Aarch64的增强型SIMD运算实验

版本：1.0



华为技术有限公司

目录

[1 课程介绍 1](#_Toc50560846)

[1.1 简介 1](#_Toc50560847)

[1.2 内容描述 1](#_Toc50560848)

[1.3 读者知识背景 1](#_Toc50560849)

[1.4 实验环境说明 1](#_Toc50560850)

[2 实验：Aarch64增强型SIMD示例程序 2](#_Toc50560851)

[2.1 实验介绍 2](#_Toc50560852)

[2.1.1 关于本实验 2](#_Toc50560853)

[2.1.2 教学目标 2](#_Toc50560854)

[2.1.3 实验内容介绍 2](#_Toc50560855)

[2.2 实验任务操作指导 5](#_Toc50560856)

[2.2.1 创建示例程序源码 5](#_Toc50560857)

[2.2.2 进行编译运行 7](#_Toc50560858)

[2.3 思考题 8](#_Toc50560859)

# 课程介绍

## 简介

本手册适用于学习ARM平台汇编课程的学生进行实验练习，完成本实验手册后，您将能掌握华为鲲鹏云服务器环境配置，并更加充分理解GNU ARM汇编代码运行环境的搭建、配置及汇编代码的编译运行。

## 内容描述

本实验指导书通过在华为鲲鹏云服务器上，编译运行SIMD(Single Instruction Multiple Data，即单指令多数据)的示例程序。完成实验操作后，读者会了解到SIMD指令的编写方式和适用范围，并掌握简单的SIMD程序编写。

## 读者知识背景

本课程为GNU ARM汇编基础课程，为了更好地掌握本书内容，阅读本手册的读者应首先具备以下基本条件：

* 具备基本的Linux命令能力。
* 了解GNU ARM汇编中的代码结构、指令格式、伪指令等基础知识。

## 实验环境说明

* 华为鲲鹏云主机、Centos7.6操作系统。
* 安装gcc7.3+版本。
* 每套实验环境可供1名学员上机操作。

# 实验：Aarch64增强型SIMD示例程序

## 实验介绍

### 关于本实验

实现ARM64架构下SIMD(Single Instruction Multiple Data，即单指令多数据)程序的编译和运行。

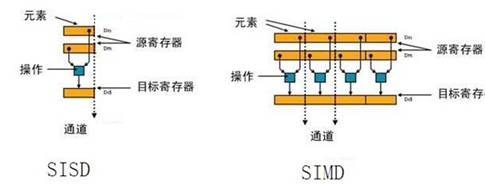
### 教学目标

掌握使用GNU ARM汇编、C语言嵌入式等方式实现ARM平台SIMD指令操作的方式。

### 实验内容介绍

Aarch64中的NEON单元专用于支持Adavanced（增强型） SIMD(Single Instruction Multiple Data)的并行运算，完全符合IEEE754双精度标准，适合多媒体数据处理等任务。

在SIMD型的CPU中，指令译码后几个执行部件同时访问内存，一次性获得所有[操作数](http://baike.baidu.com/view/420846.htm)进行运算。这个特点使SIMD特别适合于多媒体应用等数据密集型运算。SIMD与传统的SISD方式的操作原理区别下图所示：



#### A64 NEON寄存器

在ArmV8的Aarch64状态下，NEON提供了专门的32个128bit的向量寄存器，这些寄存器用于支持浮点运算和SIMD运算，记名为v0-v31。这些寄存器是独立于通用的Xn或Wn寄存器的。v0-v31的特殊用法说明：v0-v7用于传递输入参数和输出参数；v8-v15是在传递的参数数目多于8个时，需要将其低64位压栈。

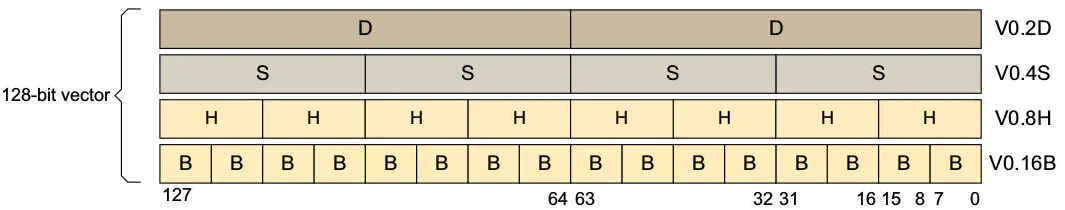
如果仅使用v0-v31这些寄存器的低64位、低32位、低16位和低8位，那么也可以记名为D0-D31（低64位）、S0-S31（低32位）、H0-H31（低16位）和B0-B31（低8位）。

在实际的使用过程中，如果需要使用这些寄存器的全部128bit，但同时也需要使用这128bit中的一部分，那么还可以在v0-v31之后添加后缀的方式来达到既使用全部又使用局部的目的，后缀的典型格式为：Vn.T（n为0-31之间的一个整数）。例如一条SIMD加法指令：ADD Vd.T, Vn.T, Vm.T。在该指令中，Vd、Vn和Vm都是Neon寄存器的名字，T是Neon寄存器的数据位宽表示方式。T的表示方式如表2-1和图2-1所示：

寄存器Vn的数据位宽表示类型

|  |  |  |  |
| --- | --- | --- | --- |
| 标识 | 位宽 | 数据类型 | 示例 |
| b | 8 bit | char | v1.b，v1.8b, v1.16b：v1寄存器的低8位，低64位，全部的128位； |
| h | 16 bit | short | v1.4h, v1.8h：v1寄存器的低64位，全部的128位 |
| s | 32 bit | int | v1.2s, v1.4s：v1寄存器的低64位，全部的128位 |
| d | 64 bit | long long | v1.1d, v1.2d：v1寄存器的低64位，全部的128位 |

例如，对2个long long类型变量进行向量加法，则指令形式为：ADD V0.2D, V0.2D, V1.2D。



Vn矢量寄存器的的数据宽度与名称

这些128bit寄存器的低64位也可以当做64位向量寄存器来使用。如果将这些128位向量寄存器的低64位或更低位作为小位宽的向量寄存器的话，那么其高位部分将被清零（除了lane插入操作之外）。

#### A64 SIMD指令修饰符

##### 前缀修饰符

在Aarch64状态下，SIMD指令中不再有“V”这种助记符前缀，但是使用S/U/F/P等四种指令数据类型的前缀修饰符，可以被添加用来说明SIMD指令的操作数是有符号、无符号、浮点数、多项式中的某一种数据类型，例如，SMAX，UMAX，FMAX指令等。如果没有这些修饰符，则表明是与数据类型无关的SIMD指令。

指令前缀Q可以指定饱和指令。饱和指令限定了各种类型的数据范围。比如加法 SQADD 和 UQADD分别表示有符号饱和加以及无符号饱和加，如果结构数据超过了最大最小界限，饱和运算会使得结果不过超过最大或最小。比如，SQADD V0.16B, V0.16B, V1.16B实现有符号饱和加功能，其三个操作数都是16个字节。

##### 后缀修饰符

Normal指令，对相同类型的数据进行操作，返回结果的数据类型与源类型相同。但是A64的SIMD还提供了两个源操作数和一个目标操作数之间长度不一致情况的指令后缀：

* 长指令，使用L作为后缀，结果数据的位数是源数据位数的两倍，例如：SADDL V0.4S, V1.4H, V2.4H。
* Wide宽指令，对一个双字数据和一个单字数据进行操作，结果将都是双字数据，使用W作为后缀，例如：SADDW V0.4S, V1.4H, V2.4S。
* Narrow指令，操作两个四字向量，得到双字向量，结果数据是源数据的一半长，使用N作为后缀，例如：SUBHN V0.4H, V1.4S, V2.4S。

后缀修饰符除了定义操作数的宽度情况之外，还将指明其它一些情况：

* 后缀P，表示矢量寄存器内部组对操作(pairwise)，即矢量寄存器中将相邻两个lane作为一组进行算法运算，并产生一个矢量结果。比如：addp v1.8B, v2.8B, v3.8B
* 后缀V，表示Across，对寄存器中跨所有lane的操作，比如：

1）addv B0, v1.8B // 将v1寄存器中的低64位中8个8位数据（跨8个B类型的lane）相加求和后，赋给v0的最低8位。

2）addv H2, v1.8B // 将v1寄存器中的低64位中8个字节（跨8个B类型的lane）依次相加求和，并将结果赋给v1第2个16位标量中。

* 指令后缀”2”，表示对源寄存器的高64位进行操作，例如：

1）smull2 v0.4S, v1.4H, v2.4H //将v1和v2高64位中每一个16位相乘，并将结果放在v0的每个32位中。

2）sqxtn2 v5.8H, v4.4S //将v4中的每个32位元素，饱和缩进到v5高64位的每个16位中。

3）sxtl2 v16.4S, v17.4H //将v17高64位中的每个16位元素，扩展到v16的每个32位元素中。

#### SIMD操作数下标(index)

NEON寄存器用于矢量运算时, 支持将16字节的NEON寄存器平均划分成若干个element或lane（车道），每个lane或element有相同类型/大小的位宽. 如v0.8h, 表示分成8个lane, 每个lane是两字节大小。Neon中含有向量功能单元，每个单元都完全实现流水化，每个时钟周期都可以开启一个新的操作。每个功能单元都可能只有单条流水线（单车道），也可以有多个并行的流水线（多车道）来并行执行相关操作。

在A64 SIMD指令中，操作数的下标(index)：表明操作数的哪一个lane被使用。比如：插入指令：INS Vd.Ts[index1], Vn.Ts[index2]

其中，Vd为目标寄存器，Vn为源寄存器，Ts指明从最低位起需要使用的lane的宽度和个数，index1和index2表明在多个lane中仅使用哪一个lane，其上限为T-1。index为0则表示最低的lane，index的最大值为Ts中表示lane的最大数目。例如Ts为b类型，则128bit的寄存器最多一共可以有16个b类型的lane；Ts为d类型，则128bit的寄存器最多一共可以有2个d类型的lane。如果Ts为4b，则index值不能超过3。

Ts的下标(index)

|  |  |  |  |
| --- | --- | --- | --- |
| Ts | lane宽度 | 下标index的总范围 | index的实际范围 |
| b | char：8bit | 0-15 | Ts为8b，则index<=7；Ts为2b，则index为0或1； |
| h | short：16bit | 0-7 | Ts为6h，则index<=5；Ts为2b，则index为0或1； |
| s | int：32bit | 0-3 | Ts为4s，则index<=3；Ts为2b，则index为0或1； |
| d | long long：64bit | 0或1 | Ts为2d，则index<=1； |

例如：

1） ld1 {v18.S}[0], [x0], x1 // 将x0地址里面的数据取32位加载到v18的最低32位，然后x0=x0+x1。

2） st1 {v0.S}[0], [x0], x2 // 将 寄存器v0的低32位数据存储到x0地址处的最低32位，然后x0=x0+x2。

#### A64 SIMD指令

A64 SIMD指令分为标量(scalar)指令和矢量(vector)指令。具体指令功能与用法参考<https://developer.arm.com/documentation/100069/0604/A64-SIMD-Vector-Instructions?lang=en>。

#### 实验案例介绍

本实验是一个4\*4的矩阵相乘的例子，乘数是4\*4矩阵的aa和bb，乘积是4\*4矩阵的cc，乘的方式是cc[i][j]=aa[i][j]\*bb[i][j]。采用内嵌汇编代码的方式，其中使用了多个simd寄存器v0-v7，也使用了ld4和st4指令。

## 实验任务操作指导

### 创建示例程序源码

以下步骤以在华为鲲鹏云服务器上执行为例。

执行以下命令，创建simd目录存放该程序的所有文件, 并进入simd目录

cd

mkdir simd

cd simd

执行以下命令，创建示例程序源码simd.c

vim simd.c

代码内容如下：

#include <stdio.h>

#include <arm\_neon.h>

static void matrix\_mul\_asm(uint16\_t \*\*aa,uint16\_t \*\*bb,uint16\_t \*\*cc)

{

uint16\_t \*a=(uint16\_t \*)aa;

uint16\_t \*b=(uint16\_t \*)bb;

uint16\_t \*c=(uint16\_t \*)cc;

\_\_asm\_\_ \_\_volatile\_\_ (

"ld4 {v0.4h-v3.4h},[%0]\n"

"ld4 {v4.4h,v5.4h,v6.4h,v7.4h},[%1]\n"

"mul v3.4h,v3.4h,v7.4h\n"

"mul v2.4h,v2.4h,v6.4h\n"

"mul v1.4h,v1.4h,v5.4h\n"

"mul v0.4h,v0.4h,v4.4h\n"

"st4 {v0.4h,v1.4h,v2.4h,v3.4h},[%2]\n"

:"+r"(a),"+r"(b),"+r"(c)

:

:"cc","memory","v0","v1","v2","v3","v4","v5","v6","v7"

);

}

int main()

{

uint16\_t aa[4][4]={

{1,2,3,4},

{5,6,7,8},

{3,6,8,1},

{2,6,7,1}

};

uint16\_t bb[4][4]={

{1,3,5,7},

{2,4,6,8},

{2,5,7,9},

{5,2,7,1}

};

uint16\_t cc[4][4]={0};

int i,j;

matrix\_mul\_asm((uint16\_t \*\*)aa,(uint16\_t \*\*)bb,(uint16\_t \*\*)cc);

for(i=0;i<4;i++)

for(j=0;j<4;j++)

printf("out is %11u \n",cc[i][j]);

return 0;

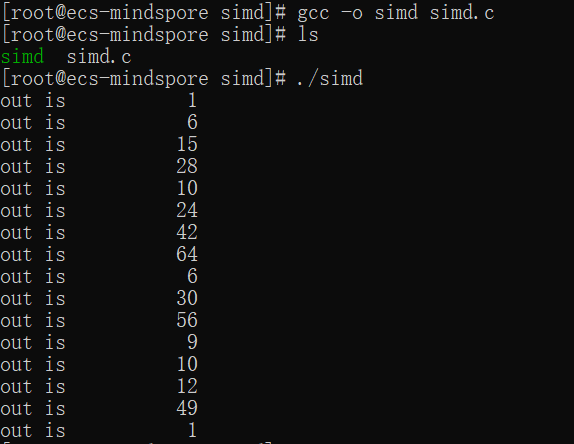
}

### 进行编译运行

保存示例源码文件，然后退出vim编辑器。在当前目录中依次执行以下命令，进行代码编译运行。

gcc -o simd simd.c

./simd



通过上述代码运行，可以看出，编写的simd程序已经在华为鲲鹏云服务器上通过编译和运行，已经成功输出结果。

## 思考题

基于上述代码，试着解决如下问题：

* 同样是实现4\*4乘法，用非SIMD的常规指令，采用内嵌汇编代码的方式实现一遍，比较一下用与不用SIMD指令的性能差异；如果是8\*8,16\*16等乘法，这种性能差异会怎样变化？
* 用NEON SIMD的intrinsic方式重新一遍。提示：Intrinsics是使用C语言的方式对NEON寄存器进行操作，因为相比于传统的使用纯汇编语言，具有可读性强，开发速度快等优势。如果需要在代码中调用NEON Intrinsics函数，需要加入头文件"arm\_neon.h"。 Intrinsic的语法可以参考<https://www.jianshu.com/p/70601b36540f>和Arm官网的内容。
* 用C代码写一遍，试着用与不用特定的gcc的neon编译选项来进行优化编译，比较一下两种方式下的性能，所用的时间测量函数参考实验2中的相关内容。