ARM NEON Development

By: Ali Nuhi

This guide will introduce the NEON subsystem as well as show how to develop NEON specific code.

Background

The NEON subsystem is an advanced SIMD (Single Instruction, Multiple Data) processing unit. This means that it can apply a single type of instruction to many pieces of data at one time in parallel. This is extremely helpful when it comes to media processing such as audio/video filters and codecs.

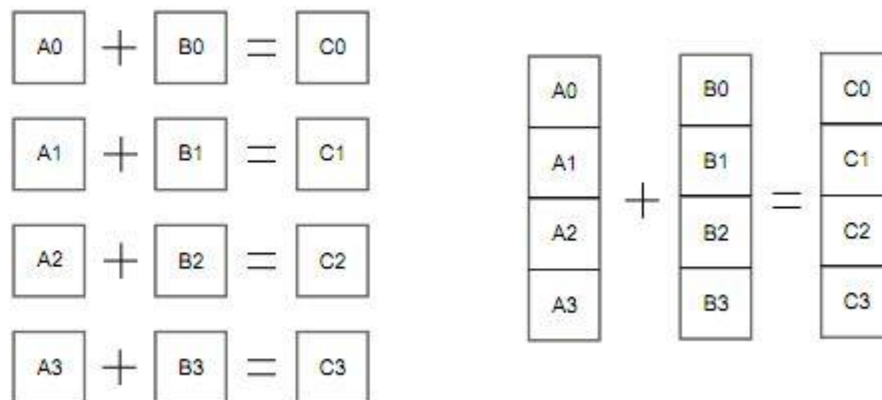
The NEON system is NOT the floating point unit of the ARM processor. There is separate FPU known as the VFP system. They use the same register space but this is taken care of by the compiler/kernel. There are a few differences between the NEON and VFP systems such as: NEON does not support double-precision floating point numbers, NEON only works on vectors and does not support advanced operations such as square root and divide.

NEON types supported:

- 32bit single precision floating point
- 64, 32, 16, 8bit signed and unsigned integers
- 16 and 8 bit polynomials

The convention used to distinguish types is to put the first letter of the type before the size. For example an unsigned 32bit int would be U32, 32 bit floating point would be F32 and so on.

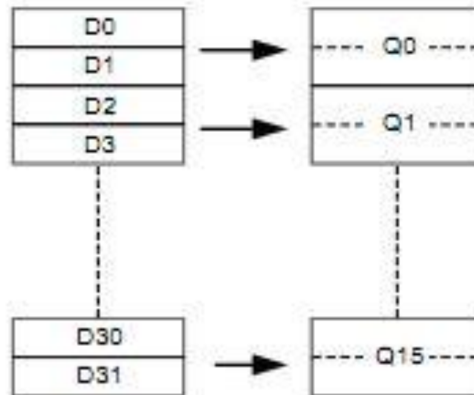
The main selling point of the NEON system is that it does vector math in parallel. Below is a visual representation of the differences.



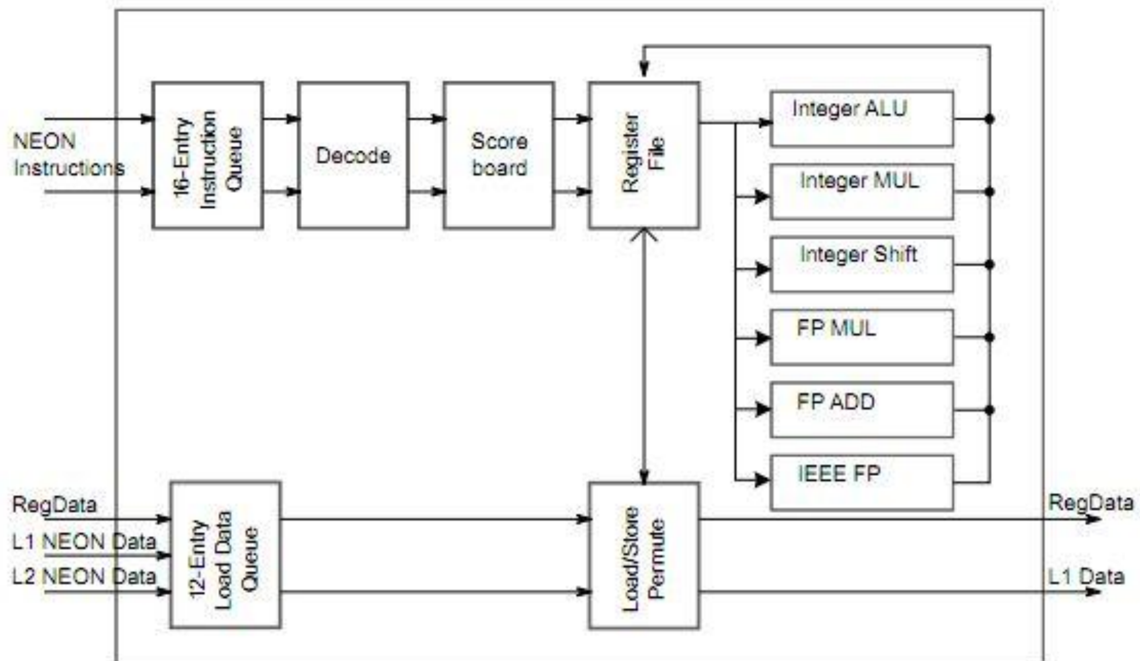
While a normal processor would do A0+ B0, A1+B1...the NEON system does this in one instruction producing the same output.

Architecture

The NEON system uses a bank of 32 64bit registers which can also be used as 16 128bit registers as seen below.



The NEON system architecture:



Development

NEON hardware specific code can be developed using assembly, C/C++ or various open source libraries.

Using NEON assembly can become quite messy because of issues with aligning data so unless you are using it for small segments it is not recommended.

Auto-vectorization:

The easiest way to utilize the NEON subsystem is to use the auto-vectorization feature in GCC for ARM. This is available in the compiler installed in the windows development guide (CodeSourcery lite). To use this feature you must add the following options to your command line compiling.

```
-ftree-vectorize
-mfpu=neon
```

This allows the compiler to vectorize applicable parts of your code which yields a speed increase. An example command line compile would look something like this.

```
$ arm-none-linux-gnueabi-gcc SAMPLE.c –mfpu=neon –ftree-vectorize –o SAMPLEPROGRAM
```

Libraries:

The advantage of these options is that you can write portable code that can be reconfigured at compile time but still utilize hardware when applicable.

The next easiest way to take advantage of this architecture is to utilize pre made NEON libraries. Two examples are Math-Neon - which have some complicated math functions that utilize NEON code and Eigen2 – a linear algebra/matrix library.

NEON Types in C:

The last method is to write your own code that uses special NEON C types. To do this you need to include the NEON header file. See Below.

```
#include <arm_neon.h>
```

The formatting for special types uses the following convention.

```
<basic type>x<number of elements>_t
```

Available NEON types in C.

64-bit type (D-register)	128-bit type (Q-register)
int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

For a simple program see ARM_neon_sample.c and the cortex A series programming guide.

```
/*

	ARM NEON sample code.
	By: ALI NUHI

	SIMD subsystem: single instruction, multiple data
	-run the same instruction on multiple piece of data

	neon registers:
	D0-D31 Registers: [...64bits...] OR Q0-Q15 Registers [...128bits...]
	So if you have 32bit numbers you can load data into the Q register like so
	[...32...|...32...|...32...|...32...] then do the same instruction such
	as a multiply on all the numbers at the same time.

	Important to realize the system sees each register as a vector of data.

	to compile: use the options in GCC
	gcc FILENAME.C -march=armv7-a -mtune=cortex-a8 -mfpu=neon -ftree-vectorize -mfloat-
abi=softfp -o OUTPUT
	if your using codesourcery gcc for cross compiling it would be
	arm-none-linux-gnueabi-gcc (or g++ for cpp)

	Dont forget to include the arm_neon.h file.~

	Example instruction:
	VADD.I8 D0, D1, D2
	this is doing a neon add and the type of numbers your adding is 8bit Ints
	D1:[ 1 | 2 | 3 | 4 | 5 |  6  | 7  | 8 ]
	             +
	D2:[ 1 | 2 | 3 | 4 | 5 |  6  | 7  | 8 ]
	             =
	D0:[ 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 ]
*/
```

```
#include <stdio.h>
#include <arm_neon.h>   //need to include this ifyou want to use intrinsics

int main(){

//vector addition 8x8 example.
      uint8x8_t vec_a, vec_b, vec_dest;  //a vector of 8 8bit ints
      vec_a = vdup_n_u8(9);
      vec_b = vdup_n_u8(10);

      vec_dest = vec_a * vec_b; //90
      vec_a = vec_dest * vec_b; //90*10 = 900
      vec_dest = vec_a * vec_b; //900*10 = 9000
      int i = 0;
      int result;

      result = vget_lane_u8( vec_dest, 0 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 1 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 2 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 3 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 4 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 5 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 6 );
      printf( "Lane %d: %d\n", i, result );
      i++;
      result = vget_lane_u8( vec_dest, 7 );
      printf( "Lane %d: %d\n", i, result );


}
```