

VFP10™ Vector Floating-point Coprocessor

(Rev 0)

Technical Reference Manual

ARM®

VFP10 Vector Floating-point Coprocessor

Technical Reference Manual

Copyright © 2000 ARM Limited. All rights reserved.

Release Information

Change history

Date	Issue	Change
9 February 2000	A	First release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

VFP10 Vector Floating-point Coprocessor

Technical Reference Manual

	Preface	
	About this document	x
	Further reading	xii
	Feedback	xiii
Chapter 1	Introduction	
	1.1 Overview	1-2
	1.2 Vector floating-point architecture features	1-3
	1.3 IEEE 754 compatibility	1-5
	1.4 Applications	1-6
Chapter 2	Programmer's Model	
	2.1 The register file	2-2
	2.2 Register banks	2-5
	2.3 The VFP10 Rev 0 pipeline	2-13
	2.4 Implementation-specific control registers	2-19
	2.5 Rationale for implementation choices	2-23
Chapter 3	Exceptions	
	3.1 Support code	3-2

3.2	Exception processing for scalar and vector instructions	3-9
3.3	Invalid operation	3-14
3.4	Division by zero	3-16
3.5	Overflow	3-17
3.6	Underflow	3-18
3.7	Inexact result	3-19
3.8	Input exceptions	3-21
3.9	Arithmetic exceptions	3-23

VFP10 Technical Reference Manual Glossary

List of Tables

VFP10 Vector Floating-point Coprocessor

Technical Reference Manual

	Change history	ii
Table 2-1	Register bank description	2-5
Table 2-2	Single-precision three-operand register usage	2-7
Table 2-3	Single-precision two-operand register usage	2-8
Table 2-4	Double-precision three-operand register usage	2-8
Table 2-5	Double-precision two-operand register usage	2-8
Table 2-6	Single-precision data memory images and byte addresses	2-9
Table 2-7	Double-precision data memory images and byte addresses	2-10
Table 2-8	VFP10 Rev 0 register contents illustrating parallel execution	2-16
Table 2-9	FPEXC bit field definitions bits 31-8	2-19
Table 2-10	Vector iteration count bit values	2-20
Table 2-11	FPEXC bit field definitions bits 7-0	2-20
Table 2-12	Access to control registers through FMRX and FMXR	2-21
Table 2-13	Bit fields for the FPSID register	2-22
Table 3-1	Possible IEEE 754 invalid operation exceptions	3-14
Table 3-2	Overflow result	3-17
Table 3-3	Key to bounce conditions table	3-21
Table 3-4	Input operand bounce conditions (non-FZ mode)	3-21
Table 3-5	LSA and USA determination	3-23
Table 3-6	USA and LSA values and conditions	3-24
Table 3-7	FMUL family bounce and exceptional thresholds	3-25

List of Tables

Table 3-8	FDIV bounce and exceptional thresholds	3-27
Table 3-9	FCVTSD bounce conditions	3-28
Table 3-10	SP Float-to-integer bounce thresholds and stored results	3-29
Table 3-11	DP Float-to-integer bounce thresholds and stored result	3-30

List of Figures

VFP10 Vector Floating-point Coprocessor

Technical Reference Manual

Figure 2-1	Single-precision data format	2-3
Figure 2-2	Register data formats	2-3
Figure 2-3	Register file decoding	2-12
Figure 2-4	VFP10 Rev 0 pipeline	2-14
Figure 2-5	FPEXC register format	2-21
Figure 2-6	FPSID register format	2-22
Figure 3-1	User status and control bit fields summary	3-2

Preface

This preface introduces the VFP10™ (Rev 0) Vector Floating-point Coprocessor and its reference documentation. It contains the following sections:

- *About this document* on page x
- *Further reading* on page xii
- *Feedback* on page xiii.

About this document

This document is the technical reference manual for the VFP10 Coprocessor.

Intended audience

This document has been written for experienced hardware and software engineers who are familiar with the ARM10 Thumb Family architecture and are conversant with IEEE 754 and its conventions for dealing with floating-point arithmetic. We recommend reading the relevant sections of the *ARM Architecture Reference Manual* before reading this manual. Only VFP10-specific implementation issues are addressed here.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the VFP10 Coprocessor.

Chapter 2 *Programmer's Model* Read this chapter for an explanation of the programmer's model. The chapter deals with the register file, register banks, pipeline and control registers.

Chapter 3 *Exceptions* Read this chapter for an explanation of how the VFP10 coprocessor handles exceptions and how it implements IEEE 754.

Glossary Refer to the glossary for definitions of terms frequently used in this manual.

Typographical conventions

The following typographical conventions are used in this book:

<i>italic</i>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
monospace	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
<u>monospace</u>	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

- monospace italic* Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
- monospace bold** Denotes language keywords when used outside example code.

Further reading

This section lists publications by ARM Limited, and by third parties.

ARM periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at: <http://www.arm.com>.

ARM publications

This document contains information that is specific to the VFP10 Vector Floating-point Coprocessor (Rev 0). Refer to the following documents for other relevant information:

- *ARM Architectural Reference Manual (ARM DUI 0100) Revision D or later.*
- *ARM1020T Technical Reference Manual (ARM DDI 0135).*

Other publications

This manual makes extensive use of the terminology and conventions of:

- ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-point Arithmetic.*

Feedback

ARM Limited welcomes feedback both on the VFP10 Vector Floating-point Coprocessor (Rev 0), and on the documentation.

Feedback on the VFP10 Vector Floating-point Coprocessor (Rev 0)

If you have any comments or suggestions about this product, please contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this document

If you have any comments about this document, please send e-mail to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1

Introduction

This chapter introduces the VFP10 Vector Floating-point Coprocessor. It contains the following sections:

- *Overview* on page 1-2
- *Vector floating-point architecture features* on page 1-3
- *IEEE 754 compatibility* on page 1-5
- *Applications* on page 1-6.

1.1 Overview

The ARM VFP10 Floating-point Coprocessor is the first implementation of the *Vector Floating-point Architecture* (VFPv1). It provides IEEE 754-compliant, low-cost floating-point computation for applications where high-performance graphics processing or signal processing capabilities are needed.

1.2 Vector floating-point architecture features

The ARM VFP10 Rev 0 specifies a subset of the ARM floating-point instruction set in hardware with the remaining instructions supported by software emulation.

The features of the ARM VFP10 Rev 0 implementation of the VFPv1 architecture include:

- Four-stage arithmetic pipeline:
 - full IEEE 754 rounding of normal results
 - four-cycle single-precision, multiply and multiply-accumulate with single-cycle latency
 - five-cycle double-precision multiply and multiply-accumulate, with two-cycle latency
 - four-cycle single-precision and double-precision operations with single-cycle latency for all other operations other than divide and square root
 - divide and square root are multi-cycle operations to conserve area.
- Fully IEEE 754-compliant multiply-accumulate family of operations which include:
 - multiply-accumulate
 - multiply-subtract
 - negate-multiply-accumulate
 - negate-multiply-subtract.

————— **Note** —————

 - These operations return the same result as the series of component operations.
 - The product of these operations is always rounded to the current rounding mode and the target precision before the negations, if applicable, and the final summation or difference.

- The following exceptions which trap to software support code to produce IEEE 754 results:
 - overflow
 - underflow
 - invalid operand
 - divide-by-zero.

- The following unsupported values will trap to software support code for IEEE 754 handling:
 - operations involving NaNs
 - operations involving infinities
 - operations involving denormal values.
- Simplified exception handling through early detection of potential exception cases and a constraint to allow only one exceptional condition in the machine at any time.
- High-performance vector operations providing for one instruction to operate on arrays of data. A vector load or store instruction and a vector arithmetic instruction can be executed in parallel.

1.3 IEEE 754 compatibility

The ARM VFP10 Rev 0 is IEEE 754-compatible. The combination of the ARM VFP10 Rev 0 hardware and support software forms an IEEE 754-compliant system. Both single-precision and double-precision operands are supported.

1.3.1 Special operand and exception handling

The ARM VFP10 Rev 0 has removed much of the special operand and exception handling to the software support code. Critical speed paths are made faster and simpler by not handling these exceptions and operands in hardware.

Software support code traps these exceptions:

- overflow
- underflow
- invalid operand
- divide-by-zero.

Software support code also traps all operations involving:

- infinities
- denormals
- NaNs.

Handling these exceptions or operands in software significantly increases the execution time of code which performs the computations dealing with these exceptions or operands. The incidence of these exceptions and operands in typical code is very small. In embedded application code their incidence is almost nonexistent. The presence of these operands and exceptions indicate situations in which the answers are not reliable. Often, their presence indicates an arithmetic situation which renders the result meaningless. Refer to Chapter 3 *Exceptions* for details.

This document is intended to be read in conjunction with the *ARM Architecture Reference Manual*. Only VFP10-specific implementation issues are addressed here.

1.4 Applications

The ARM VFP10 Rev 0 provides high-performance, low-cost floating-point computation particularly suitable for a wide spectrum of applications such as:

- personal digital assistants and smartphones for graphics, voice and user interfaces, Java interpretation and *Just In Time* (JIT) compilation
- games machines for high-resolution three-dimensional graphics and digital audio
- printers and MFP (multi-function peripheral) controllers for high-definition color rendering requiring high data memory bandwidth
- network controllers for high data bandwidth between network ports and for data compression
- set-top boxes for digital audio and digital video and three-dimensional user interfaces
- automotive applications for engine management and power train computations.

Chapter 2

Programmer's Model

This chapter details implementation-specific features of the VFP10 Rev 0 which are useful to programmers. It contains the following sections:

- *The register file* on page 2-2
- *The VFP10 Rev 0 pipeline* on page 2-13
- *Implementation-specific control registers* on page 2-19
- *Handling of IEEE 754 special operands* on page 2-23.

2.1 The register file

This chapter presents material useful to designers programming the VFP10 Rev 0 for use in conjunction with the ARM1020T on an ARM10200 reference device. Testing and timing issues are described in the ARM1020T reference manual.

The ARM VFP10 Rev 0 uses a register file which contains thirty-two 32-bit registers organized in four banks. Each register can be used to store:

- a single-precision data item
- a single-integer data item.

Alternatively, a consecutive pair of registers (R_{i+1}, R_i) can be used to store a double-precision item. Figure 2-3 on page 2-12 shows the register file decoding.

2.1.1 Register internal formats

The VFPv1 architecture provides the option of an internal data format which is different from some or all of the external formats. For the VFP10 Rev 0, data in the register file possesses the same format as data in memory. No modification to the format is performed by a load or store operation. This is also true for integer data. It is the responsibility of the programmer to be aware of the data type in each register. Hardware does not perform any checking of the agreement between data type in the source registers and the data type expected by the instruction. Hardware always interprets the data according to the precision contained in the instruction. It is recommended that for saving and restoring of VFP data registers you use the FLMDMX/FSTMX instructions for compatibility with future implementations.

The VFP10 Rev 0 does not support FLMDMX or FSTMX with a register count of 33. To get round this, perform the save and restore on each half of the register file. For example, to save the entire register file in a full descending stack, use:

```
FSTMFDX Rn!, {d0-d7}
FSTMFDX Rn!, {d8-d15}
```

To restore the same registers use:

```
FLDMFDX Rn!, {d8-d15}
FLDMFDX Rn!, {d0-d7}
```

Attempting to access a register that has not been initialized or loaded with valid data is UNPREDICTABLE. It is advised that all registers are loaded with signaling NaNs in the precision of the initial access of the register. It is further advised that the invalid exception is enabled to detect access to uninitialized register usage.

2.1.2 Single-precision data format

The single-precision data format used in the VFP10 is defined in the IEEE 754 specification. Refer to this for details concerning:

- the exponent bias
- special formats
- numerical ranges.

Figure 2-1 shows the single-precision bit fields

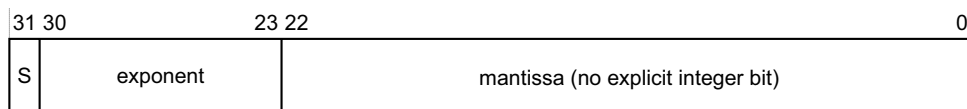


Figure 2-1 Single-precision data format

Single-precision data format comprises:

- the sign bit [bit 31]
- the exponent [bits 30:23]
- the mantissa with no explicit integer bit [bits 22:0].

2.1.3 Double-precision data format

The double-precision data format used in the VFP10 Rev 0 is defined in the IEEE 754 specification. Refer to this for details concerning:

- the exponent bias
- special formats
- numerical ranges.

Double-precision format comprises the *Most Significant Word (MSW)* and the *Least Significant Word (LSW)*. Figure 2-2 shows the bit fields of the two words in double-precision format.

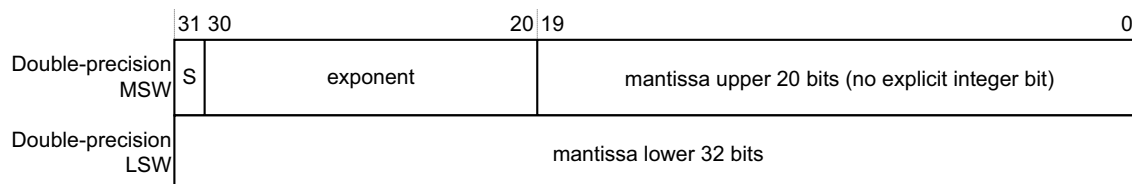


Figure 2-2 Register data formats

MSW comprises:

- the sign bit [bit 31] of the MSW
- the exponent [bits 30:20]
- the mantissa upper 20 bits with no explicit integer bit [bits 19:0].

LSW comprises the mantissa lower 32 bits.

2.2 Register banks

The register file is especially suited for vector operations. You can use four banks of registers in a circular fashion to facilitate signal processing and matrix operations. For details of this refer to the ARM Architecture Reference Manual.

Table 2-1 shows how the register banks are defined.

Table 2-1 Register bank description

Bank	Single-precision registers in bank	Double-precision registers in bank
0	s0-s7	d0-d3
1	s8-s15	d4-d7
2	s16-s23	d8-d11
3	s24-s31	d12-d15

The register file organization supports four types of operations:

- *Scalar-only operations*
- *Vector-only operations* on page 2-6
- *Vector operations with a scalar source* on page 2-6
- *Scalar operations in vector mode* on page 2-7.

2.2.1 Scalar-only operations

An operation is a scalar-only operation if the operands are treated as scalars and the result is a scalar. There are two ways to perform a scalar-only operation:

- Setting the LEN field of the Floating Point Status and Control Register (FPSCR) to 0 selects a vector length of 1. For example, if LEN = 0, then the following operation:

```
FADDS s12, s21, s22
```

 results in the sum of the single-precision values in s21 and s22 being written to s12. See Figure 3-1 on page 3-2 for details of LEN and the *Floating Point Status and Control Register (FPSCR)*.
- If the LEN field of the FPSCR is not 0, the operation will be scalar-only if the destination register is in bank 0. For example, regardless of the value of LEN, the following operation:

```
FADDD d2, d5, d14
```

results in the sum of the double-precision values in d5 and d14 being written to d2. No other operation will be performed by this instruction even though the LEN field value is nonzero. *Scalar operations in vector mode* on page 2-7 shows an example where scalar and vector operations are intermixed.

Some operations can only operate on scalar data regardless of the value of the LEN field or destination register bank number. These operations include:

- compare instructions FCMP, FCMPZ, and FCMPE
- integer conversion instructions FTOUI, FTOUIZ, FTOSI, FTOSIZ, FUITO, and FSITO
- precision conversion instructions FCVTDS and FCVTSD.

2.2.2 Vector-only operations

Vector-only operations require the LEN field to be nonzero and the destination and Fm registers are not in bank 0.

For example, if LEN = 3 (an effective vector length of 4) and STRIDE = 0 (for a vector stride of one) the following:

```
FMACS s16, s0, s8
```

results in the following being performed as an atomic operation:

```
FMACS s16, s0, s8
FMACS s17, s1, s9
FMACS s18, s2, s10
FMACS s19, s3, s11.
```

2.2.3 Vector operations with a scalar source

The VFPv1 architecture provides a mechanism for a vector to be operated on by a scalar operand. The destination must be a vector (not in bank 0) and the Fm operand must be in bank 0.

For example, if LEN = 1 (an effective vector length of 2) and STRIDE = 0 (for a vector stride of one) the following operation:

```
FMULD d12, d8, d2
```

results in the following scalar operations being performed as an atomic operation:

```
FMULD d12, d8, d2
FMULD d13, d9, d2.
```

This effectively scales the two entry vectors (d8, d9) by the value in d2 and writes the new vector to d12 and d13.

2.2.4 Scalar operations in vector mode

You can intermix scalar and vector operations by carefully selecting the source and destination register. Combining the second method of performing scalar-only operations with nonscalar operation means that it is not necessary to change the LEN field to 0 from a nonzero value in order to perform scalar operations.

For example, if LEN = 1 and STRIDE = 0 (for a vector stride of one), then the following operations:

```
FABSD d4, d8
FADDS s0, s0, s31
FMULS s24, s26, s1
```

results in the following operations being performed:

```
FABSD d4, d8           ; a vector double-precision ABS operation
FABSD d5, d9           ; on registers (d8, d9) to (d4, d5)
FADDS s0, s0, s31     ; a scalar increment of s0 by s31
FMULS s24, s26, s1    ; a vector(s26, s27) scaled by s1
FMULS s25, s27, s1    ; and written to (s24, s25)
```

Table 2-2 to Table 2-5 on page 2-8 summarize the four types of operations possible in the VFPv1 architecture. *Any* refers to the availability of all registers in the precision for specified operand. VFP10 Rev 0 supports all these operations in hardware. S refers to a scalar register only with a single register on each of the Fd, Fn, and Fm operands. V refers to a vector register with multiple registers for Fd and Fn, and possibly for Fm as well.

Table 2-2 Single-precision three-operand register usage

LEN field	Fd	Fn	Fm	Operation type
0	Any	Any	Any	S = S op S or S = S op S * S
Non-0	0-7	Any	Any	S = S op S or S = S op S * S
Non-0	8-31	Any	0-7	V = V op S or V = V op V * S
Non-0	8-31	Any	8-31	V = V op V or V = V op V * V

Table 2-3 Single-precision two-operand register usage

LEN field	Fd	Fm	Operation type
0	Any	Any	S = op S
Non-0	0-7	Any	S = op S
Non-0	8-31	0-7	V = op S
Non-0	8-31	8-31	V = op V

Table 2-4 Double-precision three-operand register usage

LEN field	Fd	Fn	Fm	Operation type
0	Any	Any	Any	S = S op S or S = S op S * S
Non-0	0-3	Any	Any	S = S op S or S = S op S * S
Non-0	4-15	Any	0-3	V = V op S or V = V op V * S
Non-0	4-15	Any	4-15	V = V op V or V = V op V * V

Table 2-5 Double-precision two-operand register usage

LEN field	Fd	Fm	Operation type
0	Any	Any	S = op S
Non-0	0-3	Any	S = op S
Non-0	4-15	0-3	V = op S
Non-0	4-15	4-15	V = op V

2.2.5 Transferring data between ARM registers and VFP10 Rev 0 registers

Floating-point data can be transferred between ARM registers and VFP10 Rev 0 registers using the MCR and MRC instructions. Single-precision data can be stored and manipulated in a single ARM register. Double-precision data requires two ARM registers. No exceptions are possible on MCR and MRC instructions. The format of the data in the ARM register or registers after an MRC operation will be the same format as in memory and in the VFP10 Rev 0 registers.

2.2.6 Data storage in memory

The format for accessing data stored in memory is determined by the CP15 control register B bit. The ARM supports both little-endian and big-endian access formats in memory.

The ARM stores 32-bit words in memory with the LSB in the lowest byte of memory regardless of the endianness selected. For a store of a single-precision data value the LSBs are located at the target address with the lower two bits of the address set to 00. The MSB is at the target address with the lower two bits set to 11. To load the single-precision data to an ARM register or to a VFP10 Rev 0 register you must set the lower two bits of the target address to 00.

For single-precision data, Table 2-6 shows the data storage in memory and the address access to each byte in both little-endian and big-endian access modes. In the examples in Table 2-6 and Table 2-7 on page 2-10 the target address is 0x40000000.

Table 2-6 Single-precision data memory images and byte addresses

Single-precision data bytes	Address in memory	Little-endian byte address	Big-endian byte address
MSB Bits[31:24]	0x40000003	0x40000003	0x40000000
Bits[23:16]	0x40000002	0x40000002	0x40000001
Bits[15:8]	0x40000001	0x40000001	0x40000002
LSB Bits[7:0]	0x40000000	0x40000000	0x40000003

For double-precision data, the location of the two words which comprise the data are stored in different locations for little-endian and big-endian data access formats. Table 2-7 shows the data storage in memory and the address to access each byte in little-endian and big-endian access modes.

Table 2-7 Double-precision data memory images and byte addresses

Double-precision data bytes	Little-endian		Big-endian	
	Address in memory	Byte address	Address in memory	Byte address
MSB Bits[63:56]	0x40000007	0x40000007	0x400000003	0x40000000
Bits[55:48]	0x40000006	0x40000006	0x40000002	0x40000001
Bits[47:40]	0x40000005	0x40000005	0x40000001	0x40000002
Bits[39:32]	0x40000004	0x40000004	0x40000000	0x40000003
Bits[31:24]	0x40000003	0x40000003	0x40000007	0x40000004
Bits[23:16]	0x40000002	0x40000002	0x40000006	0x40000005
Bits[15:08]	0x40000001	0x40000001	0x40000005	0x40000006
LSB Bits[7:0]	0x40000000	0x40000000	0x40000004	0x40000007

The memory image for the data is identical for both little-endian and big-endian within word data items. The hardware performs the transformations of the address in order to provide both little-endian and big-endian addressing to the programmer.

2.2.7 Maintaining consistency in register precisions

The VFP10 Rev 0 register file stores both single-precision and double-precision data in the same registers. For example, D6 occupies the same registers as S12 and S13. The usable format of the register or registers is a function of the last load or arithmetic instruction which wrote to the register or registers.

The hardware does not do any checking of the register contents to enforce consistent use of the current register format with the precision of the current operation. Inconsistent use of the registers is possible but UNPREDICTABLE. The data is interpreted by the hardware in the format required by the instruction regardless of the latest store or write operation to the register. It is the task of the compiler or programmer to maintain consistency in register usage.

2.2.8 Decoding the register file

Decoding into the register file involves the most significant four bits of the register index. For operations involving double-precision operands or destinations, the M, N, and D bit corresponding to a double-precision access must be zero. For single-precision register indices the most significant four bits will be in the Fx bit positions (where x is m, n, or d), and the least significant bit in the M, N, or D bits for each instruction format. Figure 2-3 on page 2-12 shows register file decoding. See the *ARM Architectural Reference Manual* for instruction formats and the positions of these bits.

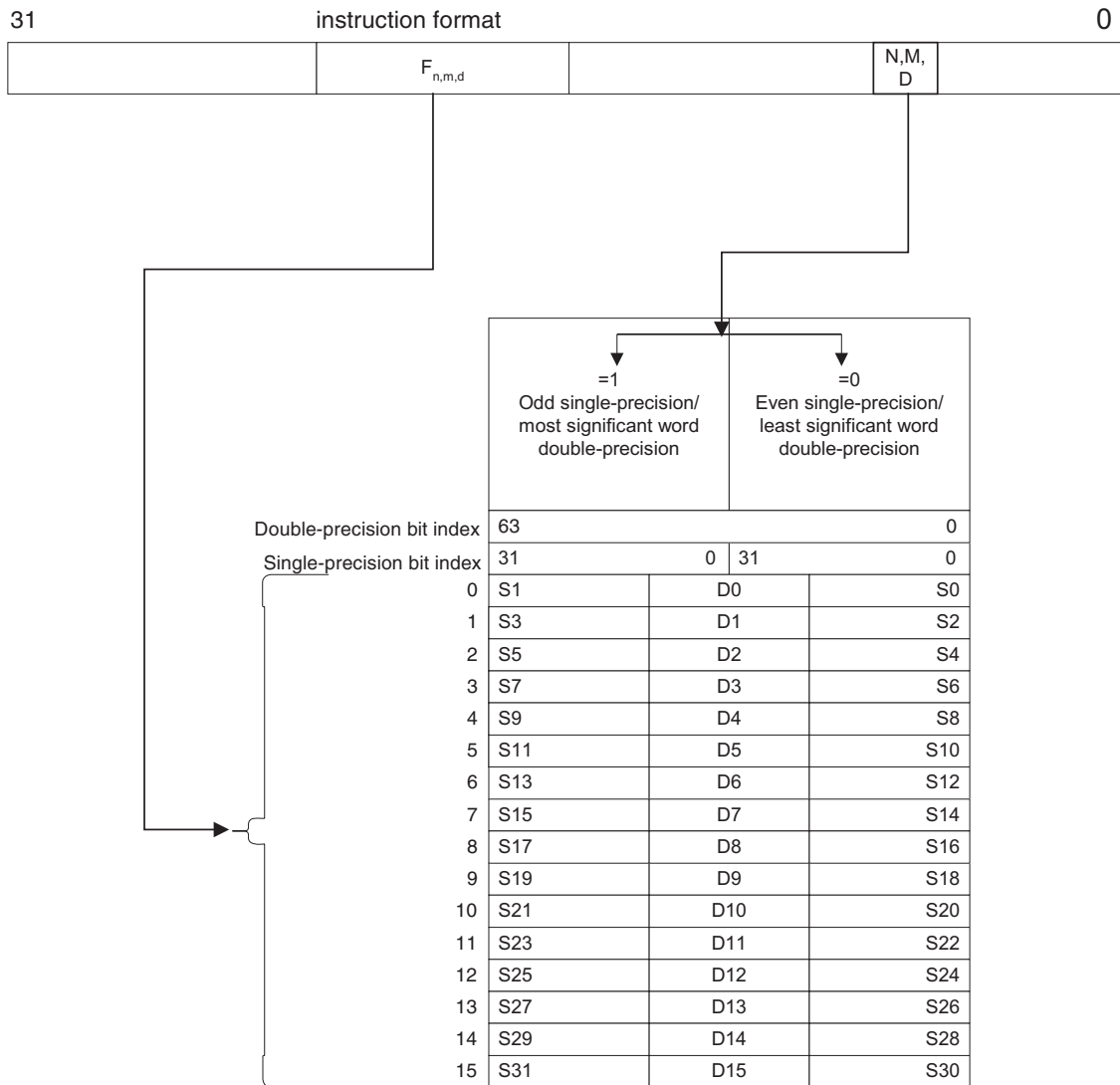


Figure 2-3 Register file decoding

2.2.9 Loading operands from ARM registers

To load a double-precision operand from two ARM registers each register is loaded to the corresponding single-precision register. A subsequent double-precision operation on this data reads the pair of registers as containing a double-precision value.

2.3 The VFP10 Rev 0 pipeline

The VFP10 Rev 0 pipeline stages are:

- Issue
- Decode
- Execute 1
- Execute 2
- Execute 3
- Execute 4
- Write.

The front end of the VFP10 Rev 0 pipeline mirrors the ARM10 pipeline, beginning with the Issue/Decode stage boundary. The instruction is issued from the ARM core to the VFP during the Issue stage. The instruction is issued in sequence with respect to the ARM code stream. Only a single instruction is issued every cycle regardless of whether it is an ARM or coprocessor instruction. During the following VFP Decode stage, the VFP decoder determines whether the instruction proceeds to the first *Four-stage Multiply-Accumulate* (FMAC) Execute stage or the load/store Execute stage. In addition, the operand registers are read in the Decode stage for arithmetic instructions or stores.

The VFP decoder can hold an instruction in the Decode stage for several reasons:

- if a register conflict exists between the incoming instruction and instructions already present in the VFP pipeline
- if the incoming instruction requires resources (the FMAC pipeline or the load/store pipeline) that are being used by a previous multicycle (vector or load/store multiple) instruction
- if the incoming instruction is serializing, and either the VFP FMAC pipeline or load/store pipeline is active.

The ARM core can also hold up the instruction in either the VFP Decode or first Execute stage in certain conditions.

If any of the conditions described above exist and the instruction is held in Decode, the entire ARM code stream is held. For this reason, code should be scheduled to minimize the number of stall cycles incurred.

2.3.1 The VFP10 Rev 0 pipeline stages

shows the VFP10 Rev 0 pipeline. The top half of the diagram shows the FMAC pipeline and the lower half, reading in from Port S, the load/store pipeline. After the Decode stage, the pipeline splits into two. One section consists of the FMAC for arithmetic instructions and the other mirrors the ARM pipeline for load/store instructions.

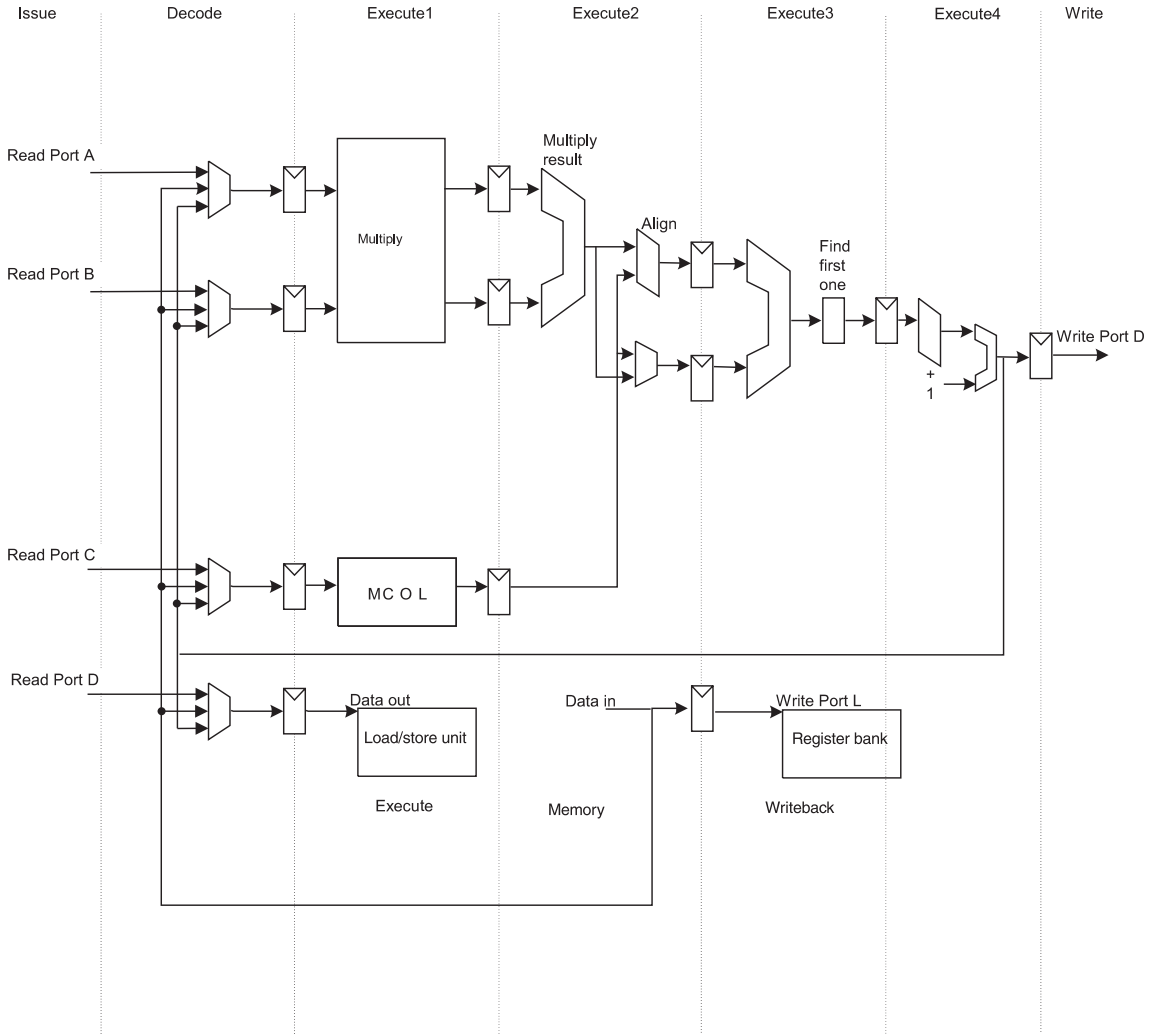


Figure 2-4 VFP10 Rev 0 pipeline

2.3.2 The four-stage multiply-accumulate (FMAC) functional block

The ARM VFP10 Rev 0 incorporates an FMAC functional block which executes the following instructions in hardware:

FMAC, FNMAC, FMSC, FNMSC	Multiply-accumulate.
FMUL, FNMUL	Multiply.
FADD	Add.
FSUB	Subtraction.
FABS	Absolute value.
FNEG	Negation.
FUITO, FSITO, FTOUTI, FTOSI	Integer conversion.
FCMP	Comparison.
FCVT	Format conversion.
FDIV	Divide.
FSQRT	Square root.
FCPY	Copy register.

The FMAC performs a chained multiply-add operation with the following sequence of operations:

1. The product of the operands in the Fn and Fm registers are multiplied.
2. The product is rounded to the current rounding mode.
3. The result is summed with the operand in the Fd register.
4. The sum is rounded to the current rounding mode and written to the Fd register.

The top half of the pipeline diagram Figure 2-4 on page 2-14 shows the relevant blocks and ports.

2.3.3 Load/store pipeline

The load/store pipeline handles all of the instructions that involve data transfer to and from the ARM, including loads, stores, move to coprocessor register (MCR), and move from coprocessor register (MRC). It remains synchronized with the ARM load/store pipeline for the duration of the instruction. It begins with the Execute stage, during which data is transferred to the ARM for store-type instructions. Load-type instruction data is received during the following memory stage.

2.3.4 Parallel execution of load/store and vector instructions

Vector instructions and load/store multiple instructions only occupy the decoder during the first iteration. Subsequent iterations are handled outside the Decode stage, freeing the Decode stage for the next floating-point instruction. The register file has independent read ports for load/store instructions and arithmetic instructions. This allows load/store instructions to be issued while a vector instruction is active or a vector instruction to be issued while a load/store is active. The two pipelines run independently and in parallel provided no register conflicts exist between any of the source or destination registers in either operations. As an example, consider the following code sequence with the VFP set up to a vector length of four:

```
FLDMS    r0, {s12-s15}
FADDS    s8, s16, s24
```

The load instruction is sent from the decoder to the load/store pipeline in cycle 1, and the first iteration (loading of registers s12, s13) is sent to the Execute stage. During cycle 2, the FADDS occupies the decoder. It is allowed to proceed to the Execute stage of the FMAC pipeline since the pipeline is available and the source and destination registers are non-overlapping with the FLDMS. shows VFP10 registers and their contents.

Table 2-8 VFP10 Rev 0 register contents illustrating parallel execution

Cycle	1	2	3	4	5
Decode	FLDMS	FADDS	-	-	-
Load/store pipeline	s12, s13	s14, s15	-	-	-
FMAC pipeline	-	s8, s16, s24	s9, s17, s25	s10, s18, s26	s11, s19, s27

2.3.5 Multicycle instructions

A multicycle or vector instruction is held in the Decode stage until all the registers required to complete the instruction are available. When an instruction enters the first Execute or Execute1 stage of either pipeline it locks all operand and result registers. When an operand has been read from the register file, the lock on that register is cleared.

When a result becomes available, the lock on the destination register is cleared. For registers that are both operands and results, the lock is not cleared until the result is available.

Examples of multicycle locking mechanisms

Example 2-1 to Example 2-3 (all with a vector length of two) illustrate the locking mechanism.

Example 2-1 Vector operation followed by load with write-after-read hazard

FMACS	s8, s16, s24
FLDS	s17, [r0, #0]

In Example 2-1, the first instruction locks all operands and results used by the entire vector operation. In this case, these are registers s8 and s9, s16 and s17, and s24 and s25. The operands used for the first iteration (s16 and s24) are immediately unlocked, since the registers are read during the first Decode cycle. The operand registers used for the next iteration, registers s17 and s25, are unlocked the next cycle. The load is stalled until the second iteration of the FMAC has read register s17.

Example 2-2 Load followed by vector operation with read-after-write hazard

FLDS	s17, [r0, #0]
FMACS	s8, s16, s24

In Example 2-2 the load locks register s17 until the data becomes available at the end of the Execute 2 cycle. Until all operands become available for all iterations of the FMAC, it is not allowed to proceed into Execute for a two-cycle stall.

Example 2-3 Vector operation followed by store with false hazard

FMACS	s8, s16, s24
FSTS	s17, [r0, #0]

In Example 2-3, a vector arithmetic operation is followed by a store. Although the second instruction only reads from the same register as the first instruction operand, the second instruction stalls until register s17 is read. (In this case, only one cycle would be wasted.) Although there is no hazard, the second instruction stalls since the locks are set and checked without regard to operation (read/write).

2.3.6 Pointers for writing optimal VFP code

The fact that the pipeline can execute some sequences of operations in parallel allows code you write to take advantage of this. Here are some brief notes about writing such code:

- You can schedule most scalar operations immediately following each other, provided there is no read-after-write hazard. Scalar double-precision multiply or multiply-accumulate instructions should be followed by either a single ARM or load/store instruction instead of an arithmetic VFP instruction.
- Try to avoid vector divides and square roots. The FMAC pipeline is unavailable for the duration of the vector divide or square root. A double-precision divide takes 33 cycles. A single precision divide takes 19 cycles.
- You must double-buffer looped vector instructions. The vector banks must be divided in half. Arithmetic operations on one half of the bank must be followed by loads or stores to the other bank.
- The first VFP instruction following a branch mispredict is serialized and will wait for all VFP instructions prior to the branch to complete. Avoid placing long load/store instructions or divide/square-root instructions before branches that are not predicted correctly a high percentage of the time.
- Moves to and from control registers are serializing. Avoid placing these in loops or time-critical code.

2.3.7 VFP10 Rev 0 treatment of branch instructions

VFP10 Rev 0 does not provide branch instructions. Instead, the result of a floating-point compare instruction can be stored in the ARM condition code flags by loading the FPSCR register to the Program Counter using the FMSTAT instruction. This allows the ARM branch instruction to be used for executing conditional FP code.

2.4 Implementation-specific control registers

Two implementation-specific mechanisms deal with exception handling:

- the support code exception status word FPEXC
- instruction word register FPINST.

These mechanisms are described in the following sections:

- *Implementation-specific system register moves*
- *The support code exception status word FPEXC* on page 2-20
- *Instruction word register (FPINST)* on page 2-21
- *Access to control registers* on page 2-21.

2.4.1 Implementation-specific system register moves

FMRX and FMXR instructions, the VFP instructions to transfer ARM registers to VFP system registers and the other way round, can be used to access FPEXC. The bit field definitions are as follows:

Table 2-9 FPEXC bit field definitions bits 31-8

Bit	Name	Description
31	EX	Exception status bit. If set, the VFP is in exception mode and will cause all following VFP instructions (except FMRX and FMXR of the FPEXC, FPINST, or FPSID registers in a Privileged Mode) to assert CPBOUNCEE .
30	EN	Enable VFP: 0 = disabled (default) 1 = enabled.
[29:11]	Reserved	-
[10:8]	VECITR	Vector iteration count. This field will contain the number of iterations remaining in a vector operation in which an iteration was exceptional.

Table 2-10 lists the iterations for vector operations.

Table 2-10 Vector iteration count bit values

Bit values for [10:8]	Iterations for FMULD/FDIV/FSQRT	Iterations for all other operations
000	0	1
001	1	2
010	2	3
011	3	4
100	4	5
101	5	6
110	6	7
111	7	0

Table 2-11 lists the FPEXC bit field definitions for bits 7-0.

Table 2-11 FPEXC bit field definitions bits 7-0

Bit	Name	Description
[7:4]	Reserved	-
3	UF	Underflow exception detected.
2	OF	Overflow exception detected.
1	DZ	Divide-by-zero exception detected.
0	IV	Invalid exception detected.

2.4.2 The support code exception status word FPEXC

In a bounce situation, the exceptional condition is recorded in the FPEXC register to provide support code information sufficient to recover from the exceptional condition or report the condition to a system or user software exception handler. The format of the FPEXC register is shown in Figure 2-5 on page 2-21. Bits [31:30] are architecturally-defined and must be present in all implementations of the VFPv1 architecture.

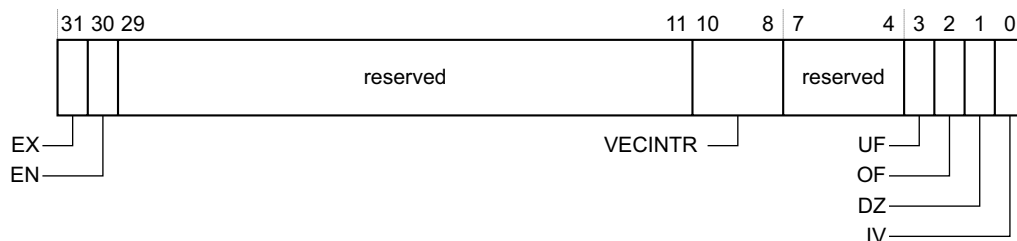


Figure 2-5 FPEXC register format

2.4.3 Instruction word register (FPINST)

The instruction word of the operation in the E stage is copied to the FPINST register with the condition code bits [31:28] forced to 1110, the AL (always) condition. The FPINST register has the same format as a CDP instruction. In the case of an exception in a vector operation the contents of the register fields in the FPINST reflects the source and destination registers of the failing iteration. This register is used by the support code to complete the instruction when any of the operands is an unsupported data type. This register is also used by the support code when a potential exception condition is detected.

The instruction word register can be accessed with an FMRX/FMXR by specifying Fn as 0b1001 in Privileged Mode. It is a read/write register.

2.4.4 Access to control registers

These control registers can only be accessed in privileged mode as detailed in Table 2-12.

Table 2-12 Access to control registers through FMRX and FMXR

Register	Trigger exception processing?	Legal modes
FPINST	No	Privileged
FPEXC	No	Privileged

2.4.5 The FPSID register

Figure 2-6 on page 2-22 shows the bit fields in the FPSID register.

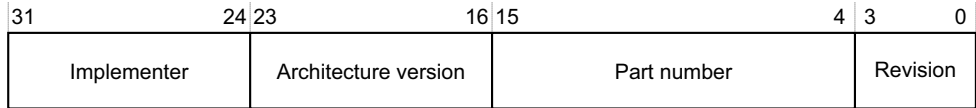


Figure 2-6 FPSID register format

The value for this register is `0x410000A0`.

Table 2-13 gives the meanings of the bit fields in FPSID.

Table 2-13 Bit fields for the FPSID register

Bits	Meaning
Bits [31:24]	Contain the ASCII code of implementers trademark, 0x41 = ARM.
Bits [23:16]	Contain the architecture version.
Bits [15:4]	Contain the 3-digit part number.
Bits [3:0]	Contain the revision number for the processor.

2.5 Rationale for implementation choices

Exception processing in the ARM VFP10 Rev 0 favors fast execution of high percentage operations and reasonable execution time for handling special and rare cases. In order to execute typical operations at the highest possible speed, handling of exceptional cases, with the exception of inexact, is relegated to the software support code.

2.5.1 Handling of IEEE 754 special operands

Input operands which are not in the range of normal operands for the target precision, such as infinities, NaNs, and denormal values (referred to collectively as unsupported values), are processed by the software support code. The vast majority of embedded software and user code does not generate these exceptional conditions or operate on these values.

2.5.2 Imprecise exception model

The ARM VFP10 Rev 0 returns exceptional cases imprecisely. For scalar and vector CDP operations, two checks are made on the input operands in the first Execute stage. The input operands are checked for unsupported values. In parallel, the exponents are operated on to produce a preliminary final exponent result. (The result of this stage is preliminary in that the final exponent could be greater due to mantissa overflow and rounding or less due to cancellation in an effective subtraction.) A check is made on this preliminary exponent to determine if the final exponent could be in the exceptional range for the destination precision. If the check of the preliminary final exponent identifies the operation as possessing the potential to cause an exception, the instruction is bounced to support code for completion.

Chapter 3

Exceptions

This chapter contains detailed information about the ARM VFP10 Rev 0 combination of hardware and software support used to provide floating-point functionality. Refer to the *ARM Architecture Reference Manual* for further information.

Refer to the glossary of this manual for VFPv1 terms. This chapter contains the following sections:

- *Support code* on page 3-2
- *Exception processing for scalar and vector instructions* on page 3-9
- *Invalid operation* on page 3-14
- *Division by zero* on page 3-16
- *Overflow* on page 3-17
- *Underflow* on page 3-18
- *Inexact result* on page 3-19
- *Input exceptions* on page 3-21
- *Arithmetic exceptions* on page 3-23.

3.1 Support code

This chapter refers to the FPSCR. Figure 3-1 shows the bit fields in summary form. The FPSCR can be accessed in any mode using FMRX/FMXR instructions

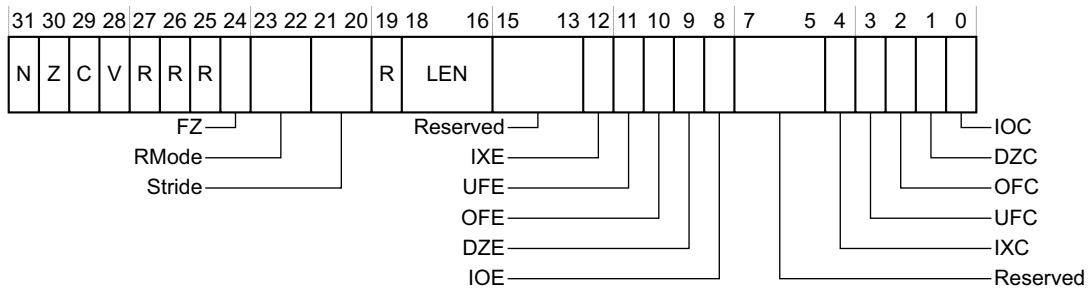


Figure 3-1 User status and control bit fields summary

The ARM VFP10 Rev 0 provides floating-point functionality through a combination of hardware and software support. Floating-point instructions are normally executed by the VFP hardware coprocessor. However, the coprocessor may use the coprocessor interface signals between it and the ARM10 to refuse to accept a floating-point instruction. This asserts the **CPBOUNCEE** signal which causes the ARM10 undefined instruction exception to occur. This is known as *bouncing* the instruction. When an instruction is bounced, software installed on the ARM10 undefined instruction vector determines why the coprocessor rejected the instruction and takes appropriate remedial action. This software is called the *VFP support code*. The support code has two components:

- a library of routines which perform floating-point arithmetic functions.
- a set of exception handlers which process exception conditions.

There are two main reasons for bouncing an instruction:

- potential floating-point arithmetic exceptions
- illegal instructions.

These are described in *Potential floating-point arithmetic exceptions* on page 3-5 and *Illegal instructions* on page 3-7, together with the remedial action taken by the support code in each case.

3.1.1 An IEEE 754-compliant implementation

The VFP coprocessor and support code together provide IEEE 754-compliant implementations of all the floating-point operations supplied by the VFP instruction set. Unless a floating-point exception occurs and the enable bit of the exception in the FPSCR is set, it will appear to the program that the floating-point instruction was

executed by the hardware. However, if the instruction is bounced and processed by the support code, it takes significantly more cycles than normal to produce the result. This only happens for cases whose incidence is typically nil or very low, and is a common practice in the industry.

The VFP support code also includes routines which perform administrative tasks such as initializing the VFP system.

3.1.2 Complete implementation of IEEE 754

The following operations from the IEEE 754 standard are not supplied by the VFP instruction set:

- Remainder.
- Round floating-point number to integer-valued floating-point number.
- Binary-to-decimal conversions.
- Decimal-to-binary conversions.
- Direct comparison of single-precision and double-precision values. This operation is not used by high-level languages like C, which first perform a single-to-double conversion and then compare the double-precision values.

To obtain a complete implementation of the IEEE 754 standard, the VFP coprocessor and support code must be augmented with library functions that implement the above operations.

3.1.3 IEEE 754 implementation choices

VFPv1 specifies how various implementation choices allowed by the IEEE 754 standard are to be made. Full details can be found in the *ARM Architecture Reference Manual*.

———— **Note** —————

References to IEEE 754 in this section appear in italicized text.

Further implementation choices are made within the VFP10 Rev 0 floating-point system, about:

- which cases are handled by the VFP10 Rev 0 hardware
- which cases are bounced to the support code.

In order to execute frequently-encountered operations as fast as possible and minimize silicon area, handling of infrequently occurring values and exceptions is relegated to the support code. Accordingly, operations on infinities, NaNs and denormal values (collectively called *unsupported values*) are processed by the support code. So are all floating-point exceptions except untrapped inexact result exceptions. The majority of embedded software and user code do not operate on these values or generate these exceptions.

The following list summarizes the main choices. Where relevant, it also describes briefly how the VFP10 Rev 0 and its support code handles them:

- The supported floating-point formats are single-precision and double-precision. No extended format is supported. The supported integer formats are:
 - unsigned 32-bit integers
 - two's complement signed 32-bit integers.
- All single-precision and double-precision values with maximum exponent field and nonzero fraction field are valid NaNs. A NaN is signaling or quiet depending on whether its most significant fraction bit is 0 or 1 respectively. Two NaN values are treated as different NaNs if they differ in any bit. There are precise rules about:
 - which NaN values are generated by untrapped invalid operation exceptions
 - how quiet NaNs propagate through floating-point operations
 - how format conversions of NaNs are performed.

The VFP10 Rev 0 hardware handles all NaNs identically. Operations which copy NaNs without a change of format copy all bits of a NaN (except that the sign bit might be changed in the cases of FNEG and FABS). All other operations with NaN inputs are bounced to the support code. The support code implements the signaling/quiet NaN distinction and the rest of the rules about NaNs.

- Copying a NaN without a change of format does not raise the invalid operation exception, nor is the instruction bounced. All load/store operations between the ARM and the VFP register transfers involving one or more floating-point value are treated as copying the value(s) without a change of format. The FCPY, FNEG and FABS instructions are also treated as copying a value without a change of format, with the latter two changing the sign bit of the value, as described in the Appendix to the IEEE 754 standard.
- Comparison results are delivered as condition codes. The condition codes used are chosen so that subsequent conditional execution of ARM instructions can test the predicates defined in the standard.

The VFP10 Rev 0 hardware handles most comparisons of numeric values itself, generating the appropriate condition code depending on whether the result is *less than*, *equal*, or *greater than*. Other comparisons are bounced to the support code, which generates an invalid operation exception and/or an appropriate condition code result.

- For the underflow exception, the *after rounding* form of *tininess* and the *denormalization loss* form of *loss of accuracy* are used.

The VFP10 Rev 0 hardware only detects whether there is a risk of *tininess* occurring. If there is, the operation is bounced and the support code implements the rules about when the underflow exception occurs, including precise determination of whether *tininess* and *loss of accuracy* did in fact occur. For more details, see *Underflow* on page 3-18 and *Arithmetic exceptions* on page 3-23.

- Exception traps can be requested by setting the relevant trap enable bits in the FPSCR. The facilities available to trap handlers are system-dependent.

The VFP10 Rev 0 hardware ensures that all potentially trapping operations are bounced to the support code. The only exceptions it handles itself are untrapped inexact result exceptions. The basic exception trap mechanism is implemented in the support code. The precise set of facilities available will also depend on the rest of the system.

3.1.4 Potential floating-point arithmetic exceptions

A floating-point instruction may be bounced because execution of an earlier floating-point instruction encountered a potentially exceptional condition. In this case, the earlier instruction is called the *potentially exceptional instruction* and the instruction that actually bounced is called the *trigger instruction*. This is an *imprecise* exception reporting mechanism.

When such a bounce occurs, the hardware sets the EX bit in the FPEXC register and set FPINST to a copy of the potentially exceptional instruction. This condition in the VFP10 Rev 0 is referred to as the *exceptional state*. Any trigger instruction currently in the VFP10 Rev 0 D stage, or issued after entering the exceptional state, is bounced.

The hardware detects potential exceptions *pessimistically*. This means an instruction bounce always occurs when there is a floating-point exception (other than a disabled inexact exception) but also occurs in some rare cases when there is no floating-point exception.

The remedial action is performed as follows:

1. The support code starts with reading the FPEXC register, determining that there is a potential exception because the EX bit is set.

2. The FPEXC register is written to clear the EX bit (failure to do this results
3. in an infinite loop of exception traps when the support code next accesses the VFP hardware).
4. The FPINST register is read to determine the earlier instruction that caused the potential exception.
5. The support code then decodes the earlier instruction, reads its operands (including implicit ones such as the FPSCR rounding mode and vector length), and determines whether a floating-point exception occurred.
6. Then:
 - If no floating-point exception occurred, the support code determines the correct result of the operation, and writes it to the destination register.
 - If one or more floating-point exceptions occurred, but all of them were disabled, the support code determines the correct result of the instruction, writes it to the destination register, and sets the corresponding cumulative exception bits in the FPSCR.
 - If one or more floating-point exceptions occurred and at least one of them was enabled, the support code calls the user-provided trap handler for that exception. (The only cases where more than one exception can occur are overflow with inexact, and underflow with inexact. In these cases, calling the overflow or underflow trap handler takes precedence over calling the inexact trap handler.)
7. If the potentially exceptional instruction specified a vector operation, any vector iterations after the one that encountered the potentially exceptional condition will not have been executed by the hardware. The support code will repeat steps 4 and 5 above for any such iterations. See *CDP vector instructions* on page 3-9 for more details.

Note

Steps 1-6 imply that the support code must be capable of performing steps 4 and 5 for any operation/operands combination, not just for those combinations which the VFP10 Rev 0 hardware treats as potentially exceptional.

1. Once the support code has completed processing the potentially exceptional instruction, it returns to the program containing the trigger instruction. The original bounce of the trigger instruction always occurs during the initial stage of the hardware coprocessor handshake, and prevents any operation(s) the trigger instructions specify from completing.

Accordingly, the support code returns to the address of the trigger instruction, causing the ARM to refetch the trigger instruction from memory and re-issue it to the VFP10 Rev 0. Unless another bounce occurs, this results in the trigger instruction being fully executed after the return. Returning in this fashion is known as *retrying* the trigger instruction.

The support code may be written to utilize the VFP10 Rev 0 hardware for its internal calculations, provided recursive bounces are avoided or handled correctly, and provided care is taken to restore the state of the original program on returning to it. This last requirement can be difficult to satisfy if the original program was executing in FIQ mode or in undefined instruction mode. It is legitimate for support code to disallow or restrict the use of VFP instructions in these two processor modes.

3.1.5 Illegal instructions

If there is no potential floating-point exception from an earlier instruction, the current instruction can still be bounced because it is architecturally undefined in some way. When this happens, the support code finds that the EX bit is zero and handles the instruction by passing it to a system-specific *undefined instruction* routine. The instruction that caused the bounce is contained in the memory word pointed to by `r14_undef - 4`.

It is possible that both conditions for an instruction to be bounced occurs simultaneously. This happens when an illegal instruction is encountered and there is also a potential floating-point exception from an earlier instruction. When this happens, the EX bit is 1 and the support code processes the potential exception in the earlier instruction. If and when it returns, it causes the illegal instruction to be retried and the sequence of events described in the paragraph above occurs.

The following types of instruction are architecturally required to be treated as illegal instructions, as described above:

- instructions with opcode bit combinations defined as Reserved in the architectural specification
- load/store instructions with (P, W, and U) bit combinations marked as UNDEFINED
- FMRX/FMXR instructions to or from a control register that is not defined
- User mode FMRX/FMXR instructions to or from a control register that can only be accessed in privileged mode.

Certain types of instruction do not have architecturally-defined behavior, even to the extent of causing the ARM undefined instruction trap to be entered. They may be treated as illegal instructions, but this should not be relied upon. The types of instructions are:

- Load/store multiple instructions with a transfer count of zero or greater than thirty-two. In this implementation this case is bounced.
- A vector operation that has a combination of precision, length, and stride that would cause the vector (not the bank) to wrap around more than once (more than one access to the same register). In this implementation this case is bounced.
- A vector operation with overlapping source and destination register addresses that are not exactly the same. In this implementation this case is not bounced and the results are UNPREDICTABLE.

3.2 Exception processing for scalar and vector instructions

This section deals with exceptions processing for:

- *Load/store instructions and ARM transfer instructions*
- *CDP scalar instructions*
- *CDP vector instructions*
- *Examples of exception detection for vector operations on page 3-10.*

3.2.1 Load/store instructions and ARM transfer instructions

No floating-point exceptions are possible with load, store, MCR, and MRC instructions. They are valid trigger instructions if the VFP10 Rev 0 is in the exceptional state when they are issued or started.

3.2.2 CDP scalar instructions

A scalar CDP determined to be exceptional causes the FPINST register to be loaded with the instruction word for the offending instruction and the FPEXC to be set with the exception condition. Once the exception is detected, the offending instruction is blocked from further execution while any previous instructions not yet retired is allowed to retire.

Two possible conditions may exist in the following situation:

- If there is not a floating-point instruction (CDP or load/store) in the VFP10 Rev 0 Decode stage, the VFP10 Rev 0 waits until one is issued. The next trigger instruction is bounced.
- If there is a trigger instruction in the VFP10 Rev 0 Decode stage, it is bounced in the cycle after the exception is detected on the offending instruction.

The FMXR and FMRX instructions accessing the FPINST or FPEXC registers are not trigger instructions in a privileged mode, and is bounced if it was the instruction following the offending instruction in any of the above situations.

The trigger instruction which was in the VFP10 Rev 0 Decode stage is retried by the ARM when the ARM returns from exception processing.

3.2.3 CDP vector instructions

For vector instructions any iteration may be exceptional. If an exceptional condition is detected for a vector iteration, the vector iterations issued prior to the offending operation are allowed to complete and retire.

Once the offending iteration of the vector operation is found to be potentially exceptional the following sequence of operations occurs:

1. The FPINST register is loaded with the operation instruction word.
2. The source and destination register addresses are modified to point to the source and destination registers of the offending iteration.
3. The EX bit in the FPEXC register is set.
4. The VECITR field is written with the coded number of the offending iteration. (See *Implementation-specific control registers* on page 2-19 for the encoding of the VECITR field.)

The VFP10 Rev 0 allows for parallel execution of CDP and load/store operations. Either operation may be issued first, and the other issued and begun if no register conflicts on the source or destination registers exist and the VFP10 Rev 0 is not in the exceptional state.

———— **Note** —————

The requirement for no conflicts with source registers is maintained in order to facilitate support code handling of exceptional operations.

In the case of a vector operation with an exceptional iteration, any load/store operation begun before the issue of the vector instruction was detected as exceptional, or after the issue of the vector instruction but before the offending iteration, is allowed to complete and retire. The exception is signaled as a bounce on the first floating-point instruction to be issued after the exception condition has been detected.

3.2.4 Examples of exception detection for vector operations

In Example 3-1 to Example 3-4 on page 3-12 code fragments illustrate the exception detection mechanism of the VFP10 Rev 0 for vector operations. The LEN field in the FPSCR is set to 0b011, for a vector length of four.

In Example 3-1 the FLDMD in Inst A will complete regardless of the exception status of the CDP (Inst B) following. Inst C will bounce if the FMULD (Inst B) is exceptional in the first iteration. Otherwise, Inst C will issue and complete and the FMULS (Inst D) will bounce and trigger exception handling for Inst B.

Example 3-1 FLDMD completes regardless of a subsequent CDP

```

FLDMD R2, {D2-D6} ;Inst A load multiple 5 dp data
FMULD D8, D12, D8 ;Inst B vector dp multiply of length 4
    
```

FSTMD R3, {D8-D11};Inst C store multiple 4 dp data
 FMULS S0, S1, S1 ;Inst D scalar sp multiply S0 = S1*S1

In Example 3-2 the FMULD in Inst A is determined to have a potential underflow exception in the second iteration. The FLDMD in Inst B is issued before the exception in the second iteration of Inst A is detected. It is allowed to issue, start, and complete. The FSTMD in Inst C triggers the exception and be restarted upon completion of the Support Code exception routine. If the vector were exceptional on an iteration beyond the second, it is possible for Inst C also to be issued and started. In that case, Inst D triggers the exception.

Example 3-2 Exceptional vector CDP followed by several load/store operations

FMULD D8, D8, D12; Inst A vector dp multiply of len 4
 FLDMD R5, {D0-D3}; Inst B load multiple 4 dp data
 FSTMD R6, {D4-D7}; Inst C store multiple of 4 dp data
 FMULS S0, S1, S1 ; Inst D scalar sp multiply S0 = S1*S1

After the exception processing has begun, the FPEXC register contains the following fields:

EX: 1 (Signaling the VFP10 Rev 0 is exceptional)
 EN: 1
 VECITR: 010 (VECITR reports 2 iteration remain)
 UF: 1 (The exception detected is Underflow)
 OF: 0
 DZ: 0
 IV: 0

The FPINST register contains the following fields (the conditional field and forced bits are not shown):

Op: 0100
 Fd/D: 1001/0 (Destination is D9 for the 2nd iteration)
 Fn/N: 1001/0 (Fn source is D9 for the 2nd iteration)
 Fm/M: 1101/0 (Fm source is D13 for the 2nd iteration)
 CpID: 1011

In Example 3-3 on page 3-12 the FMULD in Inst A is determined to have a potential underflow in the third iteration. The FLDMD in Inst B is blocked from beginning execution by a register conflict on the source operands of the FMULD. In the cycle following the detection of the exception in the FMULD third iteration, the bounce signal to the ARM is asserted. The FLDMD is restarted upon the return of the support code exception routine.

Example 3-3 Exceptional vector CDP followed by load multiple with register conflict

```
FMULD D12, D4, D12; Inst A vector dp multiply of length 4
FLDMD R3, {D4-D7} ; Inst B load multiple 4 dp data
```

The FPEXC register contains the following fields:

```
EX:    1    (Signaling the VFP10 Rev 0 is exceptional)
EN:    1
VECITR: 001 (VECITR reports 1 iteration remains)
UF:    1    (The exception detected is Underflow)
OF:    0
DZ:    0
IV:    0
```

The FPINST register contains the following fields (the conditional field and forced bits are not shown):

```
Op:    0100
Fd/D:  1110/0 (Destination is D14 for the 3rd iteration)
Fn/N:  0110/0 (Fn source is D6 for the 3rd iteration)
Fm/M:  1110/0 (Fm source is D14 for the 3rd iteration)
CpID:  1011
```

In the Example 3-4, the FMULD in Inst A is found to have an unsupported operand in the third iteration. The FMULS in Inst B will stall in the Decode stage and not be started. The FMULS is restarted upon the return of the support code exception routine.

Example 3-4 Exceptional vector CDP followed by scalar CDP with register conflict

```
FMULD D4, D4, D12 ; Inst A vector dp multiply of length 4
FMULS S0, S20, S20 ; Inst B scalar sp multiply S0 = S1*S1
```

After the exception processing has begun, the FPEXC and FPINST registers have the following:

```
FPEXC register:
EX:    1    (Signaling the VFP10 Rev 0 is exceptional)
EN:    1
VECITR: 001 (VECITR reports 1 iteration remains)
UF:    0
OF:    0
DZ:    0
IV:    1    (The exception detected is Invalid)
```

The FPINST register (the conditional field and forced bits are not shown).

Op:	0100
Fd/D:	0110/0 (Destination is D6 for the 3rd iteration)
Fn/N:	0110/0 (Fn source is D6 for the 3rd iteration)
Fm/M:	1110 (Fm source is D14 for the 3rd iteration)
CpID:	1011

3.3 Invalid operation

Table 3-1 lists the operand combinations that produce invalid operation exceptions. In addition to the conditions in Table 3-1, any CDP instruction other than FCPY, FNEG, and FABS causes an invalid operation exception if one or more of its operands is a signaling NaN (see Table 3-4 on page 3-21). When an invalid operation exception occurs, the support code is called through the ARM undefined instruction vector. What happens next depends on whether the invalid operation exception is enabled.

Table 3-1 Possible IEEE 754 invalid operation exceptions

Instruction	Invalid operation exceptions
FMAC/FNMAC	Any of the conditions which can cause an invalid exception for FADD or FMUL can cause an invalid exception for FMAC and FNMAC. The product generated by the multiply operation of the FMAC or FNMAC is considered in the determination of the invalid exception for the subsequent sum operation.
FMSC/FNMSC	Any of the conditions which can cause an invalid exception for FSUB or FMUL can cause an invalid exception for FMSC and FNMSC. The product generated by the multiply operation of the FMSC or FNMSC is considered in the determination of the invalid exception for the subsequent difference operation.
FADD	(+infinity) + (-infinity) or (-infinity) + (+infinity)
FSUB	(+infinity) - (+infinity) or (-infinity) - (-infinity)
FDIV	0/0 or infinity/infinity
FMUL/FNMUL	$0 * \pm \text{infinity}$ or $\pm \text{infinity} * 0$
FSQRT	Source is < 0
FFTOUI	Rounded result would lie outside the range $0 \leq \text{result} < 2^{32}$
FFTOSI	Rounded result would lie outside the range $-2^{31} \leq \text{result} < 2^{31}$

3.3.1 Exception enabled

If the IOE bit of the FPSCR is 1, the invalid operation trap handler you created is called. No registers are modified.

3.3.2 Exception disabled

If the IOE bit of the FPSCR is 0, the support code writes a quiet NaN to the destination register for all CDPs except integer convert instructions:

- for FFTOUI the largest integer is written to the destination register if the source overflowed, and zero if the source was negative

- for FFTOSI the largest integer with the sign of the source is written to the destination register.

The support code sets the IOC bit. For both FFTOUI and FFTOSI, 0 is written to the result register if the source was a NaN.

3.4 Division by zero

The division by zero exception is generated for a division $x/0$, where x is anything other than a zero, infinity, or a NaN. When a division by zero exception occurs, the support code is called through the ARM undefined instruction vector. What happens next depends on whether the invalid operation exception is enabled.

3.4.1 Exception enabled

If the DZE bit of the FPSCR is 1, the divide-by-zero trap handler you created is called. No registers are modified.

3.4.2 Exception disabled

If the DZE bit of the FPSCR is 0, a trap is taken to support code, which writes a correctly signed infinity to the destination register. The support code sets the DZC bit.

3.5 Overflow

Overflow is detected pessimistically based on the preliminary calculation of the final exponent value. If the pessimistic determination of overflow by the hardware is confirmed by the support code for an operation with a floating-point result, an overflow exception is generated. This confirmation consists of determining that the result of the operation after rounding exceeds the largest representable number in magnitude in the destination format. The result depends on whether the invalid operation exception is enabled.

3.5.1 Exception enabled

A trap is taken to support code and the IEEE 754 defined intermediate result is written to the destination register. The overflow trap handler you created is called.

3.5.2 Exception disabled

A trap is taken to support code. This writes a correctly signed infinity or largest finite number for the destination precision according to Table 3-2. The support code sets the OFC bit and the IXC bit.

Table 3-2 Overflow result

Rounding mode	Result
RN	Infinity, with the sign of the intermediate result.
RZ	Largest magnitude value for the destination size, with the sign of the intermediate result.
RP	For positive overflow, +infinity. For negative overflow, the largest negative value for the destination size.
RM	For positive overflow, the largest positive value for the destination size. For negative overflow, -infinity.

3.6 Underflow

If the pessimistic determination of underflow by the hardware is confirmed by the support code for an operation with a floating-point result, an underflow exception is generated. How this is confirmed depends on whether the underflow exception is enabled:

- If the FZ flag is set, all underflowing results are forced to a positive signed zero and written to the destination register. The IXC bit is set in the FPSCR. No trap is taken and the underflow exception bit is not set. If the underflow exception enable bit is set, it is ignored.
- If the FZ flag is not set what happens next depends on whether the invalid operation exception is enabled.

3.6.1 Exception enabled

Underflow is confirmed if the result of the operation after rounding is less in magnitude than the smallest normalized number in the destination format. If it is confirmed, the IEEE 754 defined intermediate result is written to the destination register and the underflow trap handler you created is called.

3.6.2 Exception disabled

Underflow is confirmed if a *denormalization loss*, as defined in the IEEE 754 standard, occurs *and* the result of the operation after rounding is less in magnitude than the smallest normalized number in the destination format. The IEEE 754 defined result of the operation is written to the destination register in either case. If underflow is confirmed, the UFC bit and the IXC bit is set in the FPSCR.

3.7 Inexact result

Floating-point arithmetic inherently has limited precision (24 significant bits for single-precision, 53 for double-precision). The result of an arithmetic operation on two floating-point values often has more significant bits than the destination register can hold. When this happens, the result is rounded to a value that the destination register can hold, and is said to be *inexact*.

The inexact exception occurs whenever:

- a result is inexact
- an untrapped overflow exception occurs
- an untrapped underflow exception occurs, because the definition of underflow implies that the result of such an operation is always inexact.

If the FZ bit is set in the FPSCR, the inexact exception additionally occurs on any operation for which a denormalized operand is treated as zero, or for which an underflowing result is forced to zero.

Note

The inexact exception occurs frequently in the course of normal floating-point calculations, and does not indicate a significant numerical error except in some specialized applications for floating-point arithmetic.

The VFP10 Rev 0 handles the inexact exception differently from the other floating-point exceptions. It has no mechanism for reporting inexact results to the software, but can handle the exception without software intervention as long as the inexact exception is not enabled (in other words, as long as the IXE bit in the FPSCR is zero).

3.7.1 Exception enabled

If the IXE bit in the FPSCR is 1, all CDP operations will be bounced to the support code without any attempt to perform the calculation. The support code is then responsible for performing the calculation, determining which, if any, exceptions have taken place, and handling them appropriately. As part of this, if it determines that an inexact exception occurs, it calls the trap handler you provide.

Note

If the IXE bit determines that the overflow or underflow exception also occurs, it gives that exception priority over the inexact exception.

3.7.2 Exception disabled

If the IXE bit in the FPSCR is zero, the VFP10 Rev 0 writes the result to the destination register regardless of whether an inexact exception occurred:

- if an inexact exception did occur, it also sets the IXC bit in the FPSCR
- if no inexact exception occurred, the IXC bit is left unchanged.

No software intervention is needed, and no bounce to support code takes place, unless a potential overflow or underflow exception is also detected by the hardware. If a potential overflow exception is detected by the hardware, the support code first determines whether the overflow exception really occurs. If it does not, or if it does but the FPSCR indicates that the overflow exception is not enabled, the support code then determines whether an inexact exception takes place, and sets the IXC bit in the FPSCR if an inexact exception takes place.

A potential underflow exception is treated analogously by the support code.

3.8 Input exceptions

Input exceptions are defined as instructions in which one or more of the input operands are not supported in hardware. The list of which operands are not supported is defined by the instruction. Table 3-4 lists the operands and the response of the ARM VFP10 Rev 0 for non-FZ mode operation.

All operand combinations that generate an IEEE 754 invalid operand or division by zero exception result in an input exception. Input exceptions also occur for some operations which are not going to cause these IEEE 754 exceptions, such as (infinity + 0) or (normalized * denormal). Most of these operations do not generate an IEEE 754 exception at all. Some of the operations involving denormals generate IEEE 754 overflow, underflow, or inexact result exceptions (see *Invalid operation* on page 3-14 to *Inexact result* on page 3-19).

For all types of input exception, the VFP10 Rev 0 hardware bounces the instruction to the support code to determine the appropriate result value and IEEE 754 exceptions, if any. A consequence is that any instruction with input operand combinations listed as bouncing in Table 3-4 are executed in a considerably greater number of cycles than normal, even if it does not result in an IEEE 754 exception. Table 3-3 provides a key to the abbreviations in Table 3-4.

Table 3-3 Key to bounce conditions table

Abbreviation	Meaning
OK	Signifies a no-bounce condition
Bnc	Signifies a bounce condition
N/a	Not applicable

Table 3-4 Input operand bounce conditions (non-FZ mode)

Instruction family	Input operand(s)				
	Norm	INF	NaN	DEN	Zero
FADD/FSUB/FCMP/FCMPE/FCMPZ/FCM/PEZ	OK	Bnc	Bnc	Bnc	OK
FMUL/FNMUL	OK	Bnc	Bnc	Bnc	OK
FMAC/FNMAC/FMSC/FNMSC	OK	Bnc	Bnc	Bnc	OK
FDIV	OK	Bnc	Bnc	Bnc	OK ^a
FSQRT	OK ^b	Bnc	Bnc	Bnc	OK

Table 3-4 Input operand bounce conditions (non-FZ mode) (continued)

Instruction family	Input operand(s)				
	Norm	INF	NaN	DEN	Zero
FCPY/FABS/FNEG	OK	OK	OK	OK	OK
FCVTDS/FCTVSD	OK	Bnc	Bnc	Bnc	OK
FUITO/FSITO	OK	N/a	N/a	N/a	OK
FTUOI/FTOIUZFTOSI/FTOSIZ	OK ^c	Bnc	Bnc	Bnc	OK

- a. Division with zero as the divisor will generate a bounce.
- b. FSQRT with a negative input will generate a bounce.
- c. Out-of-range inputs will generate a bounce.

Table 3-4 on page 3-21 shows input operand bounce conditions in non-FZ mode.

When FZ mode is used, the values in Table 3-4 on page 3-21 change in response to denormal inputs. An input denormal operand is converted to a positive zero and all further processing is done on the zero. No invalid exceptions are signaled for a denormal input in FZ mode.

3.9 Arithmetic exceptions

Arithmetic exceptions occur for instructions whose input operands are supported by hardware, but which might result in an IEEE 754 overflow or underflow exception. They are detected pessimistically, so result values that are close to overflowing or underflowing also result in an arithmetic exception.

Like input exceptions, arithmetic exceptions always bounce. The support code then determines the result value and whether any IEEE 754 exceptions occurred. Any instruction that generates an arithmetic exception therefore takes many more cycles than normal to execute.

The following sections specify the precise circumstances in which arithmetic exceptions occur for each instruction:

- *FADD/FSUB/FCMP/FCMPZ/FCMPE/FCMPEZ*
- *FMUL/FNMUL* on page 3-25
- *FMAC/FMSC/FNMAC/FNMSC* on page 3-26
- *FDIV* on page 3-26
- *FSQRT* on page 3-28
- *FCPY/FABS/FNEG* on page 3-28
- *FCVTDS/FCVTSD* on page 3-28
- *FUITO/FSITO* on page 3-28
- *FTOUI/FTOUIZ/FTOSI/FTOSIZ* on page 3-29.

3.9.1 FADD/FSUB/FCMP/FCMPZ/FCMPE/FCMPEZ

The exponent in addition or subtraction operations, and compare (which is effectively a subtraction operation) is initially set to the larger of the two input exponents. For clarity we define the operation in terms of *Like-Signed Addition* (LSA) or an *Unlike-Signed Addition* (USA). Table 3-5 specifies how this division is made. + refers to a positive operand and - refers to a negative operand.

Table 3-5 LSA and USA determination

Instruction	A sign	B sign	Operation
FADD	+	+	LSA
FADD	+	-	USA
FADD	-	+	USA
FADD	-	-	LSA
FSUB/FCMP	+	+	USA

Table 3-5 LSA and USA determination (continued)

Instruction	A sign	B sign	Operation
FSUB/FCMP	+	-	LSA
FSUB/FCMP	-	+	LSA
FSUB/FCMP	-	-	USA

For LSA, the bounce conditions are more pessimistic for overflow than they are for USA, since it is possible for an LSA operation to cause the exponent to be incremented if the mantissa overflows. The LSA ranges are made slightly more pessimistic to incorporate FMAC operations (see *FMAC/FMSC/FNMAC/FNMSC* on page 3-26).

For USA, the underflow bounce ranges are pessimistic to a greater degree to accommodate the possibility of a massive cancellation in which the result exponent might be smaller than the larger operand exponent by as much as the length of the mantissa (24 for single-precision and 53 for double-precision). The overflow range for USA is slightly pessimistic (it is set to the LSA overflow range) in order to reduce the number of logic terms. A USA operation cannot overflow for normal inputs. Table 3-6 lists the USA and LSA values and conditions.

Table 3-6 USA and LSA values and conditions

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
7FF	-	DBL NaN, Inf	-	Bounce
7FE	-	DBL Ovfl Det	-	Bounce
7FD	-	DBL Ovfl Det	-	Bounce
7FC	-	DBL Norm	-	Norm
47F	FF	SGL NaN, Inf	Bounce	Norm
47E	FE	SGL Ovfl Det	Bounce	Norm
47D	FD	SGL Ovfl Det	Bounce	Norm
47C	FC	SGL Norm	Norm	Norm
3FF	7F	e=0 bias value	Norm	Norm
3A0	20	SGL Norm (LSA)	MIN (USA)	Norm

Table 3-6 USA and LSA values and conditions (continued)

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
39F	1F	SGL Unfl (USA)	Bounce (USA) Norm (LSA)	Norm
381	01	SGL Norm (LSA)	MIN (LSA)	Norm
380	00	SGL Denormal	Bounce	Norm
040	-	DBL Norm (USA)	-	Norm (LSA) MIN (USA)
03F	-	DBL Unfl (USA)	-	Norm (LSA) Bounce (USA)
001	-	DBL Norm (LSA)	-	MIN (LSA) Bounce (USA)
000	-	DBL Denormal	-	Bounce

3.9.2 FMUL/FNMUL

The determination for potential exceptional conditions is made based on the initial product exponent, the sum of the multiplicand and multiplier exponents. Table 3-7 lists the VFP10 Rev 0 response for specific values of the initial product exponent. It is possible for the exponent to be incremented by a mantissa overflow condition. This is the cause for the additional bounce values near the real overflow threshold. The one additional value incorporated into the bounce range makes the FMUL/FNMUL overflow detection ranges identical to those of the FADD family in *FADD/FSUB/FCMP/FCMPZ/FCMPE/FCMPEZ* on page 3-23.

Table 3-7 FMUL family bounce and exceptional thresholds

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
7FF	-	DP OVFL	-	Bounce
7FF	-	DP NaN, Inf	-	Bounce
7FE	-	DP Max Norm	-	Bounce
7FD	-	DP Norm	-	Bounce

Table 3-7 FMUL family bounce and exceptional thresholds (continued)

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
7FC	-	DP Norm	-	Norm
>47F	>FF	SP OVFL	Bounce	Norm
47F	FF	SP NaN, Inf	Bounce	Norm
47E	FE	SP Max Norm	Bounce	Norm
47D	FD	SP Norm	Bounce	Norm
47C	FC	SP Norm	Norm	Norm
3FF	7F	e=0 bias value	Norm	Norm
381	01	SP Norm	Norm	Norm
380	00	SP Denormal	Bounce	Norm
<380	<00	SP UNFL	Bounce	Norm
C01	-	DP Norm	-	Norm
C00	-	DP Denormal	-	Bounce
<C00	-	DP UNFL	-	Bounce

3.9.3 FMAC/FMSC/FNMAC/FNMSC

The FMAC family of operations adds to the potential overflow range by allowing a final value in the range [0, 4). In this case it is possible for the final exponent to require incrementing by two to normalize the mantissa.

The bounce thresholds presented earlier for the FADD family and the FMUL family incorporate this additional factor. Those ranges are used to detect potential exceptions for the FMAC family.

3.9.4 FDIV

The thresholds for divide are simple and based only on the difference of the exponents of the dividend and the divisor. It is not possible in a divide operation for the mantissa to overflow and cause an increment of the exponent. However, it is possible for the mantissa to require a single bit left shift and the exponent to be decremented for normalization. The overflow ranges are the same as those of the LSA operations in

FADD/FSUB/FCMP/FCMPZ/FCMPE/FCMPEZ on page 3-23 (again, to reduce logic complexity). The underflow ranges now include the minimum normal exponent (01 for single-precision and 001 for double-precision). The complete table is shown in Table 3-8.

Table 3-8 FDIV bounce and exceptional thresholds

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
7FF	-	DP OVFL	-	Bounce
7FF	-	DP NaN, Inf	-	Bounce
7FE	-	DP Max Norm	-	Bounce
7FD	-	DP Norm	-	Bounce
7FC	-	DP Norm	-	Norm
-	>FF	SP OVFL	Bounce	Norm
47F	FF	SP NaN, Inf	Bounce	Norm
47E	FE	SP Max Norm	Bounce	Norm
47D	FD	SP Norm	Bounce	Norm
47C	FC	SP Norm	Norm	Norm
3FF	7F	e=0 bias value	Norm	Norm
382	02	SP Norm	Norm	Norm
381	01	SP Norm	Bounce	Norm
380	00	SP Denormal	Bounce	Norm
-	<00	SP UNFL	Bounce	-
002	-	DP Norm	-	Norm
001	-	DP Norm	-	Bounce
000	-	DP Denormal	-	Bounce
<000	-	DP UNFL	-	Bounce

3.9.5 FSQRT

It is not possible for FSQRT to overflow or underflow.

3.9.6 FCPY/FABS/FNEG

It is not possible for FCPY, FABS, or FNEG to bounce for any operand.

3.9.7 FCVTDS/FCVTSD

Only the FCVTSD operation is capable of overflow or underflow. Table 3-9 lists the FCVTSD bounce conditions. The FCVTDS operation is a conversion from a smaller precision to a larger precision which has greater range in both the exponent and mantissa. The overflow ranges are the same as the LSA ranges. This is to reduce logic complexity.

Table 3-9 FCVTSD bounce conditions

DP	Value	Condition (non-FZ mode) FVCTSD
>47F	SP OVFL	Bounce
47F	SP NaN, Inf	Bounce
47E	SP Max Norm	Bounce
47D	SP Norm	Bounce
47C	SP Norm	Norm
3FF	e=0 bias value	Norm
381	SP Norm	Norm
380	SP Denormal	Bounce
<380	SP UNFL	Bounce

3.9.8 FUITO/FSITO

It is not possible to generate overflow or underflow in an integer-to-float conversion.

3.9.9 FTOUI/FTOUIZ/FTOSI/FTOSIZ

Float-to-integer conversions generate only Invalid exceptions rather than overflow or underflow. The thresholds are different for the various rounding modes in order to support signed conversions with round-to-zero rounding in the maximum range possible for C, C++ and Java compiled code.

Table 3-10 and Table 3-11 on page 3-30 use the following notation:

Ex (Exception generated)

I Invalid

V None (Operation is Valid)

VFP Response

ALL These input values are bounced for all rounding modes.

S These input values are bounced for signed conversions in all rounding modes.

SnZ These input values are bounced for signed conversions in all rounding modes except round-to-zero.

U These input values are bounced for unsigned conversions in all rounding modes.

UnZ These input values are bounced for unsigned conversions in all rounding modes except round-to-zero.

Table 3-10 shows the single-precision float-to-integer bounce range and the results returned for exceptional conditions.

Table 3-10 SP Float-to-integer bounce thresholds and stored results

Float value	Integer	Unsigned result	E_x	Signed result	E_x	VFP response
NaN	-	00000000	I	00000000	I	Bounce all
7F800000	+Inf	FFFFFFFF	I	7FFFFFFF	I	Bounce all
7F7FFFFF to 4F800000	+Max Sp 2 ³²	FFFFFFFF FFFFFFFF	 I	7FFFFFFF 7FFFFFFF	 I	 Bounce all

Table 3-10 SP Float-to-integer bounce thresholds and stored results (continued)

Float value	Integer	Unsigned result	E_x	Signed result	E_x	VFP response
4F7FFFFF to 4F000000	$(2^{32} - 2^8)$ 2^{31}	FFFFFF00 80000000	 I	7FFFFFFF 7FFFFFFF	 I	 Bounce S UnZ
4EFFFFFF to 4E800000	$(2^{31} - 2^7)$ 2^{30}	7FFFFFF80 40000000	 V	7FFFFFF80 40000000	 V	 Bounce SnZ
4E7FFFFF to 00000000	$(2^{30} - 2^6)$ +0	3FFFFFFC0 00000000	 V	3FFFFFFC0 00000000	 V	 -
80000000 to CE7FFFFF	-0 $(-2^{30} + 2^6)$	00000000 00000000	 I	00000000 C0000040	 V	 Bounce U
CE800000 to CEFFFFFF	-2^{30} $(-2^{31} + 2^7)$	00000000 00000000	 I	C0000000 80000080	 V	 Bounce U Bounce U SnZ
CF000000 to FF7FFFFF	-2^{31} -Max Sp	00000000 00000000	 I	80000000 80000000	 I	 Bounce all
FF800000	-Inf	00000000	I	80000000	I	Bounce all

Table 3-11 shows the double-precision float-to-integer bounce range and the results returned for exceptional conditions.

Table 3-11 DP Float-to-integer bounce thresholds and stored result

Float value	Integer	Unsigned result	E_x	Signed result	E_x	VFP response
NaN	-	00000000	I	00000000	I	Bounce all
7FF00000_00000000	+Inf	FFFFFFFF	I	7FFFFFFF	I	Bounce all
7FEFFFFFF_FFFFFFFF to 41F00000_00000000	+Max DP 2^{32}	FFFFFFFF FFFFFFFF	 I	7FFFFFFF 7FFFFFFF	 I	 Bounce all

Table 3-11 DP Float-to-integer bounce thresholds and stored result (continued)

Float value	Integer	Unsigned result	E x	Signed result	E x	VFP response
41EFFFFFF_FFFFFFFF to 41E00000_00000000	$(2^{32} - 2^{21})$ 2^{31}	FFFFFFFF 80000000	 V	7FFFFFFF 7FFFFFFF	 I	 Bounce S UnZ
41DFFFFFF_FFFFFFFF to 41D00000_00000000	$(2^{31} - 2^{22})$ 2^{30}	80000000 40000000	 V	7FFFFFFF 40000000	 V	 Bounce SnZ
41CFFFFFF_FFFFFFFF to 00000000_00000000	$(2^{30} - 2^{23})$ +0	3FFFFFFF 00000000	 V	3FFFFFFF 00000000	 V	 -
80000000_00000000 to C1CFFFFFF_FFFFFFFF	-0 $(-2^{30} + 2^{23})$	00000000 00000000	 I	00000000 C0000001	 V	 Bounce U
C1D00000_00000000 to C1DFFFFFF_FFFFFFFF	-2^{30} $(-2^{31} + 2^{22})$	00000000 00000000	 I	C0000000 80000080	 V	 Bounce U Bounce U SnZ
C1E00000_00000000 to FFEFFFFFF_FFFFFFFF	-2^{31} -Max DP	00000000 00000000	 I	80000000 80000000	 I	 Bounce all
FFF00000_00000000	-Inf	00000000	I	80000000	I	Bounce all

VFP10 Technical Reference Manual Glossary

This glossary contains selected items from the *ARM Architecture Reference Manual*, the IEEE 754-1985 specification, and items defined within the text of the manual.

Bouncing	Asserts the CPBOUNCEE signal to the ARM. An exception reported to the operating system which is handled by the support code entirely without calling user trap handlers or otherwise interrupting the normal flow of user code.
Coprocessor Data Processing	For the VFP10 Rev 0, CDP operations are arithmetic operations rather than load or store operations.
Denormalized value	A representation of a value in the range $(-2^{E_{\min}} < x < 2^{E_{\min}})$. In the IEEE 754 format for single and double precision operands, a denormalized value, or denormal, has a zero exponent and the leading significant bit is 0 rather than 1. The IEEE 754-1985 specification requires that the generation and manipulation of denormalized operands be performed with the same precision as with normal operands.
Disabled exception	An exception that has its associated exception enable bit in the FPCSR set to 0 is referred to as <i>disabled</i> . For these exceptions the IEEE 754 specification defines the correct result to be returned. An operation that generates an exception condition may bounce to the support code to produce the IEEE 754 defined result. The exception is not reported to the user exception handler.

Enabled exception	An exception with the respective exception enable bit set to 1. In the event of an occurrence of this exception a trap to the user handler is taken. An operation that generates an exception condition might bounce to the support code to produce the IEEE 754 defined result. The exception is then reported to the user exception handler.
Exceptional state	When a potentially exceptional instruction is issued, the VFP sets the EX bit in the FPSCR and loads a copy of the instruction word for the potentially exceptional instruction. If the instruction is a vector operation, the register fields in the FPINST are altered to represent the iteration that was exceptional. When in the exceptional state, the issue of a trigger instruction to the VFP causes a bounce. <i>See also</i> Bouncing, Potentially exceptional instruction, and Trigger instruction.
Exponent	The component of a floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.
Fd	The destination register and the accumulate value in triadic operations. Sd for single-precision operations and Dd for double-precision.
Fn	The first source operand in dyadic or triadic operations. Sn for single-precision operations and Dn for double-precision.
Fm	The second source operand in dyadic or triadic operations. Sm for single-precision operations and Dm for double-precision.
Fraction	The field of the significand that lies to the right of its implied binary point.
Flush-To-Zero mode	In this mode all values in the range $(-2^{E_{min}} < x < 2^{E_{min}})$ before rounding are treated as zero, rather than converted to a denormalized value.
IEEE 754	IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, Inc. New York, New York, 10017. The standard, often referred to as the IEEE 754 standard, which defines data types, correct operation, exception types and handling, and error bounds for floating-point systems. Most processors are built in compliance with the standard in either hardware or a combination of hardware and software.
Illegal instructions	If there is no potential floating-point exception from an earlier instruction, the current instruction may still be bounced because it is architecturally undefined in some way. Such instructions are known as illegal instructions.
Infinity	An IEEE 754 special format used to represent ∞ . The exponent will be maximum for the precision and the significand will be all zeros.
Input exception	An exception condition in which one or more of the operands for a given operation are not supported by the hardware. The operation will bounce to support code for completion of the operation.

Intermediate result	An internal format used to store the result of a calculation before rounding. This format may have a larger exponent field and significand field than the destination format.
MCR	For the FPS this includes instructions which transfer data or control registers between an ARM register and a FPS register. Only 32 bits of information can be transferred using a single MCR class instruction.
MRC	For the FPS this includes instructions which transfer data or control registers between the FPS and an ARM register. Only 32 bits of information can be transferred using a single MRC class instruction.
NaN	A symbolic entity encoded in a floating-point format. There are two types of NaNs, signaling and non-signaling, or quiet. Signaling NaNs will cause an Invalid Operand exception if used as an operand. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions. The format for a NaN has the exponent field of all 1's with the significand non-zero. To represent a signaling NaN the most significant bit of the fraction is zero, while a quiet NaN will have the bit set to a one.
Potentially exceptional instruction	An instruction which is determined, based on the exponents of the operands and the sign bits, to have the potential to be exceptional (either to produce an overflow or underflow condition). Once this determination is made, the VFP enters the exceptional state and bounces the next trigger instruction issued. <i>See also</i> Bouncing, Trigger instruction, and Exceptional state.
Reserved	A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces UNPREDICTABLE results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as zero and will be read as zero.
Rounding mode	The IEEE 754 Standard requires all calculations are performed as if to an infinite precision, that is, a multiply of two single precision values must calculate accurately the significand to twice the number of bits of the significand. To represent this value in the destination precision rounding of the significand is often required. The IEEE 754 standard specifies four rounding modes: <i>Round to Nearest (RN)</i> This is accomplished by rounding at the half way point, with the tie case rounding up if it would zero the LSB of the significand, making it even. <i>Round to Zero (RZ)</i> This effectively chops any bits to the right of the significand, always rounding down, and is used by the C, C++, and Java languages in integer conversions.

Round to Plus Infinity (RP)

This is used in interval arithmetic.

Round to Minus Infinity (RM)

This is used in interval arithmetic.

Scalar operation	An operation involving a single destination register.
Significand	The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.
Trap	An exceptional condition which has the respective exception enable bit set in the FPSCR. The user provided trap handler is executed.
Trigger instruction	The instruction which causes a bounce at the time it is issued. A potentially exceptional instruction causes the VFP to enter the exceptional state. The next instruction, unless it is an FMXR or FMRX instruction accessing one of the FPExc, FPinSt, or FPSID registers, causes a bounce, beginning exception processing. The trigger instruction is not necessarily exceptional, and no processing of it is performed. It will be retried at the return from exception processing of the potentially exceptional instruction. <i>See also</i> See also Bouncing, Potentially exceptional instruction, and, Exceptional state
UNDEFINED	Indicates an instruction that generates an undefined instruction trap. See the <i>ARM Architectural Reference Manual</i> for more information on ARM exceptions.
UNPREDICTABLE	The result of an instruction or control register field value that cannot be relied upon. UNPREDICTABLE instructions or results must not represent security holes, or halt or hang the processor, or any parts of the system.
Unsupported values	Specific data values that are not processed by the hardware but bounced to the support code for completion. These data can include infinities, NaNs, denormal values, and zeros. An implementation is free to select which of these values is supported in hardware fully or partially, or requires assistance from support code to complete the operation. Any exception resulting from processing unsupported data is trapped to user code if the corresponding exception enable bit for the exception is set.
Vector operation	An operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination.
VFP Support Code	Software which must be used to complement the hardware to provide compatibility with the IEEE 754 standard. The support code is intended to have two components: <ul style="list-style-type: none"> • a library of routines that performs operations beyond the scope of the hardware, such as transcendental computations, as well as supported functions, such as divide with unsupported inputs or inputs that might generate an exception

- a set of exception handlers that process exceptional conditions in order to provide IEEE 754 compliance.

The support code is required to perform implemented functions to emulate proper handling of any unsupported data type or data representation (denormal values or decimal data types). The routines can be written to utilize the FPS in their intermediate calculations if care is taken to restore the user state at the exit of the routine.

