

RealView® Compilation Tools

Version 3.1

NEON™ Vectorizing Compiler Guide



RealView Compilation Tools

NEON Vectorizing Compiler Guide

Copyright © 2007 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
March 2007	A	Non-confidential	Release 3.1 for RVDS v3.1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein might be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document can be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

RealView Compilation Tools

NEON Vectorizing Compiler Guide

	Preface	
	About this book	vi
	Feedback	ix
Chapter 1	Introduction	
	1.1 About the NEON vectorizing compiler	1-2
Chapter 2	NEON Vectorizing features	
	2.1 Command-line options	2-2
	2.2 Keywords	2-5
	2.3 Pragmas	2-6
	2.4 ETSI basic operations	2-9
Chapter 3	Using the NEON Vectorizing Compiler	
	3.1 The NEON unit	3-2
	3.2 Writing code for NEON	3-3
	3.3 Working with automatic vectorization	3-5
	3.4 Improving performance	3-8
	3.5 Examples	3-17

Preface

This preface introduces the *RealView Compilation Tools NEON Vectorizing Compiler*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.

About this book

This book provides you with an understanding of the NEON™ vectorizing compiler and explains how to take advantage of the automatic vectorizing features.

Intended audience

This book is written for all developers who are producing applications using the NEON Vectorizing Compiler to target ARM processors with a NEON unit. It assumes that you are an experienced software developer. See the *RealView Compilation Tools Essentials Guide* for an overview of the ARM development tools provided with RVCT.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the NEON Vectorizing Compiler.

Chapter 2 *NEON Vectorizing features*

Read this chapter for a description of the command-line options supported by the NEON Vectorizing Compiler.

Chapter 3 *Using the NEON Vectorizing Compiler*

Read this chapter for a tutorial on the NEON Vectorizing Compiler. It provides you with an understanding of the NEON unit and explains how to take advantage of the automatic vectorizing features.

This book assumes that the ARM software is installed in the default location. For example, on Windows this might be *volume:\Program Files\ARM*. This is assumed to be the location of *install_directory* when referring to path names. For example *install_directory\Documentation\...* You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

- | | |
|------------------------|--|
| <code>monospace</code> | Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code. |
| <u>monospace</u> | Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name. |

monospace italic

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

monospace bold

Denotes language keywords when used outside example code.

italic

Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold

Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM Limited periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs).

ARM publications

This book contains additional reference information specific to development tools required for use with RVCT. Other publications included in the suite are:

- *RVCT Compiler User Guide* (ARM DUI 0205)
- *RVCT Compiler Reference Guide* (ARM DUI 0348)
- *RVCT Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *RVCT Essentials Guide* (ARM DUI 0202)
- *RVCT Linker and Utilities Guide* (ARM DUI 0206)
- *RVCT Assembler Guide* (ARM DUI 0204)
- *RVCT Developer Guide* (ARM DUI 0203)
- *RealView Development Suite Glossary* (ARM DUI 0324).

For full information about the base standard, software interfaces, and standards supported by ARM, see *install_directory\Documentation\Specifications\...*

In addition, see the following documentation for specific information relating to ARM products:

- *ARM7-M Architecture Reference Manual* (ARM DDI 0403)

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (ARM DDI 0406)*
- *ARM Architecture Reference Manual Advanced SIMD Extension and VFPv3 Supplement (ARM DDI 0268)*
- ARM datasheet or technical reference manual for your hardware device.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RVCT, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on this book

If you notice any errors or omissions in this book, send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1

Introduction

NEON is an implementation of the ARM Advanced Single Instruction, Multiple Data (SIMD) Extension.

This chapter introduces the NEON vectorizing compiler and contains the following section:

- *About the NEON vectorizing compiler* on page 1-2.

1.1 About the NEON vectorizing compiler

RVCT provides `armcc --vectorize`, a vectorizing version of the ARM compiler, that targets ARM processors with a NEON unit, such as the Cortex-A8.

Vectorizing means that the compiler generates NEON vector instructions directly from C or C++ code. The compiler can vectorize regular C and C++ operations like `+` and some of the ITU intrinsics from the `dspfn.h` header file.

As an alternative to compiler vectorization, RVCT also supports NEON intrinsics as an intermediate step for SIMD code generation between a vectorizing compiler and writing assembler code. For more information, see Appendix E *Using NEON Support* in the *RVCT Compiler Reference Guide*.

———— **Note** —————

The NEON vectorizing compiler requires its own separate *FLEXnet* license to be installed prior to use. This license is independent of the license required for RVDS products. If you attempt to use the NEON vectorizing compiler with your RVDS license, the NEON compiler generates an error message.

For more information about licensing, see the *ARM FLEXnet License Management Guide* (ARM DUI 0209).

Chapter 2

NEON Vectorizing features

This chapter describes the command-line options, keywords and features supported by the NEON vectorizing compiler, `armcc --vectorize`. It includes:

- *Command-line options* on page 2-2
- *Keywords* on page 2-5
- *Pragmas* on page 2-6
- *ETSI basic operations* on page 2-9.

2.1 Command-line options

This section describes command-line options supported by the NEON vectorizing compiler in addition to those supported by the ARM compiler.

2.1.1 `--diag_suppress=optimizations`

This option suppresses diagnostic messages for high-level optimizations.

Default

By default, optimization messages have Remark severity. Specifying `--diag_suppress=optimizations` suppresses optimization messages.

Note

Use the `--remarks` option to see optimization messages having Remark severity.

Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level `-O3`, for example, loop unrolling. Use this option to suppress diagnostic messages relating to these high-level optimizations.

Example

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

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

    return result;
}
```

Compiling this code with the options `-O3 -Otime --remarks --diag_suppress=optimizations` suppresses optimization messages.

See also

- `--diag_warning=optimizations` on page 2-3
- `--diag_suppress=tag[,tag,...]` on page 2-28 in the *Compiler Reference Guide*
- `--diag_warning=tag[,tag,...]` on page 2-30 in the *Compiler Reference Guide*
- `-Onum` on page 2-66 in the *Compiler Reference Guide*

- *-Otime* on page 2-69 in the *Compiler Reference Guide*.

2.1.2 --diag_warning=optimizations

This option sets high-level optimization diagnostic messages to have Warning severity.

Default

By default, optimization messages have Remark severity.

Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level *-O3 -Otime*, for example, loop unrolling. Use this option to display diagnostic messages relating to these high-level optimizations.

Example

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

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

    return result;
}
```

Compiling this code with the options *--vectorize --cpu=Cortex-A8 -O3 -Otime --diag_warning=optimizations* generates optimization warning messages.

See also

- *--diag_suppress=optimizations* on page 2-2
- *-Onum* on page 2-66 in the *Compiler Reference Guide*
- *-Otime* on page 2-69 in the *Compiler Reference Guide*
- *--diag_suppress=tag[,tag,...]* on page 2-28 in the *Compiler Reference Guide*
- *--diag_warning=tag[,tag,...]* on page 2-30 in the *Compiler Reference Guide*.

2.1.3 --[no_]vectorize

This option enables or disables the generation of NEON vector instructions directly from C or C++ code.

Default

The default is `--no_vectorize`.

Restrictions

The following options must be specified for loops to vectorize:

- `--cpu=name` Target processor must have NEON capability.
- `-Otime` Type of optimization to reduce execution time.
- `-Onum` Level of optimization. One of the following must be used:
 - `-O2` High optimization. This is the default.
 - `-O3` Maximum optimization.

Note

NEON is an implementation of the ARM Advanced *Single Instruction, Multiple Data* (SIMD) extension.

A separate *FLEXnet* license is needed to enable the use of vectorization.

Example

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

See also

- `--cpu=name` on page 2-15 in the *Compiler Reference Guide*
- `-Onum` on page 2-66 in the *Compiler Reference Guide*
- `-Otime` on page 2-69 in the *Compiler Reference Guide*.

2.2 Keywords

This section describes C/C++ keywords supported by the NEON vectorizing compiler in addition to those supported by the ARM compiler.

2.2.1 restrict

The **restrict** keyword is a C99 feature that enables you to ensure that different object pointer types and function parameter arrays do not point to overlapping regions of memory. Therefore, the compiler can perform optimizations that might otherwise be prevented because of possible aliasing.

To enable the **restrict** keyword in C90 or C++, you must specify the `--restrict` option.

The keywords `__restrict` and `__restrict__` are supported as synonyms for **restrict** and are always available, regardless of the `--restrict` option.

Example

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

See also

- *Using pointers* on page 3-11
- *restrict* on page 3-8 in the *Compiler Reference Guide*.

2.3 Pragma

This section describes the ARM-specific pragmas supported by the NEON vectorizing compiler in addition to those supported by the ARM compiler.

2.3.1 #pragma unroll [(n)]

This pragma instructs the compiler to unroll a loop by n iterations.

Note

Both vectorized and non vectorized loops can be unrolled using #pragma unroll [(n)]. That is, #pragma unroll [(n)] applies to both --vectorize and --no_vectorize.

Syntax

```
#pragma unroll
```

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

Where:

n is an optional value indicating the number of iterations to unroll.

Default

If you do not specify a value for n , the compiler assumes #pragma unroll (4).

Usage

When compiling at -O3 -Otime, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to request that the compiler to unroll a loop that has not been unrolled automatically.

Note

Use this #pragma only when you have evidence, for example from --diag_warning=optimizations, that the compiler is not unrolling loops optimally by itself.

Restrictions

#pragma unroll [(n)] can be used only immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

Example

```

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;
        }
    }
}

```

In this example, the compiler does not normally complete its loop analysis because `src2` is indexed as `src2[j][k]` but the loops are nested in the opposite order, that is, with `j` inside `k`. When `#pragma unroll` is uncommented in the example, the compiler proceeds to unroll the loop four times.

If the intention is to multiply a matrix that is not a multiple of four in size, for example an $n * n$ matrix, `#pragma unroll (m)` could be used instead, where m is some value such that n is an integral multiple of m .

See also

- `--diag_warning=optimizations` on page 2-3
- `#pragma unroll_completely`
- `--[no_]vectorize` on page 2-3
- `-Onum` on page 2-66 in the *Compiler Reference Guide*
- `-Otime` on page 2-69 in the *Compiler Reference Guide*
- *Optimizing loops* on page 4-4 in the *Compiler User Guide*.

2.3.2 #pragma unroll_completely

This pragma instructs the compiler to completely unroll a loop. It has an effect only if the compiler can determine the number of iterations the loop has.

Note

Both vectorized and non vectorized loops can be unrolled using `#pragma unroll_completely`. That is, `#pragma unroll_completely` applies to both `--no_vectorize` and `--vectorize`.

Usage

When compiling at `-O3 -Otime`, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to request that the compiler completely unroll a loop that has not automatically been unrolled completely.

Note

Use this `#pragma` only when you have evidence, for example from `--diag_warning=optimizations`, that the compiler is not unrolling loops optimally by itself.

Restrictions

`#pragma unroll_completely` can only be used immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

Using `#pragma unroll_completely` on an outer loop can prevent vectorization. On the other hand, using `#pragma unroll_completely` on an inner loop might help in some cases.

See also

- `--diag_warning=optimizations` on page 2-3
- `--[no_]vectorize` on page 2-3
- `#pragma unroll [(n)]` on page 2-6
- `-Onum` on page 2-66 in the *Compiler Reference Guide*
- `-Otime` on page 2-69 in the *Compiler Reference Guide*
- *Optimizing loops* on page 4-4 in the *Compiler User Guide*.

2.4 ETSI basic operations

RVCT supports the original ETSI family of basic operations described in the ETSI G.729 recommendation *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*. For a full list of operations see *ETSI basic operations* on page 4-97 in the *Compiler Reference Guide* for more information.

Table 2-1 shows a list of operations that can vectorize. To use the ETSI basic operations in your own code, include the header file `dspfns.h`.

Table 2-1 ETSI basic operations supported in RVCT 3.1

Intrinsics				
<code>abs_s</code>	<code>L_sub</code>	<code>L_mult</code>	<code>mult_r</code>	<code>shl</code>
<code>extract_h</code>	<code>L_deposit_h</code>	<code>L_negate</code>	<code>negate</code>	<code>shr</code>
<code>extract_l</code>	<code>L_deposit_l</code>	<code>L_shl</code>	<code>norm_s</code>	<code>shr_r</code>
<code>L_abs</code>	<code>L_mac</code>	<code>L_shr</code>	<code>norm_l</code>	<code>sub</code>
<code>L_add</code>	<code>L_msu</code>	<code>mult</code>	<code>round</code>	

Chapter 3

Using the NEON Vectorizing Compiler

This chapter provides you with an understanding of the NEON unit and explains how to take advantage of the automatic vectorizing features. It contains the following sections:

- *The NEON unit* on page 3-2
- *Writing code for NEON* on page 3-3
- *Working with automatic vectorization* on page 3-5
- *Improving performance* on page 3-8
- *Examples* on page 3-17.

3.1 The NEON unit

The NEON unit provides 32 vector registers that each hold 16 bytes of information. These 16 byte registers can then be operated on in parallel in the NEON unit. For example, in one vector add instruction you can add eight 16-bit integers to eight other 16 bit integers to produce eight 16-bit results.

The NEON unit supports 8-bit, 16-bit and 32-bit integer operations, and some 64-bit operations, in addition to 32-bit floating point operations.

———— **Note** —————

Vector floating point operations are only performed when the VFP coprocessor is operating in RunFast mode. You must compile with `--fpmode fast` to vectorize floating point code.

The NEON unit is classified as a vector SIMD unit that operates on multiple elements, in a vector register, with one instruction.

For example, array A is a 16-bit integer array with 8 elements.

Table 3-1 Array A

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

Array B has these 8 elements:

Table 3-2 Array B

80	70	60	50	40	30	20	10
----	----	----	----	----	----	----	----

To add these arrays together, fetch each vector into a vector register and use one vector SIMD instruction to obtain the result.

Table 3-3 Result

81	72	63	54	45	36	27	18
----	----	----	----	----	----	----	----

3.2 Writing code for NEON

This section gives an overview of the ways you can write code for the NEON unit. There are several ways to get code running on NEON:

- Write in assembly language, or use embedded assembly language in C, and use the NEON instructions directly.
- Write in C or C++ with the NEON C language extensions.
- Call a library routine that has been optimized to use NEON instructions.
- Use automatic vectorization to get loops vectorized for NEON.

3.2.1 NEON C extensions

The NEON C extensions are a set of new data types and intrinsic functions defined by ARM to enable access to the NEON unit from C. Most of the vector functions map directly to vector instructions available in the NEON unit and are compiled inline by the NEON enhanced ARM C compiler. With these extensions, performance can be achieved at the C level that is comparable to that obtained with assembly language coding.

See Appendix E *Using NEON Support* in the *RVCT Compiler Reference Guide* for more information.

3.2.2 Automatic vectorization

By coding in vectorizable loops instead of writing in explicit NEON instructions, code portability is preserved between processors. Performance levels similar to that of hand coded vectorization are achieved with less effort.

Example 3-1 shows the required command-line options to invoke automatic vectorization.

Example 3-1 Automatic vectorization

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

Enabling and disabling the vectorize option provides timing comparisons with scalar code when optimizing for improved performance using vectorization. See `--[no_]vectorize` on page 2-3 for more information.

———— **Note** ————

You can also compile with `-O2 -Otime`, however, this does not give the maximum code performance.

3.2.3 Performance goals

Most applications require tuning to gain the best performance from vectorization. There is always some overhead so the theoretical maximum performance cannot be reached. For example, the NEON unit can process four single-precision floats at one time. This means that the theoretical maximum performance for a floating-point application, is a factor of four over the original scalar non vectorized code. Given typical overheads, a reasonable goal for a whole floating-point application is to aim for a 50% improvement on performance over the scalar code. For large applications that are not completely vectorizable, achieving a 25% improvement on performance over the scalar code is a reasonable goal.

See *Improving performance* on page 3-8 for more information.

3.3 Working with automatic vectorization

This section gives an overview of automatic vectorization and also describes the factors affecting the vectorization process and performance of the generated code.

3.3.1 Overview of automatic vectorization

Automatic vectorization involves the high-level analysis of loops in your code. This is the most efficient way to map the majority of typical code onto the functionality of the NEON unit. For most code, the gains that can be made with algorithm-dependent parallelism on a smaller scale are very small relative to the cost of automatic analysis of such opportunities. For this reason, the NEON unit is designed as a target for simple loop-based parallelism.

Vectorization is carried out in a way that ensures that the optimized code gives the same results as the non vectorized code. In certain cases vectorization of a loop is not carried out so that the possibility of an incorrect result is avoided. This can lead to sub-optimal code, and you might need to manually tune your code to make it more suitable for automatic vectorization. See *Improving performance* on page 3-8 for more information.

3.3.2 Vectorization concepts

This section describes some concepts that are commonly used when considering vectorization of code.

Data references

Data references in your code can be classified as one of three types:

Scalar	A single location that does not change through all the iterations of the loop.
Index	An integer quantity that increments by a constant amount each pass through the loop.
Vector	A range of memory locations with a constant stride between consecutive elements.

Example 3-2 on page 3-6 shows the classification of variables in the loop:

i, j	index variables
a, b	vectors
x	scalar

Example 3-2 Categorization of a vectorizable loop

```

float *a, *b;
int i, j, n;
...
for (i = 0; i < n; i++)
{
    *(a+j) = x +b[i];
    j += 2;
};

```

When classifying variables ensure that vectorization of the loop provides the same results as the scalar version. Loops are unable to vectorize directly if there are data dependencies within the loop that calculate future iterations. See *Data dependencies* on page 3-8.

Stride patterns and data accesses

The stride pattern of data accesses in a loop is the pattern of accesses to data elements between sequential loop iterations. For example, a loop that linearly accesses each element of an array has a stride of one. Another example is a loop that accesses an array with a constant offset between each element used is described as having a constant stride.

3.3.3 Factors affecting vectorization performance

The automatic vectorization process and performance of the generated code is affected by the following:

The way loops are organized

For best performance, the innermost loop in a loop nest must access arrays with a stride of one.

The way the data is structured

For example, a single structure containing arrays of data can be more efficient than an array of structures. The data type also dictates how many data elements can be held in a NEON register, and therefore, how many operations can be performed in parallel.

The iteration counts of loops

Longer iteration counts are generally better, because the loop overhead is amortized over more iterations. Tiny iteration counts, such as two or three elements, can be faster to process with non vector instructions. Extremely long loops, accessing tens of thousands of array elements, can exceed the size of the cache and interfere with data reuse.

The data type of arrays

For example, NEON does not improve performance when double precision floating point arrays are used.

The use of memory hierarchy

Most current processors are relatively unbalanced between memory bandwidth and processor capacity. For example, performing relatively few arithmetic operations on large data sets retrieved from main memory is limited by the memory bandwidth of the system.

3.4 Improving performance

Most applications require some tuning on the part of the programmer to get the best NEON results. This section describes the different types of loops. It explains how vectorization works successfully with some loops but does not work with others. It also explains how you can modify code to achieve the best performance from the vectorized code.

3.4.1 General performance issues

Using the command-line options `-O3` and `-Otime` ensures that the code achieves significant performance benefits in addition to those of vectorization.

When optimizing for performance, you must give consideration to the high-level algorithm structure, data element size, array configurations, strict iterative loops, reduction operations and data dependency issues. Optimizing for performance requires an understanding of where in the program most of the time is spent. To gain maximum performance benefits you might need to use profiling and benchmarking of the code under realistic conditions.

Automatic vectorization can often be impeded by any prior manual optimization of the code, for example, manual loop unrolling in the source code or complex array accesses. For optimal results, the best way is to write the code using simple loops, therefore enabling the compiler to perform all the optimization. For hand-optimized legacy code, you might find it easier to rewrite critical portions based on the original algorithm using simple loops. Removing manual optimizations might impede automatic vectorization.

For more information see:

- `--[no_]vectorize` on page 2-3
- `-Onum` on page 2-66 in the *RVCT Compiler Reference Guide*
- `-Otime` on page 2-69 in the *RVCT Compiler Reference Guide*.

3.4.2 Data dependencies

A loop that has results from one iteration feeding back into a future iteration of the same loop is said to have a data dependency conflict. The conflicting values might be array elements or a scalar such as an accumulated sum.

Loops containing data dependency conflicts might not be completely optimized. To detect data dependencies involving arrays and/or pointers requires extensive analysis of the arrays used in each loop nest, and examination of the offset and stride of accesses to elements along each dimension of arrays that are both used and stored in a loop. If there is a possibility of the usage and storage of arrays overlapping on different iterations of a loop, then there is a data dependency problem. A loop cannot be safely vectorized if

the vector order of operations can change the results. In these cases, the compiler detects the problem and leaves the loop in its original form or carries out a partial vectorization of the loop. This type of data dependency must be avoided in your code to achieve the best performance.

In the loop shown in the Example 3-3, the reference to `a[i-2]` at the top of the loop conflicts with the store into `a[i]` at the bottom. Performing vectorization on this loop gives a different result, and so it is left in its original form.

Example 3-3 Non vectorizable data dependency

```
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;
};
```

Information from other array subscripts is used as part of the analysis of dependencies. The loop in Example 3-4 vectorizes because the non vector subscripts of the references to array `a` can never be equal, because `n` is not equal to `n+1`, and so gives no feedback between iterations. The references to array `a` use two different pieces of the array and so, do not share data.

Example 3-4 Vectorizable data dependency

```
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 Scalar variables

A scalar variable that is used but not set, in a NEON loop is replicated in each position in a vector register and the result used in the vector calculation.

A scalar that is set and then used in a loop is *promoted* to a vector. These variables generally hold temporary scalar values in a loop that now has to hold temporary vector values. In Example 3-5 on page 3-10, `x` is a *used* scalar and `y` is a *promoted* scalar.

Example 3-5 Vectorizable loop

```

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;
};

```

A scalar that is used and then set in a loop is called a *carry-around* scalar. These variables are a problem for vectorization because the value computed in one pass of the loop is carried forward into the next pass. In Example 3-6 x is a carry-around scalar.

Example 3-6 Non vectorizable loop

```

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;
};

```

Reduction operations

A special category of scalar usages in a loop is reduction operations. This category involves the reduction of a vector of values down to a scalar result. The most common reduction is the summation of all elements of a vector. Other reductions include: dot product of two vectors, maximum value in a vector, minimum value in a vector, product of all vector elements and location of a maximum or minimum element in a vector.

Example 3-7 on page 3-11 shows a dot product reduction where x is a reduction scalar.

Example 3-7 Dot product reduction

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

Reduction operations are worth vectorizing because they occur so often. In general, reduction operations are vectorized by creating a vector of partial reductions that are then reduced into the final resulting scalar.

3.4.4 Using pointers

For vectorization purposes, it is generally better to use arrays rather than pointers. The compiler is able to vectorize loops containing pointers if it can determine that the loop is safe. Both array references and pointer references in loops are analyzed to see if there is any vector access to memory. In some cases, the compiler creates a run-time test, and executes a vector version or scalar version of the loop depending on the result of the test.

Often function arguments are passed as pointers. If several pointer variables are passed to a function, it is possible that pointing to overlapping sections of memory can occur. Often, at runtime, this is not the case but the compiler always follows the safe method and avoids optimizing loops that involve pointers appearing on both the left and right sides of an assignment operator. For example, consider the function in Example 3-8.

Example 3-8 Non vectorizable pointers

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

In this example, if *pa* and *pb* overlap in memory in a way that causes results from one loop pass to feedback to a subsequent loop pass, then vectorization of the loop can give incorrect results. If the function is called with the following arguments, vectorization might be ambiguous:

```
int *a;

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

The compiler performs a runtime test to see if pointer aliasing occurs. If pointer aliasing does not occur, it executes a vectorized version of the code. If pointer aliasing occurs, the original non vectorized code is executed instead. This leads to a small cost in runtime efficiency and code size.

In actual practice, it is very rare for data dependence to exist because of function arguments. Programs that pass overlapping pointers are very hard to understand and debug, apart from any vectorization concerns.

See *restrict* on page 2-5 for more information.

Indirect addressing

Indirect addressing occurs when an array is accessed by a vector of values. If the array is being fetched from memory, the operation is called a *gather*. If the array is being stored into memory, the operation is called a *scatter*. In Example 3-9, a is being scattered, and b is being gathered.

Example 3-9 Non vectorizable indirect addressing

```
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]];

```

Indirect addressing is not vectorizable with the NEON unit because it can only deal with vectors that are stored consecutively in memory. If there is indirect addressing and significant calculations in a loop, it might be more efficient for you to move the indirect addressing into a separate non vector loop. This enables the calculations to vectorize efficiently.

3.4.5 Loop structure

The overall structure of a loop is important for obtaining the best performance from vectorization. Generally, it is best to write simple loops with iteration counts that are fixed at the start of the loop, and do not contain complex conditional statements or conditional exits. You might need to rewrite your loops to improve the vectorization performance of the code.

Exits from loops

Example 3-10 is also unable to vectorize because it contains an exit from the loop. In cases like this, you must rewrite the loop if possible for vectorization to succeed.

Example 3-10 Non vectorizable loop

```
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;
};
```

Loop iteration count

Loops must have a fixed iteration count at the start of the loop. Example 3-11 shows the iteration count is *n* and this is not changed through the course of the loop.

Example 3-11 Vectorizable loop

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

Example 3-12 has no fixed iteration count and is unable to vectorize automatically.

Example 3-12 Non vectorizable loop

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

The NEON unit can operate on elements in groups of 2, 4, 8, or 16. Where the iteration count at the start of the loop is known, the compiler might add a runtime test to check if the iteration count is not a multiple of the lanes that can be used for the appropriate data type in a NEON register. This increases the code size because additional non vectorized code is generated to execute any additional loop iterations.

If you know that your iteration count is one of those supported by NEON, you can indicate this to the compiler. The most efficient way to do this is to divide the number of iterations by four in the caller and multiply by four in the function that you intend to vectorize. If you cannot modify all of the calling functions, you can use an appropriate expression for your loop limit test to indicate that the loop iteration is a suitable multiple. For example, to indicate that your loop is a multiple of four iterations, use:

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

or:

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

This reduces the size of the generated code and can give a performance improvement.

3.4.6 Function calls and inlining

Calls to other functions within a loop are not accepted by the compiler because it cannot process the content of the called function. Vectorization is inhibited because the compiler cannot pass the partial result of vectorization to the other function in NEON registers.

Splitting complex operations into several functions to aid clarity is common practice. In order for these functions to be considered for vectorization, they must be marked with the `__inline` or `__forceinline` keywords. These functions are then expanded inline for vectorization. See `__inline` on page 4-9 and `__forceinline` on page 4-6 in the *RVCT Compiler Reference Guide* for more information.

———— Note —————

These keywords also imply internal static linkage.

3.4.7 Conditional statements

For efficient vectorization, loops must contain mostly assignments statements and limit the use of `if` and `switch` statements.

Simple conditions that do not change between iterations of the loop are described as being loop invariant. These can be moved before the loop by the compiler so that they do not need to be executed on each loop iteration. More complex conditional operations are vectorized by computing all pathways in vector mode and merging the results. If there is significant computation being performed conditionally, then a substantial amount of time is wasted.

Example 3-13 shows an acceptable use of conditional statements.

Example 3-13 Vectorizable condition

```
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 Example of improving performance by tuning source code

The compiler can provide diagnostic information to indicate where vectorization optimizations are successfully applied and where it failed to apply vectorization. See `--diag_suppress=optimizations` on page 2-2 and `--diag_warning=optimizations` on page 2-3 for more information.

Example 3-14 shows two functions that implement a simple sum operation on an array. This code does not vectorize.

Example 3-14 Non vectorizable code

```
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);
}
```

Using the `--diag_warnings=optimization` option produces an optimization warning message for the `addition()` function.

Adding the `__inline` qualifier to the definition of `addition()` enables this code to vectorize but it is still not optimal. Using the `--diag_warnings=optimization` option again, produces optimization warning messages to indicate that the loop vectorizes but there might be a potential pointer aliasing problem.

The compiler must generate a runtime test for aliasing and output both vectorized and scalar copies of the code. Example 3-15 shows how this can be improved using the restrict keyword if you know that the pointers are not aliased.

Example 3-15 Using restrict to improve vectorization performance

```
__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);
}
```

The final improvement that can be made is to the number of loop iterations. In Example 3-15, the number of iterations is not fixed and might not be a multiple that can fit exactly into a NEON register. This means that the compiler must test for remaining iterations to execute using non vectored code. If you know that your iteration count is one of those supported by NEON, you can indicate this to the compiler. Example 3-16 shows the final improvement that can be made to obtain the best performance from vectorization.

Example 3-16 Code tuned for best vectorization performance

```
__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 Examples

The following are examples of vectorizable code. See Example 3-17 and Example 3-18 on page 3-19.

Example 3-17 Vectorization code

```

/*
 * 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, 0xaeed, 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, 0xbd9, 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, 0xbd9, 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

```

Example 3-18 DSP vectorization code

```

/*
 * 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 "dspfn.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,
        0x8000, 0x7991, 0x72d7, 0x6bd5, 0x6492, 0x5d10, 0x5555, 0x4d65,
        0x4546, 0x3cfb, 0x348c, 0x2bfc, 0x2351, 0x1a90, 0x11bf, 0x08e2,
    };
}

```

```

    0x0000, 0xf71e, 0xee41, 0xe570, 0xdcdf, 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, 0x866f, 0x8d29, 0x942b, 0x9b6e, 0xa2f0, 0xaaab, 0xb29b,
    0xbaba, 0xc305, 0xcb74, 0xd404, 0xdcdf, 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
#endif
#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
```
