RealView® 编译工具

3.1 版

NEON™ 向量化编译器指南

ARM®

ARM DUI 0350AC

RealView 编译工具
NEON 向量化编译器指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	更改
2007 年 3 月	A	非保密	RVDS 3.1 版 v3.1

所有权声明

保密状态

本文档的内容是非保密的。根据 ARM 与 ARM 将本文档交予的参与方的协议条款，使用、复制和公开本文档内容的权利可能会受到许可限制的制约。

产品状态

本文档的信息是开发的产品的最新信息。

网址

`http://www.arm.com`

目录
NEON Vectorizing Compiler
NEON 向量化编译器指南

前言

本前言介绍 *RealView 编译工具 NEON Vectorizing Compiler*。它包含以下几节:

关于本手册

本手册介绍 NEON™ 向量化编译器，并说明如何利用自动向量化功能。

适用对象

本手册是为所有使用 NEON Vectorizing Compiler针对带有 NEON 单元的 ARM 处理器编写应用程序的开发者编写的。前提是您是一位经验丰富的软件开发人员。请参阅 *RealView 编译工具要点指南*以概要了解随 RVCT 提供的 ARM 开发工具。

使用本手册

本手册由以下章节组成：

第 1 章 *简介*

本章简要介绍 NEON Vectorizing Compiler。

第 2 章 *NEON 向量化功能*

本章介绍 NEON Vectorizing Compiler支持的命令行选项。

第 3 章 *使用 NEON 向量化编译器*

本章提供有关 NEON Vectorizing Compiler的指导信息。本章介绍 NEON 单元，并说明如何利用自动向量化功能。

本手册假定 ARM 软件安装在缺省位置。例如，在 Windows 上，这可能是 `卷:\Program Files\ARM`。引用路径名时，假定安装位置为 *`install_directory`*。例如，*`install_directory`*`\Documentation\...`。如果将 ARM 软件安装在其他位置，则可能需要更改此位置。

印刷约定

本手册使用了以下印刷约定：

`monospace` 表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

`mono`space 表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

`monospace italic`

表示此处的命令和函数的自变量可用特定值代替。

`等宽粗体` 表示在示例代码以外使用的语言关键字。

斜体	突出显示重要注释、介绍特殊术语以及表示内部交叉引用和引文。
粗体	突出显示界面元素，如菜单名称。有时候也用在描述性列表中以示强调，以及表示 ARM 处理器信号名称。

更多参考出版物

本部分列出了 ARM 公司和第三方发布的、可提供有关 ARM 系列处理器开发代码的附加信息的出版物。

ARM 公司将定期对其文档进行更新和更正。有关最新勘误表、附录以及 ARM 常见问题 (FAQ)，请访问 http://www.arm.com。

ARM 公司出版物

本手册包含专用于 RVCT 所需的开发工具的附加参考信息。该套件中包含的其他出版物有:

- *《RVCT 编译器用户指南》* (ARM DUI 0205)
- *《RVCT 编译器参考指南》* (ARM DUI 0348)
- *《RVCT 库和浮点支持指南》* (ARM DUI 0349)
- *《RVCT 要点指南》* (ARM DUI 0202)
- *《RVCT 链接器和实用程序指南》* (ARM DUI 0206)
- *《RVCT 汇编程序指南》* (ARM DUI 0204)
- *《RVCT 开发指南》* (ARM DUI 0203)
- *《RealView Development Suite 词汇表》* (ARM DUI 0324)。

有关基本标准、软件接口以及 ARM 支持的标准的完整信息，请参阅 *install_directory*\Documentation\Specifications\...。

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档:

- *《ARM7-M 体系结构参考手册》* (ARM DDI 0403)
- *《ARM 体系结构参考手册》，ARMv7-A 和 ARMv7-R 版* (ARM DDI 0406)
- *《ARM 体系结构参考手册高级 SIMD 扩展和 VFPv3 补充》* (ARM DDI 0268)
- 您的硬件设备的 ARM 数据表或技术参考手册。

反馈

ARM 欢迎您提出关于 RealView 编译工具 和文档的反馈信息。

关于 RealView 编译工具 的反馈

如果您有关于 RVCT 的任何问题，请与您的供应商联系。为便于他们快速提供有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用版本的详细信息
- 您运行的平台的详细信息，如硬件平台、操作系统类型和版本
- 能重现问题的一小段独立代码示例
- 您预期发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 能说明问题的示例输出
- 工具的版本字符串，包括版本号和内部版本号。

关于本手册的反馈

如果您发现本手册有任何错误或遗漏之处，请发送电子邮件到 errata@arm.com，并提供：

- 文档标题
- 文档编号
- 您有疑问的页码
- 问题的简要说明。

我们还欢迎您对需要增加和改进之处提出建议。

第 1 章
简介

NEON 是对 ARM 高级单指令、多数据 (SIMD) 扩展的实现。

本章介绍 NEON Vectorizing Compiler，包含以下内容:

1.1 关于 NEON Vectorizing Compiler

RVCT 提供了 `armcc --vectorize`，它是 ARM 编译器的向量化版本，它针对具有 NEON 单元的 ARM 处理器，如 Cortex-A8。

向量化意味着编译器直接从 C 或 C++ 代码生成 NEON 向量指令。编译器可向量化常规 C 和 C++ 运算（如 +），和一些来自 `dspfns.h` 头文件的 ITU 内在函数。

除了编译器向量化，RVCT 还支持 NEON 内在函数作为中间步骤，在向量化编译器和编写汇编程序代码之间生成 SIMD 代码。有关详细信息，请参阅 *《RVCT 编译器参考指南》*中的附录 E *使用 NEON 支持*。

注意

NEON 向量化编译器要求在使用前安装自己单独的 *FLEXnet* 许可证。此许可证与 RVDS 产品所需的许可证无关。如果您试图将 RVDS 许可证用于 NEON 向量化编译器，则 NEON 编译器会生成错误消息。

有关许可的详细信息，请参阅 *《ARM FLEXnet 许可证管理指南》*(ARM DUI 0209)。

第 2 章
NEON 向量化功能

本章介绍 NEON 向量化编译器 armcc --vectorize 支持的命令行选项、关键字和功能。它包括以下内容:

2.1 命令行选项

本节介绍除了 ARM 编译器所支持的选项之外，NEON 向量化编译器还支持的命令行选项。

2.1.1 --diag_suppress=optimizations

使用此选项将禁止高级优化的诊断消息。

缺省值

缺省情况下，优化消息具有“备注”严重性。指定 --diag_suppress=optimizations 会禁止显示优化消息。

注意

使用 --remarks 选项可查看具有“备注”严重性的优化消息。

用法

在优化级别 -O3 执行编译时，编译器执行某些高级向量和标量优化，如展开循环。使用此选项可以禁止与这些高级优化有关的诊断消息。

示例

```
/*int*/ factorial(int n)
{
    int result=1;

    while (n > 0)
        result *= n--;

    return result;
}
```

使用 -O3 -Otime --remarks --diag_suppress=optimizations 选项编译此代码，可禁止优化消息。

另请参阅

- 第 2-3 页的*--diag_warning=optimizations*
- *编译器参考指南*中的第 2-29 页的*--diag_suppress=tag[,tag,...]*
- *编译器参考指南*中的第 2-30 页的*--diag_warning=tag[,tag,...]*
- *编译器参考指南*中的第 2-67 页的*-Onum*
- 第 2-70 页的*-Otime*中的*编译器参考指南*

2.1.2 --diag_warning=optimizations

此选项可将高级优化诊断消息设置为警告级严重性。

缺省值

缺省情况下，优化消息具有“备注”严重性。

用法

在优化级别 `-O3 -Otime` 执行编译时，编译器执行某些高级向量和标量优化，如展开循环。使用此选项可显示与这些高级优化相关的诊断消息。

示例

```
int factorial(int n)
{
    int result=1;

    while (n > 0)
        result *= n--;

    return result;
}
```

使用 `--vectorize --cpu=Cortex-A8 -O3 -Otime --diag_warning=optimizations` 选项编译此代码，将生成优化警告消息。

另请参阅

- 第 2-2 页的*--diag_suppress=optimizations*
- *编译器参考指南*中的第 2-67 页的*-Onum*
- *编译器参考指南*中的第 2-70 页的*-Otime*
- *编译器参考指南*中的第 2-29 页的*--diag_suppress=tag[,tag,...]*
- *编译器参考指南*中的第 2-30 页的*--diag_warning=tag[,tag,...]*

2.1.3 --[no_]vectorize

使用此选项可允许或禁止直接从 C 或 C++ 代码生成 NEON 向量指令。

缺省值

缺省为 --no_vectorize。

限制

必须为要向量化的循环指定以下选项：

--cpu=*name* 目标处理器必须具有 NEON 功能。

-O*time* 缩短执行时间的优化类型。

-O*num* 优化级别。必须使用以下选项之一：

- -O2 高度优化。这是缺省设置。
- -O3 最大优化。

注意

NEON 是对 ARM 高级*单指令、多数据* (SIMD) 扩展的实现。

需要单独的 *FLEXnet* 许可证才能启用向量化。

示例

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

另请参阅

- *编译器参考指南*中的第2-16 页的*--cpu=name*
- *编译器参考指南*中的第2-67 页的*-Onum*
- *编译器参考指南*中的第2-70 页的*-Otime*

2.2 关键字

本节介绍除了 ARM 编译器所支持的关键字之外，NEON 向量化编译器还支持的 C/C++ 关键字。

2.2.1 restrict

restrict 关键字是 C99 功能，用于确保不同的对象指针类型和函数参数数组不会指向重叠的内存区域。因此，编译器可以执行优化，否则优化会因可能的别名而被禁止。

要在 C90 或 C++ 中启用关键字 **restrict**，必须指定 `--restrict` 选项。

支持关键字 **__restrict** 和 **__restrict__** 作为 **restrict** 的同义词，并且无论是否指定了 `--restrict` 选项，始终可以使用这些关键字。

示例

```
void func (int *restrict pa, int *restrict pb, int x)
{
    int i;
    for (i=0; i<100; i++)
        *(pa + i) = *(pb + i) + x;
}
```

另请参阅

- 第3-10 页的*使用指针*
- *编译器参考指南*中的第3-9 页的*restrict*

2.3 编译指示

本节介绍除了 ARM 编译器所支持的编译指示之外，NEON 向量化编译器还支持的 ARM 特定的编译指示。

2.3.1 #pragma unroll [(*n*)]

此编译指示指示编译器通过 *n* 次迭代展开循环。

注意

使用 #pragma unroll [(*n*)] 可以展开向量化和非向量化的循环。即，#pragma unroll [(*n*)] 适用于 --vectorize 和 --no_vectorize。

语法

```
#pragma unroll
#pragma unroll (n)
```

其中：

n 是一个可选值，用于指示要展开的迭代次数。

缺省值

如果没有指定 *n* 的值，则编译器假定为 #pragma unroll (4)。

用法

使用 -O3 -Otime 进行编译时，如果编译器认为展开循环比较有利，则会自动将其展开。可以使用此编译指示请求编译器展开未自动展开的循环。

注意

仅当有证据表明（例如，从 --diag_warning=optimizations 中），编译器本身没有以最优方式展开循环时，才应使用此 #pragma。

限制

只能在 **for** 循环、**while** 循环或 **do** ... **while** 循环的紧前面使用 #pragma unroll [(*n*)]。

示例

```
void matrix_multiply(float ** __restrict dest, float ** __restrict src1,
    float ** __restrict src2, unsigned int n)
{
    unsigned int i, j, k;

    for (i = 0; i < n; i++)
    {
        for (k = 0; k < n; k++)
        {
            float sum = 0.0f;
            /* #pragma unroll */
            for(j = 0; j < n; j++)
                sum += src1[i][j] * src2[j][k];
            dest[i][k] = sum;
        }
    }
}
```

在此示例中，编译器没有正常完成其循环分析，因为 src2 是以 src2[j][k] 进行索引的，而循环是按相反顺序嵌套的，即，j 位于 k 内部。如果在示例中取消 #pragma unroll 注释，编译器将继续展开循环四次。

如果目的是计算大小不是 4 的倍数的矩阵（例如 *n* * *n* 矩阵），则可以改用 #pragma unroll (*m*)，其中 *m* 是某个值，以使 *n* 是 *m* 的整数倍数。

另请参阅

- 第 2-3 页的*--diag_warning=optimizations*
- *#pragma unroll_completely*
- 第 2-4 页的*--[no_]vectorize*
- *编译器参考指南*中的第 2-67 页的*-Onum*
- *编译器参考指南*中的第 2-70 页的*-Otime*
- *编译器用户指南*中的第 4-4 页的*优化循环*。

2.3.2 #pragma unroll_completely

此编译指示指示编译器完全展开循环。仅当编译器可以确定循环包含的迭代次数时，它才有效。

注意

使用 #pragma unroll_completely 可以展开向量化和非向量化循环。即，#pragma unroll_completely 适用于 --no_vectorize 和 --vectorize。

用法

使用 -O3 -Otime 进行编译时，如果编译器认为展开循环比较有利，则会自动将其展开。可以使用此编译指示请求编译器完全展开未自动完全展开的循环。

注意

仅当有证据表明（例如，从 --diag_warning=optimizations 中），编译器本身没有以最优方式展开循环时，才应使用此 #pragma。

限制

只能在 **for** 循环、**while** 循环或 **do** ... **while** 循环的紧前面使用 #pragma unroll_completely。

在外部循环中使用 #pragma unroll_completely 可防止向量化。另一方面，在某些情况下，在内部循环中使用 #pragma unroll_completely 可能会有所帮助。

另请参阅

- 第2-3 页的*--diag_warning=optimizations*
- 第2-4 页的*--[no_]vectorize*
- 第2-6 页的*#pragma unroll [(n)]*
- *编译器参考指南*中的第2-67 页的*-Onum*
- *编译器参考指南*中的第2-70 页的*-Otime*
- *编译器用户指南*中的第4-4 页的*优化循环*

2.4 ETSI 基本运算

RVCT 支持原始 ETSI 基本运算系列（如 ETSI G.729 建议*《使用共轭结构代数码激励线性预测 (CS-ACELP) 的 8 Kb/s 语音编码》*所述）。有关运算的完整列表的详细信息，请参阅*编译器参考指南*中的第4-100 页的*ETSI 基本运算*。

表2-1 显示了可以进行向量化的运算列表。若要在自己的代码中使用 ETSI 基本运算，请包含头文件 `dspfns.h`。

表2-1 RVCT 3.1 中支持的 ETSI 基本运算

内在函数				
abs_s	L_sub	L_mult	mult_r	shl
extract_h	L_deposit_h	L_negate	negate	shr
extract_l	L_deposit_l	L_shl	norm_s	shr_r
L_abs	L_mac	L_shr	norm_l	sub
L_add	L_msu	mult	round	

第 3 章
使用 NEON 向量化编译器

本章介绍 NEON 单元，并说明如何利用自动向量化功能。它包含以下几节：

3.1 NEON 单元

NEON 单元提供 32 个向量寄存器，每个寄存器可保存 16 字节的信息。然后，这些 16 字节寄存器可以在 NEON 单元中进行并行运算。例如，在一个向量相加指令中，您可以将 8 个 16 位整数加上另外 8 个 16 位整数，以得到 8 个 16 位的结果。

NEON 单元支持 8 位、16 位、32 位整数运算和部分 64 位运算，以及 32 位浮点运算。

注意

向量浮点运算仅当 VFP 协处理器在 RunFast 模式下运行时才会执行。您必须使用 `--fpmode fast` 进行编译，才能向量化浮点代码。

NEON 单元被归类为向量 SIMD 单元，可使用一条指令在一个向量寄存器中对多个元素进行运算。

例如，数组 A 是一个有 8 个元素的 16 位整数数组。

表3-1 数组 A

1	2	3	4	5	6	7	8

数组 B 也有这 8 个元素:

表3-2 数组 B

80	70	60	50	40	30	20	10

若要将这些数组相加，提取每个向量放入向量寄存器中，并使用一个向量 SIMD 指令来得出结果。

表3-3 结果

81	72	63	54	45	36	27	18

3.2 编写用于 NEON 的代码

本节概要介绍可用于编写 NEON 单元代码的方式。可以通过几种方式来获得在 NEON 上运行的代码：

- 使用汇编语言、使用 C 中的嵌入式汇编语言以及直接使用 NEON 指令来编写代码。
- 使用具有 NEON C 语言扩展的 C 或 C++ 编写代码。
- 调用已优化为使用 NEON 指令的库例程。
- 使用自动向量化功能来获得针对 NEON 向量化的循环。

3.2.1 NEON C 扩展

NEON C 扩展是由 ARM 定义的一组新的数据类型和内在函数，用于从 C 访问 NEON 单元。大部分向量函数直接映射到 NEON 单元中提供的向量指令，并且由 NEON 增强的 ARM C 编译器进行内联编译。通过使用这些扩展，性能可以达到 C 语言的级别，这可以与通过汇编语言编码达到的性能相媲美。

有关详细信息，请参阅*《RVCT 编译器参考指南》*中的附录 E *使用 NEON 支持*。

3.2.2 自动向量化

通过使用可向量化的循环进行编码（而不是使用显式的 NEON 指令编写代码），可以保留处理器之间的代码可移植性。这样可以轻松达到与手写编码向量化类似的性能级别。

示例 3-1 显示了调用自动向量化所需的命令行选项。

示例 3-1 自动向量化

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

当使用向量化为改善性能进行优化时，启用和禁用向量化选项提供了可与标量代码相比的计时。有关详细信息，请参阅第 2-4 页的*--[no_]vectorize*。

注意

您还可以使用 `-O2 -Otime` 进行编译，但是这样不会得到最佳代码性能。

3.2.3 性能目标

大部分应用程序需要优化，以获得最佳向量化性能。开销始终会有一些，因此理论上是不可能达到最佳性能的。例如，NEON 单元可以同时处理四个单精度浮点数。这意味着，理论上浮点应用程序的最佳性能应该是原始标量非向量化代码的四倍。在典型的开销下，整个浮点应用程序的合理目标是比标量代码的性能提高 50%。对于不可完全向量化的大型应用程序，比标量代码的性能提高 25% 是比较合理的目标。

有关详细信息，请参阅第3-7 页的*改善性能*。

3.3 使用自动向量化

本节概要介绍自动向量化，还介绍影响向量化进程和生成代码性能的因素。

3.3.1 自动向量化概述

自动向量化涉及对您代码中循环的高级分析。这是将大部分典型代码映射到 NEON 单元功能的最有效的方法。对于大部分代码，小规模使用算法相关的并行处理所得到的收获与自动分析的开销相比是微不足道的。鉴于此原因，NEON 单元专门针对基于简单循环的并行处理。

向量化的执行方式确保优化代码得出的结果与非向量化代码相同。在某些情况下，不执行循环的向量化，以避免出现错误结果的可能性。这可能会导致代码不是最优，您可能需要手动优化代码，以使其更加适合自动向量化。有关详细信息，请参阅第3-7 页的*改善性能*。

3.3.2 向量化概念

本节介绍在考虑代码向量化时的一些常用概念。

数据引用

代码中的数据引用可以分为以下三种类型：

标量 是在所有循环迭代过程中均保持不变的单一位置。

索引 是随着每次循环而增加一个常数的整数。

向量 连续元素间具有常数跨度的内存位置范围。

第3-5 页的示例 3-2 显示了循环中变量的分类：

i、j	索引变量
a、b	向量
x	标量

示例 3-2 可向量化循环的类别

```
float *a, *b;
int i, j, n;
...
for (i = 0; i < n; i++)
{
```

```
    *(a+j) = x +b[i];
    j += 2;
};
```

将变量分类可确保循环的向量化提供与标量版本相同的结果。如果循环中存在计算未来迭代的数据相关性，则循环不能直接向量化。请参阅第3-7 页的*数据相关性*。

跨度模式和数据访问

循环中数据访问的跨度模式是一种对顺序循环迭代间数据元素的访问模式。例如，线性访问数组中每个元素的循环的跨度为 1。另一个例子是，访问所使用的每个元素间具有常数偏移量的数组的循环称为具有常数跨度。

3.3.3 影响向量化性能的因素

自动向量化进程和生成代码的性能受以下因素影响：

组织循环的方式

要获得最佳性能，循环嵌套中最内层的循环必须访问跨度为 1 的数组。

组织数据的方式

例如，包含数据数组的单个结构比结构数组更有效。数据类型还指示可以在 NEON 寄存器中保存的数据元素的数量，以及因此可并行执行的运算数量。

循环的迭代计数

通常较长的迭代计数比较好，因为循环的开销可以分摊在更多迭代中。较小的迭代计数（如两个或三个元素）可以通过非向量指令更快地进行处理。访问数以万计数组元素的特别长的循环会超出高速缓存的大小，并妨碍数据重用。

数组的数据类型

例如，在使用双精度浮点数组时，NEON 不会改善性能。

内存层次结构的使用

相对而言，大部分当前的处理器在内存带宽和处理器容量之间都是不平衡的。例如，对从主内存检索的大型数据集执行相对较少的算术运算时会受到系统内存带宽的限制。

3.4 改善性能

大部分应用程序要求程序员执行一些优化，以获得最佳 NEON 结果。本节介绍不同类型的循环。它解释了向量化如何对某些循环起作用，而对其他循环不起作用。它还解释了可以如何修改代码以使向量化代码获得最佳性能。

3.4.1 一般性能问题

使用命令行选项 `-O3` 和 `-Otime` 可确保除了向量化的好处之外，代码还可获得显著的性能优势。

优化性能时，您必须考虑到高级算法结构、数据元素大小、数组配置、严格的迭代循环、缩减运算和数据相关性问题。性能优化要求了解程序中最花费时间之处。若要获得最佳性能优势，可能需要使用符合实际条件的代码分析及确定代码基准。

自动向量化经常受到任何之前手动代码优化的妨碍，例如，在源代码中手动循环展开或复杂的数组访问。若要获得最佳结果，最好的办法是使用简单循环编写代码，从而使编译器执行全部优化。对于人工优化的旧代码，可能根据原始算法使用简单循环重新编写关键部分会更容易。删除手动优化可能会妨碍自动向量化。

有关详细信息，请参阅：

- 第 2-4 页的 *--[no_]vectorize*
- *《RVCT 编译器参考指南》*中的第 2-67 页的 *-Onum*
- *《RVCT 编译器参考指南》*中的第 2-70 页的 *-Otime*

3.4.2 数据相关性

循环中来自一个迭代的结果如果反馈到同一循环的未来迭代，则这种循环存在数据相关性冲突。冲突的值可能是数组元素或标量，如累加求和。

包含数据相关性冲突的循环可能未完全优化。要检测与数组和/或指针有关的数据相关性，要求广泛分析在每个循环嵌套中使用的数组，并对在循环中使用和存储的数组，检查对其每个维度元素的访问的偏移量和跨度。如果在循环的不同迭代中存在数组使用和存储重叠的可能性，则会出现数据相关性问题。如果运算的向量顺序会更改结果，则循环无法安全地向量化。在这类情况下，编译器检测该问题，并使循环保持原始格式或执行循环的部分向量化。在您的代码中必须避免这种类型的数据相关性，以获得最佳性能。

在示例 3-3 显示的循环中，对循环顶部的 `a[i-2]` 的引用与循环底部的存入 `a[i]` 相冲突。对此循环执行的向量化会得出不同的结果，因此该循环保持原始格式。

示例 3-3 不可向量化的数据相关性

```
float a[99], b[99], t;
int i;
for (i = 3; i < 99; i++)
{
    t = a[i-1] + a[i-2];
    b[i] = t + 3.0 + a[i];
    a[i] = sqrt(b[i]) - 5.0;
};
```

来自其他数组下标的信息用作相关性分析的一部分。示例 3-4 中的循环进行了向量化，这是因为对数组 a 引用的非向量下标永远不可能相等，因为 n 不等于 n+1，因此不会提供迭代间的反馈。对数组 a 的引用使用两个不同的数组，因此不会共享数据。

示例 3-4 可向量化的数据相关性

```
float a[99][99], b[99][99], c[99];
int i, n, m;
...
for (i = 1; i < m; i++) a[n][i] = a[n+1][i-1] * b[i] + c[i];
```

3.4.3 标量变量

在 NEON 循环中使用但未设置的标量变量，会复制到向量寄存器的每个位置和向量计算中使用的每个结果中。

设置后在循环中使用的标量*升级*为向量。这些变量通常保存循环中的临时标量值，现在需要保存临时向量值。在第3-8 页的示例 3-5 中，x 是一个*已使用的*标量，y 是一个*升级的*标量。

示例 3-5 可向量化的循环

```
float a[99], b[99], x, y;
int i, n;
...
for (i = 0; i < n; i++)
{
```

```
    y = x + b[i];
    a[i] = y + 1/y;
};
```

使用后在循环中设置的标量称为*传递*标量。这些变量对于向量化是个问题，因为在一轮循环中计算的值将传递到下一轮循环。在 示例 3-6 中，x 是一个传递标量。

示例 3-6 不可向量化的循环

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = x + b[i];
    x = a[i] + 1/x;
};
```

缩减运算

循环中一种特殊类别的标量用途是缩减运算。这种类别包括将值的向量缩减为标量结果。最常见的缩减是将向量的所有元素相加。其他缩减包括：两个向量的标量积、向量中的最大值、向量中的最小值、所有向量元素的积和向量中最大或最小元素的位置。

第3-9 页的示例 3-7 显示了标量积缩减，其中 x 是缩减标量。

示例 3-7 标量积缩减

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++) x += a[i] * b[i];
```

缩减运算有必要向量化，因为它们经常出现。通常，通过创建部分缩减的向量，然后将其缩减为最终结果标量的方式来向量化缩减运算。

3.4.4 使用指针

对于向量化来说，通常使用数组比指针更好。如果编译器能够确定循环是安全的，则可以向量化包含指针的循环。循环中的数组引用和指针引用都要分析，以查看是否存在任何对内存的向量访问。在某些情况下，编译器创建运行时测试，并根据测试结果决定执行循环的向量版本还是标量版本。

通常函数自变量是作为指针传递的。如果将多个指针变量传递给函数，则可能出现指向对内存的重叠部分。通常情况下，在运行时事实并非如此，而是编译器始终遵循安全的方法并避免优化包含同时出现在赋值运算符左侧和右侧的指针的循环。例如，考虑示例 3-8 中的函数。

示例 3-8 不可向量化的指针

```
void func (int *pa, int *pb, int x)
{
    for (i = 0; i < 100; i++) *(pa + i) = *(pb + i) + x;
};
```

在此示例中，如果 pa 和 pb 在内存中的重叠方式会导致来自一轮循环的结果反馈到后续循环，则该循环的向量化会产生错误结果。如果使用以下自变量调用函数，则向量化可能不明确：

```
int *a;

func (a, a-1);
```

编译器执行运行时测试，以查看是否出现指针别名。如果未出现指针别名，则执行代码的向量化版本。如果出现指针别名，则执行原始的非向量化代码。这会导致运行时效率和代码大小方面的少量开销。

在实际操作中，很少存在函数自变量的原因引起的数据相关性。除了向量化的问题之外，传递重叠指针的程序很难了解和调试。

有关详细信息，请参阅第2-5 页的*restrict*。

间接寻址

当值的向量访问数组时，发生间接寻址。如果要从内存获取数组，则该运算称为*集中*。如果数组要存储在内存中，则该运算称为*分散*。在示例 3-9 中，a 进行分散，b 进行聚合。

示例 3-9 不可向量化的间接寻址

```
float a[99], b[99];
int ia[99], ib[99], i, n, j;
...
for (i = 0; i < n; i++) a[ia[i]] = b[j + ib[i]];
```

间接寻址对 NEON 单元不可向量化，因为它只处理内存中连续存储的向量。如果循环中存在间接寻址和大量计算，则将间接寻址移到单独的非向量循环中可能会更加有效。这样可以使向量化计算更有效。

3.4.5 循环结构

若要从向量化中获取最佳性能，循环的整体结构是很重要的。通常，最好编写简单的循环，其迭代计数在循环开始时即固定，并且循环中不包含复杂的条件语句或条件退出。您可能需要重新编写循环，以改善代码的向量化性能。

从循环中退出

示例 3-10 也无法进行向量化，因为它在循环中包含退出。在这种情况下，如果可能，必须重新编写该循环，以使向量化成功执行。

示例 3-10 不可向量化的循环

```
int a[99], b[99], c[99], i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = b[i] + c[i];
    if (a[i] > 5) break;
};
```

循环迭代计数

循环在开始时即必须有固定的迭代计数。示例 3-11 的迭代计数为 n，并且该计数不随循环的进行而变化。

示例 3-11 可向量化的循环

```
int a[99], b[99], c[99],i, n;
...
for (i = 0; i < n; i++) a[i] = b[i] + c[i];
```

示例 3-12 没有固定的迭代计数，因而无法自动进行向量化。

示例 3-12 不可向量化的循环

```
int a[99], b[99], c[99], i, n;
...
while (i < n)
{
    a[i] = b[i] + c[i];
    i += a[i];
};
```

NEON 单元可以对元素数为 2、4、8 或 16 的组中的元素进行运算。在循环一开始迭代计数即已知的情况下，编译器可能会增加一个运行时测试来检查迭代计数是否是可用于 NEON 寄存器中相应数据类型路线的倍数。这会增加代码的大小，因为生成了执行任何附加循环迭代的附加非向量化代码。

如果您知道迭代计数是 NEON 所支持的迭代计数之一，则可以向编译器指示此计数。执行此操作最有效的方式是在调用方将迭代数除以四，在要向量化的函数中将迭代数乘以四。如果您无法修改所有的调用函数，则可以将适当的表达式用于循环限制测试，以指示循环迭代是一个合适的倍数。例如，若要指示循环迭代次数是四的倍数，可使用：

```
for(i = 0; i < (n >> 2 << 2); i++)
```

或：

```
for(i = 0; i < (n & ~3); i++)
```

这会缩减生成代码的大小，并改善性能。

3.4.6 函数调用和内联

编译器不接受对循环内其他函数的调用，因为它无法处理被调用函数的内容。向量化无法进行，因为编译器无法将向量化的部分结果传递给 NEON 寄存器中的其他函数。

通常将复杂的运算拆分为多个函数，可使运算更加明了。为了在向量化时将这些函数考虑在内，必须使用 __inline 或 __forceinline 关键字对其进行标记。然后，这些函数扩展为内联，以进行向量化。有关详细信息，请参阅《*RVCT 编译器参考指南*》中的第4-9 页的*__inline* 和第4-6 页的*__forceinline*。

——注意——

这些关键字还暗含内部静态链接。

3.4.7 条件语句

若要进行有效的向量化，循环中必须包含大部分赋值语句，并限制使用 **if** 和 **switch** 语句。

循环迭代之间不会变化的简单条件称为循环不变量。编译器可在循环之前将这些不变量移走，以便不在每个循环迭代中执行它们。通过计算向量模式下的所有路径并合并结果来对更复杂的条件运算进行向量化。如果要有条件地执行大量计算，则会浪费大量时间。

示例 3-13 显示了一种可接受的条件语句的用法。

示例 3-13 可向量化的条件

```
float a[99], b[99], c[i];
int i, n;
...
for (i = 0; i < n; i++)
{
    if (c[i] > 0) a[i] = b[i] - 5.0;
    else a[i] = b[i] * 2.0;
};
```

3.4.8 通过优化源代码来改善性能的示例

编译器可提供诊断信息来指示在何处成功应用了向量化优化，在何处应用向量化失败。有关详细信息，请参阅第2-2 页的*--diag_suppress=optimizations* 和第2-3 页的*--diag_warning=optimizations*。

示例 3-14 显示了对数组实现简单相加运算的两个函数。此代码未向量化。

示例 3-14 不可向量化的代码

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

void add_int(int *pa, int *pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

使用 --diag_warnings=optimization 选项将对 addition() 函数产生优化警告消息。

向 addition() 的定义添加 __inline 限定符可使此代码向量化，但仍未优化。再次使用 --diag_warnings=optimization 选项会产生优化警告消息，指示循环得以向量化，但是可能存在潜在的指针别名问题。

编译器必须生成针对别名的运行时测试，并输出代码的向量化副本和标量副本。示例 3-15 演示了当您知道指针没有别名时，如何使用 restrict 关键字进行改善。

示例 3-15 使用 restrict 改善向量化性能

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

void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

最后可进行的改善是循环迭代的次数。在示例 3-15 中，迭代的次数是不固定的，并且可能不是能够准确符合 NEON 寄存器所需数字的倍数。这意味着，编译器必须进行测试，以使用非向量化的代码执行剩余的迭代。如果您知道该迭代计数是 NEON 所支持的迭代计数之一，则可以向编译器指示此计数。示例 3-16 显示了可以进行的最终改善，以从向量化获取最佳性能。

示例 3-16 为获得最佳向量化性能而优化的代码

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

void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < (n & ~3); i++) *(pa + i) = addition(*(pb + i),x);
    /* n is a multiple of 4 */
}
```

3.5 示例

以下是可向量化的代码的示例：请参阅示例 3-17 和第3-18 页的示例 3-18。

示例 3-17 向量化代码

```
/*
 * Vectorizable example code.
 * Copyright 2006 ARM Limited. All rights reserved.
 *
 * Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
 *
 * Build using:
 *   armcc --vectorize -c vector_example.c --cpu Cortex-A8 -Otime
 *   armlink -o vector_example.axf vector_example.o --entry=init_cpu
 */

#include <stdio.h>

void fir(short *__restrict y, const short *x, const short *h, int n_out, int n_coefs)
{
    int n;
    for (n = 0; n < n_out; n++)
    {
        int k, sum = 0;
        for (k = 0; k < n_coefs; k++)
        {
            sum += h[k] * x[n - n_coefs + 1 + k];
        }
        y[n] = ((sum>>15) + 1) >> 1;
    }
}

int main()
{
    static const short x[128] =
    {
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
```

```
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
    };

    static const short coeffs[8] =
    {
        0x0800, 0x1000, 0x2000, 0x4000,
        0x4000, 0x2000, 0x1000, 0x0800
    };

    int i, ok = 1;
    short y[128];
    static const short expected[128] =
    {
        0x1474, 0x1a37, 0x1fe9, 0x2588, 0x2b10, 0x307d, 0x35cc, 0x3afa,
        0x4003, 0x44e5, 0x499d, 0x4e27, 0x5281, 0x56a9, 0x5a9a, 0x5e54,
        0x61d4, 0x6517, 0x681c, 0x6ae1, 0x6d63, 0x6fa3, 0x719d, 0x7352,
        0x74bf, 0x6de5, 0x66c1, 0x5755, 0x379e, 0x379e, 0x5755, 0x66c1,
        0x6de5, 0x74bf, 0x7352, 0x719d, 0x6fa3, 0x6d63, 0x6ae1, 0x681c,
        0x6517, 0x61d4, 0x5e54, 0x5a9a, 0x56a9, 0x5281, 0x4e27, 0x499d,
        0x44e5, 0x4003, 0x3afa, 0x35cc, 0x307d, 0x2b10, 0x2588, 0x1fe9,
        0x1a37, 0x1474, 0x0ea5, 0x08cd, 0x02f0, 0xfd10, 0xf733, 0xf15b,
        0xeb8c, 0xe5c9, 0xe017, 0xda78, 0xd4f0, 0xcf83, 0xca34, 0xc506,
        0xbffd, 0xbb1b, 0xb663, 0xb1d9, 0xad7f, 0xa957, 0xa566, 0xa1ac,
        0x9e2c, 0x9ae9, 0x97e4, 0x951f, 0x929d, 0x905d, 0x8e63, 0x8cae,
        0x8b41, 0x8a1b, 0x893f, 0x88ab, 0x8862, 0x8862, 0x88ab, 0x893f,
        0x8a1b, 0x8b41, 0x8cae, 0x8e63, 0x905d, 0x929d, 0x951f, 0x97e4,
        0x9ae9, 0x9e2c, 0xa1ac, 0xa566, 0xa957, 0xad7f, 0xb1d9, 0xb663,
        0xbb1b, 0xbffd, 0xc506, 0xca34, 0xcf83, 0xd4f0, 0xda78, 0xe017,
        0xe5c9, 0xebcc, 0xf229, 0xf96a, 0x02e9, 0x0dd8, 0x1937, 0x24ce,
    };

    fir(y, x + 7, coeffs, 128, 8);

    for (i = 0; i < sizeof(y)/sizeof(*y); ++i)
    {
        if (y[i] != expected[i])
        {
            printf("mismatch: y[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, y[i], i, expected[i]);
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
    return ok ? 0 : 1;
}
```

```
#ifdef __TARGET_ARCH_7_A
__asm void init_cpu() {
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE
#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4

    IMPORT __main
    B __main
}
#endif
```

示例 3-18 DSP 向量化代码

```
/*
* DSP Vectorizable example code.
* Copyright 2006 ARM Limited. All rights reserved.
*
* Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
*
* Build using:
*   armcc -c dsp_vector_example.c --cpu Cortex-A8 -Otime --vectorize
*   armlink -o dsp_vector_example.axf dsp_vector_example.o --entry=init_cpu
*/

#include <stdio.h>
#include "dspfns.h"

void fn(short *__restrict r, int n, const short *__restrict a, const short *__restrict b)
{
    int i;
    for (i = 0; i < n; ++i)
    {
        r[i] = add(a[i], b[i]);
    }
}
```

```
int main()
{
    static const short x[128] =
    {
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
    };
    static const short y[128] =
    {
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
    };
    short r[128];
    static const short expected[128] =
    {
        0x8000, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
        0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
```

```
        0x8000, 0x7991, 0x72d7, 0x6bd5, 0x6492, 0x5d10, 0x5555, 0x4d65,
        0x4546, 0x3cfb, 0x348c, 0x2bfc, 0x2351, 0x1a90, 0x11bf, 0x08e2,

        0x0000, 0xf71e, 0xee41, 0xe570, 0xdcaf, 0xd404, 0xcb74, 0xc305,
        0xbaba, 0xb29b, 0xaaab, 0xa2f0, 0x9b6e, 0x942b, 0x8d29, 0x866f,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
        0x8000, 0x866f, 0x8d29, 0x942b, 0x9b6e, 0xa2f0, 0xaaab, 0xb29b,
        0xbaba, 0xc305, 0xcb74, 0xd404, 0xdcaf, 0xe570, 0xee41, 0xf71e,
        0x0000, 0x08e2, 0x11bf, 0x1a90, 0x2351, 0x2bfc, 0x348c, 0x3cfb,
        0x4546, 0x4d65, 0x5555, 0x5d10, 0x6492, 0x6bd5, 0x72d7, 0x7991,
    };
    int i, ok = 1;

    fn(r, sizeof(r)/sizeof(*r), x, y);

    #if 0
        printf("r[] = {");
        for (i = 0; i < sizeof(r)/sizeof(*r); ++i)
        {
            if (i % 8 == 0) printf("\n ");
            printf(" 0x%04x,", (unsigned short)r[i]);
        }
        printf("\n};\n");
    #endif

    for (i = 0; i < sizeof(r)/sizeof(*r); ++i)
    {
        if (r[i] != expected[i])
        {
            printf("mismatch: r[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, r[i], i, expected[i]);
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
        return ok ? 0 : 1;
}


#ifdef __TARGET_ARCH_7_A
__asm void init_cpu()
{
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
```



```
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE

#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4

    IMPORT __main
    B __main
}
#endif
```

