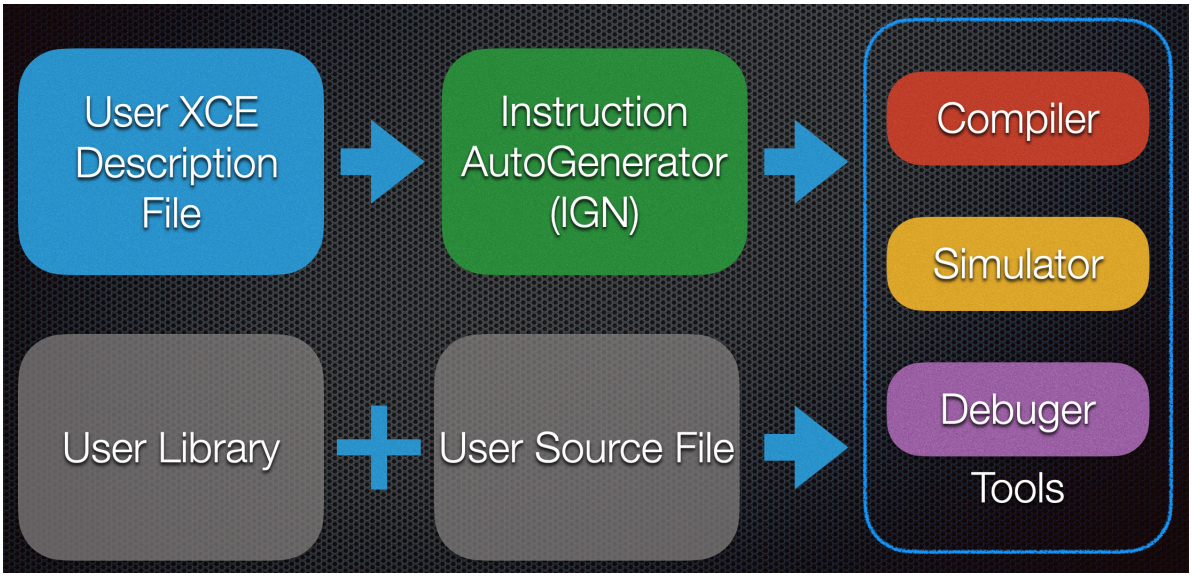


图 1: 工具链对输入的 XCE 描述文件的处理流程

第二章　快速开始

2.1　定义指令规范

一般我们通过模拟器或者性能分析工具，确认 C 代码的热点，根据热点指令来确定需要设计/定义的指令，此处我们假设已经确定热点，根据热点指令确定要扩展一条整形加法指令 **add**。

虽然自定义指令的定义可以以任意的形式出现，但是遵循 RISC-V 社区的标准规范能使我们自定义的指令更易懂，维护也更方便。我们需要定义一条整形加法指令，按照规范，一般我们需要以下几步来设计指令：

- 指令汇编语法，指令功能一般有寄存器与寄存器运算的，也有寄存器与立即数运算的，这里我们以寄存器与寄存器运算的 add 指令为例定义汇编语法格式。

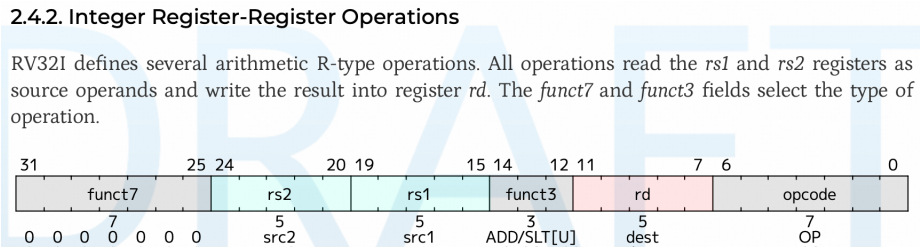
  我们把指令名称定义成 **xt.add**，需要在指令名称前面增加厂商简称约定，此处以 XuanTie（**xt**）为例：

```
xt.add rd, rs1, rs2
```

- 指令行为，一般我们用类似 C 语言的伪代码描述指令的功能：

```
rd = rs1 + rs2;
```

- 指令编码，根据 RISCV 编码规范，我们的 add 编码格式如下：



2.2　编写 XCE 描述文件

我们可以把之前传统的指令定义方法，统一描述在 **XCE 描述文件**中：

```
{
    "type": "integer",                          // 指令类型，由IGN工具定义
    "instructions": [
        {
            "asm": "xt.add $rd, $rs1, $rs2",// 指令汇编语法，由用户定义
            "encoding": [                       // 指令编码域，由用户定义
                {
                    "size": 7,                  // 指令编码域宽度(单位：bit)，由用户定义
                    "value": "OP"               // 指令编码域值，由用户定义
                },
                {
                    "size": 5,
                    "value": "rd",
                    "constraint": "="           // 指令编码域约束，由IGN工具定义
                },
                {
                    "size": 3,
                    "value": "0b000"
                },
                {
                    "size": 5,
                    "value": "rs1"
                },
                {
                    "size": 5,
                    "value": "rs2"
                },
                {
                    "size": 7,
                    "value": "0b0000000"
                }
            ],

            "execute": {
                "c": [
                    "rd = rs1 + rs2;"           // 指令行为c语言功能描述，由用户定义
                ]
            }
        }
    ]
}
```

更详细的描述语法请移步下面章节查看：

- *XCE 描述文件语法定义*

更详细的描述案例请移步下面章节查看：

- *XCE 描述文件案例*

2.3 生成工具链插件

一般 **IGN** 工具随我们的编译器工具一起发布，在编译器的安装根目录下可以找到 IGN 工具 **ctb**。ctb 是 IGN 工具的 driver，通过 ctb 命令来使用 IGN 工具生成插件。下面是我们编译器工具的目录结构：

```
├── bin
├── include
├── lib
├── libexec
├── riscv64-unknown-linux-gnu
├── share
├── sysroot
└── xuantie
    └── tools
        └── ctb
```

我们可以通过 `-c iconfig=` 命令行参数来生成插件：

```
ctb -m igen --update=igen -c iconfig=</path/to/xce description file>
```

默认情况下，会在当前目录生成 *install-riscv64-linux-glibc* 目录，里面有我们对应生成的头文件和插件，生成的相关资源名称都会带上用户在 *xce 描述文件*中描述的 “**name**” 字段内容（缺省情况下为 XCE 描述文件名，这里我们以 **xti** 举例）。

```
install-riscv64-linux-glibc/
├── libxti.h
└── libxti-compiler.so
└── libxti-disassembler.so
└── libxti-simulator.so
```

当然我们也可以通过参数 `--prefix=</path/to/install dir>` 指定输出内容的安装目录，更多 *igen* 使用命令，请移步下面章节查看：

- *IGN 工具帮助手册*

如果是在 Windows 系统下使用，需要配置相关环境，详见 *Windows 系统环境配置*

2.4 调整用户代码

我们在设计/生成自定义指令功能后，需要调整 C 代码，以确保我们能把预期的自定义指令功能用起来。用法也是相当简单，我们只需要包含自定义指令功能接口文件就行，此处就是 *libxti.h*。

目前 RISCV 社区已经有一套较为完善的指令 C 接口定义规范，整体上 IGN 生成的接口名称都遵循社区的规范。一般来说，接口形式如下：

```
INTRINSIC ::= PREFIX NAME [ '_' TYPE ]
PREFIX ::= "__riscv_"
NAME ::= Name of the intrinsic function.
TYPE ::= Optional type postfix.
```

此处以 xt.add 为例：

```
#include "libxti.h"
size_t __riscv_xt_add(size_t rs1, size_t rs2);
```

更多针对 C 接口定义规范的细节，请详见社区规范：

- 通用 intrinsic 接口规范
- Vector intrinsic 接口规范

2.5 工具链使用插件

我们在编译自定义扩展指令的用户程序时，不仅要输入用户程序，同时也要指定用户自定义指令的 xce 描述文件生成的插件，此处以用户程序 *main.c / main.elf* 为例：

2.5.1 编译器

```
clang main.c -o main.elf -miconfig=</path/to/libxti-compiler.so>
```

2.5.2 模拟器

```
qemu-riscv64 -dsa </path/to/libxti-simulator.so> main.elf
```

2.5.3 调试器

```
gdb main.elf
set disassembler-option -Mdsa=</path/to/libxti-disassembler.so>
```

2.5.4 反汇编器

```
llvm-objdump -d main.elf -mllvm -load -mllvm </path/to/libxti-disassembler.so>
```

第三章　XCE 描述文件语法定义

XCE 描述文件主要由三层结构组成：**模块**，**指令**和 **指令编码 / 指令行为**组成。

3.1　模块 module

可以描述一类指令的集合，也可以把多种类型的指令描述在一个模块中。

```
module = {
    string name = ?;// 模块名称
    string type;    // 指令类型，如果指令“instruction”没有描述类型，则使用该描述
    list instructions<instruction> = []; // 指令描述集合
    // 文档描述，第一个文档元素表示文档的简述
    list doc<string> = [];
}
```

3.2　指令 instruction

描述一条指令，类似 LLVM 中的 class Instruction 作用

```
instruction = {
    string asm;                          // 指令汇编语法
    list encoding<instructionEnc> = []; // 编码域

    int size = ?                         // 长度（单位：bit）
    string type ?= MODULE.type;          // 类型

    list uses<string> = ?;               // 使用额外的寄存器描述
    list defs<string> = ?;               // 破坏额外的寄存器描述

    list etype<string> = ?;              // 操作数元素类型
    list group<string> = ?;              // 操作数寄存器个数
    list mask<string> = ?;               // 指令mask功能描述
```

(续下页)

(接上页)

```
    int cycle ?= 1;                         // 指令执行周期数

    // 行为描述
    dict execute = { "c" : [], "depend" : [] };

    // 文档描述，第一个文档元素表示文档的简述
    list doc<string> = [];
}
```

3.3 指令编码 instructionEnc

```
instructionEnc = {
    int offset = ?;                 // 偏移（从0开始）
    int size;                       // 宽度（单位：bit）
    string type = ?;                // 类型
    string value ?= "0";            // 值
    int voffset ?= 0;               // 一个变量分多段编码时的偏移

    string constraint = "";         // 约束
    bool optional = ?;              // 操作数是否可省略

    // 在操作数是变量的情况下，操作数元素类型
    string etype ?= for etype in INSTRUCTION.etype;
    string group ?= for group in INSTRUCTION.group;
}
```

3.4 指令行为 execute

目前指令的行为描述支持 C 语言描述

```
execute = {
    "c" : [
        // c语言描述列表
    ],
    "depend" : [
        // 执行依赖描述列表
    ]
}
```

3.5 描述文件关键字定义

3.5.1 指令类型（type）

表 1: 指令类型

类型描述	作用
（缺省）	没有类型，根据描述文件推断
integer	整形指令类型
float	浮点指令类型
vector	向量指令类型

3.5.2 操作数类型（type）

表 2: 只读数据类型

类型描述	作用
（缺省）	没有类型，根据 **value** 描述推断
bits	• 描述二进制数据，如 “0b110” • 描述无符号整型数据，如 “20”、“0x14”
imm	描述有符号整型数据类型
float	描述浮点常量数据类型

表 3: 变量（寄存器）数据类型

类型描述	作用
（缺省）	没有类型，根据 **value** 描述推断
r	GPR 寄存器类型
f	float 寄存器类型
vr	vector 寄存器类型
vm	vector 中的 mask 寄存器类型

这部分的类型描述我们与 RISCV 的编译器社区保持一致，详见 RISCV 寄存器 spec

3.5.3 操作数约束（constraint）

表 4: 约束类型

类型描述	作用
（缺省）	设置为“读”操作数
“=”	设置为“写”操作数
“+”	设置为读和写
“&”	设置当前操作数的寄存器不能与其它操作数的寄存器一样

这部分的类型描述我们与 RISCV 的编译器社区保持一致，详见 RISCV 约束 spec

3.5.4 操作数值（value）

这里所说的操作数，主要指 **encoding** 中的各个域的描述，其中如果某个域的值（value）会表现在指令的 **asm** 中，一般称这种域的类型是变量类型。变量类型的操作数我们需要增加一个 **$** 前缀出现在 **asm** 中。

3.5.5 操作数元素类型（etype）

操作数元素类型缺省情况下由指令的 **etype** 决定，而指令 **etype** 在缺省的情况下为“size_t”。

表 5: 元素类型

类型描述	C 语言数据类型描述
i8	int8_t
u8	uint8_t
i16	int16_t
u16	uint16_t
i32	int32_t
u32	uint32_t
i64	int64_t
u64	uint64_t
f16	_Float16
f32	float
f64	double
8	相当于同时支持 i8 和 u8
16	相当于同时支持 i16 和 u16。如果是浮点指令，则相当于 f16
32	相当于同时支持 i32 和 u32。如果是浮点指令，则相当于 f32
64	相当于同时支持 i64 和 u64。如果是浮点指令，则相当于 f64
i128	int128_t / __int128
u128	uint128_t / unsigned __int128
bf16	__bf16
f8e4	uint8_t
f8e5	uint8_t

3.5.6 操作数寄存器个数（group）

在某些指令设计中，有可能需要多个寄存器来存储更大的数据量，比如 vector，这里我们通过描述 group 列表来表示操作数所支持的寄存器个数情况。缺省情况下，操作数寄存器个数由指令的 **group** 决定，而指令的 **group** 在缺省情况下为 [“1”]

3.5.7 指令行为（execute）

目前指令的行为描述支持 C 语言描述。**IGN** 工具提供了大量内置变量和函数接口，用户在描述 C 语言过程当中可以直接使用，提升描述效率：

- 第一类是寄存器资源变量，主要是 CSR 寄存器，用户可以直接引用、操作。
- 第二类是由用户在指令 **asm** 描述中出现的变量，用户也可以直接引用、操作。

- 第三类是内置函数接口，一般情况下以 *ign* 函数名前缀出现。

详细的接口使用文档，请移步*IGN 内置接口手册* 查看。

同时 **IGN** 工具允许客户使用第三方库/源代码文件为指令的行为描述提供支持，目前支持两种方式：

- 一种我们通过 IGN 工具的命令行参数 `-c cflags=` 来指定，使用方法和使用编译器参数类似：

```
ctb -m igen --update=igen -c iconfig=</path/to/xce description file> \
    -c cflags="-l<libname> -L</path/to/lib dir> -I</path/to/header dir>"
```

```
ctb -m igen --update=igen -c iconfig=</path/to/xce description file> \
    -c cflags="</path/to/source> -I</path/to/header dir>"
```

- 另一种我们通过在 **execute** 中的 **depend** 来描述需要依赖的库、源文件或者头文件。依赖文件的路径是以 **XCE 描述文件**的相对路径，如：

```
"execute": {
    "c": [
        "for (int i = 0; i < vl; i++) {",
        "  if (VM || vm(i))",
        "    vd[i] = xt_float(ESIZE, add, vs2[i], rs1);",
        "}"
    ],
    "depend": [
        "xt_float.c"
    ]
}
```

第四章　XCE 描述文件案例

4.1　标量 XCE 描述

4.2　向量 XCE 描述

4.2.1　xtvdot

xt.vmaqa.vx

8-bit 矢量标量有符号乘累加链加指令

- 指令语法：

```
xt.vmaqa.vx vd, rs1, vs2, vm
```

- 指令编码：

31 - 27	26	25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
10000	1	vm	vs2	rs1	110	vd	0001011

指令说明

- vd 数据宽度为 VSEW=32-bit，vs2 数据宽度为 VSEW/4=8-bit，其他 VSEW 配置非法。
- VLMUL 支持 1/2/4/8/0.5，其他配置非法。
- mask 对源操作数起作用

更多信息，详见 xtvdot 指令 spec

指令 XCE 描述

```
{
    "type": "vector",
    "instructions": [
```

(续下页)

(接上页)

```
        {
            "asm": "xt.vmaqa.vx $vd, $rs1, $vs2, $vm",
            "encoding": [
                {
                    "size": 7,
                    "value": "custom-0"
                },
                {
                    "size": 5,
                    "value": "vd",
                    "constraint": "=&"
                },
                {
                    "size": 3,
                    "value": "0b110"
                },
                {
                    "size": 5,
                    "value": "rs1"
                },
                {
                    "size": 5,
                    "value": "vs2",
                    "etype": "$etype / 4"
                },
                {
                    "size": 1,
                    "value": "vm",
                    "etype": "$etype / 4"
                },
                {
                    "size": 6,
                    "value": "0b100001"
                }
            ],

            "etype": ["i32"],
            "group": ["1", "2", "4", "8", "f2"],

            "execute": {
                "c": [
                    "signed char operand1[4];",
                    "for (int i = 0; i < 4; i++) {",
```

(续下页)

(接上页)

```
                    "  operand1[i] = ign_bits(rs1, (i + 1) * 8 - 1, i * 8);",
                    "}",
                    "for(int i = 0; i < VLMAX; i++) {",
                    "  int32_t temp = 0;",
                    "  for(int j = 0; j < 4; j++) {",
                    "    if (VM || vm(i * 4 + j))",
                    "      temp += vs2[i * 4 + j] * operand1[j];",
                    "  }",
                    "  if (i < vl)",
                    "    vd[i] += temp;",
                    "}"
                ]
            }
        }
    ]
}
```

4.2.2 xtvector

xt.vadd.vv

- 指令语法：

```
xt.vadd.vv vd, vs2, vs1, vm
```

- 指令编码：

31 - 26	25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
000000	vm	vs2	vs1	000	vd	1010111

指令说明

- 无，标准的 vector 指令。

更多信息，详见 xtvector 指令 spec

指令 XCE 描述

```
{
    "type": "vector",
    "instructions": [
        {
```

(续下页)

(接上页)

```
            "asm": "xt.vadd.vv $vd, $vs2, $vs1, $vm",
            "encoding": [
                {
                    "size": 7,
                    "value": "OP-V"
                },
                {
                    "size": 5,
                    "value": "vd",
                    "constraint": "=&"
                },
                {
                    "size": 3,
                    "value": "OPIVV"
                },
                {
                    "size": 5,
                    "value": "vs1"
                },
                {
                    "size": 5,
                    "value": "vs2"
                },
                {
                    "size": 1,
                    "value": "vm"
                },
                {
                    "size": 6,
                    "value": "0b000000"
                }
            ],

            "execute": {
                "c": [
                    "for (int i = 0; i < vl; i++) {",
                    "  if (VM || vm(i))",
                    "    vd[i] = vs2[i] + vs1[i];",
                    "}"
                ]
            }
        }
    ]
}
```

xt.vfadd.vf

- 指令语法：

```
xt.vfadd.vf vd, vs2, rs1, vm
```

- 指令编码：

31 - 26	25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
000000	vm	vs2	rs1	101	vd	1010111

指令说明

- 标准的 vector 浮点指令。
- 会产生浮点异常，破坏 fflags 寄存器
- 使用 frm 寄存器

更多信息，详见 xtvector 指令 spec

指令 XCE 描述

```
{
    "type": "vector",
    "instructions": [
        {
            "asm": "xt.vfadd.vf $vd, $vs2, $fs1, $vm",
            "encoding": [
                {
                    "size": 7,
                    "value": "OP-V"
                },
                {
                    "size": 5,
                    "value": "vd",
                    "constraint": "=&"
                },
                {
                    "size": 3,
                    "value": "OPFVF"
                },
                {
                    "size": 5,
                    "value": "fs1"
                },
```

(续下页)

(接上页)

```
                {
                    "size": 5,
                    "value": "vs2"
                },
                {
                    "size": 1,
                    "value": "vm"
                },
                {
                    "size": 6,
                    "value": "0b000000"
                }
            ],

            "etype": ["f16", "f32", "f64"],

            "uses": ["vtype", "vl", "frm"],
            "defs": ["fflags"],

            "execute": {
                "c": [
                    "for (int i = 0; i < vl; i++) {",
                    "  if (VM || vm(i))",
                    "    vd[i] = ign_float(ESIZE, add, vs2[i], fs1);",
                    "}"
                ]
            }
        }
    ]
}
```

第五章　IGN 内置接口手册

5.1　IGN 符号

5.1.1　编码域符号

支持 RISCV 社区指令集手册的标准命名：

Table 74. RISC-V base opcode map, inst[1:0]=11

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	OP-VE	custom-3/rv128	≥80b

5.1.2　CSR 寄存器

支持 RISCV 社区指令集手册定义的标准寄存器名称，如 **vl** 、**vtype** 等。

5.1.3　指令操作数

用户在指令 **asm** 描述中出现的变量，用户也可以直接引用、操作。

备注： 目前有一个寄存器比较特殊，就是 vector 中的 mask 寄存器，由于 C 语言没有 bit 数据类型，所以在描述 mask 寄存器的时候，我们不能像 C 语言数组一样操作，我们需要把取数组下标的运算符 *[]* 修改成 *()* 即可。

大写的描述表示这条指令是否支持 mask 功能：

Mask Encoding

Where available, masking is encoded in a single-bit `vm` field in the instruction (`inst[25]`).

vm	Description
0	vector result, only where v0.mask[i] = 1
1	unmasked

5.1.4 其它

- ESIZE

  表示当前上下文中指令操作的元素位宽，如 32、64 等。

- VLMAX

  表示 vector 指令操作的最大元素个数。

5.2 IGN 接口

5.2.1 工具接口

Defines

ign_bits(DATA, MAX, MIN)

从 *DATA* 整形数据中提取 [*MAX* : *MIN*] 位域。

ign_bit(DATA, BIT)

从 *DATA* 整形数据中提取第 *BIT* 位域。

ign_abit(ARRAY, BIT)

从 *ARRAY* 整形数组数据中提取第 *BIT* 位域。

ign_zext64(DATA, BITS)

对 *DATA* 整形数据低 *BITS* 位数据进行 0 扩展。

ign_sext64(DATA, BITS)

对 *DATA* 整形数据低 *BITS* 位数据进行有符号扩展，第 *BITS-1* 位就是符号位。

ign_float(esize, op, args...)

该接口是对浮点执行相关接口的封装，对应的浮点接口为：ign_f<esize>_<op>(args…)

ign_bfloat(esize, op, args...)

该接口是对 bfloat 执行相关接口的封装，对应的浮点接口为：ign_bf<esize>_<op>(args…)

5.2.2 寄存器接口

Defines

ign_get_csr(csr)

获取 csr 寄存器

参数

- **csr** –**[out]** 寄存器

返回

true or false, true 表示成功

ign_set_csr(csr)

设置 csr 寄存器

参数

- **csr** –**[in]** 寄存器

返回

true or false, true 表示成功

5.2.3 浮点接口

整型转换为浮点

param data

[in] 源整型操作数

return

转换后的浮点数

float16 **ign_i8_to_f16**(int8_t data)

float16 **ign_i16_to_f16**(int16_t data)

float16 **ign_i32_to_f16**(int32_t data)

float16 **ign_i64_to_f16**(int64_t data)

float16 **ign_u8_to_f16**(uint8_t data)

float16 **ign_u16_to_f16**(uint16_t data)

float16 **ign_u32_to_f16**(uint32_t data)

float16 **ign_u64_to_f16**(uint64_t data)

float32 **ign_i16_to_f32**(int16_t data)

float32 **ign_i32_to_f32**(int32_t data)

float32 **ign_i64_to_f32**(int64_t data)

float32 **ign_u16_to_f32**(uint16_t data)

float32 **ign_u32_to_f32**(uint32_t data)

float32 **ign_u64_to_f32**(uint64_t data)

float64 **ign_i16_to_f64**(int16_t data)

float64 **ign_i32_to_f64**(int32_t data)

float64 **ign_i64_to_f64**(int64_t data)

float64 **ign_u16_to_f64**(uint16_t data)

float64 **ign_u32_to_f64**(uint32_t data)

float64 **ign_u64_to_f64**(uint64_t data)

bfloat16 **ign_i8_to_bf16**(int8_t data)

bfloat16 **ign_i16_to_bf16**(int16_t data)

bfloat16 **ign_i32_to_bf16**(int32_t data)

bfloat16 **ign_i64_to_bf16**(int64_t data)

bfloat16 **ign_u8_to_bf16**(uint8_t data)

bfloat16 **ign_u16_to_bf16**(uint16_t data)

bfloat16 **ign_u32_to_bf16**(uint32_t data)

bfloat16 **ign_u64_to_bf16**(uint64_t data)

浮点转换为整型

param data
[in] 源浮点操作数

return
转换后的整型数

int8_t **ign_f16_to_i8**(*float16* data)

int16_t **ign_f16_to_i16**(*float16* data)

int32_t **ign_f16_to_i32** (*float16* data)

int64_t **ign_f16_to_i64** (*float16* data)

uint8_t **ign_f16_to_u8** (*float16* data)

uint16_t **ign_f16_to_u16** (*float16* data)

uint32_t **ign_f16_to_u32** (*float16* data)

uint64_t **ign_f16_to_u64** (*float16* data)

int8_t **ign_bf16_to_i8** (*bfloat16* data)

int16_t **ign_bf16_to_i16** (*bfloat16* data)

int32_t **ign_bf16_to_i32** (*bfloat16* data)

int64_t **ign_bf16_to_i64** (*bfloat16* data)

uint8_t **ign_bf16_to_u8** (*bfloat16* data)

uint16_t **ign_bf16_to_u16** (*bfloat16* data)

uint32_t **ign_bf16_to_u32** (*bfloat16* data)

uint64_t **ign_bf16_to_u64** (*bfloat16* data)

int16_t **ign_f32_to_i16** (*float32* data)

int32_t **ign_f32_to_i32** (*float32* data)

int64_t **ign_f32_to_i64** (*float32* data)

uint16_t **ign_f32_to_u16** (*float32* data)

uint32_t **ign_f32_to_u32** (*float32* data)

uint64_t **ign_f32_to_u64** (*float32* data)

int16_t **ign_f64_to_i16** (*float64* data)

int32_t **ign_f64_to_i32** (*float64* data)

int64_t **ign_f64_to_i64** (*float64* data)

uint16_t **ign_f64_to_u16** (*float64* data)

uint32_t **ign_f64_to_u32** (*float64* data)

uint64_t **ign_f64_to_u64** (*float64* data)

浮点转换为浮点

param data
[in] 源浮点操作数

return
转换后的浮点数

float16 **ign_f32_to_f16** (*float32* data)

float32 **ign_f16_to_f32** (*float16* data)

float16 **ign_f64_to_f16** (*float64* data)

float64 **ign_f16_to_f64** (*float16* data)

bfloat16 **ign_f32_to_bf16** (*float32* data)

float32 **ign_bf16_to_f32** (*bfloat16* data)

bfloat16 **ign_f64_to_bf16** (*float64* data)

float64 **ign_bf16_to_f64** (*bfloat16* data)

float64 **ign_f32_to_f64** (*float32* data)

float32 **ign_f64_to_f32** (*float64* data)

舍入到整数，将浮点数的值舍入到整数，输出的仍是浮点的格式

param data
[in] 源浮点操作数

return
舍入后的浮点数

float16 **ign_f16_round_to_int** (*float16* data)

bfloat16 **ign_bf16_round_to_int** (*bfloat16* data)

float32 **ign_f32_round_to_int** (*float32* data)

float64 **ign_f64_round_to_int** (*float64* data)

浮点加法，源操作数 1 + 源操作数 2

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_add**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_add**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_add**(*float32* data1, *float32* data2)

float64 **ign_f64_add**(*float64* data1, *float64* data2)

浮点减法，源操作数 1 - 源操作数 2

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_sub**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_sub**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_sub**(*float32* data1, *float32* data2)

float64 **ign_f64_sub**(*float64* data1, *float64* data2)

浮点乘法，源操作数 1 x 源操作数 2

param data1
[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

计算结果

float16 **ign_f16_mul**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_mul**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_mul**(*float32* data1, *float32* data2)

float64 **ign_f64_mul**(*float64* data1, *float64* data2)

浮点乘加，源操作数 1 x 源操作数 2 + 源操作数 3

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

param data3

[in] 源浮点操作数 3

param flags

[in] 特殊设置 FloatMuladdFlags，为 0 则无特殊行为

return

计算结果

float16 **ign_f16_muladd**(*float16* data1, *float16* data2, *float16* data3, int flags)

bfloat16 **ign_bf16_muladd**(*bfloat16* data1, *bfloat16* data2, *bfloat16* data3, int flags)

float32 **ign_f32_muladd**(*float32* data1, *float32* data2, *float32* data3, int flags)

float64 **ign_f64_muladd**(*float64* data1, *float64* data2, *float64* data3, int flags)

浮点除法，源操作数 1 / 源操作数 2

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_div**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_div**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_div**(*float32* data1, *float32* data2)

float64 **ign_f64_div**(*float64* data1, *float64* data2)

浮点取较小值

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_min**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_min**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_min**(*float32* data1, *float32* data2)

float64 **ign_f64_min**(*float64* data1, *float64* data2)

浮点取较大值

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_max**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_max**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_max**(*float32* data1, *float32* data2)

float64 **ign_f64_max**(*float64* data1, *float64* data2)

浮点取较小值，IEEE754-2018 minNum 标准，如果一个操作数是 qNaN，另一个是正常值，则返回正常值

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_minnum**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_minnum**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_minnum**(*float32* data1, *float32* data2)

float64 **ign_f64_minnum**(*float64* data1, *float64* data2)

浮点取较大值，IEEE754-2018 maxNum 标准，如果一个操作数是 qNaN，另一个是正常值，则返回正常值

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_maxnum**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_maxnum**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_maxnum**(*float32* data1, *float32* data2)

float64 **ign_f64_maxnum**(*float64* data1, *float64* data2)

浮点取较小值，在 minnum 类函数功能的基础上，先比较数值，如果相等则比较符号，如果不相等直接返回绝对值较小的数

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_minnummag**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_minnummag**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_minnummag**(*float32* data1, *float32* data2)

float64 **ign_f64_minnummag**(*float64* data1, *float64* data2)

浮点取较大值，在 maxnum 类函数功能的基础上，先比较数值，如果相等则比较符号，如果不相等直接返回绝对值较大的数

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float16 **ign_f16_maxnummag**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_maxnummag**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_maxnummag**(*float32* data1, *float32* data2)

float64 **ign_f64_maxnummag**(*float64* data1, *float64* data2)

浮点取较小值，IEEE754-2019 minimumNumber 标准，在 minnum 类函数功能的基础上，如果两个操作数都是 NaN，返回一个 qNaN，如果任意一个操作数是 sNaN，设置 invalid 标志，但除非两个操作数都是 NaN，否则 SNaN 会被忽略，不会转换为 QNaN

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

计算结果

float16 **ign_f16_minimum_number**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_minimum_number**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_minimum_number**(*float32* data1, *float32* data2)

float64 **ign_f64_minimum_number**(*float64* data1, *float64* data2)

浮点取较大值，IEEE754-2019 maximumNumber 标准，在 maxnum 类函数功能的基础上，如果两个操作数都是 NaN，返回一个 qNaN，如果任意一个操作数是 sNaN，设置 invalid 标志，但除非两个操作数都是 NaN，否则 SNaN 会被忽略，不会转换为 QNaN

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

计算结果

float16 **ign_f16_maximum_number**(*float16* data1, *float16* data2)

bfloat16 **ign_bf16_maximum_number**(*bfloat16* data1, *bfloat16* data2)

float32 **ign_f32_maximum_number**(*float32* data1, *float32* data2)

float64 **ign_f64_maximum_number**(*float64* data1, *float64* data2)

浮点开方计算

param data

[in] 源浮点操作数

return

计算结果

float16 **ign_f16_sqrt** (*float16* data)

bfloat16 **ign_bf16_sqrt** (*bfloat16* data)

float32 **ign_f32_sqrt** (*float32* data)

float64 **ign_f64_sqrt** (*float64* data)

浮点比较

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
比较结果

FloatRelation **ign_f16_compare** (*float16* data1, *float16* data2)

FloatRelation **ign_bf16_compare** (*bfloat16* data1, *bfloat16* data2)

FloatRelation **ign_f32_compare** (*float32* data1, *float32* data2)

FloatRelation **ign_f64_compare** (*float64* data1, *float64* data2)

浮点比较，如果两个源操作数有 NaN 但不是 sNaN 时，不生成 invalid 标志

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
比较结果

FloatRelation **ign_f16_compare_quiet** (*float16* data1, *float16* data2)

FloatRelation **ign_bf16_compare_quiet** (*bfloat16* data1, *bfloat16* data2)

FloatRelation **ign_f32_compare_quiet** (*float32* data1, *float32* data2)

FloatRelation **ign_f64_compare_quiet** (*float64* data1, *float64* data2)

浮点是否是 qNaN

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_quiet_nan**(*float16* data)

bool **ign_bf16_is_quiet_nan**(*bfloat16* data)

bool **ign_f32_is_quiet_nan**(*float32* data)

bool **ign_f64_is_quiet_nan**(*float64* data)

浮点是否是 sNaN

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_signaling_nan**(*float16* data)

bool **ign_bf16_is_signaling_nan**(*bfloat16* data)

bool **ign_f32_is_signaling_nan**(*float32* data)

bool **ign_f64_is_signaling_nan**(*float64* data)

将 sNAN 源操作数转换为一个 qNaN，保留尾数低位

param data
[in] 源浮点操作数

return
计算结果

float16 **ign_f16_silence_nan**(*float16* data)

bfloat16 **ign_bf16_silence_nan**(*bfloat16* data)

float32 **ign_f32_silence_nan** (*float32* data)

float64 **ign_f64_silence_nan** (*float64* data)

判断源操作数是否为任意 NaN

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_any_nan** (*float16* data)

bool **ign_bf16_is_any_nan** (*bfloat16* data)

bool **ign_f32_is_any_nan** (*float32* data)

bool **ign_f64_is_any_nan** (*float64* data)

判断源操作数是否为负数

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_neg** (*float16* data)

bool **ign_bf16_is_neg** (*bfloat16* data)

bool **ign_f32_is_neg** (*float32* data)

bool **ign_f64_is_neg** (*float64* data)

判断源操作数是否为无穷

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_infinity**(*float16* data)

bool **ign_bf16_is_infinity**(*bfloat16* data)

bool **ign_f32_is_infinity**(*float32* data)

bool **ign_f64_is_infinity**(*float64* data)

判断源操作数是否为零

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_zero**(*float16* data)

bool **ign_bf16_is_zero**(*bfloat16* data)

bool **ign_f32_is_zero**(*float32* data)

bool **ign_f64_is_zero**(*float64* data)

判断源操作数是否为零或非规格数

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_zero_or_denormal**(*float16* data)

bool **ign_bf16_is_zero_or_denormal**(*bfloat16* data)

bool **ign_f32_is_zero_or_denormal**(*float32* data)

bool **ign_f64_is_zero_or_denormal**(*float64* data)

判断源操作数是否为规格数

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f16_is_normal** (*float16* data)

bool **ign_bf16_is_normal** (*bfloat16* data)

bool **ign_f32_is_normal** (*float32* data)

bool **ign_f64_is_normal** (*float64* data)

浮点求绝对值

param data
[in] 源浮点操作数

return
计算结果

float16 **ign_f16_abs** (*float16* data)

bfloat16 **ign_bf16_abs** (*bfloat16* data)

float32 **ign_f32_abs** (*float32* data)

float64 **ign_f64_abs** (*float64* data)

浮点符号位取反

param data
[in] 源浮点操作数

return
计算结果

float16 **ign_f16_chs** (*float16* data)

bfloat16 **ign_bf16_chs** (*bfloat16* data)

float32 **ign_f32_chs** (*float32* data)

float64 **ign_f64_chs** (*float64* data)

浮点设置符号位

param data
[in] 源浮点操作数

param sign
[in] 符号值，最低 1bit 有效

return
计算结果

float16 **ign_f16_set_sign**(*float16* data, int sign)

bfloat16 **ign_bf16_set_sign**(*bfloat16* data, int sign)

float32 **ign_f32_set_sign**(*float32* data, int sign)

float64 **ign_f64_set_sign**(*float64* data, int sign)

比较浮点是否相等

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
true 或者 false

bool **ign_f16_eq**(*float16* data1, *float16* data2)

bool **ign_bf16_eq**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_eq**(*float32* data1, *float32* data2)

bool **ign_f64_eq**(*float64* data1, *float64* data2)

比较源操作数 1 是否小于或等于源操作数 2

param data1
[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

true 或者 false

bool **ign_f16_le**(*float16* data1, *float16* data2)

bool **ign_bf16_le**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_le**(*float32* data1, *float32* data2)

bool **ign_f64_le**(*float64* data1, *float64* data2)

比较源操作数 1 是否小于源操作数 2

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

true 或者 false

bool **ign_f16_lt**(*float16* data1, *float16* data2)

bool **ign_bf16_lt**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_lt**(*float32* data1, *float32* data2)

bool **ign_f64_lt**(*float64* data1, *float64* data2)

源操作数 1 与源操作数 2 对比较结果是否是 unordered

param data1

[in] 源浮点操作数 1

param data2

[in] 源浮点操作数 2

return

true 或者 false

bool **ign_f16_unordered**(*float16* data1, *float16* data2)

bool **ign_bf16_unordered**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_unordered**(*float32* data1, *float32* data2)

bool **ign_f64_unordered**(*float64* data1, *float64* data2)

比较浮点是否相等，如果两个源操作数有 NaN 但不是 sNaN 时，不生成 invalid 标志

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
true 或者 false

bool **ign_f16_eq_quiet**(*float16* data1, *float16* data2)

bool **ign_bf16_eq_quiet**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_eq_quiet**(*float32* data1, *float32* data2)

bool **ign_f64_eq_quiet**(*float64* data1, *float64* data2)

比较源操作数 1 是否小于或等于源操作数 2，如果两个源操作数有 NaN 但不是 sNaN 时，不生成 invalid 标志

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
true 或者 false

bool **ign_f16_le_quiet**(*float16* data1, *float16* data2)

bool **ign_bf16_le_quiet**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_le_quiet**(*float32* data1, *float32* data2)

bool **ign_f64_le_quiet**(*float64* data1, *float64* data2)

比较源操作数 1 是否小于源操作数 2，如果两个源操作数有 NaN 但不是 sNaN 时，不生成 invalid 标志

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
true 或者 false

bool **ign_f16_lt_quiet**(*float16* data1, *float16* data2)

bool **ign_bf16_lt_quiet**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_lt_quiet**(*float32* data1, *float32* data2)

bool **ign_f64_lt_quiet**(*float64* data1, *float64* data2)

源操作数 1 与源操作数 2 对比较结果是否是 unordered，如果两个源操作数有 NaN 但不是 sNaN 时，不生成 invalid 标志

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
true 或者 false

bool **ign_f16_unordered_quiet**(*float16* data1, *float16* data2)

bool **ign_bf16_unordered_quiet**(*bfloat16* data1, *bfloat16* data2)

bool **ign_f32_unordered_quiet**(*float32* data1, *float32* data2)

bool **ign_f64_unordered_quiet**(*float64* data1, *float64* data2)

生成架构默认的 NaN

return
默认 NaN

float16 **ign_f16_default_nan**()

bfloat16 **ign_bf16_default_nan**()

float32 **ign_f32_default_nan**()

float64 **ign_f64_default_nan**()

浮点求余数，源操作数 1 / 源操作数 2 的余数

param data1
[in] 源浮点操作数 1

param data2
[in] 源浮点操作数 2

return
计算结果

float32 **ign_f32_rem**(*float32* data1, *float32* data2)

float64 **ign_f64_rem**(*float64* data1, *float64* data2)

浮点求 e 的次方，e^ 源操作数 1

param data
[in] 源浮点操作数

return
计算结果

float32 **ign_f32_exp2**(*float32* data)

浮点求以 2 为底的对数

param data
[in] 源浮点操作数

return
计算结果

float32 **ign_f32_log2**(*float32* data)

float64 **ign_f64_log2** (*float64* data)

判断源操作数是否为非规格数

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f32_is_denormal** (*float32* data)

bool **ign_f64_is_denormal** (*float64* data)

判断源操作数是否为零或规格数

param data
[in] 源浮点操作数

return
true 或者 false

bool **ign_f32_is_zero_or_normal** (*float32* data)

bool **ign_f64_is_zero_or_normal** (*float64* data)

用符号部分、指数部分、尾数部分打包 32 位浮点数

param sign
[in] 符号位

param exp
[in] 指数

param sig
[in] 尾数

return
32 位浮点数

float32 **ign_f32_pack** (bool sign, int exp, uint32_t sig)

源操作数 1 的指数部分加上源操作数 2，源操作数 1 x 2^ 源操作数 2

param data

[in] 源浮点操作数

param scale

[in] 源操作数 2，放缩数

return

计算结果

float16 **ign_f16_scalbn**(*float16* data, int scale)

float16 **ign_bf16_scalbn**(*bfloat16* data, int scale)

float32 **ign_f32_scalbn**(*float32* data, int scale)

float64 **ign_f64_scalbn**(*float64* data, int scale)

浮点类型

浮点数据类型

内置软浮点库使用整数来模拟浮点，因此不使用高级语言内置的 float、double 等类型，使用相同宽度的无符号整数进行运算作为浮点使用的整数二进制位格式如下，与标准浮点相同：

typedef uint16_t **float16**

1 位符号位，5 位指数位，10 位尾数位

typedef uint32_t **float32**

1 位符号位，8 位指数位，23 位尾数位

typedef uint64_t **float64**

1 位符号位，11 位指数位，52 位尾数位

typedef uint16_t **bfloat16**

1 位符号位，8 位指数位，7 位尾数位

Enums

enum **FloatRelation**

Values:

enumerator **float_relation_less**

小于

enumerator **float_relation_equal**

等于

enumerator **float_relation_greater**

大于

enumerator **float_relation_unordered**

无法比较

enum **FloatMuladdFlags**

Values:

enumerator **float_muladd_negate_c**

对操作数 3 的符号位取反

enumerator **float_muladd_negate_product**

对乘法的中间结果的符号位取反

enumerator **float_muladd_negate_result**

对乘加最后结果的符号位取反

enumerator **float_muladd_halve_result**

对乘加最后结果的指数位减一

第六章　IGN 工具帮助手册

IGN 工具内置了文档系统，通过 **ctb** 命令 `ctb --usage=igen` 来显示 IGN 工具的帮助文档，如：

```
====> module ['igen']:

--> option:

    cflags:     append any cflags for this component
    iconfig:    set config file for instructions

--> doc:

    build igen
    ==========
    ctb -m igen --update=igen -c iconfig=</path/to/description file> \
        --tcprefix=</path/to/toolchain>/bin
```

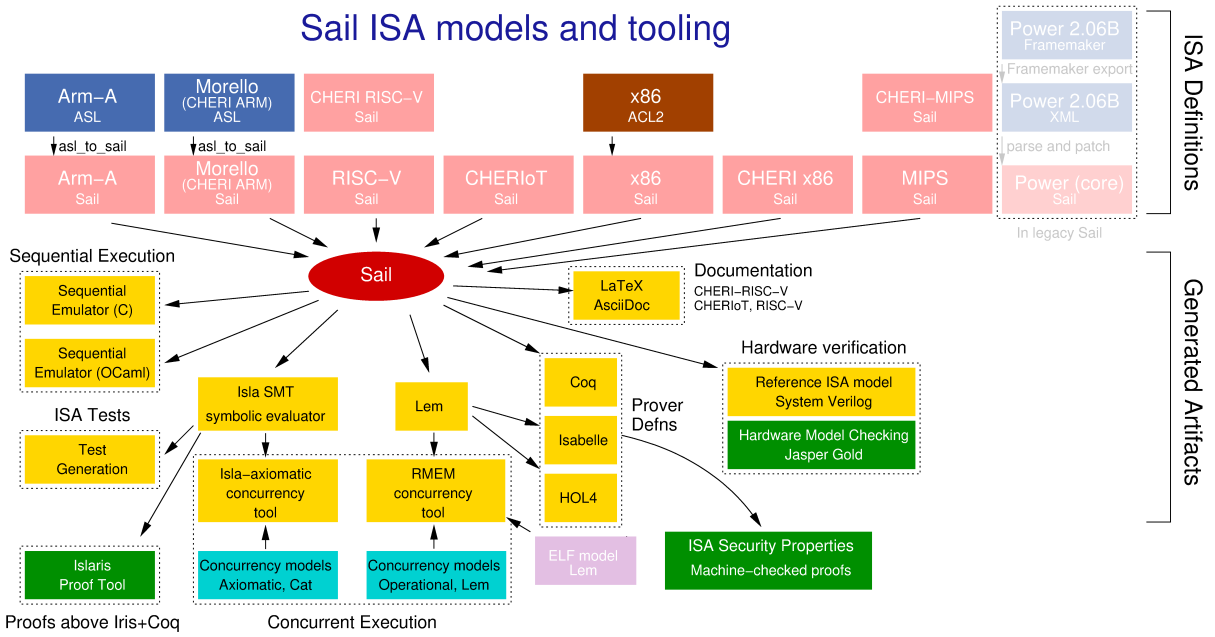
最新的工具文档请使用发布的 **ctb** 工具来显示。

第七章　高级主题

7.1　XCE-SAIL

7.1.1　简介

SAIL 是一门专门用来表达处理器指令集架构 ISA 语义的语言。以前，处理器厂商一般用自然语言或伪代码来描述指令的语义，往往失之精确性和一致性。而有了 SAIL，我们可以精确描述指令语义，并且可以利用工具处理 SAIL 指令描述，自动生成一致的编译工具链和相关文档。



当我们要自定义指令时，我们可以把指令的编码、汇编、执行语义等信息用 sail 语言描述到 sail 文件中。**IGN** 工具可以根据 sail 描述文件自动生成支持自定义指令的工具链：

```
ctb -m igen --update=igen -c iconfig=</path/to/sail description file>
```

如果是在 Windows 系统下使用，需要配置相关环境，详见*Windows 系统环境配置*

7.1.2 快速开始

这里我们也同样用 **xt.add** 这条指令举例，使用 sail 语言来描述：

```
$include <xce/xce.sail>

union clause ast = XT_ADD : (bits(5), bits(5), bits(5))  // 声明指令的语法树

mapping clause encdec =                                  // 定义指令的编码
    XT_ADD(rd, rs1, rs2) <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011

mapping clause assembly =                                // 定义指令的汇编
    XT_ADD(rd, rs1, rs2) <-> "xt.add" ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(rs1) ^
↪ sep() ^ reg_name(rs2)

function clause execute XT_ADD(rd, rs1, rs2) = {         // 定义指令的执行语义
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let result = rs1_val + rs2_val;
    X(rd) = result;
    RETIRE_SUCCESS
}
```

7.1.3 描述文件语法定义

XCE-SAIL 是基于 SAIL 的一套运行时环境，包括*SAIL 语言* 支持、运行时库等。

使用 XCE-SAIL 环境描述自定义指令集，可以基于其内置的大量 API 快速描述指令，从而避免陷入 sail 语言的基础细节中，本质上来说，基于 XCE-SAIL 环境描述自定义指令集就是基于 XCE-SAIL 的 API 来描述自定义指令集。

SAIL 描述文件可以包含任意条指令，每条指令的描述应包含四要素：

- union clause ast
- mapping clause encdec
- mapping clause assembly
- function clause execute

指令语法树 ast

```
(** 定义指令语法树类型
 *  指令参数(编码域)类型根据不同的指令设置
```

(续下页)

(接上页)

```
 *)
union clause ast = <id> : (<type0>, <type1>, ..., <typeN>)
```

指令编码 encdec

```
(** 定义指令编码
 *  指令参数(编码域)根据不同的指令设置，每个编码域之间用“@”符号连接
 *)
mapping clause encdec = <id> : (<param0>, <param1>, ..., <paramN>) <-> <param0> @
↪<param1> @ ... @ <paramN>
```

指令汇编语法 assembly

```
(** 定义指令汇编语法
 *  指令参数(编码域)根据不同的指令设置，每个编码域之间用“^”符号连接
 *)
mapping clause assembly = <id> : (<param0>, <param1>, ..., <paramN>) <-> <param0> ^
↪<param1> ^ ... ^ <paramN>
```

指令行为 execute

```
function clause execute <id> (<param0>, <param1>, ..., <paramN>) = {
    (** 指令行为描述 *)
}
```

描述文件基础 API 定义

数据类型 指令描述使用到的数据类型大体有两类，一类是 sail 语言支持的数据类型，还有一类是 XCE 定义的数据类型，一般我们使用 XCE 定义的数据类型，使指令描述的表达更易懂。

表 1: XCE 数据类型

类型描述	SAIL 语言数据类型描述
xlen	Int = 64 / Int = 32
xlen_bytes	Int = 8 / Int = 4
xlenbits	bits(xlen)
regidx	bits(5)

API 接口

```
(** @功能 生成空格，用于描述汇编字符串
 *  @return string
 *)
val spc : unit <-> string

(** @功能 生成寄存器名字，用于描述汇编字符串
 *  @param[0] regidx 寄存器编号
 *  @return string 寄存器的名称
 *)
val reg_name : regidx <-> string

(** @功能 整形寄存器读/写
 *  用法示例：
 *  let rs_val = X(rs)  //读取rs寄存器的值到rs_val变量中
 *  X(rd) = rd_val      //将rd_val写入到rd寄存器中
 *)
overload X = {rX, wX}

(** @功能 bits有符号扩展
 *  @param[0] int 指定长度
 *  @param[1] bits(n) 待扩展数据
 *  @return bits(m) 有符号扩展后的比特数
 *)
val sign_extend : forall 'n 'm, 'm >= 'n. (implicit('m), bits('n)) -> bits('m)

(** @功能 bits零扩展
 *  @param[0] int 指定长度
 *  @param[1] bits(n) 待扩展数据
 *  @return bits(m) 零扩展后的比特数
 *)
val zero_extend : forall 'n 'm, 'm >= 'n. (implicit('m), bits('n)) -> bits('m)

(** @功能 半/单/双精度浮点寄存器读/写。
 *  用法示例：
 *  let rs_val = F_or_X_H(rs) //读取rs寄存器的值到rs_val变量中
 *  F_or_X_H(rd) = rd_val     //将rd_val写入到rd寄存器中
 *)
overload F_or_X_H = {rF_or_X_H, wF_or_X_H}
overload F_or_X_S = {rF_or_X_S, wF_or_X_S}
overload F_or_X_D = {rF_or_X_D, wF_or_X_D}

(** @功能 读取矢量寄存器
 *  @param[0] int('n) 待读取的元素数量
```

(续下页)

(接上页)

```
 *  @param[1] int('m) sew
 *  @param[2] int('p) lmul之指数
 *  @param[3] regidx 矢量寄存器的起始编号
 *  @return vector('n, bits('m)) 所读取的矢量寄存器的值，其中'n为元素数量，
↪'m为sew对应的位数
 *)
val read_vreg : forall 'n 'm 'p, 'n >= 0 & 'm >= 0. (int('n), int('m), int('p),␣
↪regidx) -> vector('n, bits('m))

(** @功能 写入矢量寄存器
 *  @param[0] int('n) 待读取的元素数量
 *  @param[1] int('m) sew
 *  @param[2] int('p) lmul之指数
 *  @param[3] regidx 矢量寄存器的起始编号
 *  @param[4] vector('n, bits('m)) 待写入的vector值，其中'n为元素数量，
↪'m为sew对应的位数
 *  @return unit
 *)
val write_vreg : forall 'n 'm 'p, 'n >= 0 & 'm >= 0. (int('n), int('m), int('p),␣
↪regidx, vector('n, bits('m))) -> unit

(** @功能 判断两个vector寄存器是否overlap
 *  @param[0] regidx 源寄存器编号
 *  @param[1] regidx 目的寄存器编号
 *  @param[2] int 源EMUL之指数
 *  @param[3] int 目的EMUL之指数
 *  @return bool
 *)
val valid_reg_overlap : (regidx, regidx, int, int) -> bool
```

更多的接口使用文档，请移步内置接口手册 查看。

7.1.4 描述文件案例

标量

addi

```
$include <xce/xce.sail>

union clause ast = ADDI : (bits(5), bits(5), bits(12))

mapping clause encdec =
```

(续下页)

(接上页)

```
  ADDI(rd, rs1, imm1) <-> imm1 @ rs1 @ 0b000 @ rd @ 0b0010011

mapping clause assembly =
  ADDI(rd, rs1, imm2) <-> "addi" ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(rs1) ^␣
↪sep() ^ hex_bits_signed_12(imm2)

function clause execute ADDI(rd, rs1, imm) = {
  let rs1_val = X(rs1);
  let immext : xlenbits = sign_extend(imm);
  let result = rs1_val + immext;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

add & sub

```
enum xop = {XT_ADD, XT_SUB}

$include <xce/xce.sail>

union clause ast = XOP : (xop, bits(5), bits(5), bits(5))

mapping encdec_xop : xop <-> bits(7) = {
  XT_ADD <-> 0b0000000,
  XT_SUB <-> 0b0000001
}

mapping clause encdec = XOP(op, rs2, rs1, rd)
  <-> encdec_xop(op) @ rs2 @ rs1 @ 0b000 @ rd @0b0110011

mapping mnemonic_xop : xop <-> string = {
  XT_ADD <-> "xt.add",
  XT_SUB <-> "xt.sub"
}

mapping clause assembly = XOP(op, rs2, rs1, rd)
  <-> mnemonic_xop(op) ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(rs1) ^ sep() ^ reg_
↪name(rs2)

function clause execute (XOP(op, rs2, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
```

(续下页)

(接上页)

```
    XT_ADD => rs1_val + rs2_val,
    XT_SUB => rs1_val - rs2_val
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

fadd.s

```
$include <xce/xce.sail>

union clause ast = FADD_S : (bits(5), bits(5), rounding_mode, bits(5))

mapping clause encdec = FADD_S(rs2, rs1, rm, rd)
  <-> 0b0000000 @ rs2 @ rs1 @ encdec_rounding_mode (rm) @ rd @ 0b1010011

mapping clause assembly = FADD_S(rs2, rs1, rm, rd)
  <-> "fadd.s " ^ freg_or_reg_name(rd) ^ "," ^ freg_or_reg_name(rs1) ^ "," ^ freg_or_
↪reg_name(rs2) ^ "," ^ frm_mnemonic(rm)

function clause execute FADD_S(rs2, rs1, rm, rd) = {
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        riscv_f32Add (rm_3b, rs1_val_32b, rs2_val_32b);
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

向量

vfadd.vf

```
$include <xce/xce.sail>

union clause ast = VFADD_VF : (bits(1), regidx, regidx, regidx)
```

(续下页)

(接上页)

```
mapping clause encdec = VFADD_VF(vm, vs2, rs1, vd)
  <-> 0b000000 @ vm @ vs2 @ rs1 @ 0b101 @ vd @ 0b1010111

mapping clause assembly = VFADD_VF(vm, vs2, rs1, vd)
  <-> "vfadd.vf" ^ spc() ^ vreg_name(vd) ^ sep() ^ vreg_name(vs2) ^ sep() ^ reg_
↪name(rs1) ^ maybe_vmask(vm)

function clause execute(VFADD_VF(vm, vs2, rs1, vd)) = {
  let rm_3b    = fcsr[FRM];
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_
↪FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b00000);
  let rs1_val : bits('m)                   = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  var result  : vector('n, dec, bits('m)) = undefined;
  var mask    : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = fp_add(rm_3b, vs2_val[i], rs1_val)
    }
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

vwadd.vv

```
$include <xce/xce.sail>

union clause ast = VWADD_VV : (bits(1), bits(5), bits(5), bits(5))

mapping clause encdec = VWADD_VV(vm, vs2, vs1, vd)
  <-> 0b110001 @ vm @ vs2 @ vs1 @ 0b010 @ vd @ 0b1010111

mapping clause assembly = VWADD_VV(vm, vs2, vs1, vd)
  <-> "vwadd.vv " ^ vreg_name(vd) ^ "," ^ vreg_name(vs1) ^ "," ^ vreg_name(vs2) ^␣
↪maybe_vmask(vm)

function clause execute VWADD_VV(vm, vs2, vs1, vd) = {
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if  illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
      not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b00000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_
↪widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_
↪widen, vs2);
  var result  : vector('n, dec, bits('o)) = undefined;
  var mask    : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_
↪val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i]))
    }
  };
```

(续下页)

(接上页)

```
  write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

7.1.5 内置接口手册

符号

常见运算符

表 2: 运算符

运算符	功能
+	加
-	减
*	乘
/	除
%	取模
&	按位与
|	按位或
^	按位异或
>>	逻辑右移
<<	逻辑左移
>>>	循环右移
<<<	循环左移
<_s	有符号小于
>_s	有符号大于
<=_s	有符号小于等于
>=_s	有符号大于等于
<_u	无符号小于
>_u	无符号大于
<=_u	无符号小于等于
>=_u	无符号大于等于

sail-riscv 函数库

SAIL 语言为我们描述指令语义提供了基本手段，比如读写寄存器，这样我们就可以用 SAIL 语言去描述一个完整的指令集 ISA 了。但是，如果我们要基于 RISC-V CPU 核添加自定义指令，从头去描述整个 RISC-V 指令集模型就显得太笨重了，这时我们可以直接基于 sail-riscv 函数库 做指令描述。这就类似于，在 C 语言编程

中，我们不用自己去实现常见的求正弦余弦的数学函数，而是直接去调用 C 语言数学函数库中的 sin/cos 函数。sail-riscv 函数库提供了大量的 API，方便我们基于 RISC-V CPU 核添加自定义指令。基于这些 API 描述指令语义，可以大大提高效率。

整形 API

```
/** @功能 读取整型寄存器的值
 *  @param[0] regno 寄存器编号
 *  @return bits(xlen) 所读取寄存器的值
 */
val rX : regno -> xlenbits

/** @功能 将值写入整型寄存器
 *  @param[0] regno 寄存器编号
 *  @param[1] xlenbits 待写入的值
 *  @return unit
 */
val wX : (regno, xlenbits) -> unit

/** @功能 是rX和wX的overload
 *  用法示例:
 *  let rs_val = X(rs)  //读取rs寄存器的值到rs_val变量中
 *  X(rd) = rd_val      //将rd_val写入到rd寄存器中
 */
overload X = {rX, wX}

/** @功能 bits有符号扩展
 *  @param[0] int 指定长度
 *  @param[1] bits(n) 待扩展数据
 *  @return bits(m) 有符号扩展后的比特数
 */
val sign_extend : forall 'n 'm, 'm >= 'n. (implicit('m), bits('n)) -> bits('m)

/** @功能 bits零扩展
 *  @param[0] int 指定长度
 *  @param[1] bits(n) 待扩展数据
 *  @return bits(m) 零扩展后的比特数
 */
val zero_extend : forall 'n 'm, 'm >= 'n. (implicit('m), bits('n)) -> bits('m)

/** @功能 取两个整数的最大值
 *  @param[0] int 操作数1
 *  @param[1] int 操作数2
 *  @return int 返回较大者
 */
```

(续下页)

(接上页)

```
val max = forall 'x 'y. (int('x), int('y)) -> {'z, ('x >= 'y & 'z == 'x) ('x < 'y &
↪'z == 'y). int('z)}
(** @功能 取两个整数的最小值
 *  @param[0] int 操作数1
 *  @param[1] int 操作数2
 *  @return int 返回较小者
 *)
val min = forall 'x 'y. (int('x), int('y)) -> {'z, ('x >= 'y & 'z == 'x) ('x < 'y &
↪'z == 'y). int('z)}

(** @功能 生成寄存器名字，用于描述汇编字符串
 *  @param[0] regidx 寄存器编号
 *  @return string 寄存器的名称
 *)
val reg_name : regidx <-> string
```

浮点 API

```
(** @功能 从浮点寄存器（若开启了浮点扩展）或整型寄存器读取半精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @return bits(16) 半精度浮点数
 *)
val rF_or_X_H : regidx -> bits(16)

(** @功能 向浮点寄存器（若开启了浮点扩展）或整型寄存器写入半精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @param[1] bits(16) 待写入的值
 *  @return unit
 *)
val wF_or_X_H : (regidx, bits(16)) -> unit

(** @功能 是rF_or_X_H和wF_or_X_H的overload。
 *  用法示例：
 *  let rs_val = F_or_X_H(rs) //读取rs寄存器的值到rs_val变量中
 *  F_or_X_H(rd) = rd_val     //将rd_val写入到rd寄存器中
 *)
overload F_or_X_H = {rF_or_X_H, wF_or_X_H}

(** @功能 从浮点寄存器（若开启了浮点扩展）或整型寄存器读取单精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @return bits(32) 半精度浮点数
 *)
val rF_or_X_S : regidx -> bits(32)
```

(续下页)

(接上页)

```
(** @功能 向浮点寄存器（若开启了浮点扩展）或整型寄存器写入单精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @param[1] bits(32) 待写入的值
 *  @return unit
 *)
val wF_or_X_S : (regidx, bits(32)) -> unit

(** @功能 是rF_or_X_S和wF_or_X_S的overload。
 *  用法示例：
 *  let rs_val = F_or_X_S(rs) //读取rs寄存器的值到rs_val变量中
 *  F_or_X_S(rd) = rd_val //将rd_val写入到rd寄存器中
 *)
overload F_or_X_S = {rF_or_X_S, wF_or_X_S}

(** @功能 从浮点寄存器（若开启了浮点扩展）或整型寄存器读取双精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @return bits(64) 双精度浮点数
 *)
val rF_or_X_D : regidx -> bits(64)

(** @功能 向浮点寄存器（若开启了浮点扩展）或整型寄存器写入双精度浮点数
 *  @param[0] regidx 寄存器编号
 *  @param[1] bits(64) 待写入的值
 *  @return unit
 *)
val wF_or_X_D : (regidx, bits(64)) -> unit

(** @功能 是rF_or_X_D和wF_or_X_D的overload。
 *  用法示例：
 *  let rs_val = F_or_X_D(rs) //读取rs寄存器的值到rs_val变量中
 *  F_or_X_D(rd) = rd_val //将rd_val写入到rd寄存器中
 *)
overload F_or_X_D = {rF_or_X_D, wF_or_X_D}

(** @功能 生成规范化的NaN
 *  @return 规范化的，16或32或64或128位的NaN
 *)
val canonical_NaN : forall 'n, 'n in {16, 32, 64, 128} . (implicit('n)) -> bits('n)

(** @功能 写fcsr寄存器
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits(5) 浮点异常标志位(NV|DZ|OF|UF|NX)
```

(续下页)

(接上页)

```
 *  @return unit
 *)
val write_fcsr : (bits(3), bits(5)) -> unit

(** @功能 将新的fflags与fcsr中原有的fflags执行按位或运算，并把结果更新到fcsr中
 *  @param[0] bits(5) 浮点异常标志位(NV|DZ|OF|UF|NX)
 *  @return unit
 *)
val accrue_fflags : (bits(5)) -> unit

(** @功能 判断是否为-inf
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_neg_inf : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为负的规格化浮点数 （注： 规格化浮点数是指尾数部分满足`1. Mantissa`␣
↪的表示形式，其中小数点左侧的1是隐含的（隐含位），不需要显式存储；␣
↪非规格化浮点数用于表示非常接近 0 的小数，当指数全为 0␣
↪时，浮点数进入非规格化表示形式 ）
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_neg_norm : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为负的非规格化浮点数
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_neg_subnorm : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为负的零
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_neg_zero : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为正的零
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_pos_zero : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool
```

(续下页)

(接上页)

```
(** @功能 判断是否为正的非规格化浮点数
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_pos_subnorm : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为正的规格化浮点数
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_pos_norm : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为Singaling NaN
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_SNaN : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为Quiet NaN
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_QNaN : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 判断是否为NaN
 *  @param[0] bits('m) 浮点数
 *  @return bool
 *)
val f_is_NaN : forall 'm, 'm in {16, 32, 64}. bits('m) -> bool

(** @功能 求相反数
 *  @param[0] bits('m) 浮点数
 *  @return bits('m) 该浮点数的相反数
 *)
val negate_fp : forall 'm, 'm in {16, 32, 64}. bits('m) -> bits('m)

(** @功能 浮点加法
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 操作数1
 *  @param[2] bits('m) 操作数2
 *  @return bits('m) 操作数的和
 *)
val fp_add: forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m)) -> bits('m)
```

(续下页)

(接上页)

```
(** @功能 浮点减法
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 操作数1
 *  @param[2] bits('m) 操作数2
 *  @return bits('m) 操作数的差
 *)
val fp_sub: forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m)) -> bits('m)

(** @功能 浮点取小者
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bits('m) 两个操作数中的较小者
 *)
val fp_min : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bits('m)

(** @功能 浮点取大者
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bits('m) 两个操作数中的较大者
 *)
val fp_max : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bits('m)

(** @功能 判断两浮点数是否相等
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bool
 *)
val fp_eq : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bool

(** @功能 比较浮点数是否大于
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bool
 *)
val fp_gt : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bool

(** @功能 比较浮点数是否大于等于
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bool
 *)
val fp_ge : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bool
```

(续下页)

(接上页)

```
(** @功能 比较浮点数是否小于
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bool
 *)
val fp_lt : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bool

(** @功能 比较浮点数是否小于等于
 *  @param[0] bits('m) 操作数1
 *  @param[1] bits('m) 操作数2
 *  @return bool
 *)
val fp_le : forall 'm, 'm in {16, 32, 64}. (bits('m), bits('m)) -> bool

(** @功能 浮点乘法
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 操作数1
 *  @param[2] bits('m) 操作数2
 *  @return bits('m) 操作数的积
 *)
val fp_mul : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m)) -> bits('m)

(** @功能 浮点除法
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 操作数1
 *  @param[2] bits('m) 操作数2
 *  @return bits('m) 操作数的商
 *)
val fp_div : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m)) -> bits('m)

(** @功能 浮点乘加 (a * b + c)
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 乘法操作数1
 *  @param[2] bits('m) 乘法操作数2
 *  @param[3] bits('m) 加法操作数
 *  @return bits('m) 计算结果
 *)
val fp_muladd : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m), bits(
↪'m)) -> bits('m)

(** @功能 浮点乘加的变体 (-a * b + c)
 *  @param[0] bits(3) 舍入模式
```

(续下页)

(接上页)

```
 *  @param[1] bits('m) 乘法操作数1
 *  @param[2] bits('m) 乘法操作数2
 *  @param[3] bits('m) 加法操作数
 *  @return bits('m) 计算结果
 *)
val fp_nmuladd : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m), bits(
↪'m)) -> bits('m)

(** @功能 浮点乘减 (a * b - c)
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 乘法操作数1
 *  @param[2] bits('m) 乘法操作数2
 *  @param[3] bits('m) 减法操作数
 *  @return bits('m) 计算结果
 *)
val fp_mulsub : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m), bits(
↪'m)) -> bits('m)

(** @功能 浮点乘减的变体 (-a * b - c)
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits('m) 乘法操作数1
 *  @param[2] bits('m) 乘法操作数2
 *  @param[3] bits('m) 减法操作数
 *  @return bits('m) 计算结果
 *)
val fp_nmuladd : forall 'm, 'm in {16, 32, 64}. (bits(3), bits('m), bits('m), bits(
↪'m)) -> bits('m)

(** @功能 浮点扩位操作（半精度->单精度，单精度->双精度）
 *  @param[0] bits('m) 源操作数
 *  @return bits('m * 2) 扩位后的结果，宽度为源操作数的2倍
 *)
val fp_widen : forall 'm, 'm in {16, 32}. bits('m) -> bits('m * 2)

(** @功能 求浮点数的倒平方根 （结合查表法和规格化处理 ）
 *  @param[0] bits('m) 源操作数
 *  @param[1] bool 是否需要对尾数进行规格化
 *  @return bits_D 倒平方根
 *)
val rsqrt7 : forall 'm, 'm in {16, 32, 64}. (bits('m), bool) -> bits_D

(** @功能 求浮点数的倒数 （结合查表法和规格化处理 ）
 *  @param[0] bits('m) 源操作数
```

(续下页)

(接上页)

```
 *  @param[1] bits(3) 摄入模式
 *  @param[2] bool 是否需要对尾数进行规格化
 *  @return (bool, bits_D)␣
↪前一部分为布尔值，表示是否出现特殊处理（如溢出、舍入）；后一部分为计算倒数的结果
 *)
val recip7 : forall 'm, 'm in {16, 32, 64}. (bits('m), bits(3), bool) -> (bool, bits_
↪D)

(** @功能 舍入到整数
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits(16) 浮点操作数
 *  @param[2] bool 指示操作是否需要处理精度问题，当设置为`true`␣
↪时，它表示需要额外检查非零丢弃位来确定操作是否存在未捕获的舍入误差
 *  @return (bits_fflags, bits(16))␣
↪前一部分为异常标志位fflags，类型为bits(5)，后一部分是舍入结果
 *)
val riscv_f16roundToInt : (bits_rm, bits(16), bool) -> (bits_fflags, bits(16))

(** @功能 舍入到整数
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits(32) 浮点操作数
 *  @param[2] bool 指示操作是否需要处理精度问题，当设置为`true`␣
↪时，它表示需要额外检查非零丢弃位来确定操作是否存在未捕获的舍入误差
 *  @return (bits_fflags, bits(32))␣
↪前一部分为异常标志位fflags，类型为bits(5)，后一部分是舍入结果
 *)
val riscv_f32roundToInt : (bits_rm, bits(32), bool) -> (bits_fflags, bits(32))

(** @功能 舍入到整数
 *  @param[0] bits(3) 舍入模式
 *  @param[1] bits(64) 浮点操作数
 *  @param[2] bool 指示操作是否需要处理精度问题，当设置为`true`␣
↪时，它表示需要额外检查非零丢弃位来确定操作是否存在未捕获的舍入误差
 *  @return (bits_fflags, bits(64))␣
↪前一部分为异常标志位fflags，类型为bits(5)，后一部分是舍入结果
 *)
val riscv_f64roundToInt : (bits_rm, bits(64), bool) -> (bits_fflags, bits(64))

(** @功能 浮点类型转换函数。
 *  命名具有如下统一形式：riscv_{SrcTy}To{DstTy}
 *  其中SrcTy是源操作数的类型，如f16表示半精度浮点类型，DstTy是目的操作数的类型，
 *  如i32表示32位整型。这些转换函数将具有SrcTy类型的源操作数转换为相等或接近
 *  （根据舍入模式）的具有DstTy类型的数值。
```

(续下页)

(接上页)

```
 *  @param[0] bits_rm 舍入模式
 *  @param[1] {SrcTy} SrcTy类型的源操作数
 *  @return (bits_fflags, {DstTy}) 前一部分为异常标志位fflags,
 *  后一部分是DstTy类型的转换结果
 *)
val riscv_f16ToI32 : (bits_rm, bits_H) -> (bits_fflags, bits_W)
val riscv_f16ToUi32 : (bits_rm, bits_H) -> (bits_fflags, bits_WU)
val riscv_i32ToF16 : (bits_rm, bits_W) -> (bits_fflags, bits_H)
val riscv_ui32ToF16 : (bits_rm, bits_WU) -> (bits_fflags, bits_H)
val riscv_f16ToI64 : (bits_rm, bits_H) -> (bits_fflags, bits_L)
val riscv_f16ToUi64 : (bits_rm, bits_H) -> (bits_fflags, bits_LU)
val riscv_i64ToF16 : (bits_rm, bits_L) -> (bits_fflags, bits_H)
val riscv_ui64ToF16 : (bits_rm, bits_LU) -> (bits_fflags, bits_H)
val riscv_f32ToI32 : (bits_rm, bits_S) -> (bits_fflags, bits_W)
val riscv_f32ToUi32 : (bits_rm, bits_S) -> (bits_fflags, bits_WU)
val riscv_i32ToF32 : (bits_rm, bits_W) -> (bits_fflags, bits_S)
val riscv_ui32ToF32 : (bits_rm, bits_WU) -> (bits_fflags, bits_S)
val riscv_f32ToI64 : (bits_rm, bits_S) -> (bits_fflags, bits_L)
val riscv_f32ToUi64 : (bits_rm, bits_S) -> (bits_fflags, bits_LU)
val riscv_i64ToF32 : (bits_rm, bits_L) -> (bits_fflags, bits_S)
val riscv_ui64ToF32 : (bits_rm, bits_LU) -> (bits_fflags, bits_S)
val riscv_f64ToI32 : (bits_rm, bits_D) -> (bits_fflags, bits_W)
val riscv_f64ToUi32 : (bits_rm, bits_D) -> (bits_fflags, bits_WU)
val riscv_i32ToF64 : (bits_rm, bits_W) -> (bits_fflags, bits_D)
val riscv_ui32ToF64 : (bits_rm, bits_WU) -> (bits_fflags, bits_D)
val riscv_f64ToI64 : (bits_rm, bits_D) -> (bits_fflags, bits_L)
val riscv_f64ToUi64 : (bits_rm, bits_D) -> (bits_fflags, bits_LU)
val riscv_i64ToF64 : (bits_rm, bits_L) -> (bits_fflags, bits_D)
val riscv_ui64ToF64 : (bits_rm, bits_LU) -> (bits_fflags, bits_D)
val riscv_f16ToF32 : (bits_rm, bits_H) -> (bits_fflags, bits_S)
val riscv_f16ToF64 : (bits_rm, bits_H) -> (bits_fflags, bits_D)
val riscv_f32ToF64 : (bits_rm, bits_S) -> (bits_fflags, bits_D)
val riscv_f32ToF16 : (bits_rm, bits_S) -> (bits_fflags, bits_H)
val riscv_f64ToF16 : (bits_rm, bits_D) -> (bits_fflags, bits_H)
val riscv_f64ToF32 : (bits_rm, bits_D) -> (bits_fflags, bits_S)

(** @功能 浮点比较函数。
 *  命名具有如下统一形式: riscv_f{16,32,64}{CmpOp}
 *  其中CmpOp表示具体是哪种比较, 包括: Lt, Le, Eq。
 *  这些比较函数会对两个操作数执行指定的比较, 并返回比较结果。
 *  如果任一操作数是NaN, 立即触发硬件异常。
 *  @param[0] bits('m) 源操作数0
 *  @param[1] bits('m) 源操作数1
```

(续下页)

(接上页)

```
 *  @return (bits_fflags, bool) 前一部分为异常标志位fflags，后一部分是比较结果
 *)
val riscv_f16Lt : (bits_H, bits_H) -> (bits_fflags, bool)
val riscv_f16Le : (bits_H, bits_H) -> (bits_fflags, bool)
val riscv_f16Eq : (bits_H, bits_H) -> (bits_fflags, bool)
val riscv_f32Lt : (bits_S, bits_S) -> (bits_fflags, bool)
val riscv_f32Le : (bits_S, bits_S) -> (bits_fflags, bool)
val riscv_f32Eq : (bits_S, bits_S) -> (bits_fflags, bool)
val riscv_f64Lt : (bits_D, bits_D) -> (bits_fflags, bool)
val riscv_f64Le : (bits_D, bits_D) -> (bits_fflags, bool)
val riscv_f64Eq : (bits_D, bits_D) -> (bits_fflags, bool)

(** 功能 浮点比较函数（quiet版本）。
 *  命名具有如下统一形式：riscv_f{16,32,64}{CmpOp}_quiet
 *  其中CmpOp表示具体是哪种比较，包括：Lt, Le。
 *  这些比较函数会对两个操作数执行指定的比较，并返回比较结果。
 *  当有操作数是NaN时，不会触发硬件异常，而是静默地传播下去。
 *  @param[0] bits('m) 源操作数0
 *  @param[1] bits('m) 源操作数1
 *  @return (bits_fflags, bool) 前一部分为异常标志位fflags，后一部分是比较结果
 *)
val riscv_f16Lt_quiet : (bits_H, bits_H) -> (bits_fflags, bool)
val riscv_f16Le_quiet : (bits_H, bits_H) -> (bits_fflags, bool)
val riscv_f32Lt_quiet : (bits_S, bits_S) -> (bits_fflags, bool)
val riscv_f32Le_quiet : (bits_S, bits_S) -> (bits_fflags, bool)
val riscv_f64Lt_quiet : (bits_D, bits_D) -> (bits_fflags, bool)
val riscv_f64Le_quiet : (bits_D, bits_D) -> (bits_fflags, bool)

(** @功能 从fcsr的相应位获取rounding mode
 *  @return rounding_mode 可能为RM_RNE, RM_RTZ, RM_RDN, RM_RUP, RM_RMM, RM_DYN
 *)
val get_fp_rounding_mode : unit -> rounding_mode

(** @功能 读取rounding mode。若指令编码域提供了显式的rounding␣
↪mode，使用之；否则，读取fcsr寄存器提供的rounding mode
 *  @param[0] bits(3) 指令编码中提供的舍入模式
 *  @return option(rounding_mode) 要么是RM_RNE, RM_RTZ, RM_RDN, RM_RUP, RM_
↪RMM之一，具体含义参见riscv spec，要么是None，表示编码错误
 *)
val select_instr_or_fcsr_rm : rounding_mode -> option(rounding_mode)

(** @功能 生成rounding mode在指令中的编码
 *  @return bits(3) 对应的编码
```

(续下页)

(接上页)

```
 *)
val encdec_rounding_mode : rounding_mode -> bits(3)

(** @功能 生成rounding mode在汇编中的字符串
 *  @return 对应的汇编字符串
 *)
val frm_mnemonic : rounding_mode -> string

(** @功能 针对浮点指令，生成浮点寄存器（若开启了浮点扩展）或整型寄存器名字
 *  @param[0] bits(5) 寄存器编号
 *  @return string 寄存器的名称
 *)
val freg_or_reg_name : regidx <-> string
```

矢量 API

```
(** @功能 获取sew（读取vtype寄存器的对应位）
 *  @return {8, 16, 32, 64}
 *)
val get_sew : unit -> {8, 16, 32, 64}

(** @功能 获取lmul（读取vtype寄存器的对应位）
 *  @return {-3, -2, -1, 0, 1, 2, 3}（真正的lmul还需要做以2为底的幂运算）
 *)
val get_lmul_pow : unit -> {-3, -2, -1, 0, 1, 2, 3}

(** @功能 获取vma（读取vtype寄存器的对应位）
 *  @return agtype UNDISTURBED, AGNOSTIC之一
 *)
val get_vtype_vma : unit -> agtype

(** @功能 获取vta（读取vtype寄存器的对应位）
 *  @return agtype UNDISTURBED, AGNOSTIC之一
 *)
val get_vtype_vta : unit -> agtype

(** @功能 判断两个vector寄存器是否overlap
 *  @param[0] regidx 源寄存器编号
 *  @param[1] regidx 目的寄存器编号
 *  @param[2] int 源EMUL之指数
 *  @param[3] int 目的EMUL之指数
 *  @return bool
 *)
```

(续下页)

(接上页)

```
val valid_reg_overlap : (regidx, regidx, int, int) -> bool

(** @功能 计算矢量操作的元素数量
 *  @param[0] int lmul之指数
 *  @param[1] int sew
 *  @return nat 矢量操作的实际元素数量
 *)
val get_num_elem : (int, int) -> nat

(** @功能 读取矢量寄存器
 *  @param[0] int('n) 待读取的元素数量
 *  @param[1] int('m) sew
 *  @param[2] int('p) lmul之指数
 *  @param[3] regidx 矢量寄存器的起始编号
 *  @return vector('n, bits('m)) 所读取的矢量寄存器的值，其中'n为元素数量，
↪'m为sew对应的位数
 *)
val read_vreg : forall 'n 'm 'p, 'n >= 0 & 'm >= 0. (int('n), int('m), int('p),␣
↪regidx) -> vector('n, bits('m))

(** @功能 写入矢量寄存器
 *  @param[0] int('n) 待读取的元素数量
 *  @param[1] int('m) sew
 *  @param[2] int('p) lmul之指数
 *  @param[3] regidx 矢量寄存器的起始编号
 *  @param[4] vector('n, bits('m)) 待写入的vector值，其中'n为元素数量，
↪'m为sew对应的位数
 *  @return unit
 *)
val write_vreg : forall 'n 'm 'p, 'n >= 0 & 'm >= 0. (int('n), int('m), int('p),␣
↪regidx, vector('n, bits('m))) -> unit

(** @功能 读取vmask
 *  @param[0] int('n) 待读取的元素数量
 *  @param[1] bits(1) vm编码位
 *  @param[2] regidx 矢量寄存器编号（对于riscv始终为0）
 *  @return vector('n, bool) 由一组mask位组成的vector，其中'n为元素数量
 *)
val read_vmask : forall 'n, 'n >= 0. (int('n), bits(1), regidx) -> vector('n, bool)

(** @功能 写入vmask
 *  @param[0] int('n) 待读取的元素数量
 *  @param[1] regidx 矢量寄存器编号（对于riscv始终为0）
```

(续下页)

(接上页)

```
 *  @param[2] vector('n, bool) 待写入的值，其中n为元素数量
 *  @return unit
 *)
val write_vmask : forall 'n, 'n >= 0. (int('n), regidx, vector('n, bool)) -> unit

(** @功能 生成vector寄存器名字
 *  @param[0] regidx 寄存器编号
 *  @return string vector寄存器的名称
 *)
val vreg_name : regidx <-> string

(** @功能 生成vmask寄存器名字，如"v0.t"
 *  @param[0] bits(1) 指令编码中对应于vmask的编码位
 *  @return string␣
↪vmask寄存器的名称，类型为字符串。如果参数为0b1，返回空字符串；如果为0b0，返回"v0.t"
 *)
val maybe_vmask : bits(1) -> string
```

杂项 API

```
(** @功能 生成全0比特数
 *  @param[0] implicit('n) 指定长度
 *  @return bits('n) 全0比特数
 *)
val zeros : forall 'n, 'n >= 0 . implicit('n) -> bits('n)

(** @功能 生成全1比特数
 *  @param[0] implicit('n) 指定长度
 *  @return bits('n) 全1比特数
 *)
val ones : forall 'n, 'n >= 0 . implicit('n) -> bits('n)

(** @功能 将某一长度的比特数转换为有符号整数。
 *  @param[0] bits('n) 待转化数据
 *  @return range(- (2 ^ ('n - 1)), 2 ^ ('n - 1) - 1) 转换结果，处于`[(2 ^ ('n - 1)),␣
↪2 ^ ('n - 1) - 1]`之间的一个有符号整数
 *)
val signed : forall 'n, 'n > 0. bits('n) -> range(- (2 ^ ('n - 1)), 2 ^ ('n - 1) - 1)

(** @功能 将某一长度的比特数转换为无符号整数。
 *  @param[0] bits('n) 待转化数据
 *  @return range(0, 2 ^ 'n - 1) 转换结果，处于`[0, 2 ^ 'n - 1]`之间的一个无符号整数
 *)
```

(续下页)

(接上页)

```
val unsigned : forall 'n. bits('n) -> range(0, 2 ^ 'n - 1)

(** @功能 将有符号整数转换为指定长度的比特数
 *  @param[0] int('l) 指定长度
 *  @param[1] int 待转化数据
 *  @return bits('l) 转换结果，'l为指定长度
 *)
val to_bits : forall 'l, 'l >= 0.(int('l), int) -> bits('l)

(** @功能 字节内比特倒序
 *  @param[0] bits(8) 原始数据
 *  @return bits(8)␣
↪倒序后的比特数，其第7位是原始数的第0位，第6位是原始数的第1位，依次类推
 *)
val reverse : bits(8) -> bits(8)

(** @功能 对{1, 2, 4, 8, 16, 32, 64}范围内的整数求对数
 *  @param[0] {1, 2, 4, 8, 16, 32, 64} 给定的整数
 *  @return int 给定整数的以2为底的对数
 *)
val log2 : forall 'n, 'n in {1, 2, 4, 8, 16, 32, 64}. int('n) -> int

(** @功能 生成空格，用于描述汇编字符串
 *  @return string
 *)
val spc : unit <-> string

(** @功能 生成分隔符，用于描述汇编字符串
 *  @return string
 *)
val sep : unit <-> string
```

7.1.6 SAIL 语言

本节简要介绍 SAIL 语言的基本要素和用法。关于语言的完整介绍，请参见：SAIL 语言规范

变量

- 变量命名：SAIL 与其它语言没有区别，可以参考 C 语言命名规范。
- 变量绑定：可以用 *let a = 1* 为变量绑定一个值。
- 变量可变性：SAIL 中变量分为两种，不可变的和可变的。其中不可变的用 *let*，可变的用 *var*：

```
let x = 3;  // 不可变变量x
var y = 2;  // 可变变量y
y = y + 1;  // 修改y的值为3
```

寄存器

SAIL 可以用 *register* 定义一个寄存器：

```
register PC : bits(64)  // 64位寄存器PC
register X1 : bits(64)  // 64位寄存器X1
```

基本类型

SAIL 中每个值都有确定的数据类型，总的来说可以分为两类：基本类型和复合类型。基本类型意味着它们往往是一个最小化原子类型，无法解构为其它类型，包含以下几种：

- 数值类型：整数 int, 自然数 nat, 范围 range(n, m)
- 比特类型：单个比特 bit，多个比特 bits(n)
- 布尔类型：bool，有两个值：true、false
- 字符串类型：字符串 string，主要用于错误报告和调试
- 单元类型：即 () ，其唯一的值也是 ()，类似 C/C++ 语言中 void 类型

类型推导和标注 与 Python、JavaScript 等动态语言不同，SAIL 是一门静态类型语言，也就是在编译期必须知道所有变量的的类型。但是与 C、Java 等显式声明类型的语言也不同，SAIL 是一门隐式类型的语言，这意味着我们不需要为每个变量指定类型。SAIL 编译器可以根据变量的值和上下文中的使用方式来自动推导变量的类型，这叫类型推导，但是在某些情况下，它无法推导出变量类型，需要手动去给予一个类型标注：

```
let a = 3;              // 自动推导出a的类型为int(3)
var v : bits(8) = 0xFF; // 手动指定v的类型为bits(8)
```

表达式和块

SAIL 的函数体由一系列表达式组成，每个表达式都会返回一个值，例如 *1 + 3* 求值结果为 *4*。块也是一种表达式，它将表达式用 *{* 和 *}* 包裹起来，它会将括号内最后一个表达式的求值结果作为整个块的值：

```
{
    let x : int = 3;
    var y : int = 2;
    y = y + 1;
```

(续下页)

(接上页)

```
    x + y
}
```

上面这个块求值结果为 *6*。

函数

SAIL 中我们可以定义函数。函数由两部分组成：用 *val* 声明类型签名，用 *function* 定义函数体：

```
val foo : (int, int) -> int

function foo (x, y) = {
    x + y
}

// 对函数foo的使用
let sum = foo (3, 5);
```

条件表达式

SAIL 中用 *if* 来表达分支条件：

```
if illegal(insn) then handle_illegal();
if illegal(insn) then handle_illegal() else {...}
```

循环表达式

SAIL 中一般使用 *foreach* 来表达循环：

```
foreach (i from 0 to (num_elem - 1)) {
    result[i] = vs2_val[i] + vs1_val[i]
}
```

注释

SAIL 中可以用 *//* 添加行注释。

7.1.7 FAQ

Windows 系统环境配置

SAIL 工具使用 MinGW 编译器来生成工具链插件，因此需要确保 Windows 机器上已经安装了 MinGW 编译器。一般我们在 Windows 上安装 MinGW 编译器是通过 MSYS2 来安装的：

```
pacman -S mingw64/mingw-w64-ucrt-x86_64-gcc
pacman -S mingw64/mingw-w64-x86_64-z3
```

这里我们仅安装了两个必要的组件。安装后，我们需要把相关的资源路径加入到系统环境变量 **path** 中。如：C:\msys64\mingw64\bin

第八章　FAQ

8.1　Windows 系统环境配置

CTB 工具使用 MSVC 编译器来生成工具链插件，因此需要确保 Windows 机器上已经安装了 MSVC 编译器。一般我们在 Windows 上安装编译器是通过微软的 Visual Studio 来安装的：

这里我们仅安装了两个必要的组件。安装后，我们需要把相关的资源路径加入到系统环境变量 **path** 中。如：C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build

索引

B

F

I