11 Floating Point Operations in ARM

CPE 221

The University of Alabama in Huntsville

Rahul Bhadani

March 25, 2025

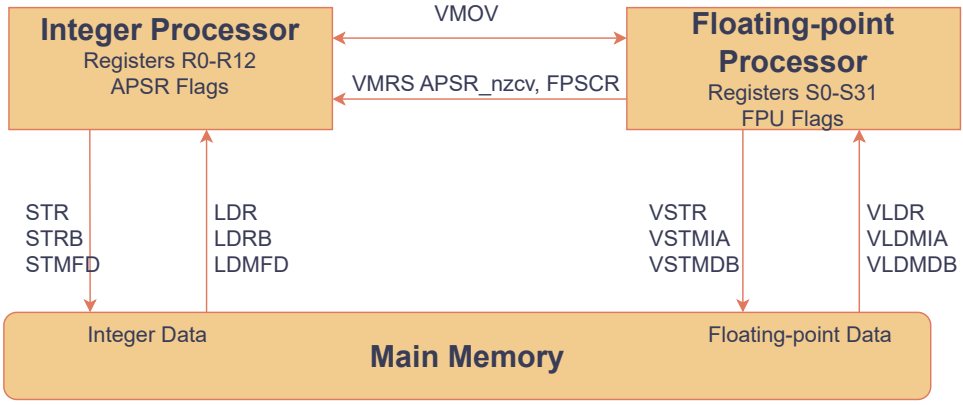
Introduction to Floating-Point Support in ARMv7

0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 0 0 1 0 0
0 1 0 1 1 0 0 1 1 1 0 1 0 0 0 0 1 1 0 0
0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 0 1 0

ARMv7 VFP Architecture Overview

- ARMv7 provides hardware floating-point support through:
  - Vector Floating Point (VFP) extension
  - Advanced Single Instruction, Multiple Data (SIMD), NEON extension
- CPULator supports these extensions
- VFP supports IEEE 754 single-precision and double-precision operations
- Key components:
  - Dedicated register set
  - Special instructions for floating-point operations
  - Control and status register (FPSCR)

Two Separate Processors



Floating-Point Register Set

1 1 0 0 1 0 1 1 0 0 0 0 0 1 1 1 1 0 0 1

1 0 1 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 1 0

0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 0 1 1

Floating-Point Register Organization

- ARMv7 VFP provides 32 single-precision registers (S0-S31)
- These can be viewed as:
  - 16 double-precision registers (D0-D15)
  - S0/S1 overlap with D0, S2/S3 with D1, etc.
- Advanced implementations support 32 double-precision registers (D0-D31)
- VFP/NEON registers are separate from general-purpose registers

Single-precision	S0	S1	...
Double-precision	D0		...

FPSCR: Floating-Point Status and Control Register

- Special register that controls VFP operation
- Contains fields for:
  - Exception flags (N, Z, C, V)
  - Rounding mode configuration
  - Exception handling controls
  - Vector length/stride control
- Similar role to the CPSR for integer operations
- Accessible via special instructions:
  - VMRS - Move from FPSCR to ARM register
  - VMSR - Move to FPSCR from ARM register
  - APSR_nzcv (Application Program Status Register, NZCV flags): The APSR holds the condition flags for integer operations. The _nzcv suffix specifies that only the N (Negative), Z (Zero), C (Carry), and V (Overflow) flags should be updated.

Basic Floating-Point Operations

1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0

1 0 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1

0 1 1 1 0 0 0 0 0 0 1 0 1 1 0 1 1 1 0 1

VMOV

```
1  .data
2      pi_val: .float 3.14159
3      double_val: .double 1.618
4
5  .text
6  .global _start
7  _start:
8      VMOV S0,0.5
9  done: B done
```

VMOV only works with a limited set of constants. VMOV constants are limited to $\pm m/2^k$. Bit 7: Sign bit. Bits 6:4: $0 \leq k \leq 7$, Bits 3:0: $16 \leq m \leq 31$
This is not supported in CPULator.

Loading and Storing Floating-Point Values

```
1  .data
2      pi_val: .float 3.14159
3      double_val: .double 1.618
4
5  .text
6  .global _start
7  _start:
8      LDR R0, =pi_val
9      VLDR S0, [R0]          @ Load single-precision constant
10
11     LDR R0, =double_val
12     VLDR D1, [R0]          @ Load double-precision constant
13
14 done: B done
```

Supported in CPULator.

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Basic Arithmetic Operations

```
1  @ Single-precision operations
2  VADD.F32 S0, S1, S2          @ S0 = S1 + S2
3  VSUB.F32 S3, S4, S5          @ S3 = S4 - S5
4  VMUL.F32 S6, S7, S8          @ S6 = S7 * S8
5  VDIV.F32 S9, S10, S11        @ S9 = S10 / S11
6  @ Double-precision operations
7  VADD.F64 D0, D1, D2          @ D0 = D1 + D2
8  VSUB.F64 D3, D4, D5          @ D3 = D4 - D5
9  VMUL.F64 D6, D7, D8          @ D6 = D7 * D8
10 VDIV.F64 D9, D10, D11        @ D9 = D10 / D11
```

Division Operation

```
1  .global _start
2  _start:
3      // Load the floating-point numbers into registers using PC-relative addressing
4      VLDR.F32 S0, float1   // Load 37.24 into register S0
5      VLDR.F32 S1, float2   // Load 7.51 into register S1
6      // Perform the division: S2 = S0 / S1
7      VDIV.F32 S2, S0, S1   // S2 = S0 / S1
8      // At this point, S2 contains the result of 37.24 / 7.51
9      // Exit the program (optional, depending on your environment)
10     MOV R7, #1            // syscall: exit
11 done: B done
12     // Define the floating-point constants in a literal pool
13     .ltorg                // Place the literal pool here
14 float1: .float 37.24
15 float2: .float 7.51
```

Loading First Floating-Point Value

```
1  VLDR.F32 S0, float1   // Load 37.24 into register S0
```

- `VLDR.F32`: Vector Load instruction for 32-bit floating-point values
- `S0`: Destination register (single-precision floating-point)
- `float1`: Label referencing memory location with value 37.24
- Uses PC-relative addressing to locate the value

Loading Second Floating-Point Value

```
1  VLDR.F32 S1, float2   // Load 7.51 into register S1
```

- Similar to previous instruction
- Loads 32-bit floating-point value 7.51 from memory
- Places value into register S1
- Prepares second operand for division operation

Performing Floating-Point Division

```
1  VDIV.F32 S2, S0, S1   // S2 = S0 / S1
```

- `VDIV.F32`: Vector Floating-Point Division instruction
- Divides value in S0 (37.24) by value in S1 (7.51)
- Result stored in register S2
- Calculates $37.24 \div 7.51 \approx 4.96$

Literal Pool Directive

```
1   .ltorg                  // Place the literal pool here
```

- `.ltorg`: Directive to place a literal pool at this location
- Literal pool: Storage area for constants not directly encodable in instructions
- Ensures floating-point constants are accessible to VLDR instructions
- Important for proper memory layout and addressing

Floating-Point Constant

```
1 float1: .float 37.24
2 float2: .float 7.51
```

- `float1:`: Label to reference this memory location
- `.float 37.24`: Allocates space for 32-bit floating-point constant
- Defines the first operand for our division operation
- Referenced by the first VLDR instruction
- `float2:`: Label for memory reference
- `.float 7.51`: Allocates space for 32-bit floating-point constant
- Defines the second operand (divisor)
- Referenced by the second VLDR instruction

Comparison and Conditional Operations

```
1  @ Compare floating-point values
2  VCMP.F32 S0, S1              @ Compare S0 with S1
3  VMRS APSR_nzcv, FPSCR        @ Transfer flags to APSR
4  BEQ equal_label              @ Branch if S0 == S1
5  BGT greater_than_label       @ Branch if S0 > S1
6
7  @ Using VPSEL (ARMv8 but supported in some ARMv7 implementations)
8  VCMP.F32 S0, S1
9  VMRS APSR_nzcv, FPSCR
10 VMOVGT.F32 S2, S0            @ Move S0 to S2 if S0 > S1
```

Data Conversion

1 0 0 0 1 1 0 0 0 0 0 1 0 0 1 1 1 0 1 0

1 1 1 1 1 1 0 0 1 0 0 0 1 1 0 0 0 1 1 0

0 1 1 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 1 1

Converting Between Formats

```
1  @ Integer to floating-point
2  VMOV S0, R0                  @ Move R0 value to S0
3  VCVT.F32.S32 S0, S0          @ Convert signed 32-bit int to float
4  @ Floating-point to integer
5  VCVT.S32.F32 S1, S0          @ Convert float to signed 32-bit int
6  VMOV R1, S1                  @ Move S1 value to R1
7
8  @ Single to double precision
9  VCVT.F64.F32 D0, S0          @ Convert single to double
10
11 @ Double to single precision
12 VCVT.F32.F64 S2, D0          @ Convert double to single
```

Working with FPSCR

1 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0 1

0 0 0 0 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1

0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1

Accessing FPSCR

```
1  @ Reading FPSCR
2  VMRS R0, FPSCR              @ Read FPSCR into R0
3  @ Writing to FPSCR
4  VMSR FPSCR, R0              @ Write R0 to FPSCR
5
6  @ Moving flags from FPSCR to APSR
7  VMRS APSR_nzcv, FPSCR       @ Copy condition flags
8
9  @ Setting rounding mode (example: round to nearest)
10 MOV R0, #0x0                @ Round to nearest mode
11 BIC R1, R0, #0x00C00000     @ Clear rounding mode bits
12 ORR R1, R1, #0x00000000     @ Set round to nearest (00)
13 VMSR FPSCR, R1              @ Update FPSCR
```

Loading Floating Data Memory to Multiple Registers

0 1 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0

0 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 0 0 1 1

0 1 1 0 1 1 1 1 1 1 0 0 1 0 0 1 1 0 0 0

VLDMIA

Load Multiple FPU Registers, Increment After.

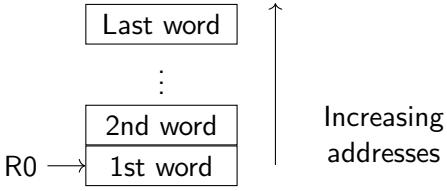
Syntax:

```
VLDMIA Rn!,register list
```

- FP registers → memory, 1st address in Rn
- Updates Rn only if write-back flag (!) is appended to Rn.

Example:

```
// Copy starting at mem[R0]
VLDMIA R0!,{S0,S1,S2}
```



VLDMDB

Load Multiple FPU Registers, Decrement Before.

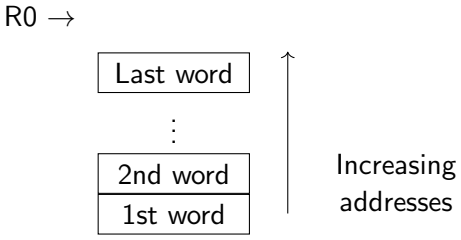
Syntax:

`VLDMDB Rn!,register list`

- FP registers → memory, addresses end just before address in Rn
- Must append (!) and always updates Rn.

Example:

```
//Copy ending before mem[R0]
VLDMDB R0!,{S0,S1,S2}
```



Examples

0 1 0 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0 0

1 1 0 1 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0

1 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 1 0 0

Example 1: Computing the Area of a Circle

```
1   .data
2   pi: .float 3.14159265        @ Single-precision PI
3   radius: .float 5.0           @ Circle radius
4   area: .float 0.0             @ Result storage
5   .text
6   .global _start
7   _start:
8   @ Load the radius and PI into VFP registers
9   LDR R0, =radius
10  VLDR S0, [R0]                @ S0 = radius
11  LDR R0, =pi
12  VLDR S1, [R0]                @ S1 = pi
13
14  @ Compute radius squared
15  VMUL.F32 S2, S0, S0          @ S2 = radius * radius
16
17  @ Compute area = pi * radius^2
18  VMUL.F32 S3, S1, S2          @ S3 = pi * radius^2
19
20  @ Store the result
21  LDR R0, =area
22  VSTR S3, [R0]                @ Store result to memory
23
24  done: B done
```

Example 2: Temperature Conversion

```
1   .data
2   temp_f: .float 98.6          @ Temperature in Fahrenheit
3   temp_c: .float 0.0           @ Will hold Celsius result
4   const_32: .float 32.0        @ Constant for conversion
5   const_5_9: .float 0.5555555 @ 5/9 as a floating-point constant
6   .text
7   .global _start
8   _start:
9   @ Load the Fahrenheit temperature
10  LDR R0, =temp_f
11  VLDR S0, [R0]                @ S0 = fahrenheit temperature
12  @ Load the constants
13  LDR R0, =const_32
14  VLDR S1, [R0]                @ S1 = 32.0
15  LDR R0, =const_5_9
16  VLDR S2, [R0]                @ S2 = 5/9
17  @ Formula: C = (F - 32) * 5/9
18  VSUB.F32 S3, S0, S1          @ S3 = F - 32
19  VMUL.F32 S4, S3, S2          @ S4 = (F - 32) * 5/9
20  @ Store the result
21  LDR R0, =temp_c
22  VSTR S4, [R0]                @ Store Celsius result
23
24  done: B done
```

Example 3: Working with FPSCR for Exception Handling

```
1  .data
2  val1: .float 1.0
3  val2: .float 0.0            @ Will cause division by zero
4  .text
5  .global _start
6  _start:
7  @ Enable floating-point exceptions (division by zero)
8  VMRS R0, FPSCR              @ Read current FPSCR
9  ORR R0, R0, #(1 << 2)       @ Set DZE (Division by Zero) exception
   ↪  bit
10 VMSR FPSCR, R0              @ Update FPSCR
11 @ Load values
12 LDR R0, =val1
13 VLDR S0, [R0]               @ S0 = 1.0
14 LDR R0, =val2
15 VLDR S1, [R0]               @ S1 = 0.0
```

Example 3: Working with FPSCR for Exception Handling (Continued)

```
1   @ Try division that will cause exception
2   VDIV.F32 S2, S0, S1          @ S2 = S0 / S1 (div by zero)
3   @ Check for exception
4   VMRS R0, FPSCR               @ Read FPSCR after operation
5   TST R0, #(1 << 2)            @ Check if DZC (Division by Zero) flag
    ↪  is set
6   BNE division_error           @ Branch if exception occurred
7   B continue                   @ No exception, continue
8   division_error:
9   @ Handle the error here
10  MOV R0, #1                   @ Error code
11  B end
12  continue:
13  MOV R0, #0                   @ Success code
14  end:
15  done: B done
```

The End