Introducing NEON™

Development Article

ARM®

ARM DHT 0002A

Introducing NEON
Development Article

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
2 June 2009	A	Non-Confidential	Issue 1

Proprietary Notice

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.

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

`http://www.arm.com`

Contents
Introducing NEON Development Article

Chapter 1
Introducing NEON

This article introduces the NEON technology first implemented in the ARM Cortex™-A8 processor. It introduces the generic *Single Instruction Multiple Data* (SIMD) concept in addition to the NEON architecture and gives a high-level description of how to utilize it. It contains the following sections:

1.1 What is SIMD?

Some modern software, particularly media codecs and graphics accelerators, operate on large amounts of data that is less than word-sized. 16-bit data is common in audio applications, and 8-bit data is common in graphics and video.

When performing these operations on a 32-bit microprocessor, parts of the computation units are unused, but continue to consume power. To make better use of the available resources, SIMD technology uses a single instruction to perform the same operation in parallel on multiple data elements of the same type and size. This way, the hardware that normally adds two 32-bit values instead performs four parallel additions of 8-bit values in the same amount of time.

1.1.1 ARM SIMD instructions

ARMv6 architecture introduced a small set of SIMD instructions, operating on multiple 16-bit or 8-bit values packed into standard 32-bit general purpose registers. This permits certain operations to execute twice or four times as quickly, without implementing additional computation units. The mnemonics for these instructions are recognized by having 8 or 16 appended to the base form, indicating the size of data values operated on.

Figure 1-1 shows the operation of the `UADD8 R0, R1, R2` instruction. This operation performs a parallel addition of four *lanes* of 8-bit *elements* packed into *vectors* stored in general purpose registers R1 and R2, and places the result into a vector in register R0.

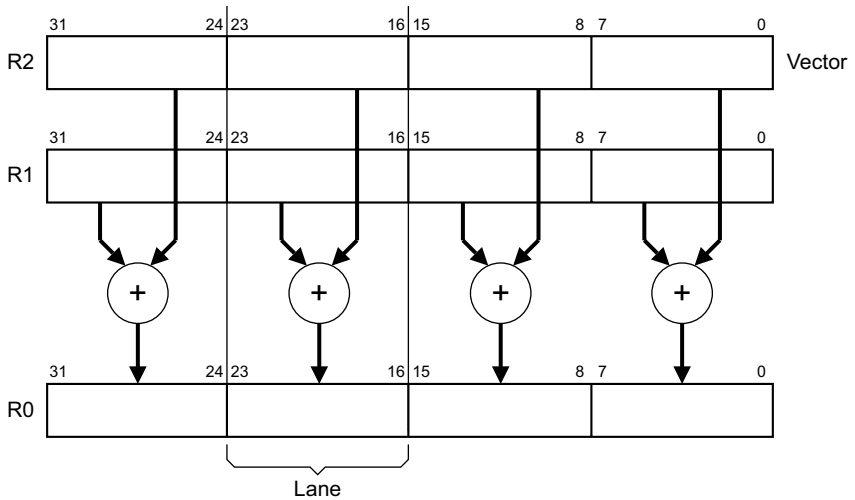

Figure 1-1 4-way 8-bit unsigned integer add operation

1.2 What is NEON?

ARMv7 architecture introduced the Advanced SIMD extension as an optional extension to the ARMv7-A and ARMv7-R profiles. It extends the SIMD concept by defining groups of instructions operating on vectors stored in 64-bit D, doubleword, registers and 128-bit Q, quadword, vector registers.

The implementation of the Advanced SIMD extension used in ARM processors is called NEON, and this is the common terminology used outside architecture specifications. NEON technology is implemented on all current ARM Cortex-A series processors.

NEON instructions are executed as part of the ARM or Thumb instruction stream. This simplifies software development, debugging, and integration compared to using an external accelerator. Traditional ARM or Thumb instructions manage all program flow and synchronization. The NEON instructions perform:

- memory accesses
- data copying between NEON and general purpose registers
- data type conversion
- data processing.

Figure 1-2 shows how the `VADD.I16 Q0, Q1, Q2` instruction performs a parallel addition of eight lanes of 16-bit elements from vectors in Q1 and Q2, storing the result in Q0.

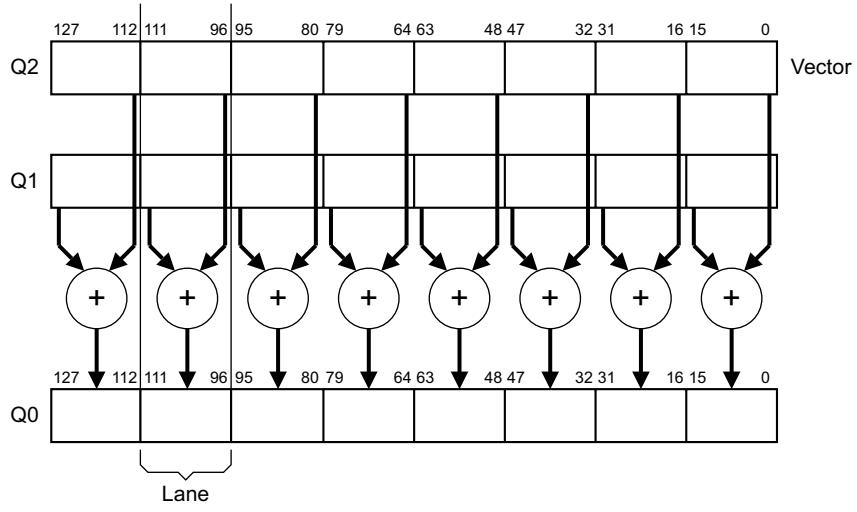

Figure 1-2 8-way 16-bit integer add operation

1.3 NEON architecture overview

The ARM architecture defines the Advanced SIMD extension as a part of coprocessors 10 and 11, that are also used for the Vector Floating Point (VFP) extension. There is no architectural requirement for a processor to implement both VFP and NEON, but the common features in the programmers models for these extensions mean an operating system that supports VFP requires little or no modifications to also support NEON.

When optimizing NEON code for a particular processor, you might have to consider implementation defined aspects of how that processor integrates the NEON technology. This means that a sequence of instructions optimized for a specific processor might have different timing characteristics on a different processor even if the NEON instruction cycle timings are identical.

See the *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* for more information about the Advanced SIMD extension, including instruction listings and encodings. This is available on request from `http://infocenter.arm.com`.

1.3.1 Supported data types

The NEON instructions support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers.

NEON also supports 32-bit single-precision floating point elements, and 8-bit and 16-bit polynomials.

The `VCVT` instruction converts elements between single-precision floating-point and:

- 32-bit integer
- fixed-point
- half-precision floating point, if the processor implements the half-precision extensions.

1.3.2 NEON registers

The NEON register bank consists of 32 64-bit registers. If both Advanced SIMD and VFPv3 are implemented, they share this register bank. In this case, VFPv3 is implemented in the VFPv3-D32 form that supports 32 double-precision floating-point registers. This integration simplifies implementing context switching support, because the same routines that save and restore VFP context also save and restore NEON context.

The NEON unit can view the same register bank as:

- sixteen 128-bit quadword registers, Q0-Q15
- thirty-two 64-bit doubleword registers, D0-D31.

The NEON D0-D31 registers are the same as the VFPv3 D0-D31 registers and each of the Q0-Q15 registers map onto a pair of D registers. Figure 1-3 shows the different views of the shared NEON and VFP register bank. All of these views are accessible at any time. Software does not have to explicitly switch between them, because the instruction used determines the appropriate view.

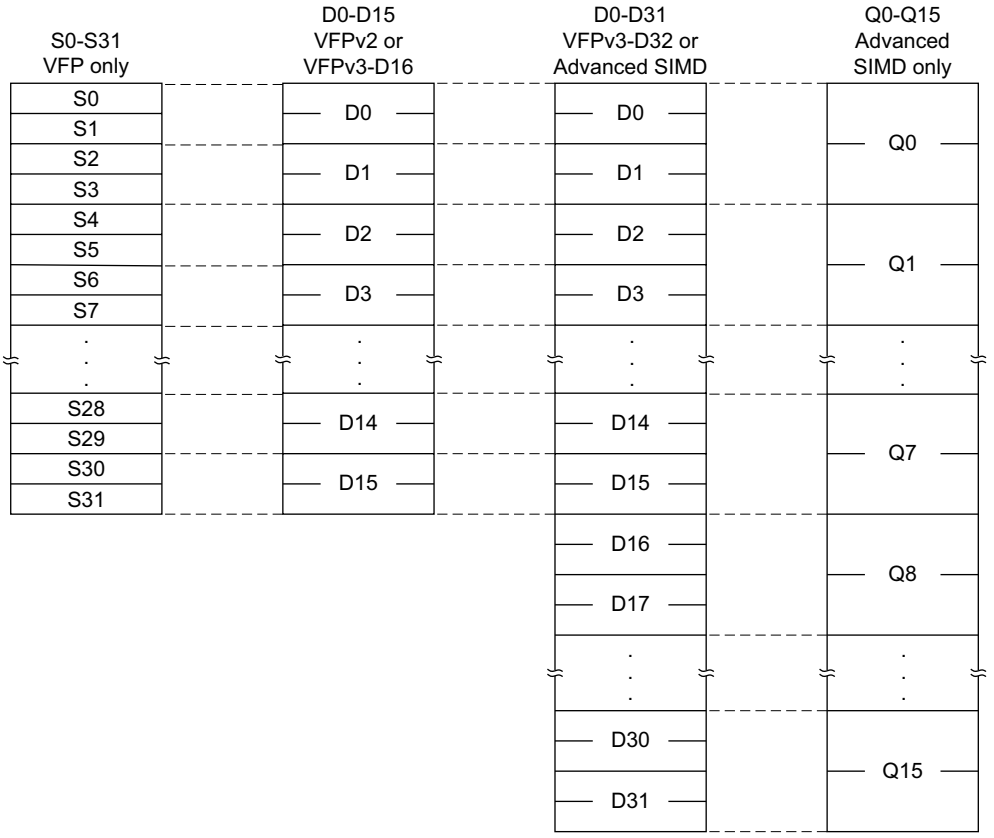

Figure 1-3 NEON and VFP register set

1.3.3 NEON instructions

The NEON instructions provide data processing and load/store operations only, and are integrated into the ARM and Thumb instruction sets. Standard ARM and Thumb instructions manage all program flow control.

The encodings for NEON instructions correspond to coprocessor operations affecting coprocessors 10 and 11, the same as VFP instructions. NEON and VFP instructions are also grouped together alphabetically because all mnemonics begin with a capital V.

Most instructions can operate on different data types, specified in the instruction encoding. Software indicates the size required by appending a suffix to the instruction mnemonic. The number of elements operated on is indicated by the specified register size. For example, `VADD.I16 q0, q1, q2` indicates an operation on 16-bit integer elements stored in 128-bit Q registers. This means that the operation is on eight 16-bit lanes in parallel.

Some instructions can have different size input and output registers. For example, `VMULL.S16 Q0, D2, D3` multiplies four 16-bit lanes in parallel, producing four 32-bit products in a 128-bit destination vector.

To improve code density and performance, the NEON instruction set includes structured load and store instructions that can load or store single or multiple values from or to single or multiple lanes in a vector register. It also includes instructions that transfer complete data structures between several vector registers and memory, with interleaving and de-interleaving.

1.4 Developing for NEON

To benefit from new features, you must use up-to-date versions of your compilation tools. Recent versions of both the GNU tools and *RealView® Compilation Tools* (RVCT) support the NEON instructions.

1.4.1 Assembler

The direct way to utilize the NEON unit is by writing assembly code. The consistent design of the NEON instruction set makes this less complex than you might expect.

The GNU and RVCT assemblers use the same instruction format, but other syntax differs. The differences include:

- assembler directives
- format of labels
- comment indicators.

Example 1-1 shows an assembler function executing a NEON instruction with the GNU assembler (Gas), and Example 1-2 shows the same code in RVCT format. Both examples use hardware floating-point linkage, meaning that the software passes and returns parameters in NEON registers.

Example 1-1 Simple NEON assembler example for Gas

```
    .text
    .arm
    .global double_elements
double_elements:
    vadd.i32 q0,q0,q0
    bx       lr
    .end
```

To assemble the code in Example 1-1 using Gas, add `-mfpu=neon` to the assembler command line. This specifies that NEON instructions are permitted. For example:

```
arm-none-linux-gnueabi-as -mfpu=neon asm.s
```

Example 1-2 Simple NEON assembler example for RVCT

```
    AREA RO, CODE, READONLY
    ARM
    EXPORT double_elements
double_elements
```

```
    VADD.I32 Q0, Q0, Q0
    BX       LR
    END
```

To assemble the code in Example 1-2 on page 1-7 using RVCT, you must specify a target processor that supports NEON instructions. For example:

```
armasm --cpu=Cortex-A8 asm.s
```

1.4.2 Intrinsics

Intrinsic functions and data types, or *intrinsics*, provide similar functionality to inline assembly, and provide additional features like type checking and automatic register allocation. An intrinsic function appears as a function call in C or C++, but is replaced during compilation by a sequence of low-level instructions. This means you can express low-level architectural behavior in a high-level language.

In addition to giving the programmer direct access to instructions that do not normally map well onto high-level language statements, using intrinsics means the compiler can optimize the operation to improve performance. Using intrinsics means the developer does not have to consider register allocation and interlock issues, because the compiler handles these.

GCC and RVCT support the same NEON intrinsic syntax, making C or C++ code portable between the toolchains. To add support for NEON intrinsics, include the header file `arm_neon.h`. Example 1-3 implements the same functionality as the assembler examples, using intrinsics in C code instead of assembler instructions.

Example 1-3 NEON intrinsics

```
#include <arm_neon.h>

uint32x4_t double_elements(uint32x4_t input)
{
    return(vaddq_u32(input, input));
}
```

Compiling the example

Although the GNU and RVCT development tools support the same syntax for NEON intrinsics, the command line syntax differs significantly between the two. The methods for compiling this example are described separately in:

- *NEON intrinsics with GCC* on page 1-9

- *NEON intrinsics with RVCT.*

NEON intrinsics with GCC

To use NEON intrinsics in GCC, you must specify `-mfpu=neon` on the compiler command line:

```
arm-none-linux-gnueabi-gcc -mfpu=neon intrinsic.c
```

Depending on your toolchain, you might also have to add -mfloat-abi=softfp to indicate to the compiler that NEON variables must be passed in general purpose registers.

A complete list of supported intrinsics can be found at `http://gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html`

NEON intrinsics with RVCT

RVCT accepts NEON instrinsics if you specify, on the compiler command line, a target processor that supports the NEON instructions. For example:

```
armcc --cpu=Cortex-A9 intrinsic.c
```

For information about the supported intrinsic functions and vector data types, see the *RealView Compilation Tools Compiler Reference Guide*, available from `http://infocenter.arm.com`.

1.4.3 Automatic vectorization

The compiler can also perform *automatic vectorization* on your C or C++ source code. This gives access to high NEON performance without writing assembly code or using intrinsics. This permits your source code to remain portable between different tools and target platforms.

Because the C language does not specify parallelizing behavior, you might have to give the compiler additional hints about where this is safe and optimal. You can do this without compromising the portability of the source code between different platforms or toolchains.

Example 1-4 on page 1-10 shows a small function that the compiler can safely and optimally vectorize. This is possible because the programmer has used the `__restrict` keyword to guarantee that the pointers `pa` and `pb` will not address overlapping regions of memory. The programmer has also forced the for loop to always execute a multiple of four times by masking off the bottom two bits of `n` for the limit test. This extra information makes it safe for the compiler to vectorize this function into NEON load and store operations.

Example 1-4 NEON vectorization

```
void add_ints(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;

    for(i = 0; i < (n & ~3); i++)
        pa[i] = pb[i] + x;
}
```

Compiling the examples

Although the GNU and RVCT development tools support the same source syntax, the command line syntax differs significantly between the two. The methods for compiling this example are described separately in:

- *Automatic vectorization with GCC*
- *Automatic vectorization with RVCT*.

Automatic vectorization with GCC

To enable automatic vectorization, you must add `-mfpu=neon` and `-ftree-vectorize` to the GCC command line. For example:

```
arm-none-linux-gnueabi-gcc -mfpu=neon -ftree-vectorize -c vectorized.c
```

Depending on your toolchain, you might also have to add `-mfloat-abi=softfp` to indicate that NEON variables must be passed in general purpose registers.

You can request more verbose compiler output by adding `-ftree-vectorizer-verbose=1` to the command line. This makes the compiler output information about:

- code that it has vectorized
- code that it could not vectorize, and hints of why this was not done.

You can use this information to modify the code into a format that the compiler can vectorize. Some versions of GCC support verbosity values higher than 1, providing even more detail about vectorization.

Automatic vectorization with RVCT

To enable automatic vectorization, you must specify a target processor that includes NEON technology, compile for optimization level `-O2` or higher, and add `-Otime and --vectorize` to the command line. For example:

```
armcc --cpu=Cortex-A9 -O3 -Otime --vectorize -c vectorized.c
```

Note

When you specify `--vectorize`, automatic vectorization is enabled only if you also specify `-Otime` and an optimization level of `-O2` or `-O3`.

Because parallel accumulations of floating-point values can reduce the precision gained by sorting input data, these are disabled unless you specify `--fpmode=fast` on the command line.

You can request more verbose compiler output by adding `--remarks` to the command line. This provides additional information about many aspects of the compilation. For NEON vectorization, this includes:

- code that the compiler has vectorized
- code that could not be vectorized, and hints of why this was not done.

This information can be used to modify the code into a format which the compiler is able to vectorize.

1.4.4 Using NEON optimized libraries

The easiest way of utilizing the NEON technology in your system is to simply use libraries that are optimized for NEON.

OpenMAX

OpenMAX is a royalty-free cross platform API standard created and distributed by the Khronos Group. ARM has created an ARMv7 NEON optimized implementation of the OpenMAX Development Layer (DL). You can download this from `http://www.arm.com`

Example 1-5 calculates the dot product of the values in two vectors of signed 16-bit integers by calling the OpenMAX function `omxSP_DotProd_S16()`. This function is implemented using NEON vector operations when using the ARMv7 optimized OpenMAX DL library.

Example 1-5 OpenMAX example

```
#include <omxSP.h>

OMX_S16 source1[] = {42, 23, 983, 7456, 124, 11111, 4554, 10002};
OMX_S16 source2[] = {242, 423, 9832, 746, 1124, 1411, 2254, 1298};

OMX_S32 source_dotproduct(void)
{
```

```
    OMX_INT len = sizeof(source1)/sizeof(OMX_S16);

    return omxSP_DotProd_S16(source1, source2, len);
}
```

Appendix A
Revisions

This appendix describes the technical changes between released issues of this book.

Table A-1 Issue A

Change	Location	Affects
First release	-	-