Modern Assembly Language Programming with the ARM processor

Chapter 9: ARM Vector Floating Point Processor

1. ARM VFP
2. Load/Store Instructions
3. Data Processing Instructions
4. Data Movement Instructions
5. Data Conversion Instructions
6. Floating Point Sine Function

VFP Versions

VFPv1: Obsolete

VFPv2: An optional extension to the ARMv5 and ARMv6 processors. VFPv2 has 16 64-bit FPU registers.

VFPv3: An optional extension to the ARMv7 processors. It is backwards compatible with VFPv2, except that it cannot trap floating-point exceptions. VFPv3-D32 has 32 64-bit FPU registers. Some processors have VFPv3-D16, which supports only 16 64-bit FPU registers. VFPv3 adds several new instructions to the VFP instruction set.

VFPv4: Implemented on some Cortex ARMv7 processors. VFPv4 has 32 64-bit FPU registers. It adds both half-precision extensions and multiply-accumulate instructions to the features of VFPv3. Some processors have VFPv4-D16, which supports only 16 64-bit FPU registers.

Additional Registers

Core register
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11 (fp)
r12 (ip)
r13 (sp)
r14 (lr)
r15 (pc)

CPSR

s1	s0	d0	Bank 0
s3	s2	d1	
s5	s4	d2	
s7	s6	d3	
s9	s8	d4	Bank 1
s11	s10	d5	
s13	s12	d6	
s15	s14	d7	
s17	s16	d8	Bank 2
s19	s18	d9	
s21	s20	d10	
s23	s22	d11	
s25	s24	d12	Bank 3
s27	s26	d13	
s29	s28	d14	
s31	s30	d15	
		d16	Bank 4
		d17	
		d18	
		d19	
		d20	Bank 5
		d21	
		d22	
		d23	
		d24	Bank 6
		d25	
		d26	
		d27	
		d28	Bank 7
		d29	
		d30	
		d31	

FPSCR

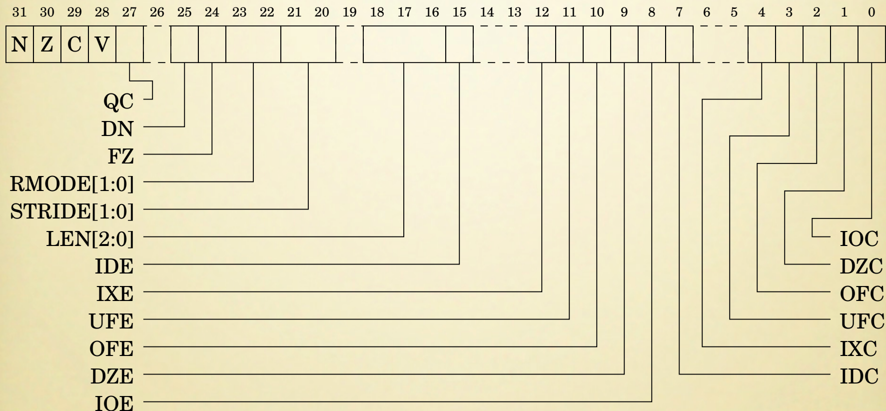
Overview

- Adds about 23 new instructions (depending on version).
- Instructions are provided to:
  - transfer floating point values between VFP registers,
  - transfer floating-point values between the VFP coprocessor registers and main memory,
  - transfer 32-bit values between the VFP coprocessor registers and the ARM integer registers,
  - perform addition, subtraction, multiplication and division, involving two source registers and a destination register,
  - compute the square root of a value,
  - perform combined multiply-accumulate operations,
  - perform conversions between various integer, fixed point, and floating point representations, and
  - compare floating-point values.

Register Rules

- Registers `d0` through `d7` are *volatile*. They are used for passing arguments, returning results, and for holding local variables. They do not need to be preserved by subroutines.
- Registers `d8` through `d15` are *non-volatile*. The contents of these registers must be preserved across subroutine calls.
- Registers `d16` through `d31` (if present) are also considered *volatile*.

FPSCR



FPSCR – Most Important Bits

N	The Negative flag is set to one by `vcmp` if Fd<Fm.
Z	The Zero flag is Set to one by `vcmp` if Fd=Fm.
C	The Carry flag is set to one by `vcmp` if Fd=Fm, or Fd>Fm, or Fd and Fm are unordered.
V	The oVerflow flag is set to one by `vcmp` if Fd and Fm are unordered.
RMODE	Rounding mode:

- 00 Round to Nearest (RN).
- 01 Round towards Plus infinity (RP).
- 10 Round towards Minus infinity (RM).
- 11 Round towards Zero (RZ).

FPSCR – RunFast Mode

DN Default NaN enable:

- 0: Disable Default NaN mode. NaN operands propagate through to the output of a floating-point operation.
- 1: Enable Default NaN mode. Any operation involving one or more NaNs returns the default NaN.

Default NaN mode does not comply with IEEE 754 standard, but may increase performance.

FZ Flush-to-Zero enable:

- 0: Disable Flush-to-Zero mode.
- 1: Enable Flush-to-Zero mode.

Flush-to-Zero mode replaces subnormal numbers with 0. This does not comply with IEEE 754 standard, but may increase performance.

RunFast Mode: When DN=1, FZ=1, and all exceptions disabled (IDE through IOE all set to zero).

- Higher Performance
- Not IEEE-754 compliant

FPSCR – Vector Mode

STRIDE Sets the stride (distance between items) for vector operations:

00	Stride is 1.
01	Reserved.
10	Reserved.
11	Stride is 2.

LEN Sets the vector length for vector operations:

000	Vector length is 1 (scalar mode).
001	Vector length is 2.
010	Vector length is 3.
011	Vector length is 4.
100	Vector length is 5.
101	Vector length is 6.
110	Vector length is 7.
111	Vector length is 8.

If LEN is not zero, then certain instructions will operate on vectors.

Scalar Mode

```
Op   Fd,Fn,Fm
Op   Fd,Fm
```

- the LEN field is set to zero (scalar mode), or
- the destination operand, `Fd`, is in Bank 0 or Bank 4.

The operation acts on `Fm` (and `Fn` if the operation uses two operands) and places the result in `Fd`.

Mixed Mode

```
Op   Fd,Fn,Fm
Op   Fd,Fm
```

- the LEN field is not set to zero, and
  - `Fm` is in Bank 0 or Bank 4, but
  - `Fd` is not.

If the operation has only one operand, then the operation is applied to `Fm` and copies of the result are stored into each register in the destination vector.

If the operation has two operands, then it is applied with the scalar `Fm` and each element in the vector starting at `Fn`, and the result is stored in the vector beginning at `Fd`.

Vector Mode

```
Op   Fd,Fn,Fm
Op   Fd,Fm
```

- the LEN field is not set to zero, and
- neither `Fd` nor `Fm` is in Bank 0 or Bank 4.

If the operation has only one operand, then the operation is applied to the vector starting at `Fm` and the results are placed in the vector starting at `Fd`.

If the operation has two operands, then it is applied with corresponding elements from the vectors starting at `Fm` and `Fn`, and the result is stored in the vector beginning at `Fd`.

Load/Store Single Register

- Operations:

  vldr Load VFP Register, and

  vstr Store VFP Register.

- Syntax:

```
v<op>r{<cond>}{.<prec>}  Fd, [Rn{,#offset}]
v<op>r{<cond>}{.<prec>}  Fd, =label
```

  - `<op>` may be either `ld` or `st`.
  - `Fd` may be any single or double precision register.
  - `Rn` may be any ARM integer register.
  - `<cond>` is an optional condition code.
  - `<prec>` may be either `f32` or `f64`.

- Examples:

```
1    vldr        s5, [r0]    @ load s5 from address in r0
2    vstr.f64    d4, [r2]    @ store d4 using address in r2
3    vstreq.f32  s0, [r1]    @ if eq condition is true,
4                            @ store s0 using address in r1
```

Load/Store Multiple Register

- Operations:

  vldm Load Multiple VFP Registers, and

  vstm Store Multiple VFP Registers.

- Syntax:

```
v<op>m<mode>{<cond>}{.<prec>}  Rn{!},<list>
vpush{<cond>}{.<prec>}         <list>
vpop{<cond>}{.<prec>}          <list>
```

  - `<op>` may be either `ld` or `st`.
  - `<mode>` is one of

    `ia` Increment address after each transfer.

    `db` Decrement address before each transfer.
  - `Rn` may be any ARM integer register.
  - `<cond>` is an optional condition code.
  - `<prec>` may be either `f32` or `f64`.
  - `<list>` may be any set of *contiguous* single precision registers, or any set of *contiguous* double precision registers.
  - If `mode` is `db` then the `!` is required.

Load/Store Multiple Register Continued

- **Examples:**

```
1     vstmdb    sp!,{s0-s3}     @ Store s0 through s3 on stack
2     vstmia    r1,{s0-s31}     @ Store all fp registers
3                               @ at address in r1
4     vldmia    sp!,{d4-d7}     @ Pop four doubles from the stack
5     vldmiaeq  sp!,{d4-d7}     @ If eq, then pop four doubles
6                               @ from the stack
```

Copy, Absolute Value, Negate, and Square Root

- Operations:

vcpy	Copy VFP Register (equivalent to move),
vabs	Absolute Value,
vneg	Negate, and
vsqrt	Square Root,

- Syntax:

```
v<op>{<cond>}.<prec>  Fd, Fm
```

  - `<op>` is one of `cpy`, `abs`, `neg`, or `sqrt`.
  - `<cond>` is an optional condition code.
  - `<prec>` may be either `f32` or `f64`.

- Examples:

```
1    vabs    d3, d5    @ Store absolute value of d1 in d3
2    vnegmi  s15, s15  @ if mi, then negate s15
```

Add, Subtract, Multiply, and Divide

- Operations:

vadd	Add,
vsub	Subtract,
vmul	Multiply,
vnmul	Negate and Multiply, and
vdiv	Divide.

- Syntax:

```
v<op>{<cond>}.<prec>  Fd, Fn, Fm
```

  - `<op>` is one of `add`, `sub`, `mul`, `nmul`, or `div`.
  - `<cond>` is an optional condition code.
  - `<prec>` may be either `f32` or `f64`.

- Examples:

```
1    vadd.f64   d0, d1, d2     @ d0 <- d1 + d2
2    vaddgt.f32 s0, s1, s2     @ if (gt) then s0 <- s1 + s2
3    vnmul.f32  s10, s10, s14 @ s10 <- -(s10 * s14)
4    vdivlt.f64 d0, d7, d8     @ if lt, then d0 <- d7 / d8
```

Compare

The compare instruction subtracts the value in `Fm` from the value in `Fd` and set the flags in the FPSCR based on the result.

- Operations:

  vcmp   Compare, and

  vcmpe   Compare with Exception.

- Syntax:

```
    vcmp{e}{<cond>}.<prec>    Fd, Fm
```

  - If `e` is present, an exception is raised if either operand is any kind of NaN. Otherwise, an exception is raised only if either operand is a signaling NaN.
  - `<cond>` is an optional condition code.
  - `<prec>` may be either `f32` or `f64`.

- Examples:

```
1     vcmp.f32     s0, s1     @ Subtract s1 from s0 and set
2                             @ FPSCR flags
```

Moving Between Two VFP Registers

- Operations:

  vmov   Move Between VFP Registers.

- Syntax:

```
vmov{<cond>}{.<prec>} Fd, Fm
```

  - F can be s or d.
  - Fd and Fm must be the same size.
  - <cond> is an optional condition code.
  - <prec> is either f32 or f64.

- Examples:

```
1    vmov.f64 d3,d4            @ d3 <- d4
2    vmov.f32 s5,s12           @ s5 <- s12
```

Moving Between VFP and Single ARM Register

- Operations:

  vmov   Move Between VFP and One ARM Integer Register.

- Syntax:

```
    vmov{<cond>}    Rd,  Sn
    vmov{<cond>}    Sn,  Rd
```

  - `Rd` is an ARM integer register.
  - `Sd` is a VFP single precision register.
  - `<cond>` is an optional condition code.

- Examples:

```
1     vmov r3,s4            @ r2 <= s4
2     vmov s12,r8           @ s12 <- r8
```

Moving Between VFP and Two ARM Registers

- Operations:

  vmov Move Between VFP and Two ARM Integer Registers.

- Syntax:

```
vmov{<cond>}   destination(s), source(s)
```

  - Source and destination must be VFP or integer registers. The following table shows the possible choices for sources and destinations.

ARM Integer	Floating Point
Rl,Rh	Dd
	Sd,Sd'

  - Sd and Sd' must be adjacent, and Sd' must be the higher-numbered register.
  - <cond> is an optional condition code.

- Examples:

```
1    vmov d9,r0,r1          @ d9 <- r1:r0
2    vmov r2,r3,d12         @ r3:r2 <- d12
3    vmov s1,s2,r2,r4       @ s1 <- r2, s2 <- r4
4    vmov r5,r7,s0,s1       @ r1 <- s0, r7 <- s1
```

Between ARM Register and VFP System Register

There are two instructions which allow the programmer to examine and change bits in the VFP system register(s):

- Operations:

  vmrs Move From VFP System Register to ARM Register, and
  vmsr Move From ARM Register to VFP System Register.

  User programs should only access the FPSCR to check the flags and control vector mode.

- Syntax:

```
vmrs{<cond>} Rd, VFPsysreg
vmsr{<cond>} VFPsysreg, Rd
```

  - `VFPsysreg` can be any of the VFP system registers.
  - `Rd` can be `APSR_nzcv` or any ARM integer register.,
  - `<cond>` is an optional condition code.

- Examples:

```
1    vmrs   APSR_nzcv,fpscr   @ Copy flags from FPSCR to CPSR
2    vmrs   r3, FPSCR         @ Copy FPSCR to ARM register r3
3    vmsr   FPSCR,r5          @ Copy R5 to FPSCR
```

Convert Between Floating Point and Integer

- Operations:

  vcvt Convert Between Floating Point and Integer

  vcvtr Convert Floating Point to Integer with Rounding

- Syntax:

```
vcvt{r}{<cond>}.<type>.f64   Sd, Dm
vcvt{r}{<cond>}.<type>.f32   Sd, Sm
vcvt{<cond>}.f64.<type>      Dd, Sm
vcvt{<cond>}.f32.<type>      Sd, Sm
```

  - The optional `r` makes the operation use the rounding mode specified in the FPSCR. The default is to round toward zero.
  - `<cond>` is an optional condition code.
  - The `<type>` can be either `u32` or `s32` to specify unsigned or signed integer.
  - These instructions can also convert to from fixed point to floating point if combined with an appropriate `vmul`.

Convert Between Floating Point and Integer Cont.

- **Examples:**

```
1      vcvt.f64.u32   d5, s7  @ Convert unsigned integer to double
2      vcvt.f64.f32   d0, s4  @ Convert signed integer to double
3      vcvt.u32.f64   s0, d7  @ Convert double to unsigned integer
4      vcvt.s32.f64   s1, d4  @ Convert double to signed integer
5      @@ Convert s10 to an S(15,16)
6      consta:      .float  65536.0
7      ⋮
8      vldr.f32     s11,consta   @ Load floating point constant
9      vmul.f32     s10,s10,s11  @ Multiply equates to shift
10     vcvt.s32.f32 s10,s10      @ Convert single to S(15,16)
```

Convert Between Fixed Point and Single Precision

- Operations:

  vcvt Convert To or From Fixed Point.

- Syntax:

```
vcvt{<cond>}.<td>.f32   Sd, Sm, #fbits
vcvt{<cond>}.f32.<td>   Sd, Sm, #fbits
```

  - `<cond>` is an optional condition code.
  - `<td>` specifies the type and size of the fixed point number, and must be one of the following:

s32	signed 32 bit value,
u32	unsigned 32 bit value,
s16	signed 16 bit value, or
u16	unsigned 16 bit value.

  - `#fbits` specifies the number of fraction bits in the fixed point number, and must be less than or equal to the size of the fixed point number indicated by `<td>`.

- Examples:

```
1   vcvt.f32.u16  s0,s0,#4 @ Convert from U(12,4) to single
2   vcvt.s32.f32  s1,s1,#8 @ Convert from single to S(23,8)
```

sin x Using IEEE Single Precision

```
 1          .data
 2          @@ The following is a table of constants used in the
 3          @@ Taylor series approximation for sine
 4          .align  5                @ Align to cache
 5  ctab:   .word   0xBE2AAAAA       @ -1.666666e-01
 6          .word   0x3C088889       @  8.333334e-03
 7          .word   0xB9500D00       @ -1.984126e-04
 8          .word   0x3638EF1D       @  2.755732e-06
 9          .word   0xB2D7322A       @ -2.505210e-08
10  @@@ --------------------------------------------------------------
11          .text
12          .align 2
13          @@ sin_a_f implements the sine function using IEEE single
14          @@ precision floating point. It computes sine by summing
15          @@ the first six terms of the Taylor series.
16          .global sin_a_f
17  sin_a_f:
18          @@ set runfast mode and rounding to nearest
19          fmrx    r1, fpscr        @ get FPSCR contents in r1
20          bic     r2, r1, #(0b1111<<23)
21          orr     r2, r2, #(0b1100<<23)
22          fmxr    fpscr, r2 @ store in FPSCR
23          @@ initialize variables
24          vmul.f32        s1,s0,s0 @ s1 <- x^2
25          vmul.f32        s3,s1,s0 @ s3 <- x^3
26          ldr             r0,=ctab @ load pointer to coefficients
27          mov             r3,#5    @ load loop counter
```

$\sin x$ Using IEEE Single Precision

```
1  loop:   vldr.f32        s4,[r0]   @ load coefficient
2          add             r0,r0,#4 @ increment pointer
3          vmul.f32        s4,s3,s4 @ s4 <- next term
4          vadd.f32        s0,s0,s4 @ add term to result
5          subs            r3,r3,#1 @ decrement and test loop count
6          vmulne.f32      s3,s1,s3 @ s4 <- x^2n
7          bne             loop     @ loop five times
8          @@ restore original FPSCR
9          fmxr    fpscr, r1
10         mov     pc,lr
```

$\sin x$ Using IEEE Single Precision Vector Mode

```
1          .data
2          .align  6                  @ Align to cache
3  ctab:   .word   0xBE2AAAAB         @ -1.666667e-01
4          .word   0x3C088889         @  8.333334e-03
5          .word   0xB9500D01         @ -1.984127e-04
6          .word   0x3638EF1D         @  2.755732e-06
7          .word   0xB2D7322B         @ -2.505211e-08
8  @@@ --------------------------------------------------------------
9          .text
10         .align 2
11         .global sin_v_f
12 sin_v_f:@@ set runfast mode and rounding to nearest
13         vmrs     r1, fpscr         @ get FPSCR contents in r1
14         bic      r2, r1, #(0b1111<<23)
15         orr      r2, r2, #(0b1100<<23)
16         vmsr     fpscr, r2         @ store settings in FPSCR
17         vmul.f32 s1,s0,s0          @ s1 = x^2
18         ldr      r0,=ctab          @ get address of coefficients
19         vldmia   r0!,{s16-s20}     @ load all coefficients into Bank 2
20         vmul.f32 s8,s0,s1          @ s8 = x^3
21         vmul.f32 s9,s8,s1          @ s9 = x^5
22         vmul.f32 s10,s9,s1         @ s10 = x^7
23         vmul.f32 s11,s10,s1        @ s11 = x^9
24         vmul.f32 s12,s11,s1        @ s12 = x^11
```

$\sin x$ Using IEEE Single Precision Vector Mode

```
1         @@ Set VFP for vector mode
2         bic      r2, r2, #(0b11111<<16) @ set rounding, stride to 1,
3         orr      r2, r2, #(0b00100<<16) @ and vector length to 5
4         vmsr     fpscr, r2      @ store settings in FPSCR
5         vmul.f32 s24,s8,s16     @ VECTOR operation x^(2n+1) * coeff[n]
6         vmsr     fpscr, r1      @ restore original FPSCR
7         @@ Add terms in Bank 3 to the result in s0
8         vadd.f32 s24,s24,s25
9         vadd.f32 s26,s26,s27
10        vadd.f32 s0,s0,s24
11        vadd.f32 s26,s26,s28
12        vadd.f32 s0,s0,s26
13        mov      pc,lr
```

sin x Using IEEE Double Precision Vector Mode

```
 1          .data
 2          @@ The following is a table of constants used in the
 3          @@ Taylor series approximation for sine
 4          .align 7        @ Align for efficient caching
 5  ctab:   .word 0x55555555, 0xBFC55555     @ -1.666666666666667e-01
 6          .word 0x11111111, 0x3F811111     @  8.333333333333333e-03
 7          .word 0x1A01A01A, 0xBF2A01A0     @ -1.984126984126984e-04
 8          .word 0xA556C734, 0x3EC71DE3     @  2.755731922398589e-06
 9          .word 0x67F544E4, 0xBE5AE645     @ -2.505210838544172e-08
10          .word 0x13A86D09, 0x3DE61246     @  1.605904383682161e-10
11          .word 0xE733B81F, 0xBD6AE7F3     @ -7.647163731819816e-13
12          .word 0x7030AD4A, 0x3CE952C7     @  2.811457254345521e-15
13          .word 0x46814157, 0xBC62F49B     @ -8.220635246624329e-18
14
15  @@@ -------------------------------------------------------------
16          .text
17          .align 2
18          @@ sin_a_d implements the sine function using IEEE
19          @@ double precision floating point.  It takes advantage
20          @@ of the ARM VFP vector processing instructions and
21          @@ computes sine by summing the first ten terms of the
22          @@ Taylor series.
23          .global sin_v_d
24  sin_v_d:
25          vmul.f64 d1,d0,d0       @ d1 <- x^2
26          vmrs    r1, fpscr       @ get FPSCR contents in r1
27          .if SET_RUNFAST
```

$\sin x$ Using IEEE Double Precision Vector Mode

```
1          @@ set runfast mode and rounding to nearest
2          bic     r2, r1, #(0b1111<<23)
3          orr     r2, r2, #(0b1100<<23)
4          vmsr    fpscr, r2 @ store settings in FPSCR
5          .endif
6          @@ Set up vector of the initial powers of x in Bank 1
7          @@      vmul.f64 d4,d0,d1       @ d8 <- x^3
8          @@      vmul.f64 d5,d4,d1       @ d9 <- x^5
9          @@      vmul.f64 d6,d5,d1       @ d10 <- x^7
10         @@ (The second and third multiply each require the result
11         @@  from the previous multiply, so the instructions are
12         @@  spread out for better scheduling to get 5% better
13         @@  performance overall.)
14         vmul.f64 d4,d0,d1       @ d8 <- x^3
15         @@ load vector of coefficients into Bank 2
16         ldr      r0,=ctab       @ get address of coefficient table
17         vmul.f64 d5,d4,d1       @ d9 <- x^5
18         vldmia   r0!,{d8-d10}   @ load first three coefficients
19         @@ Make three copies of x^6 in Bank 3
20         vmul.f64 d12,d5,d0      @ d12 <- x^6
21         vmul.f64 d6,d5,d1       @ d10 <- x^7
22         vmov.f64 d13,d12        @ d13 <- x^6
23         vmov.f64 d14,d12        @ d14 <- x^6
24         @@ Set VFP for vector mode (stride = 1, vector length = 3)
25         .if SET_RUNFAST
26         bic      r2, r2, #(0b11111<<16)
27         .else
```

$\sin x$ Using IEEE Double Precision Vector Mode

```
1          bic      r2, r1, #(0b11111<<16)
2          .endif
3          orr      r2, r2, #(0b00010<<16)
4          vmsr     fpscr, r2
5          @@ Multiply powers by coefficients. Put results in Bank 3
6          vmul.f64 d8,d8,d4       @ VECTOR operation
7          @@ Add terms in Bank 3 to the result in d0
8          vadd.f64 d3,d8,d9
9          vadd.f64 d0,d0,d10
10         mov      r3,#2          @ load loop counter
11         vadd.f64 d0,d0,d3
12 loop:   @@ load vector of next three coefficients into Bank 2
13         vldmia   r0!,{d8-d10}
14         @@ Set up vector of the required powers of x in Bank 1
15         vmul.f64 d4,d4,d12      @ VECTOR operation
16         @@ Multiply powers by coefficients Put results in Bank 2
17         vmul.f64 d8,d8,d4       @ VECTOR operation
18         @@ Add terms in Bank 2 to the result in d0
19         vadd.f64 d3,d8,d9
20         vadd.f64 d0,d0,d10
21         subs     r3,r3,#1       @ decrement and perform loop test
22         vadd.f64 d0,d0,d3       @ placed here for performance
23         bne      loop           @ perform loop twice
24         @@ restore original FPSCR
25         vmsr     fpscr, r1
26         mov      pc,lr
```

Performance

Optimization	Implementation	CPU seconds
None	Single Precision Scalar Assembly	2.96
	Single Precision Vector Assembly	2.63
	Single Precision C	8.75
	Double Precision Scalar Assembly	4.59
	Double Precision Vector Assembly	3.75
	Double Precision C	9.21
Full	Single Precision Scalar Assembly	2.16
	Single Precision Vector Assembly	2.06
	Single Precision C	2.59
	Double Precision Scalar Assembly	3.88
	Double Precision Vector Assembly	3.16
	Double Precision C	8.49

Summary

- The ARM VFP provides hardware support for the most common IEEE 754 formats for floating point numbers.
- Vector mode adds a significant performance improvement.
- Access to the vector features is only possible through assembly language.