龙芯中科
LOONGSON TECHNOLOGY

LOONGSON

龙芯 3A4000/3B4000 处理器
向量指令软件开发手册

V1.00

2020 年 5 月

龙芯中科技术有限公司

北京市海淀区温泉镇中关村环保科技示范园龙芯产业园2号楼 100095
Loongson Industrial Park, building 2，Zhongguancun environmental protection park
Haidian District，Beijing

www.loongson.cn

版权声明

免责声明

本文档仅提供阶段性信息，所含内容可根据产品的实际情况随时更新，恕不另行通知。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

龙芯中科技术有限公司

Loongson Technology Corporation Limited

地址：北京市海淀区中关村环保科技示范园龙芯产业园 2 号楼

Building No.2, Loongson Industrial Park,

Zhongguancun Environmental Protection Park, Haidian District, Beijing

电话(Tel)：010-62546668

传真(Fax)：010-62600826

阅读指南

本手册描述了龙芯 3A4000/3B4000 处理器上使用向量指令进行软件开发所需的相关内容。

本手册内容主要面向汇编程序开发人员，尤其是那些需要在汇编代码中使用向量指令的编程人员。

版本历史

文档更新记录	文档名	龙芯 3A4000/3B4000 处理器向量指令软件开发手册
	版本号	1.00
	创建人	芯片研发部
	创建日期	2020/05/13

更新历史

版本号	更新日期	更新内容
0.01	2020/05/13	内部开发版本。完成了整个文档的框架，以及除软件编程和指令延迟吞吐率之外的内容。
0.02	2020/05/17	内部开发版本。更换文档模板，将所有字体调整为开源免费字体。
0.03	2020/05/25	内部开发版本。完成文档所有内容，供内部评审。
1.00	2020/06/01	对外发布版本 1.0。

手册信息反馈：service@loongson.cn

也可通过问题反馈网站 http://bugs.loongnix.org/ 向我司提交产品使用过程中的问题，并获取技术支持。

目　录

1　向量指令功能介绍 1

1.1　向量指令功能概述 1

1.1.1　向量寄存器 1

1.1.2　向量数据类型 1

1.1.3　向量浮点运算 2

1.1.4　向量访存 2

1.2　向量指令编程 3

1.2.1　向量数据类型定义 3

1.2.2　简单代码示例 4

1.2.3　ABI 及函数传参约定 5

1.2.4　手写汇编函数 6

1.2.5　编译执行 7

2　向量指令优化指导 9

2.1　向量指令优化相关的流水线特征 9

2.2　向量指令的延迟与吞吐率 10

2.2.1　概念介绍 10

2.2.2　向量整数运算指令 10

2.2.3　向量整数比较指令 12

2.2.4　向量位运算指令 13

2.2.5　向量浮点运算指令 14

2.2.6　向量浮点比较指令 15

2.2.7　向量浮点转换指令 15

2.2.8　向量定点数运算指令 16

2.2.9　向量转移指令 16

2.2.10　向量访存和数据搬运指令 16

2.2.11　向量数据混洗指令 17

3　附录 A 定点数格式定义 18

图 目 录

图 3-1 Q31 型定点数据格式 18
图 3-2 定点数表示范围 18
图 3-3 Q7 型定点数真值计算 19

表 目 录

表 1-1 GCC 中向量数据类型定义..... 3

1 向量指令功能介绍

龙芯 3A4000/3B4000 处理器中实现了单指令多数据流（Single Instruction Multiple Data，简称 SIMD）功能的指令，亦称之为向量指令。龙芯 3A4000/3B4000 处理器中所实现的向量指令集在功能上兼容 MIPS 指令集的 SIMD 扩展指令集（简称 MSA）。因此，本手册中将只对向量指令的功能做概要介绍，详细的指令功能定义请参阅 MIPS 公司发布的文档*《MIPS® Architecture for Programmers IV-j: The MIPS64® SIMD Architecture Module》（Document Number: MD00868, Revision 1.12）*。

1.1 向量指令功能概述

龙芯 3A4000/3B4000 处理器所实现的向量指令集包含数百条指令，功能涉及运算、转移和访存。整个向量指令集仍然采用 RISC 指令集的设计理念，因此只有向量访存指令才会将内存作为操作数，其余所有向量指令的操作数都来自于寄存器。除部分向量指令还会操作通用寄存器（GR）外，向量指令的操作数主要存放在向量寄存器中。

1.1.1 向量寄存器

向量寄存器有 32 个。汇编编程时记作$w0 ~ $w31。

向量寄存器的数据位宽为 128 位。从软件的视角来看，每个向量寄存器中存放的数据是一个 128 位的位向量，第 0 位位于向量寄存器最右边，第 127 位位于向量寄存器的最左边。

向量指令操作向量寄存器时，总是以 128 位为一个基本寻址单位。也就是说，不存在将每个 128 位的高、低 64 位，或是 4 个 32 位分开来独立寻址的情况。

每个向量寄存器的低半部分与同号的浮点寄存器是重合的。也就是说，当执行某条浮点指令更新了浮点寄存器 fn（n=0..31），那么$wn 的[63:0]位被更新为相同值，但是请注意，此时$wn 的[127:64]的值是不确定的。

使用向量指令时，浮点寄存器必须呈现为 32 个 64 位宽的状态，即 CP0.Status.FR=1。这也意味着在 o32 ABI 下是无法使用向量指令的。我们推荐编程人员使用 n64 ABI。

龙芯 3A4000/3B4000 处理器的向量指令集不实现向量寄存器分块（Vector Registers Partitioning）机制，即 MSAIR.WRP=0。

1.1.2 向量数据类型

向量指令所操作的向量的元素以位宽作为划分依据，分为：比特、字节（8 比特）、半字（16 比特）、字（32 比特）、双字（64 比特）。各类元素构成的向量存放在向量寄存器中时，都是从向量寄存器的低位向高位依次摆放。

如果指令是将数据作为整数（integer）进行操作的，那么又可进一步细分为：有符号字节（对应 C 语言中的 char）、无符号字节（对应 C 语言中的 unsigned char）、有符号半字（对应 C 语言中的 short）、无符号半字（对应 C 语言中的 unsigned short）、有符号字（对应 C 语言中的 int）、无符号字（对应 C 语言中的 unsigned int）、有符号双字（对应 C 语言中的 long long）、无符号双字（对应 C 语言中的 unsigned long long）。

如果指令是将数据作为定点数（fixed-point）进行操作的，那么包括：Q15 和 Q31 两种格式的定点数。有关 Q15 和 Q31 定点数的具体定义请见本文档附录 A 定点数格式定义。

如果指令是将数据作为浮点数（floating-point）进行操作的，那么包括：半精度浮点数、单精度浮点数和双精度浮点数。其中半精度浮点数仅在浮点类型转换指令中使用，没有直接对半精度浮点数进行加、减、乘等运算的指令。

1.1.3 向量浮点运算

龙芯 3A4000/3B4000 处理器中的向量浮点运算都遵循 IEEE 754TM-2008 标准。

龙芯 3A4000/3B4000 处理器中的向量浮点运算支持非规格化数（Subnormal）的处理，即向量浮点运算的源操作数或目的操作数出现非规格化数的时候，由硬件直接处理，不会触发标志为 E（Unimplemented Operation）的向量浮点例外。

龙芯 3A4000/3B4000 处理器中的向量浮点运算不支持 Flush to Zero 机制。即无论 MSACSR.FS 置为何值，硬件都按照 MSACSR.FS=0 的定义来处理。

1.1.4 向量访存

尾端

所有龙芯处理器均只支持小尾端寻址模式。因此一个向量操作数存放在内存中时，其中的各个元素是从内存地址的低位向内存地址的高位依次摆放。

地址对齐

龙芯 3A4000/3B4000 中的向量访存指令的没有地址对齐的要求，即向量访存指令执行时不会仅因为地址最低 4 位不为全 0 而触发地址错例外。不过，当向量访存指令的地址对齐时，其执行性能会优于地址不对齐的情况。

需要注意的是，当一条地址不对齐的向量访存指令的访问跨越了页（或者段）的边界，而所跨两侧的 Cache 属性或是访问权限不同时，将触发地址错例外（ADEL、ADES）。

原子性

对于单个线程来说，在绝大多数情况下，无论地址是否对齐，单条向量访存指令的访存总是原子的。其原子性的一个典型效果是，例外（含中断）事件要么在一个向量访存指令产生完整的执行效果之前发生，要么在其产生完整的执行效果之后发生，不会出现向量访存指令已经且只完成了部分执行效果的时候发生例

外事件。单线程原子性另一个典型效果是，一个向量访存指令的访存动作只可能全部完成在一个栅障（barrier）之前或之后，不会出现一个地址不对齐的向量访存指令的不同部分的访存动作分布在一个栅障的前后两侧。

之所以说向量访存指令对于单个线程的原子性是“绝大多数情况”而不是“所有情况”，是因为存在一些极为少见的情况会破坏这种原子性。譬如，一个跨 Cache 行的向量 store 指令在访问第二个 Cache 行的时候发生了 ECC 校验错例外。

在多线程多核执行场景下，除非向量访存指令的地址是对齐的，否则不保证该指令的访存操作被同一个共享一致域上的所有观察者（observer）观察为原子的。不过，由于龙芯 3A4000/3B4000 处理器实现的是弱一致性访存模型，同时向量访存指令不会用来构建栅障，所以一个正确的程序自然会通过插入栅障来确保地址不对齐的向量访存指令的访存操作被所有观察者观察为原子的。

1.2 向量指令编程

在使用向量指令编程过程中，开发人员可能需要了解一些编译器对向量的支持工作，以加快开发效率，协助程序调试调优。具体的，它涉及到 ABI 中关于寄存器使用的约定、程序编译选项、内连函数的调用方法等方面。下面希望通过部分代码示例来进行讲述。

1.2.1 向量数据类型定义

GCC 编译器（GNU C Compiler）对向量数据类型的定义包含在 <msa.h> 头文件中。

虽然处理器在执行向量指令时没有地址对齐的要求，但为了配合使用 C 语言中“数组”这种数据结构，GCC 在定义向量数据类型时所采用的最小对齐单位为——向量的元素宽度。为了提升性能，推荐对齐到向量寄存器宽度。

表 1-1 GCC 中向量数据类型定义

向量元素类型	对齐宽度	C 语言定义
有符号字节	向量（16 字节）	typedef signed char **v16i8** __attribute__ ((vector_size(16), aligned(16)));
	字节	typedef signed char **v16i8_b** __attribute__ ((vector_size(16), aligned(1)));
无符号字节	向量（16 字节）	typedef unsigned char **v16u8** __attribute__ ((vector_size(16), aligned(16)));
	字节	typedef unsigned char **v16u8_b** __attribute__ ((vector_size(16), aligned(1)));
有符号半字	向量（16 字节）	typedef short **v8i16** __attribute__ ((vector_size(16), aligned(16)));
	半字（2 字节）	typedef short **v8i16_h** __attribute__ ((vector_size(16), aligned(2)));

向量元素类型	对齐宽度	C 语言定义
无符号半字	向量（16 字节）	typedef unsigned short **v8u16** __attribute__ ((vector_size(16), aligned(16)));
	半字（2 字节）	typedef unsigned short **v8u16_h** __attribute__ ((vector_size(16), aligned(2)));
有符号字	向量（16 字节）	typedef int **v4i32** __attribute__ ((vector_size(16), aligned(16)));
	字（4 字节）	typedef int **v4i32_w** __attribute__ ((vector_size(16), aligned(4)));
无符号字	向量（16 字节）	typedef unsigned int **v4u32** __attribute__ ((vector_size(16), aligned(16)));
	字（4 字节）	typedef unsigned int **v4u32_w** __attribute__ ((vector_size(16), aligned(4)));
有符号双字	向量（16 字节）	typedef long long **v2i64** __attribute__ ((vector_size(16), aligned(16)));
	双字（8 字节）	typedef long long **v2i64_d** __attribute__ ((vector_size(16), aligned(8)));
无符号双字	向量（16 字节）	typedef unsigned long long **v2u64** __attribute__ ((vector_size(16), aligned(16)));
	双字（8 字节）	typedef unsigned long long **v2u64_d** __attribute__ ((vector_size(16), aligned(8)));
单精度浮点	向量（16 字节）	typedef float **v4f32** __attribute__ ((vector_size(16), aligned(16)));
	字（4 字节）	typedef float **v4f32_w** __attribute__ ((vector_size(16), aligned(4)));
双精度浮点	向量（16 字节）	typedef double **v2f64** __attribute__ ((vector_size(16), aligned(16)));
	双字（8 字节）	typedef double **v2f64_d** __attribute__ ((vector_size(16), aligned(8)));

1.2.2 简单代码示例

此部分内容包括汇编语句、C 语言中内嵌汇编、C 语言中调用指令内联函数的简单示例。

汇编指令语句

```
addv.w      $w5,$w1,$w2
fill.w      $w6,$2
addvi.w     $w7,$w1,17
splati.w    $w8,$w2[2]
```

其中，“$w5” 表示 5 号向量寄存器，“$2”表示 2 号通用寄存器，“17”表示立即数 17。

C 语言中内嵌汇编

```
v4i32 t;
v4f32 a, b, c;
__asm__ volatile (
        "fadd.w    %w[a],%w[b],%w[c]          \n"
        "fslt.w      %w[t],%w[b],%w[c]     \n"
        : [a] "=&f"(a), [t] "=&f"(t)
        : [b]     "f"(b), [c]    "f"(c) );
```

在内嵌汇编中，如果输入输出操作数为向量数据类型，则其寻址类型限定符（operand constraint）采用“f”。

C 语言中调用指令内联函数

```
v4i32 t;
v4f32 a, b, c;
a = __msa_fadd_w(b, c);
t = __msa_fslt_w(b, c);
```

GCC 编译器中提供可实现与向量指令相同功能的内联函数（intrinsics），这些内联函数的声明包含在<msa.h> 头文件中。

不同向量指令所对应的内敛函数的调用方式请参阅*《MIPS® SIMD Architecture》（Document Number: MD00926, Revision 1.03）*的 “*Section 6*”章节，另外请注意在内联函数名字前添加“__msa_”前缀。

1.2.3 ABI 及函数传参约定

支持 n32 和 n64 ABI，推荐使用 n64 ABI。

C 代码中当以向量数据类型作为函数参数时，不论向量的内部元素是何类型，均借助通用寄存器以实现值传递。而实现值传递所借助的通用寄存器具体是哪些，则参见不同 ABI 下通用寄存器的传参约定。

```
#include <msa.h>
v8i16 a = {0, 1, 2, 3, 4, 5, 6, 7};
v8i16 b = {0, 1, 2, 3, 4, 5, 6, 7};
v8i16 c;
v8i16 my_func (v8i16 i, v8i16 j);
int main()
{
```

```
    c = my_func(a, b);
    return 0;
}
```

如上所示，定义向量数据类型 a、b、c，调用用户自定义子函数 my_func 进行运算，结果存入 c 中。

以 n64 ABI 为例，a 数据借助通用寄存器{$5,$4}传入子函数 my_func 中，其中$4 存储 a 向量的低 64 位，$5 存储 a 向量的高 64 位；b 数据借助通用寄存器{$7,$6}传入子函数 my_func 中。my_func 函数运行结束后返回值借助通用寄存器{$3,$2}返回给调用者函数（此处为 main 函数）。（由调用者函数完成结果通用寄存器到 c 向量的赋值。）

1.2.4 手写汇编函数

本节讲述如何实现下述需求：用 C 语言编写一个调用者函数；用汇编语言编写一个子函数，该子函数接受向量数据类型参数，使用向量指令执行任务逻辑操作，并返回向量类型结果。

衔接第 1.2.4 节中内容，调用者函数已写好，下面以 my_func()子函数为例，在 n64 ABI 下，如果用汇编语言实现该子函数，应当涵盖如下要点。

```
    .text ;
    .globl   my_func ;
    .ent     my_func ;
    .type    my_func, @function ;
    ....                                    ### 完成函数声明
my_func:
    …
    insert.d     $w0[0],$4
    insert.d     $w1[0],$6              ### 将输入参数值置入向量寄存器中
    insert.d     $w0[1],$5
    insert.d     $w1[1],$7
    …
    addv.d       $w0,$w0,$w1            ### 使用向量指令完成任务逻辑
    …
    copy_s.d     $2,$w0[0]
    copy_s.d     $3,$w0[1]              ### 借助通用寄存器完成结果值传递
    jr   $31                            ### 函数返回
    nop
    ....
    .end     msa_addv_d                 ### 函数尾
```

还要注意，在子函数内使用向量寄存器时要考虑 ABI 中关于浮点寄存器的约定，这是因为向量寄存器的低半部分即是同号的浮点寄存器。比如我们想要在子函数内使用$w24 向量寄存器时，一定要先保存后使用，否则将破坏$f24 浮点寄存器的使用约定。

1.2.5 编译执行

在 C 语言中使用向量数据类型或调用向量指令内联函数，需要包含<msa.h> 头文件。

使用向量指令编写代码，在编译时应添加“-mmsa”选项。

2 向量指令优化指导

本章对在 3A4000/3B4000 处理器上用向量指令进行汇编程序开发以优化程序性能给与一些指导性信息。向量指令优化是一项通过使用向量指令开发应用中的数据并行性来获得性能加速的优化技术。但是，这并不意味着一个具有丰富数据并行性的应用通过向量指令优化后就一定能获得大幅度的性能提升。通常我们建议编程人员在对一个应用开展手动向量优化之时，首先通过性能分析工具（如 perf）明确如下几个条件是否成立：

- 待优化热点的性能瓶颈不是因为大量长延迟的 Cache Miss；
- 待优化热点的性能瓶颈不是因为大量的 TLB 缺失；
- 待优化热点的性能瓶颈不是因为大量预测错误的转移指令；
- 待优化热点的 IPC 已经较高，且呈现出计算密集的特征。

在上述条件都满足的前提下，如果密集的计算又能够采用单指令多数据流的方式予以编程，那么向量指令优化后才有可能获得预期的性能提升。

对于龙芯 3A4000/3B4000 这样一款具有较大指令调度窗口的动态乱序调度执行处理器而言，汇编编程人员在进行向量指令优化时，首要的任务是减少执行指令的数目，其次才是结合处理器的流水线特征和向量指令的延迟与吞吐率进行人工的指令调度。

减少执行指令数目的优化与具体的微结构设计关联度角度，它更多涉及如何将算法更高效的向量化。常用的几个优化建议包括：

- 尽量直接采用向量访存指令在内存与向量寄存器之间搬运数据，而不是通过“向量寄存器—通用寄存器—标量访存指令”这种间接搬运通路。必要时，可以先通过数据拷贝将数据的存储格式进行调整。
- 尽量让向量访存指令的地址对齐。
- 尽量利用向量条件选择指令构建起向量条件执行序列，而不是将向量逐元素拆分后使用标量分支指令依次判断。

有关算法向量化的优化建议不在此处展开叙述。本节接下来将主要介绍与向量指令优化相关的处理器微结构方面的特性。

2.1 向量指令优化相关的流水线特征

龙芯 3A4000/3B4000 处理器的前端流水线的理论峰值带宽是 4 操作/周期。除了后端阻塞外，影响前端流水线供给带宽的因素主要有：分支误预测、指令 Cache miss、指令 TLB miss、密集的跳转分支、连续 4 条指令跨越 32 字节边界。部分指令在前端流水线将被拆分成多条操作，所以如果以指令作为统计口径来看

达不到每周期 4 条指令。不过绝大多数情况下，此时的性能瓶颈来自于后端发射的带宽，所以汇编编程人员并不需要在意。

龙芯 3A4000/3B4000 处理器的后端有 8 条执行流水线，其发射峰值带宽是 8 操作/周期。具体来说，每个周期至多发射 4 个定点运算操作、2 个浮点/向量运算操作以及 2 个访存操作。绝大多数操作在后端流水线执行时，只要源操作数都准备好就可以发射，即所谓乱序发射机制。当同一时刻存在多个准备好可以发射的操作时，通常会选择最早的一个操作。不过，由于整个发射机制是乱序加推测的，所以实际执行过程中的发射顺序可能未必和软件人员静态推演的一致。不过，整个发射机制总体上还是能够保证，如果静态推演情况下执行效率提升，那么最终实际执行效率不会下降。

龙芯 3A4000/3B4000 处理器后端的 4 条定点运算和 2 条浮点/向量运算执行流水线，每一条执行流水线只有一个结果写回总线，所以当同一条执行流水线先后发射执行了若干延迟不同的操作，那么就有可能出现同一周期内多个操作竞争结果总线的情况。此时，总是执行延迟最短的操作抢到结果总线。由于这个机制的影响，程序在实际执行时的指令执行延迟有可能会高于 2.2 节中向量指令的延迟与吞吐率中列举的指令延迟，且增加的周期数要视具体的指令序列情况而定。

2.2 向量指令的延迟与吞吐率

2.2.1 概念介绍

在本节中，我们将逐条列举出龙芯 3A4000/3B4000 处理器中各向量指令的延迟和吞吐率信息。这里延迟和吞吐率的定义如下：

- 延迟——单位为时钟周期。其含义是，在该指令发射执行之后，使用该指令结果作为源操作数的指令最快能在几个时钟周期后发射执行。
- 吞吐率——单位为指令条数。其含义是，在最理想的情况下，每个时钟周期内最多能够同时发射执行几条该指令。

提供延迟和吞吐率信息旨在为汇编编程人员对汇编代码进行人工指令调度提供参数依据，以最大化处理器执行程序时的指令并发度，尽量消除因为数据相关和结构相关所引发的流水线阻塞。不过需要说明的是，此处给出的延迟和吞吐率信息均是在理想场景下的数值，程序实际执行过程中各指令最终所体现的数据可能与之存在差异。

2.2.2 向量整数运算指令

指令助记符	延迟	吞吐率
ADDV.B/H/W/D	1	2
ADDVI.B/H/W/D	1	2
ADD_A.B/H/W/D	2	2

指令助记符	延迟	吞吐率
ADDS_A.B/H/W/D	2	2
ADDS_S.B/H/W/D	1	2
ADDS_U.B/H/W/D	1	2
HADD_S.H/W/D	2	2
HADD_U.H/W/D	2	2
ASUB_S.B/H/W/D	2	2
ASUB_U.B/H/W/D	2	2
AVE_S.B/H/W	1	2
AVE_S.D	2	2
AVE_U.B/H/W	1	2
AVE_U.D	2	2
AVER_S.B/H/W	1	2
AVER_S.D	2	2
AVER_U.B/H/W	1	2
AVER_U.D	2	2
DOTP_S.H/W/D	4	2
DOTP_U.H/W/D	4	2
DPADD_S.H/W/D	4	2
DPADD_U.H/W/D	4	2
DPSUB_S.H/W/D	4	2
DPSUB_U.H/W/D	4	2
DIV_S.B	1+(3-7)*8+1	2/延迟
DIV_S.H	1+(3-9)*4+1	2/延迟
DIV_S.W	1+(3-13)*2+1	2/延迟
DIV_S.D	1+(3-21)*1+1	2/延迟
DIV_U.B	1+(3-7)*8+1	2/延迟
DIV_U.H	1+(3-9)*4+1	2/延迟
DIV_U.W	1+(3-13)*2+1	2/延迟
DIV_U.D	1+(3-21)*1+1	2/延迟
MADDV.B/H/W/D	4	2
MAX_A.B/H/W/D	2	2
MIN_A.B/H/W/D	2	2
MAX_S.B/H/W	1	2
MAX_S.D	2	2
MAX_U.B/H/W	1	2
MAX_U.D	2	2
MAXI_S.B/H/W	1	2
MAXI_S.D	2	2

指令助记符	延迟	吞吐率
MAXI_U.B/H/W	1	2
MAXI_U.D	2	2
MIN_S.B/H/W	1	2
MIN_S.D	2	2
MIN_U.B/H/W	1	2
MIN_U.D	2	2
MINI_S.B/H/W	1	2
MINI_S.D	2	2
MINI_U.B/H/W	1	2
MINI_U.D	2	2
MSUBV.B/H/W/D	4	2
MULV.B/H/W/D	4	2
MOD_S.B	1+(3-8)*8+1	2/延迟
MOD_S.H	1+(3-10)*4+1	2/延迟
MOD_S.W	1+(3-14)*2+1	2/延迟
MOD_S.D	1+(3-22)*1+1	2/延迟
MOD_U.B	1+(3-8)*8+1	2/延迟
MOD_U.H	1+(3-10)*4+1	2/延迟
MOD_U.W	1+(3-14)*2+1	2/延迟
MOD_U.D	1+(3-22)*1+1	2/延迟
SAT_S.B/H/W/D	2	2
SAT_U.B/H/W/D	2	2
SUBS_S.B/H/W/D	1	2
SUBS_U.B/H/W/D	1	2
HSUB_S.H/W/D	2	2
HSUB_U.H/W/D	2	2
SUBSUU_S.B/H/W/D	2	2
SUBSUS_U.B/H/W/D	2	2
SUBV.B/H/W/D	1	2
SUBVI.B/H/W/D	1	2

2.2.3 向量整数比较指令

指令助记符	延迟	吞吐率
CEQ.B/H/W/D	1	2
CEQI.B/H/W/D	1	2
CLE_S.B/H/W	1	2
CLE_S.D	2	2

指令助记符	延迟	吞吐率
CLE_U.B/H/W	1	2
CLE_U.D	2	2
CLEI_S.B/H/W	1	2
CLEI_S.D	2	2
CLEI_U.B/H/W	1	2
CLEI_U.D	2	2
CLT_S.B/H/W	1	2
CLT_S.D	2	2
CLT_U.B/H/W	1	2
CLT_U.D	2	2
CLTI_S.B/H/W	1	2
CLTI_S.D	2	2
CLTI_U.B/H/W	1	2
CLTI_U.D	2	2

2.2.4 向量位运算指令

指令助记符	延迟	吞吐率
AND.V	1	2
ANDI.B	1	2
BCLR.B/H/W/D	2	2
BCLRI.B/H/W/D	2	2
BINSL.B/H/W/D	2	2
BINSLI.B/H/W/D	2	2
BINSR.B/H/W/D	2	2
BINSRI.B/H/W/D	2	2
BMNZ.V	1	2
BMNZI.B	1	2
BMZ.V	1	2
BMZI.B	1	2
BNEG.B/H/W/D	2	2
BNEGI.B/H/W/D	2	2
BSEL.V	1	2
BSELI.B	1	2
BSET.B/H/W/D	2	2
BSETI.B/H/W/D	2	2
NLOC.B/H/W/D	1	2
NLZC.B/H/W/D	1	2

指令助记符	延迟	吞吐率
NOR.V	1	2
NORI.B	1	2
PCNT.B/H/W/D	2	2
OR.V	1	2
ORI.B	1	2
SLL.B/H/W/D	1	2
SLLI.B/H/W/D	1	2
SRA.B/H/W/D	1	2
SRAI.B/H/W/D	1	2
SRAR.B/H/W/D	2	2
SRARI.B/H/W/D	2	2
SRL.B/H/W/D	1	2
SRLI.B/H/W/D	1	2
SRLR.B/H/W/D	2	2
SRLRI.B/H/W/D	2	2
XOR.V	1	2
XORI.B	1	2

2.2.5 向量浮点运算指令

指令助记符	延迟	吞吐率
FADD.W/D	5	2
FDIV.W	1+(3-11)*2+1	2/延迟
FDIV.D	1+(3-18)+1	2/延迟
FEXP2.W/D	4	2
FLOG2.W/D	4	2
FMADD.W/D	5	2
FMAX.W/D	2	2
FMAX_A.W/D	2	2
FMSUB.W/D	5	2
FMIN.W/D	2	2
FMIN_A.W/D	2	2
FMUL.W/D	5	2
FRCP.W	1+(3-11)*2+1	2/延迟
FRCP.D	1+(3-18)+1	2/延迟
FRINT.W/D	4	2
FRSQRT.W	1+(6-28)*2+1	2/延迟
FRSQRT.D	1+(6-49)+1	2/延迟

指令助记符	延迟	吞吐率
FSQRT.W	1+(3-17)*2+1	2/延迟
FSQRT.D	1+(3-31)+1	2/延迟
FSUB.W/D	5	2
FCLASS.W/D	2	2

2.2.6 向量浮点比较指令

指令助记符	延迟	吞吐率
FCAF.W/D	2	2
FCUN.W/D	2	2
FCOR.W/D	2	2
FCEQ.W/D	2	2
FCUNE.W/D	2	2
FCUEQ.W/D	2	2
FCNE.W/D	2	2
FCLT.W/D	2	2
FCULT.W/D	2	2
FCLE.W/D	2	2
FCULE.W/D	2	2
FSAF.W/D	2	2
FSUN.W/D	2	2
FSOR.W/D	2	2
FSEQ.W/D	2	2
FSUNE.W/D	2	2
FSUEQ.W/D	2	2
FSNE.W/D	2	2
FSLT.W/D	2	2
FSULT.W/D	2	2
FSLE.W/D	2	2
FSULE.W/D	2	2

2.2.7 向量浮点转换指令

指令助记符	延迟	吞吐率
FEXDO.H/W	3	1
FEXUPL.W/D	3	1
FEXUPR.W/D	3	1

指令助记符	延迟	吞吐率
FFINT_S.W/D	4	2
FFINT_U.W/D	4	2
FFQL.W/D	5	1
FFQR.W/D	5	1
FTINT_S.W/D	4	2
FTINT_U.W/D	4	2
FTRUNC_S.W/D	4	2
FTRUNC_U.W/D	4	2
FTQ.H/W	5	1

2.2.8 向量定点数运算指令

指令助记符	延迟	吞吐率
MADD_Q.H/W	4	2
MADDR_Q.H/W	4	2
MSUB_Q.H/W	4	2
MSUBR_Q.H/W	4	2
MUL_Q.H/W	4	2
MULR_Q.H/W	4	2

2.2.9 向量转移指令

指令助记符	延迟	吞吐率
BNZ.B/H/W/D	2	
BZ.B/H/W/D	2	
BNZ.V	2	
BZ.V	2	

2.2.10 向量访存和数据搬运指令

指令助记符	延迟	吞吐率
CFCMSA	1	1
CTCMSA	1	1
LD	4 (Dcache Hit)	2
LDI.B/H/W/D	1	2
MOVE.V	1	2
SPLAT.B/H/W/D	3	1

指令助记符	延迟	吞吐率
SPLATI.B/H/W/D	1	2
FILL.B/H	2	1
FILL.W/D	1	1
INSERT.B/H/W/D	2	1
INSVE.B/H/W/D	1	2
COPY_S.B/H/W/D	2	1
COPY_U.B/H/W	2	1
ST		

2.2.11 向量数据混洗指令

指令助记符	延迟	吞吐率
ILVEV.B/H/W/D	1	2
ILVOD.B/H/W/D	1	2
ILVL.B/H/W/D	1	2
ILVR.B/H/W/D	1	2
PCKEV.B/H/W/D	1	2
PCKOD.B/H/W/D	1	2
SHF.B/H/W	1	2
SLD.B/H/W/D	3	1
SLDI.B/H/W/D	1	2
VSHF.B/H/W/D	1	2

3 附录 A 定点数格式定义

出于大小、成本、能效比等原因，常见 DSP 算法的实现过程中经常使用定点数。定点数是指小数点位置固定不变的一种数据表示方法。定点数与浮点数是相对立的一组概念。

定点数常用 Q 型格式来记录，具体形式为 Qm.n，其中 Q 表示当前数据的类型为定点数，m 表示此数值在二进制补码形式下整数部分的位数，n 表示二进制补码小数的位数。又因为二进制补码是带符号位的，所以一个数据用定点数形式表示需要（m+n+1）位。常见的，m 置 0，Q0.n 就简记为 Qn，表示仅有小数部分（纯小数）。

±	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}	2^{-21}	2^{-22}	2^{-23}	2^{-24}	2^{-25}	2^{-26}	2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}

图 3-1 Q31 型定点数据格式

举例如下。一个 32 位字恰好可以存储一个 Q31 型数据。最高位（最左侧位）表示符号位，其余 31 位表示小数位，采用二进制补码形式存储，如图 3.1 所示。可表示数据范围为［-1.0 ， +0.999999999…）。

依此类推， 一个 16 位半字恰好可以存储一个 Q15 型数据，符号位在最左侧，其后 15 位小数位，可表示范围为 ［-1.0, +0.9999…） 。一个 8 位字节恰好可以存储一个 Q7 型数据，符号位在最左侧，其后 7 位小数位，可表示范围为［-1.0, +0.99…）。

Q15 和 Q31 可表示数据范围有限，见图 3-2。可精确表示-1.0 但不可精确表示+1.0。 +1.0 用 0x7FFFFFFF 来近似评估。因此请注意，两个-1 相乘会产生溢出例外，因为+1 无法精确表示！饱和类指令（Saturating instructions）必须检查这种情况，并通过将结果限制为最大可表示值来防止溢出。

类型	定义	16 进制表示	十进制表示
Q15 最小值	$-2^{15}/2^{15}$	0x8000	-1.0
Q15 最大值	$(2^{15}-1)/2^{15}$	0x7FFF	0.999969482421875
Q31 最小值	$-2^{31}/2^{31}$	0x80000000	-1.0
Q31 最大值	$(2^{31}-1)/2^{31}$	0x7FFFFFFF	0.99999999953433871269226074218 75

图 3-2 定点数表示范围

对于某固定格式的定点数，我们可以借助每个比特位上的权重来计算存储值（二进制补码形式）所对应的真值。以 Q7 格式为例，见图 3-3。

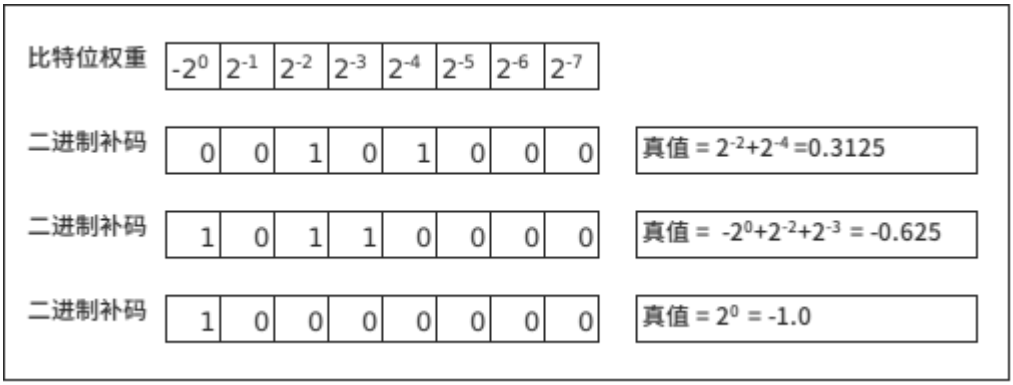

图 3-3 Q7 型定点数真值计算