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

<table>
<thead>
<tr>
<th>r0</th>
<th>s1</th>
<th>s0</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>s3</td>
<td>s2</td>
<td>d1</td>
</tr>
<tr>
<td>r2</td>
<td>s5</td>
<td>s4</td>
<td>d2</td>
</tr>
<tr>
<td>r3</td>
<td>s7</td>
<td>s6</td>
<td>d3</td>
</tr>
<tr>
<td>r4</td>
<td>s9</td>
<td>s8</td>
<td>d4</td>
</tr>
<tr>
<td>r5</td>
<td>s11</td>
<td>s10</td>
<td>d5</td>
</tr>
<tr>
<td>r6</td>
<td>s13</td>
<td>s12</td>
<td>d6</td>
</tr>
<tr>
<td>r7</td>
<td>s15</td>
<td>s14</td>
<td>d7</td>
</tr>
<tr>
<td>r8</td>
<td>s17</td>
<td>s16</td>
<td>d8</td>
</tr>
<tr>
<td>r9</td>
<td>s19</td>
<td>s18</td>
<td>d9</td>
</tr>
<tr>
<td>r10</td>
<td>s21</td>
<td>s20</td>
<td>d10</td>
</tr>
<tr>
<td>r11(fp)</td>
<td>s23</td>
<td>s22</td>
<td>d11</td>
</tr>
<tr>
<td>r12(ip)</td>
<td>s25</td>
<td>s24</td>
<td>d12</td>
</tr>
<tr>
<td>r13(sp)</td>
<td>s27</td>
<td>s26</td>
<td>d13</td>
</tr>
<tr>
<td>r14(lr)</td>
<td>s29</td>
<td>s28</td>
<td>d14</td>
</tr>
<tr>
<td>r15(pc)</td>
<td>s31</td>
<td>s30</td>
<td>d15</td>
</tr>
</tbody>
</table>

CPSR

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 – Most Important Bits

N  The Negative flag is set to one by \( \text{vcmp} \) if \( F_d < F_m \).
Z  The Zero flag is Set to one by \( \text{vcmp} \) if \( F_d = F_m \).
C  The Carry flag is set to one by \( \text{vcmp} \) if \( F_d = F_m \), or \( F_d > F_m \), or \( F_d \) and \( F_m \) are unordered.
V  The \text{Overflow} flag is set to one by \( \text{vcmp} \) if \( F_d \) and \( F_m \) 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

\[ \text{Op} \quad \text{Fd}, \text{Fn}, \text{Fm} \]
\[ \text{Op} \quad \text{Fd}, \text{Fm} \]

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

If the operation has only one operand, then the operation is applied to \text{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 \text{Fm} and each element in the vector starting at \text{Fn}, and the result is stored in the vector beginning at \text{Fd}.
Vector Mode

\[ \text{Op} \quad \text{Fd}, \text{Fn}, \text{Fm} \]
\[ \text{Op} \quad \text{Fd, Fm} \]

- the LEN field is not set to zero, and
- neither \( \text{Fd} \) nor \( \text{Fm} \) is in Bank 0 or Bank 4.

If the operation has only one operand, then the operation is applied to the vector starting at \( \text{Fm} \) and the results are placed in the vector starting at \( \text{Fd} \).

If the operation has two operands, then it is applied with corresponding elements from the vectors starting at \( \text{Fm} \) and \( \text{Fn} \), and the result is stored in the vector beginning at \( \text{Fd} \).
Load/Store Single Register

- **Operations:**
  - vldr  Load VFP Register, and
  - vstr  Store VFP Register.

- **Syntax:**
  
  \[
  \begin{align*}
  v<op>r\{<cond>\}\{.<prec>\} & \text{ Fd, } [\text{Rn}\{,#\text{offset}\}] \\
  v<op>r\{<cond>\}\{.<prec>\} & \text{ Fd, } =\text{label}
  \end{align*}
  \]

  - \(<op>\) 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,
     @ store s0 using address in r1
Load/Store Multiple Register

- Operations:
  - vldm  Load Multiple VFP Registers, and
  - vstm  Store Multiple VFP Registers.

- Syntax:

  \[
  \begin{align*}
  & v\langle\text{op}\rangle m\langle\text{mode}\rangle \{\langle\text{cond}\rangle\}\{.\langle\text{prec}\rangle\} \ Rn\{!\},<\text{list}> \\
  & vpush\{\langle\text{cond}\rangle\}\{.\langle\text{prec}\rangle\} \ <\text{list}> \\
  & vpop\{\langle\text{cond}\rangle\}\{.\langle\text{prec}\rangle\} \ <\text{list}>
  \end{align*}
  \]

- \textbf{<op>} may be either \texttt{ld} or \texttt{st}.
- \textbf{<mode>} is one of
  - \texttt{ia} Increment address after each transfer.
  - \texttt{db} Decrement address before each transfer.
- \textbf{Rn} may be any ARM integer register.
- \textbf{<cond>} is an optional condition code.
- \textbf{<prec>} may be either \texttt{f32} or \texttt{f64}.
- \textbf{<list>} may be any set of \textit{contiguous} single precision registers, or any set of \textit{contiguous} double precision registers.
- If \textbf{mode} is \texttt{db} then the \texttt{!} is required.
Examples:

1. `vstmdb sp!,{s0-s3}`  @ Store s0 through s3 on stack
2. `vstmia r1,{s0-s31}`  @ Store all fp registers at address in r1
3. `vldmia sp!,{d4-d7}`  @ Pop four doubles from the stack
4. `vldmiaeq sp!,{d4-d7}`  @ If eq, then pop four doubles 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:**
  - \texttt{vadd} Add,
  - \texttt{vsub} Subtract,
  - \texttt{vmul} Multiply,
  - \texttt{vnmul} Negate and Multiply, and
  - \texttt{vdiv} Divide.

- **Syntax:**

  \[
  \texttt{v<op>{<cond>}.<prec> Fd, Fn, Fm}
  \]

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

- **Examples:**

  1. \texttt{vadd.f64 d0, d1, d2} @ \( d0 \leftarrow d1 + d2 \)
  2. \texttt{vaddgt.f32 s0, s1, s2} @ if (gt) then \( s0 \leftarrow s1 + s2 \)
  3. \texttt{vnmul.f32 s10, s10, s14} @ \( s10 \leftarrow -(s10 \times s14) \)
  4. \texttt{vdivlt.f64 d0, d7, d8} @ if \( lt \), then \( d0 \leftarrow d7 / d8 \)
Compare

The compare instruction subtracts the value in \( F_m \) from the value in \( F_d \) and set the flags in the FPSCR based on the result.

- **Operations:**
  
  \[
  \text{vcmp} \quad \text{Compare, and}
  
  \text{vcmpe} \quad \text{Compare with Exception}.
  \]

- **Syntax:**

  \[
  \text{vcmp}\{e}\{<\text{cond}>\}.<\text{prec}> \quad \text{Fd, Fm}
  \]

  - 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.
  - \( <\text{cond}> \) is an optional condition code.
  - \( <\text{prec}> \) may be either \text{f32} or \text{f64}.

- **Examples:**

  1
  \[
  \text{vcmp}.\text{f32} \quad \text{s0, s1} \quad \text{@ Subtract s1 from s0 and set @ FPSCR flags}
  \]
Moving Between Two VFP Registers

- Operations:
  - `vmov` Move Between VFP Registers.

- Syntax:
  
  \[
  \text{vmov}\{\text{<cond>}}\{.\text{<prec>}\} \ Fd, \ Fm
  \]

- `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\cdot.f64 \ d3,\ d4`  \(\@\ d3 \leftarrow \ d4\)
  2. `vmov\cdot.f32 \ s5,\ s12`  \(\@\ s5 \leftarrow \ s12\)
Moving Between VFP and Single ARM Register

- Operations:
  - `vmov` Move Between VFP and One ARM Integer Register.

- Syntax:
  
  \[
  \begin{align*}
  \text{vmov}\{<\text{cond}>\} & \quad \text{Rd, Sn} \\
  \text{vmov}\{<\text{cond}>\} & \quad \text{Sn, Rd}
  \end{align*}
  \]

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

- Examples:

  \[
  \begin{align*}
  1. \quad \text{vmov } r3, s4 & \quad @ \ r2 <= s4 \\
  2. \quad \text{vmov } s12, r8 & \quad @ \ s12 <- r8
  \end{align*}
  \]
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.

<table>
<thead>
<tr>
<th>ARM Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rl,Rh</td>
<td>Dd</td>
</tr>
<tr>
<td>Sd,Sd'</td>
<td></td>
</tr>
</tbody>
</table>

- `<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:**

```plaintext
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 flags to CPSR
3  vmsr  FPSCR, r5         @ Copy FPSCR flags to CPSR
```
Convert Between Floating Point and Integer

- Operations:
  - `vcvt` Convert Between Floating Point and Integer
  - `vcvtr` Convert Floating Point to Integer with Rounding

- Syntax:

  \[
  \begin{align*}
  &\text{vcvt}\{r\}\{<\text{cond}>\}.<\text{type}>.f64 \quad \text{Sd, Dm} \\
  &\text{vcvt}\{r\}\{<\text{cond}>\}.<\text{type}>.f32 \quad \text{Sd, Sm} \\
  &\text{vcvt}\{<\text{cond}>\}.f64.<\text{type}> \quad \text{Dd, Sm} \\
  &\text{vcvt}\{<\text{cond}>\}.f32.<\text{type}> \quad \text{Sd, Sm}
  \end{align*}
  \]

- The optional \( r \) makes the operation use the rounding mode specified in the FPSCR. The default is to round toward zero.

- \( <\text{cond}> \) is an optional condition code.

- The \( <\text{type}> \) 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 \( \text{vmul} \).
### Examples:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>vcvt.f64.u32 d5, s7</code> @ Convert unsigned integer to double</td>
</tr>
<tr>
<td>2</td>
<td><code>vcvt.f64.f32 d0, s4</code> @ Convert signed integer to double</td>
</tr>
<tr>
<td>3</td>
<td><code>vcvt.u32.f64 s0, d7</code> @ Convert double to unsigned integer</td>
</tr>
<tr>
<td>4</td>
<td><code>vcvt.s32.f64 s1, d4</code> @ Convert double to signed integer</td>
</tr>
<tr>
<td></td>
<td><code>@@ Convert s10 to an S(15,16)</code></td>
</tr>
<tr>
<td></td>
<td><code>consta: .float 65536.0</code></td>
</tr>
<tr>
<td>5</td>
<td><code>vldr.f32 s11,consta</code> @ Load floating point constant</td>
</tr>
<tr>
<td>6</td>
<td><code>vmul.f32 s10,s10,s11</code> @ Multiply equates to shift</td>
</tr>
<tr>
<td>7</td>
<td><code>vcvt.s32.f32 s10,s10</code> @ Convert single to S(15,16)</td>
</tr>
</tbody>
</table>
Convert Between Fixed Point and Single Precision

- **Operations:**
  - `vcvt`  Convert To or From Fixed Point.
- **Syntax:**
  
  \[
  \text{vcvt}\{\text{<cond>}\}.\langle\text{td}\rangle.\text{f32} \ Sd, Sm, \ #\text{fbits} \\
  \text{vcvt}\{\text{<cond>}\}.\text{f32}.<\text{td}\rangle \ Sd, Sm, \ #\text{fbits}
  \]

  - `<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,
    - `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)
sinx Using IEEE Single Precision

.data
@ The following is a table of constants used in the
@ Taylor series approximation for sine
.align 5 @ Align to cache
ctab:
.word 0xBE2AAAAA @ -1.666666e-01
.word 0x3C088889 @ 8.333334e-03
.word 0xB9500D00 @ -1.984126e-04
.word 0x3638EF1D @ 2.755732e-06
.word 0xB2D7322A @ -2.505210e-08

.text
.align 2
@ sin_a_f implements the sine function using IEEE single
@ precision floating point. It computes sine by summing
@ the first six terms of the Taylor series.
global sin_a_f

sin_a_f:
@ set runfast mode and rounding to nearest
fmrx r1, fpscr @ get FPSCR contents in r1
bic r2, r1, #(0b1111<<23)
or r2, r2, #(0b1100<<23)
fmxr fpscr, r2 @ store in FPSCR
@ initialize variables
vmul.f32 s1,s0,s0 @ s1 <- x^2
vmul.f32 s3,s1,s0 @ s3 <- x^3
ldr r0,=ctab @ load pointer to coefficients
mov r3,#5 @ load loop counter
\textbf{sin}\textit{x} Using IEEE Single Precision

\begin{verbatim}
loop:    vldr.f32     s4,[r0]  @ load coefficient
    add          r0,r0,#4  @ increment pointer
    vmul.f32     s4,s3,s4  @ s4 <- next term
    vadd.f32     s0,s0,s4  @ add term to result
    subs         r3,r3,#1  @ decrement and test loop count
    vmulne.f32   s3,s1,s3  @ s4 <- x^{2n}
    bne          loop     @ loop five times
@@ restore original FPSCR
    fmxr      fpscr, r1
    mov       pc,lr
\end{verbatim}
sin\,x Using IEEE Single Precision Vector Mode

```
.data
.align 6 @ Align to cache
tab:
.word 0xBE2AAAAB @ -1.666667e-01
.word 0x3C088889 @ 8.333334e-03
.word 0xB9500D01 @ -1.984127e-04
.word 0x3638EF1D @ 2.755732e-06
.word 0xB2D7322B @ -2.505211e-08

@@@ ---------------------------------------------------------------
.text
.align 2
.global sin_v_f
sin_v_f:@@ set runfast mode and rounding to nearest
vmrs r1, fpscr @ get FPSCR contents in r1
bic r2, r1, #(0b1111<<23)
or r2, r2, #(0b1100<<23)
vmsr fpscr, r2 @ store settings in FPSCR
vmul.f32 s1,s0,s0 @ s1 = x^2
ldr r0,=ctab @ get address of coefficients
vldmia r0!,{s16-s20} @ load all coefficients into Bank 2
vmul.f32 s8,s0,s1 @ s8 = x^3
vmul.f32 s9,s8,s1 @ s9 = x^5
vmul.f32 s10,s9,s1 @ s10 = x^7
vmul.f32 s11,s10,s1 @ s11 = x^9
vmul.f32 s12,s11,s1 @ s12 = x^11
```
sin(x) Using IEEE Single Precision Vector Mode

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

```
.data
@@ The following is a table of constants used in the
@@ Taylor series approximation for sine
.align 7   @ Align for efficient caching
ctab: .word 0x55555555, 0xBFC55555  @ -1.666666666666667e-01
 .word 0x11111111, 0x3F811111  @ 8.333333333333333e-03
 .word 0x1A01A01A, 0xBF2A01A0  @ -1.984126984126984e-04
 .word 0xA556C734, 0x3EC71DE3  @ 2.755731922398589e-06
 .word 0x67F544E4, 0xBE5AE645  @ -2.505210838544172e-08
 .word 0x13A86D09, 0x3DE61246  @ 1.605904383682161e-10
 .word 0xE733B81F, 0xBD6AE7F3  @ -7.647163731819816e-13
 .word 0x7030AD4A, 0x3CE952C7  @ 2.811457254345521e-15
 .word 0x46814157, 0xBC62F49B  @ -8.220635246624329e-18

@@ @---------------------------------------------------------------
.text
.align 2
@@ sin_a_d implements the sine function using IEEE
datale precision floating point. It takes advantage
@@ of the ARM VFP vector processing instructions and
@@ computes sine by summing the first ten terms of the
@@ Taylor series.
.global sin_v_d
.sin_v_d:
vmul.f64 d1,d0,d0  @ d1 <- x^2
vmrs  r1, fpscr   @ get FPSCR contents in r1
.if SET_RUNFAST
```

\[ \sin x \] Using IEEE Double Precision Vector Mode

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

bic r2, r1, #(0b11111<<16)
.endif
orr r2, r2, #(0b00010<<16)
vmsr fpscr, r2

@@ Multiply powers by coefficients. Put results in Bank 3
vmul.f64 d8,d8,d4 @ VECTOR operation
@@ Add terms in Bank 3 to the result in d0
vadd.f64 d3,d8,d9
vadd.f64 d0,d0,d10
mov r3,#2 @ load loop counter
vadd.f64 d0,d0,d3

loop: @@ load vector of next three coefficients into Bank 2
vldmia r0!,{d8-d10}
@@ Set up vector of the required powers of x in Bank 1
vmul.f64 d4,d4,d12 @ VECTOR operation
@@ Multiply powers by coefficients Put results in Bank 2
vmul.f64 d8,d8,d4 @ VECTOR operation
@@ Add terms in Bank 2 to the result in d0
vadd.f64 d3,d8,d9
vadd.f64 d0,d0,d10
subs r3,r3,#1 @ decrement and perform loop test
vadd.f64 d0,d0,d3 @ placed here for performance
bne loop @ perform loop twice
@@ restore original FPSCR
vmsr fpscr, r1
mov pc,lr
## Performance

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Implementation</th>
<th>CPU seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>None</strong></td>
<td>Single Precision Scalar Assembly</td>
<td>2.96</td>
</tr>
<tr>
<td></td>
<td>Single Precision Vector Assembly</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>Single Precision C</td>
<td>8.75</td>
</tr>
<tr>
<td></td>
<td>Double Precision Scalar Assembly</td>
<td>4.59</td>
</tr>
<tr>
<td></td>
<td>Double Precision Vector Assembly</td>
<td>3.75</td>
</tr>
<tr>
<td></td>
<td>Double Precision C</td>
<td>9.21</td>
</tr>
<tr>
<td><strong>Full</strong></td>
<td>Single Precision Scalar Assembly</td>
<td>2.16</td>
</tr>
<tr>
<td></td>
<td>Single Precision Vector Assembly</td>
<td>2.06</td>
</tr>
<tr>
<td></td>
<td>Single Precision C</td>
<td>2.59</td>
</tr>
<tr>
<td></td>
<td>Double Precision Scalar Assembly</td>
<td>3.88</td>
</tr>
<tr>
<td></td>
<td>Double Precision Vector Assembly</td>
<td>3.16</td>
</tr>
<tr>
<td></td>
<td>Double Precision C</td>
<td>8.49</td>
</tr>
</tbody>
</table>
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.