Modern Assembly Language Programming with the ARM processor

Chapter 10: ARM NEON Extensions

Contents

NEON Overview

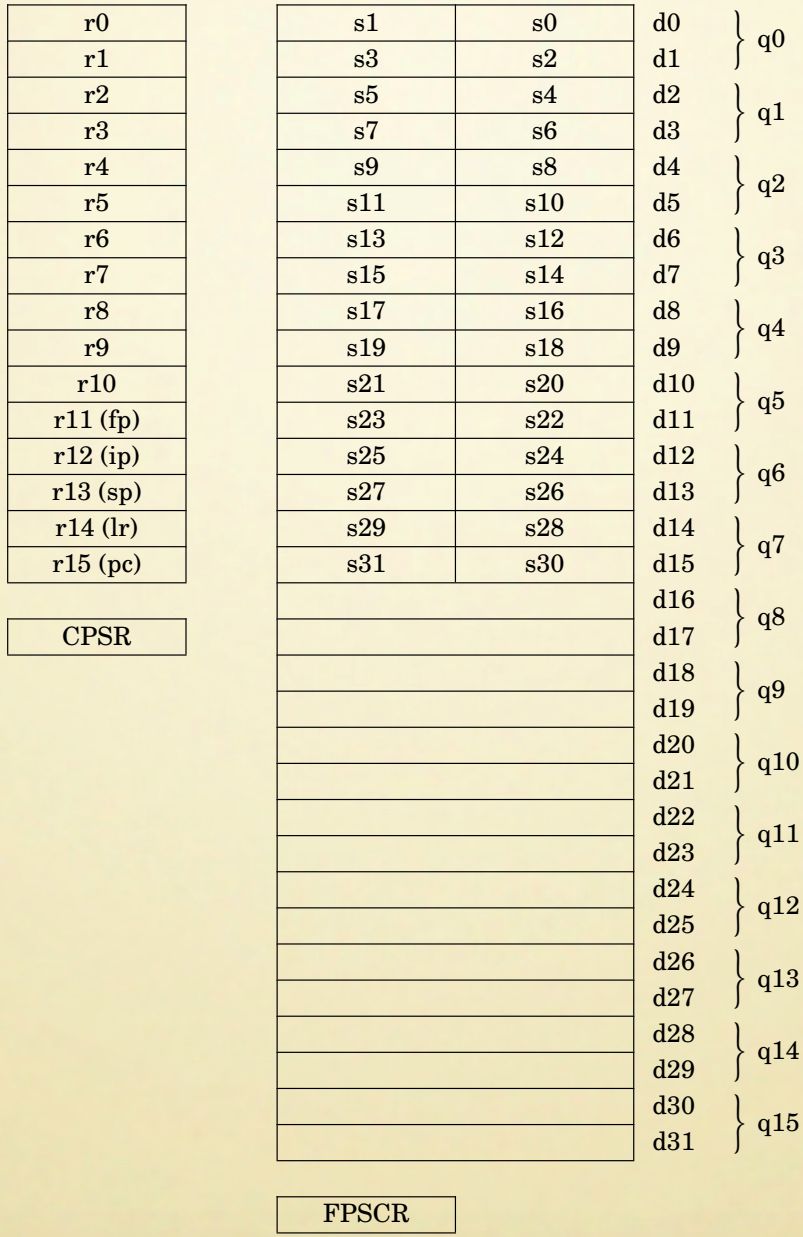
NEON

- extends the VFP instruction set with about 125 instructions and pseudo-instructions to support
  - integer,
  - fixed point, and
  - floating point,
- provides Single Instruction, Multiple Data (SIMD) operations,
- adds the ability to view the register set as sixteen 128-bit (quadruple-word) registers, named `q0` through `q15`, and
- deprecates the use of VFP vector mode.

Using vector mode on a NEON processor will result in ***extremely*** poor performance.

A single NEON instruction can operate on up to 128 bits, which may represent multiple integer, fixed point, or floating point numbers.

NEON and ARM Integer Registers



Registers

NEON:

- always provides the full set of 32 double-word registers
- views each register as containing a vector of 1, 2, 4, 8, or 16 elements, all of the same size and type
  - elements of each vector can also be accessed as scalars
  - a scalar can be 8 bits, 16 bits, 32 bits, or 64 bits
- instruction syntax is extended to refer to scalars using an index, `x`
  - `Dm[x]` is element `x` in register `Dm`
  - the size of the elements is given as part of the instruction
  - instructions that access scalars can access any element in the register bank

There are over 100 instructions, so we will only visit a few interesting ones. Refer to the textbook for others.

Pixel Data

Images may be stored in memory as an array of pixel structures:

```
1    typedef struct{
2      uint8_t red;
3      uint8_t green;
4      uint8_t blue;
5    }pixel;
```

Loading a group of eight pixels into three registers may result in this:

$green_2$	red_2	$blue_1$	$green_1$	red_1	$blue_0$	$green_0$	red_0	d0
red_5	$blue_4$	$green_4$	red_4	$blue_3$	$green_3$	red_3	$blue_2$	d1
$blue_7$	$green_7$	red_7	$blue_6$	$green_6$	red_6	$blue_5$	$green_5$	d2

But it may be better to load them like this:

red_7	red_6	red_5	red_4	red_3	red_2	red_1	red_0	d0
$green_7$	$green_6$	$green_5$	$green_4$	$green_3$	$green_2$	$green_1$	$green_0$	d1
$blue_7$	$blue_6$	$blue_5$	$blue_4$	$blue_3$	$blue_2$	$blue_1$	$blue_0$	d2

Load or Store Single Structure Using One Lane

```
v<op><n>.<size> <list>,[Rn{:<align>}]{!}
v<op><n>.<size> <list>,[Rn{:<align>}],Rm
```

- `<op>` must be either `ld` or `st`.
- `<n>` must be one of `1`, `2`, `3`, or `4`.
- `<size>` must be one of `8`, `16`, or `32`.
- `<list>` specifies the list of registers. There are four list formats:
  1. `{Dd[x]}`
  2. `{Dd[x], D(d+a)[x]}`
  3. `{Dd[x], D(d+a)[x], D(d+2a)[x]}`
  4. `{Dd[x], D(d+a)[x], D(d+2a)[x], D(d+3a)[x]}`

  where `a` can be either 1 or 2. Every register in the list must be in the range `d0-d31`.
- `Rn` is the ARM register containing the base address.
- `<align>` specifies an optional alignment.
- The optional `!` indicates that `Rn` is updated.
- `Rm` is an ARM register containing an offset from the base address. If `Rm` is present, `Rn` is updated to `Rn + Rm` after the address is used to access memory.

Load or Store Single Structure Using One Lane

NEON instructions can be quite complex!

Examples:

```
1    vld3.8    {d0[0],d1[0],d2[0]},[r0]! @ load first pixel in lane 0
2    vld3.8    {d0[1],d1[1],d2[1]},[r0]!
3    vld3.8    {d0[2],d1[2],d2[2]},[r0]!
4    vld3.8    {d0[3],d1[3],d2[3]},[r0]!
5    vld3.8    {d0[4],d1[4],d2[4]},[r0]!
6    vld3.8    {d0[5],d1[5],d2[5]},[r0]!
7    vld3.8    {d0[6],d1[6],d2[6]},[r0]!
8    vld3.8    {d0[7],d1[7],d2[7]},[r0]! @ load eigth pixel in lane 7
```

There are several other variations on the load, store, load multiple, and store multiple instructions.

Data Movement

NEON extends the `vmov` instruction to support scalars and quadwords.

```
1     @ 32 bit moves ( x can be 0 or 1 )
2     vmov.32  d0[0],r6     @ d0[0] <- r6
3     vmov.f32 r7,d1[1]     @ r7 <- d1[1]
4     vmov.u32 r8,d2[0]     @ r8 <- d2[0]
5     vmoveq.s32 r9,d2[1]   @ if eq, r9 <- d2[1]
6
7     @ 16 bit moves ( x can be 0, 1, 2, or 3)
8     vmov.16  d0[1],r8     @ d0[1] <- r8
9     @ (least significant 16 bits)
10    vmov.s16 r7,d1[2]     @ r7 <- d1[2] (sign extend)
11
12    @ 8 bit moves ( x can be 0, 1, 2, 3, 4, 5, 6, or 7)
13    vmov.8  d0[5],r6     @ d0[5] <- r6
14    @ (least significant 8 bits)
15    vmov.u8 r5,d1[5]     @ r5 <- d1[5] (no sign extend)
```

Other Data Moves

NEON also adds

- the ability to perform one's complement during a move,
- instructions for moving immediate data into a register,
- the ability to copy duplicate a scalar across all scalars in a register,
- instructions for swapping and re-ordiring vector elements in several ways, and
- instructions to move between scalars and ARM integer registers.

Integer Comparison

NEON adds the ability to perform integer comparisons between vectors.

- the comparison instructions set one element in a result vector for each pair of items.
- each element of the result vector will have every bit set to zero (for false) or one (for true).
- if the elements of the result vector are interpreted as signed two's-complement numbers, then
  - the value 0 represents false and
  - the value -1 represents true.

Examples:

```
1    vceq.i8  d0,d1,d2  @ 8 8-bit comparisons
2    vcge.s16 d0,d1,d2  @ 4 16-bit signed comparisons
3    vcgt.u16 q0,q1,q2  @ 8 16-bit unsigned comparisons
4    vcle.f32 d0,d1,d2  @ 2 single precision comparisons
5    vclt.f32 q0,q1,q2  @ 4 single precision comparisons
6    vceq.i8  q0,q1,#0  @ 16 8-bit comparisons
7    vcge.s16 d0,d1,#0  @ 8 8-bit signed comparisons
8    vcgt.f32 d0,d1,#0  @ 2 single precision comparisons
```

There are several more variations on the vector compare instructions.

Logical Operations

NEON includes vector versions of the following five logical operations:

- `vand` Bitwise AND,
- `veor` Bitwise Exclusive-OR,
- `vorr` Bitwise OR,
- `vorn` Bitwise Complement and OR, and
- `vbic` Bit Clear.

All of them involve two source operands and a destination register.
Examples:

```
1    vand.i64  q0,q1,q2    @ q0=q1 & q2
2    vbic.i32  d3,d3,d5    @ if (eq) then d3=d3 & !d4
3    vorr.i8   q0,q1,q2    @ q0=q1 | q2
4    vorr.i64  q0,q1,q2    @ q0=q1 | q2
```

There are also bitwise instructions which use immediate data, and instructions which insert selected bits from one register into another.

Shift Instructions

The NEON shift instructions operate on vectors.
These instructions shift each element in a vector left by an immediate value:

`vshl` Shift Left Immediate,

`vqshl` Saturating Shift Left Immediate,

`vqshlu` Saturating Shift Left Immediate Unsigned, and

`vshll` Shift Left Immediate Long.

Examples:

```
1     vshl.s16    q1,q6,#4   @ shift each 16-bit word left
2     vqshl.u8    d1,d6,#1   @ Multiply each byte by two
```

There are also instructions for shifting right and for shifting by a variable amount, as well as instructions for shifting and combining bits from two sources.

Add and Subtract

The following eight instructions perform vector addition and subtraction:

`vadd`	Add
`vqadd`	Saturating Add
`vaddl`	Add Long
`vaddw`	Add Wide
`vsub`	Subtract
`vqsub`	Saturating Subtract
`vsubl`	Subtract Long
`vsubw`	Subtract Wide

Examples:

```
1     vadd.s8     q1,q6,q8   @ Add elements
2     vqadd.s8    q1,q6,q8   @ Add elements and saturate
```

There are also instructions for producing results that are narrower or wider than the source elements, dividing the results by two, adding elements of a vector pairwise, computing the absolute difference, computing absolute value, negating elements, and selecting maximum and minimum elements.

Multiplication

These instructions are used to multiply the corresponding elements from two vectors:

- `vmul` Multiply
- `vmla` Multiply Accumulate
- `vmls` Multiply Subtract
- `vmull` Multiply Long
- `vmlal` Multiply Accumulate Long
- `vmlsl` Multiply Subtract Long

Examples

```
1     vmul.i8     q1,q6,q8   @ Multiply elements
2     vmlal.s8    q0,d4,d5   @ Multiply-accumulate long
```

There are also instructios for multplying each element of a vector by a scalar, performing a multiply-accumulate, and for discarding the high or low half of the result.

Division

There is no NEON instruction for division, but the VFP instructions are available for dividing single and double precision scalars.

NEON vector division is accomplished by computing the reciprocal of the divisor(s) and performing a multiplication.

These instructions perform the initial estimates of the reciprocal values:

`vrecpe` Reciprocal Estimate

`vrsqrte` Reciprocal Square Root Estimate

Examples:

```
1    vrecpe.u32 q1,q6  @ Get initial reciprocal estimates
2    vrecpe.f32 d4,d5  @ Get initial reciprocal estimates
```

These instructions are used to perform one Newton-Raphson step for improving the reciprocal estimates:

`vrecps` Reciprocal Step

`vrsqrts` Reciprocal Square Root Step

Division Example

```
1     @ Divide elements of q0 by elements of q1 and store in q3
2     @ Doing a loop and testing for convergence would be slow,
3     @ so we will just do two improvement steps and hope it is
4     @ close enough.
5     vrecpe.f32   q3,q1    @ Get initial reciprocal estimates
6     vrecps.f32   q4,q1,q3 @ Improve estimates
7     vmul.f32     q3,q3,q4 @ Finish improvement step
8     vrecps.f32   q4,q1,q3 @ Improve estimates
9     vmul.f32     q3,q3,q4 @ Finish improvement step
10    vmul.f32     q3,q3,q0 @ Perform division
```

NEON Single Precision Sine

```
@@@ sin_N_f implements the sine function using NEON single
@@@ precision floating point. It computes sine by summing
@@@ the first 7 terms of the Taylor series.
@@@ ------------------------------------------------------------
        .data
        @@ The following is a table of constants used in the
        @@ Taylor series approximation for sine
        .align  8               @ Align to cache (256-byte boundary)
ctab:   .word   0x3F800000      @  1.000000000000000
        .word   0xBE2AAAAB      @ -0.166666671633720
        .word   0x3C088889      @  0.008333333767951
        .word   0xB9500D01      @ -0.000198412701138
        .word   0x3638EF1D      @  0.000002755731884
        .word   0xB2D7322B      @ -0.000000025052108
 @@ ------------------------------------------------------------
        .text
        .align 2
        .global sin_N_f
sin_N_f:
        @@ Load the entire table into d16-d18
        ldr             r0,=ctab
        vldmia          r0,{d16-d18}
```

NEON Single Precision Sine

```
1        @@ Calculate vectors holding powers of x as follows:
2        @@ d0 <- x, x^3
3        @@ d1 <- x^5, x^7
4        @@ d2 <- x^9, x^11
5        vmul.f32        s8,s0,s0    @ Put x^2 in s8 (d4[0])
6        vmul.f32        s9,s8,s8    @ Put x^4 in s9 (d4[1])
7        vmul.f32        s1,s8,s0    @ Put x^3 in s1 (d0[1])
8        vmov.f32        s8,s9       @ d4 <- 2 copies of x^4
9        vmul.f32        d1,d0,d4    @ Get x^5 and x^7
10       vmul.f32        d3,d0,d16   @ Do first 2 multiplies
11       vmul.f32        d2,d1,d4    @ Get x^9, x^11
12       vmla.f32        d3,d1,d17   @ Accumulate 2 multiplies
13       vmla.f32        d3,d2,d18   @ Accumulate last 2 multiplies
14       vadd.f32        s0,s6,s7    @ Final addition
15       mov             pc,lr       @ Return result in s0
```

NEON Double Precision Sine

```
1  @@@ sin_N_d implements the sine function using NEON double
2  @@@ precision floating point by summing the first ten terms of the
3  @@@ Taylor series.
4  @@@ Versions of NEON before ARMv8 do not support vectors of
5  @@@ double precision floating point, but we can use loop
6  @@@ unrolling and lots of registers to get good performance.
7  @@@ ----------------------------------------------------------
8          .data
9          @@ The following is a table of constants used in the
10         @@ Taylor series approximation for sine
11         .align  8               @ Align to cache (256-byte boundary)
12 ctab:   .word   0x55555555,0xBFC55555 @ -0.166666666666667
13         .word   0x11111111,0x3F811111 @  0.008333333333333
14         .word   0x1A01A01A,0xBF2A01A0 @ -0.000198412698413
15         .word   0xA556C734,0x3EC71DE3 @  0.000002755731922
16         .word   0x67F544E4,0xBE5AE645 @ -0.000000025052108
17         .word   0x13A86D09,0x3DE61246 @  0.000000000160590
18         .word   0xE733B81F,0xBD6AE7F3 @ -0.000000000000765
19         .word   0x7030AD4A,0x3CE952C7 @  0.000000000000003
20         .word   0x46814157,0xBC62F49B @ -0.000000000000000
21 @@@ ----------------------------------------------------------
22         .text
23         .align 2
```

NEON Double Precision Sine

```
1         .global sin_N_d
2 sin_N_d:
3         ldr       r0,=ctab      @ Load pointer to coefficients
4         vmul.f64  d5,d0,d0      @ Put x^2 in d5
5         vmov      d2,d0         @ Copy x to d2
6         vldmia    r0!,{d4}      @ load first coefficient
7         vmul.f64  d3,d5,d0      @ Put x^3 in d3
8         vmul.f64  d5,d5,d5      @ Put x^4 in d5
9         vldmia    r0!,{d24,d25} @ load 2 more coefficients
10        vmla.f64  d0,d3,d4      @ d0 <- x - ((x^3)/3!) = t_1 + t_2
11        vmul.f64  d6,d5,d5      @ Put x^8 in d6
12        vmul.f64  d16,d2,d5     @ d16 <- x^5 (x*x^4)
13        vmul.f64  d17,d3,d5     @ d17 <- x^7 (x^3*x^4)
14        vldmia    r0!,{d26,d27} @ load 2 more coefficients
15        vmul.f64  d18,d2,d6     @ d18 <- x^9 ( x*x^8)
16        vmul.f64  d19,d3,d6     @ d19 <- x^11 (x^3*x^8)
17        vmul.f64  d20,d16,d6    @ d20 <- x^13 (x^5*x^8)
18        vmul.f64  d21,d17,d6    @ d21 <- x^15 (x^7*x^8)
19        vldmia    r0!,{d28-d29} @ load 2 more coefficients
20        vmul.f64  d22,d18,d6    @ d22 <- x^17 (x^9*x^8)
21        vmul.f64  d23,d19,d6    @ d23 <- x^19 (x^11*x^8)
22        @@ Calculate all of the remaining terms
23        vmul.f64  d16,d16,d24   @ d16 <- (x^5)/5! = t3
```

NEON Double Precision Sine

```
1         vmul.f64  d17,d17,d25   @ d17 <- -(x^7)/7! = t4
2         vldmia    r0!,{d30,d31} @ load 2 more coefficients
3         vmul.f64  d18,d18,d26   @ d18 <- (x^9)/9! = t5
4         vmul.f64  d19,d19,d27   @ d19 <- -(x^11)/11! = t6
5         vmul.f64  d20,d20,d28   @ d20 <- (x^13)/13! = t7
6         vmul.f64  d21,d21,d29   @ d21 <- -(x^15)/15! = t8
7         vmul.f64  d22,d22,d30   @ d22 <- (x^17)/17! = t9
8         vmul.f64  d23,d23,d31   @ d23 <- -(x^19)/19! = t10
9         @@ Sum all of the terms
10        vadd.f64  d16,d16,d17   @ d16 <- t_3 + t_4
11        vadd.f64  d17,d18,d19   @ d17 <- t_5 + t_6
12        vadd.f64  d18,d20,d21   @ d18 <- t_7 + t_8
13        vadd.f64  d19,d22,d23   @ d19 <- t_9 + t_10
14        vadd.f64  d16,d16,d17   @ d16 <- t_3 + t_4 + t_5 + t_6
15        vadd.f64  d17,d18,d19   @ d17 <- t_7 + t_8 + t_9 + t_10
16        vadd.f64  d16,d16,d17   @ d16 <- sum of t_3 to t_10
17        vadd.f64  d0,d0,d16     @ final sum
18        mov       pc,lr
```

Performance Comparison

Optimization	Implementation	CPU seconds
None	Single Precision VFP scalar Assembly	1.74
	Single Precision VFP vector Assembly	27.09
	Single Precision NEON Assembly	1.32
	Single Precision C	4.36
	Double Precision VFP scalar Assembly	2.83
	Double Precision VFP vector Assembly	106.46
	Double Precision NEON Assembly	2.24
	Double Precision C	4.59
Full	Single Precision VFP scalar Assembly	1.11
	Single Precision VFP vector Assembly	27.15
	Single Precision NEON Assembly	0.96
	Single Precision C	1.69
	Double Precision VFP scalar Assembly	2.56
	Double Precision VFP vector Assembly	107.53
	Double Precision NEON Assembly	2.05
	Double Precision C	4.27

Summary

- NEON can dramatically improve performance of algorithms that can take advantage of data parallelism.
- Compiler support for automatically vectorizing and using NEON instructions is still immature.
- A careful assembly language programmer can usually beat the compiler, sometimes by a wide margin.