ARMv8 中的 SIMD 运算

发表于 2019-03-01 | 分类于 arm ， simd | 阅读次数：4035
本文字数： 8.1k | 阅读时长 ≈ 7 分钟

NEON 是一种压缩的 SIMD 架构，主要是给多媒体使用，结果并行计算的问题。

> NEON 是 ARMv7-A 和 ARMv7-R 引入的特性，在后面的 ARMv8-A 和 ARMv8-R 中也扩展其功能.1288bit 的向量运算

	ARMv7-A/R	ARMv8-A/R	ARMv8-A
		AArch32	AArch64
Floating-point	32-bit	16-bit*/32-bit	16-bit*/32-bit
Integer	8-bit/16-bit/32-bit	8-bit/16-bit/32-bit/64-bit	8-bit/16-bit/32-bit/64-bit

ARMv8 与 ARMv7 的区别

- 1. 与 通用寄存器 相同的助记符

CPU	通用	SIMD
ARMv7	mul, r0, r0, r1	vmul d0, d0, d1
ARMv8	mul x0, x0, x1	mul v0.u8, v0.u8, v1.u8

> **注意：在 ARMv7 中所有的 SIMD 汇编的操作码如 `mul` 的前缀都有 `v` 如 vml**

- 2.ARMv8 的寄存器是 ARMv7 的两倍
  - ARMv8 拥有 32 个 128-bit 寄存器
  - ARMv7 拥有 16 个 128-bit 寄存器

- 3. 不同的指令语法

SIMD 寄存器

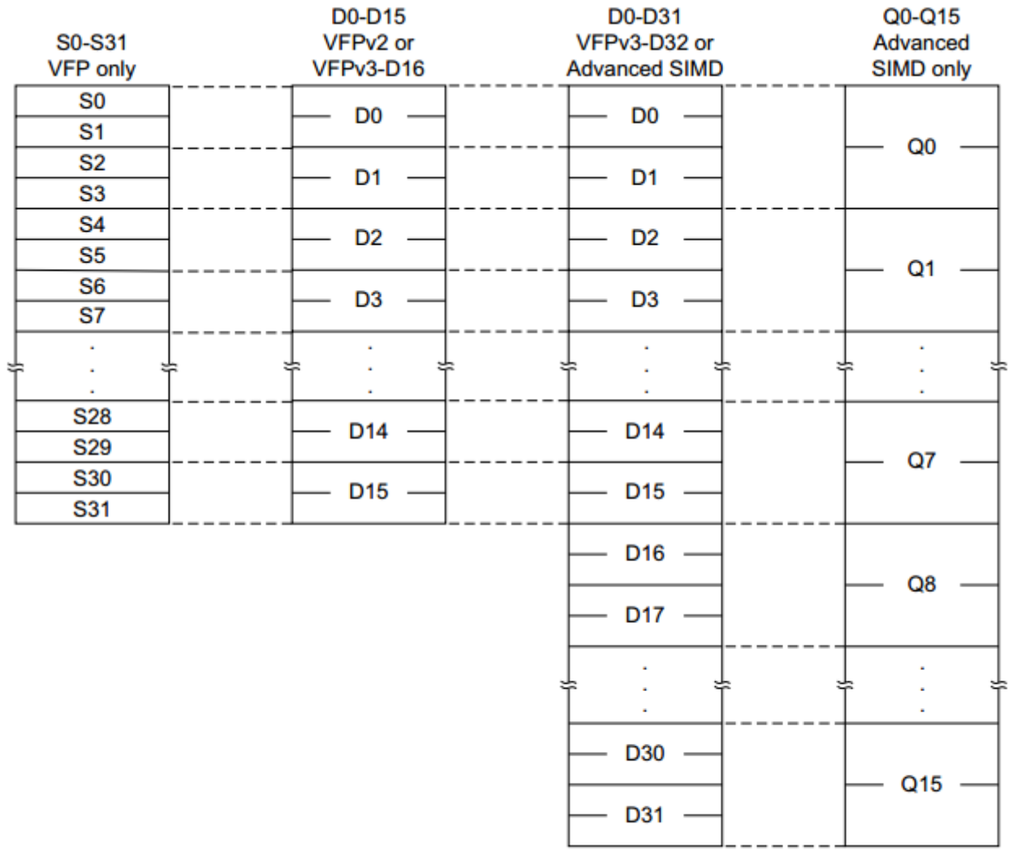


Figure 1-3 NEON and VFP register set

armv8SIMD 寄存器

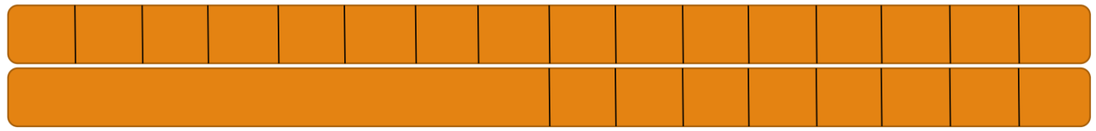
寄存器	个数	位宽	数据类型
D 寄存器（ D0-D31 ）	32 个	64-bit	双字（double word）
Q 寄存器（ Q0-Q15 ）	16 个	128-bit	四字

矢量寄存器 V0-V31：包装

armv8SIMD 寄存器标识 vx

打包 V0-V31 中的数据，方便数据操作

Example: 16 bytes or 8 bytes

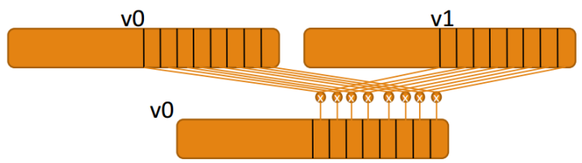


Example: 8 half-words or 4 half-words

ARMv8SIMD 寄存器打包

矢量包装

mul v0.**u8**, v0.**u8**, v1.**u8**



- v0.**8b**, v0.**16b**: 8 bytes or 16 bytes (8 bit)
- v0.**4h**, v0.**8h**: 4 half-words or 8 half-words (16 bit)
- v0.**2s**, v0.**4s**: 2 words or 4 words (32-bit)
- v0.**2d**: 2 double-words (64-bit)

ARMvc8

主要定义每一个矢量 Vn 的数据位宽

标识	位宽	数据类型	示例
b	8bit	char	v0.8b,v0.16b: 8 个 bit16 个 bit
h	16bit	short	v0.4h，v0.8h： 4 或 8 个半字（short 类型）
s	32bit	int	v0.2s，v0.4s：2 或 4 个字
d	64bit	long long	v0.2d：2 个 double word

指令语法

- <prefix> Represents data type (signed, unsigned, float, poly) **[S, U, F, P]**
- <op> Instruction mnemonic, for example **[mul]** or **[add]**
- <suffix> For special purpose functions, e.g. pairwise operations
- <T> Packing format. **[8B, 16B, 4H, 8H, 2S, 4S, 2D].** B=byte, H=halfword (16-bit), S=word (32-bit), D=doubledord (64-bit)

ARMv8SIMD 指令 op

```
1   ld4 {v0.4h-v3.4h}, [%0]
```

等同于：

```
1   ld4 {v0.4h, v1.4h, v2.4h, v3.4h}, [%0]
```

内联函数编程

NEON 内在函数在头文件 `arm_neon.h` 中定义。头文件既定义内在函数，也定义一组向量类型

> NEON 操作函数
>
> - arm_neon.h

内嵌汇编编程

```
1  asm volatile(
2      "mnemonic+operand \n\t"
3      "mnemonic+operand \n\t"
4      "mnemonic+operand \n\t"
5
6      : //output operand list  /*输出操作数列表*/
7      : //input operand list   /*输入操作数列表*/
8      : //Dirty registers etc  /*被改变资源列表*/
9  );
```

操作符 & 修饰符

```
1  asm volatile(
2          "add %0, %1, %2"
```

```
3
4           : "=r" (ret)
5           : "r" (a), "r" (b)
6           );
```

操作符	含义
r	通用寄存器
m	一个有效的内存地址
I	数据处理中的立即数
X	被修饰的操作符只能作为输出

修饰符	含义
无	只读
=	只写
+	可读可写
&	只能作为输出

传参

参数序列

```
1   asm volatile(
2           "add %0, %1, %2"
3
4           : "=r" (ret)
5           : "r" (a), "r" (b)
6           );
```

> - ret：%0，第一个参数
> - a：%1，第二个参数
> - b：%2，第三个参数

参数名

```
asm volatile(
        "add %[result], %[a], %[b]"

        : [result] "=r" (ret)
        : [a] "r" (a), [b] "r" (b)
        );
```

> 传入参数不依赖参数序列

示例

4x4 矩阵乘法

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/time.h>

#if __aarch64__
#include <arm_neon.h>
#endif

static void dump(uint16_t **x)
{
    int i, j;
    uint16_t *xx = (uint16_t *)x;

    printf("%s:\n", __func__);

    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
            printf("%3d ", *(xx + (i << 2) + j));
        }

        printf("\n");
    }
}

static void matrix_mul_c(uint16_t aa[][4], uint16_t bb[][4], uint16_t cc[][4]
{
    int i = 0, j = 0;

    printf("===> func: %s, line: %d\n", __func__, __LINE__);

    for(i = 0; i < 4; i++) {
        for(j = 0; j < 4; j++) {
```

```
            cc[i][j] = aa[i][j] * bb[i][j];
        }
    }

}

#if __aarch64__
static void matrix_mul_neon(uint16_t **aa, uint16_t **bb, uint16_t **cc)
{
    printf("===> func: %s, line: %d\n", __func__, __LINE__);
#if 1
    uint16_t (*a)[4] = (uint16_t (*)[4])aa;
    uint16_t (*b)[4] = (uint16_t (*)[4])bb;
    uint16_t (*c)[4] = (uint16_t (*)[4])cc;

    printf("aaaaaaaa\n");
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
    uint16x4_t _cc0;
    uint16x4_t _cc1;
    uint16x4_t _cc2;
    uint16x4_t _cc3;

    uint16x4_t _aa0 = vld1_u16((uint16_t*)a[0]);
    uint16x4_t _aa1 = vld1_u16((uint16_t*)a[1]);
    uint16x4_t _aa2 = vld1_u16((uint16_t*)a[2]);
    uint16x4_t _aa3 = vld1_u16((uint16_t*)a[3]);

    uint16x4_t _bb0 = vld1_u16((uint16_t*)b[0]);
    uint16x4_t _bb1 = vld1_u16((uint16_t*)b[1]);
    uint16x4_t _bb2 = vld1_u16((uint16_t*)b[2]);
    uint16x4_t _bb3 = vld1_u16((uint16_t*)b[3]);

    _cc0 = vmul_u16(_aa0, _bb0);
    _cc1 = vmul_u16(_aa1, _bb1);
    _cc2 = vmul_u16(_aa2, _bb2);
    _cc3 = vmul_u16(_aa3, _bb3);

    vst1_u16((uint16_t*)c[0], _cc0);
    vst1_u16((uint16_t*)c[1], _cc1);
    vst1_u16((uint16_t*)c[2], _cc2);
    vst1_u16((uint16_t*)c[3], _cc3);
    asm("nop");
    asm("nop");
    asm("nop");
    asm("nop");
#else
    printf("bbbbbbbb\n");
```

```
85      int i = 0;
86      uint16x4_t _aa[4], _bb[4], _cc[4];
87      uint16_t *a = (uint16_t*)aa;
88      uint16_t *b = (uint16_t*)bb;
89      uint16_t *c = (uint16_t*)cc;
90
91      for(i = 0; i < 4; i++) {
92          _aa[i] = vld1_u16(a + (i << 2));
93          _bb[i] = vld1_u16(b + (i << 2));
94          _cc[i] = vmul_u16(_aa[i], _bb[i]);
95          vst1_u16(c + (i << 2), _cc[i]);
96      }
97
98  #endif
99  }
100
101 static void matrix_mul_asm(uint16_t **aa, uint16_t **bb, uint16_t **cc)
102 {
103     printf("===> func: %s, line: %d\n", __func__, __LINE__);
104
105     uint16_t *a = (uint16_t*)aa;
106     uint16_t *b = (uint16_t*)bb;
107     uint16_t *c = (uint16_t*)cc;
108
109 #if 0
110     asm volatile(
111         "ldr d3, [%0, #0]           \n\t"
112         "ldr d2, [%0, #8]           \n\t"
113         "ldr d1, [%0, #16]          \n\t"
114         "ldr d0, [%0, #24]          \n\t"
115
116         "ldr d7, [%1, #0]           \n\t"
117         "ldr d6, [%1, #8]           \n\t"
118         "ldr d5, [%1, #16]          \n\t"
119         "ldr d4, [%1, #24]          \n\t"
120
121         "mul v3.4h, v3.4h, v7.4h    \n\t"
122         "mul v2.4h, v2.4h, v6.4h    \n\t"
123         "mul v1.4h, v1.4h, v5.4h    \n\t"
124         "mul v0.4h, v0.4h, v4.4h    \n\t"
125
126         //"add v3.4h, v3.4h, v7.4h    \n\t"
127         //"add v2.4h, v2.4h, v6.4h    \n\t"
128         //"add v1.4h, v1.4h, v5.4h    \n\t"
129         //"add v0.4h, v0.4h, v4.4h    \n\t"
130
131         "str d3, [%2,#0]            \n\t"
132         "str d2, [%2,#8]            \n\t"
133         "str d1, [%2,#16]           \n\t"
134         "str d0, [%2,#24]           \n\t"
```

```
135
136         : "+r"(a),   //%0
137           "+r"(b),   //%1
138           "+r"(c)    //%2
139         :
140         : "cc", "memory", "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7"
141     );
142 #else
143     // test, OK
144     asm("nop");
145     asm("nop");
146     asm("nop");
147     asm("nop");
148     asm("nop");
149     asm volatile(
150         //"ld4 {v0.4h, v1.4h, v2.4h, v3.4h}, [%0] \n\t"
151         "ld4 {v0.4h-v3.4h}, [%0]                \n\t"
152         "ld4 {v4.4h, v5.4h, v6.4h, v7.4h}, [%1] \n\t"
153 
154         "mul v3.4h, v3.4h, v7.4h                \n\t"
155         "mul v2.4h, v2.4h, v6.4h                \n\t"
156         "mul v1.4h, v1.4h, v5.4h                \n\t"
157         "mul v0.4h, v0.4h, v4.4h                \n\t"
158 
159         "st4 {v0.4h, v1.4h, v2.4h, v3.4h}, [%2] \n\t"
160 
161         : "+r"(a),   //%0
162           "+r"(b),   //%1
163           "+r"(c)    //%2
164         :
165         : "cc", "memory", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "v7"
166     );
167     asm("nop");
168     asm("nop");
169     asm("nop");
170     asm("nop");
171     asm("nop");
172 #endif
173 }
174 #endif
175 
176 int main(int argc, const char *argv[])
177 {
178     uint16_t aa[4][4] = {
179         {1, 2, 3, 4},
180         {5, 6, 7, 8},
181         {3, 6, 8, 1},
182         {2, 6, 7, 1}
183     };
184 
```

```
    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;
    struct timeval tv;
    long long start_us = 0, end_us = 0;

    dump((uint16_t **)aa);
    dump((uint16_t **)bb);
    dump((uint16_t **)cc);

    /* ******** C **********/
    gettimeofday(&tv, NULL);
    start_us = tv.tv_sec + tv.tv_usec;

    matrix_mul_c(aa, bb, cc);

    gettimeofday(&tv, NULL);
    end_us = tv.tv_sec + tv.tv_usec;
    printf("aa[][]*bb[][] C time %lld us\n", end_us - start_us);
    dump((uint16_t **)cc);

#if __aarch64__
    /* ******** NEON **********/
    memset(cc, 0, sizeof(uint16_t) * 4 * 4);
    gettimeofday(&tv, NULL);
    start_us = tv.tv_sec + tv.tv_usec;

    matrix_mul_neon((uint16_t **)aa, (uint16_t **)bb, (uint16_t **)cc);

    gettimeofday(&tv, NULL);
    end_us = tv.tv_sec + tv.tv_usec;
    printf("aa[][]*bb[][] neon time %lld us\n", end_us - start_us);
    dump((uint16_t **)cc);

    /* ******** asm **********/
    memset(cc, 0, sizeof(uint16_t) * 4 * 4);
    gettimeofday(&tv, NULL);
    start_us = tv.tv_sec + tv.tv_usec;

    matrix_mul_asm((uint16_t **)aa, (uint16_t **)bb, (uint16_t **)cc);

    gettimeofday(&tv, NULL);
    end_us = tv.tv_sec + tv.tv_usec;
    printf("aa[][]*bb[][] asm time %lld us\n", end_us - start_us);
```

```
235      dump((uint16_t **)cc);
236  #endif
237
238      return 0;
239  }
```

```
1  aarch64-linux-gcc -O3  matrix_4x4_mul.c
```

> gcc –march=armv8-a [input file] -o [output file]

8x8 矩阵乘法

```
1  static void matrix_mul_asm(uint16_t **aa, uint16_t **bb, uint16_t **cc)
2  {
3      printf("===> func: %s, line: %d\n", __func__, __LINE__);
4
5      uint16_t *a = (uint16_t*)aa;
6      uint16_t *b = (uint16_t*)bb;
7      uint16_t *c = (uint16_t*)cc;
8
9      asm volatile(
10         "ld4 {v0.8h, v1.8h, v2.8h, v3.8h}, [%0]     \n\t"
11         "ld4 {v8.8h, v9.8h, v10.8h, v11.8h}, [%1]   \n\t"
12
13         "mul v0.8h, v0.8h, v8.8h                    \n\t"
14         "mul v1.8h, v1.8h, v9.8h                    \n\t"
15         "mul v2.8h, v2.8h, v10.8h                   \n\t"
16         "mul v3.8h, v3.8h, v11.8h                   \n\t"
17
18         "st4 {v0.8h, v1.8h, v2.8h, v3.8h}, [%2]     \n\t"
19
20
21         "add x1, %0, #64                            \n\t"
22         "add x2, %1, #64                            \n\t"
23         "add x3, %2, #64                            \n\t"
24
25         //"ld4 {v4.8h-v7.8h}, [x1]                    \n\t"
26         "ld4 {v4.8h, v5.8h, v6.8h, v7.8h}, [x1]     \n\t"
27         "ld4 {v12.8h, v13.8h, v14.8h, v15.8h}, [x2] \n\t"
28
29         "mul v4.8h, v4.8h, v12.8h                   \n\t"
30         "mul v5.8h, v5.8h, v13.8h                   \n\t"
31         "mul v6.8h, v6.8h, v14.8h                   \n\t"
32         "mul v7.8h, v7.8h, v15.8h                   \n\t"
33
34         "st4 {v4.8h, v5.8h, v6.8h, v7.8h}, [x3]     \n\t"
35
```

```
36          : "+r"(a),   //%0
37            "+r"(b),   //%1
38            "+r"(c)    //%2
39          :
40          : "cc", "memory", "x1", "x2", "x3", "v0", "v1", "v2", "v3", "v4", "v5'
41              "v8", "v9", "v10", "v11", "v12", "v13", "v14", "v15"
42        );
43  }
```

> 内嵌汇编实现方式 8x8

参考

- ARMv8 Neon Programming
- Introducing NEON
- Coding for NEON - Part 1: Load and Stores
- Coding for NEON - Part 2: Dealing With Leftovers
- Coding for NEON - Part 3: Matrix Multiplication
- Coding for NEON - Part 4: Shifting Left and Right
- Coding for NEON - Part 5: Rearranging Vectors
- ARM® Cortex®-A72 MPCore Processor Technical Reference Manual

相关文章

- X86 平台下的 SIMD 运算
- AVX VMOVDQA slower than two SSE MOVDQA?

simd

ARMv8-aarch64 寄存器和指令集

ARM64 基本的汇编语法