VFP10™ Vector Floating-point Coprocessor

(Rev 1)

Technical Reference Manual

ARM

ARM DDI 0178B

VFP10™ Vector Floating-point Coprocessor
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
22 May, 2001	A	First release
24 May, 2001	B	Second release, addition of FPINST and FPINST2 <reg> field addresses

Proprietary Notice

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is final (information on 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 About the VFP10 coprocessor 1-2
1.2 Coprocessor interface 1-4
1.3 The VFP10 coprocessor pipeline 1-5
1.4 Modes of operation 1-12
1.5 Short vector instructions 1-15
1.6 Parallel execution of instructions 1-16
1.7 VFP10 coprocessor treatment of branch instructions 1-17
1.8 Writing optimal VFP10 coprocessor code 1-18
1.9 Clocking 1-19
1.10 Testing 1-20
1.11 Modifications from VFP10 coprocessor (Rev 0) 1-21

Chapter 2 VFP10 Register File

2.1 About the register file 2-2

2.2 Register file internal formats 2-3
2.3 Decoding the register file 2-5
2.4 Loading operands from ARM registers 2-7
2.5 Maintaining consistency in register precisions 2-9
2.6 Data transfer between memory and VFP10 coprocessor registers 2-10
2.7 Access to register banks in CDP operations 2-12

Chapter 3 VFP10 Programmer’s Model
3.1 About the programmer’s model 3-2
3.2 Compliance with IEEE-754 3-4
3.3 ARM v5TE coprocessor extensions 3-11
3.4 Summary of VFP coprocessor system control registers 3-17
3.5 FPSCR register 3-23

Chapter 4 Instruction Execution in the VFP10 Coprocessor
4.1 About instruction execution in the VFP10 coprocessor 4-2
4.2 Serializing instructions 4-4
4.3 Interrupting VFP10 coprocessor instructions 4-5
4.4 Forwarding 4-6
4.5 Hazard and resource stall conditions 4-11
4.6 Parallel execution of operations 4-21
4.7 Execution timing 4-23

Chapter 5 Exception Handling
5.1 About exception processing 5-2
5.2 Support code 5-3
5.3 Illegal instructions 5-6
5.4 Determination of the trigger instruction 5-7
5.5 Input subnormal 5-12
5.6 Invalid operation 5-13
5.7 Division by zero 5-16
5.8 Overflow 5-17
5.9 Underflow 5-19
5.10 Inexact result 5-21
5.11 Input exceptions 5-22
5.12 Arithmetic exceptions 5-23

Chapter 6 Design for Test
6.1 About DFT 6-2
6.2 VFP10 DFT 6-3
6.3 VFP10 Core 6-4
6.4 VFP10 test wrapper 6-6
6.5 VFP10 clocking 6-10
6.6 Test Pins 6-11

Glossary

List of Tables
VFP10 Vector Floating-point Coprocessor Technical Reference Manual

	Change history	ii
Table 2-1	MCR transfers	2-7
Table 2-2	MRC transfers	2-7
Table 2-3	MCRR transfers	2-8
Table 2-4	MRRC transfers	2-8
Table 2-5	Single-precision data memory images and byte addresses	2-10
Table 2-6	Double-precision data memory images and byte addresses	2-11
Table 2-7	Register bank description	2-12
Table 2-8	Single-precision three-operand register usage	2-15
Table 2-9	Single-precision two-operand register usage	2-16
Table 2-10	Double-precision three-operand register usage	2-16
Table 2-11	Double-precision two-operand register usage	2-16
Table 3-1	Default NaN values	3-6
Table 3-2	Access to control registers	3-17
Table 3-3	FPEXC bit field descriptions	3-19
Table 3-4	Vector iteration count bit values	3-20
Table 3-5	FPSID bit fields	3-21
Table 3-6	Vector length/stride combinations	3-25
Table 3-7	Exception status and control bits	3-26
Table 4-1	Single-precision source register locking and clearing in non-RunFast mode	4-12
Table 4-2	Double-precision source register locking and clearing in non-RunFast mode	4-12

Table 4-3 Single-precision source register locking and clearing in RunFast mode 4-13
Table 4-4 Double-precision source register locking and clearing in RunFast mode 4-13
Table 4-5 Instruction cycles for example 1 4-14
Table 4-6 Instruction cycles for example 2 4-15
Table 4-7 Instruction cycles for short vector MULS example 4-16
Table 4-8 Instruction cycles for short vector FMULS example 4-16
Table 4-9 Instruction cycles for example in Run Fast mode 4-17
Table 4-10 Instruction cycles for vector MULS example 4-18
Table 4-11 Instruction cycles for example 2 4-19
Table 4-12 Pipeline stages for load multiple, vector MULS, scalar FADDS 4-20
Table 4-13 Parallel execution in three pipelines 4-22
Table 4-14 Throughput and latency cycle timings for VFP10 CDP operations 4-23
Table 5-1 Possible IEEE 754 invalid operation exceptions 5-13
Table 5-2 Default results for positive invalid inputs 5-15
Table 5-3 Overflow result 5-18
Table 5-4 LSA and USA determination 5-23
Table 5-5 USA and LSA values and conditions 5-24
Table 5-6 FMUL family bounce and exceptional thresholds 5-26
Table 5-7 FDIV bounce and exceptional thresholds 5-27
Table 5-8 FCVTSD bounce conditions 5-28
Table 5-9 SP Float-to-integer bounce thresholds and stored results 5-30
Table 5-10 DP Float-to-integer bounce thresholds and stored results 5-32
Table 6-1 Scan chain configuration 6-4
Table 6-2 Internal scan chain configuration 6-5
Table 6-3 Wrapper cell control and observation configurations 6-9
Table 6-4 VFP10 macrocell test ports 6-11
Table 6-5 VFP10 test signals during core scan test 6-12
Table 6-6 VFP10 coprocessor test signals in functional mode 6-13
Table 6-7 VFP10 test pins in VFP10 coprocessor external test wrapper mode 6-13
Table G-8 Register banks in single-precision and double-precision registers Glossary-4

List of Figures
VFP10 Vector Floating-point Coprocessor Technical Reference Manual

Figure 1-1	FMAC pipeline	1-6
Figure 1-2	Divide and square root pipeline	1-8
Figure 1-3	Load/Store pipeline	1-10
Figure 2-1	Integer format	2-3
Figure 2-2	Single-precision data format	2-4
Figure 2-3	Register data formats	2-4
Figure 2-4	Register file format	2-6
Figure 3-1	FMDRR bit fields	3-11
Figure 3-2	FMRRD bit fields	3-12
Figure 3-3	FMSRR bit fields	3-13
Figure 3-4	FMRRS bit fields	3-15
Figure 3-5	FPEXC register format	3-19
Figure 3-6	FPSID register format	3-21
Figure 3-7	User status and control bit fields summary	3-23
Figure 6-1	Dedicated input wrapper cell	6-6
Figure 6-2	Dedicated output wrapper cell	6-7
Figure 6-3	Reset dedicated wrapper cell	6-8
Figure 6-4	VFP10 production scan mode clocking	6-10
Figure 6-5	VFP10 serial core test clocking requirement	6-10

Preface

This preface introduces the VFP10™ (Rev1) 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 (Rev1).

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.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an overview of the VFP10 coprocessor pipelines, modes of operation and a summary of the differences between this revision and the previous revision.

Chapter 2 *VFP10 Register File*

Read this chapter for a description of the VFP10 coprocessor register file.

Chapter 3 *VFP10 Programmer’s Model*

Read this chapter for details of the programmer’s model and VFP10 coprocessor registers.

Chapter 4 *Instruction Execution in the VFP10 Coprocessor*

Read this chapter for details of instruction execution in VFP10 coprocessor.

Chapter 5 *Exception Handling*

Read this chapter for a description of VFP10 coprocessor exception handling.

Chapter 6 *Design for Test*

Read this chapter for a description of VFP10 coprocessor design for test features.

Typographical conventions

The following typographical conventions are used in this book:

bold	Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
`typewriter`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`typewriter` (type underlined)	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code and ARM processor signal names.

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/DevSupp/Sales+Support/faq`

ARM publications

This document contains information that is specific to the VFP10 Vector Floating-point Coprocessor (Rev 1). Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DUI 0100) Revision D or later
- *AFS Firmware Suite Version 1.3 Reference Guide* (ARM DUI 0102).
- *ARM1020E Technical Reference Manua*l (ARM DDI 0177)
- *ARM10200E Test Chip Implementation Guide* (ARM DXI 0106).

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 (Rev1), and on the documentation.

Feedback on the VFP10 Vector Floating-point Coprocessor (Rev1)

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 email 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:

- *About the VFP10 coprocessor* on page 1-2
- *Coprocessor interface* on page 1-4
- *The VFP10 coprocessor pipeline* on page 1-5
- *Modes of operation* on page 1-12
- *Short vector instructions* on page 1-15
- *Parallel execution of instructions* on page 1-16
- *VFP10 coprocessor treatment of branch instructions* on page 1-17
- *Writing optimal VFP10 coprocessor code* on page 1-18
- *Clocking* on page 1-19
- *Testing* on page 1-20
- *Modifications from VFP10 coprocessor (Rev 0)* on page 1-21.

1.1 About the VFP10 coprocessor

The ARM VFP10 Floating-point Coprocessor is the first implementation of the *Vector Floating-point Architecture* (VFPv2). It provides IEEE 754-compliant, low-cost floating-point computation for applications where high-performance graphics processing or signal processing capabilities are required.

The VFP10 coprocessor is optimized for:

- high data transfer bandwidth through 64-bit split load and store buses
- fast hardware execution of a high percentage of operations on normalized data resulting in higher overall performance while providing full IEEE 754 support when required
- parallel divide and square-root operations in parallel with other arithmetic operations to reduce the impact of long latency operations
- full IEEE 754 compatibility in RunFast mode without support code assistance, providing determinable run-time calculations for all input data
- low power consumption, small die size and reduced kernel code.

The VFP10 coprocessor is a high-performance, low-power ARM enhanced numeric coprocessor macrocell that provides high throughput IEEE 754-compatible operations. Designed to be incorporated with the ARM10 family of cores, the VFP10 coprocessor provides full support of single-precision and double-precision addition, subtraction, multiplication, division, and multiply with accumulate operations. Conversions between floating-point data formats and ARM integer word format are provided, with special operations to perform the conversion in *Round-To-Zero* (RZ) rounding for high-level language support.

The VFP10 coprocessor delivers high performance in general purpose applications, such as Java, and an excellent performance-power-area solution for embedded applications.

Note

This document is intended to be read in conjunction with the Vector Floating-point Architecture section of the *ARM Architecture Reference Manual*. Only VFP10-specific implementation issues are described in this book.

1.1.1 Applications

The VFP10 coprocessor is built with full-scan for high coverage testability. Advanced power-saving support is incorporated to take advantage of the power-saving modes of the ARM1020E macrocell. The VFP10 coprocessor 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 *Multi-Function Peripheral* (MFP) 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.

1.2 Coprocessor interface

The VFP10 coprocessor is designed to be integrated with an ARM10 family device through a general-purpose ARM1020E coprocessor interface. This interface is further defined in the *ARM1020E Technical Reference Manua*l.

The VFP10 coprocessor uses coprocessor ID numbers 10 and 11, mainly for single-precision and double-precision operations, respectively. In some cases, such as mixed precision operations, the coprocessor ID represents the destination precision. In a system containing a VFP10 coprocessor, these coprocessor ID numbers must not be used by another coprocessor.

For the VFP10 coprocessor to operate at the maximum frequency specified, the coprocessor interface between the ARM1020E and the VFP10 coprocessor must be implemented with care to minimize the physical distance between the ARM1020E device and the VFP10 coprocessor, and to make the interconnect wires as short as possible. See the *ARM10200E Implementation Guide* for more information.

1.3 The VFP10 coprocessor pipeline

The VFP10 coprocessor comprises three separate pipelines:

- the multiply-accumulate pipeline (FMAC)
- the *Divide and square root* pipeline (DS).
- the *Load/Store* pipeline (LS)

These are each capable of operating independently of the other pipelines and in parallel with them. Each of the three pipelines share the first two pipeline stages, Issue and Decode. These two stages and the first cycle of the Execute stage of each pipeline remains in lockstep with the ARM pipeline stage but effectively one cycle behind the ARM pipeline. When the ARM is in the Decode stage for a particular VFP instruction, the VFP10 coprocessor is in the Issue stage for the same instruction. This lockstep mechanism maintains in-order issue between the ARM processor and the VFP10 coprocessor.

The three pipelines are capable of operating in parallel, enabling more than 1 instruction to be completed per cycle. Instructions issued to the FMAC pipeline can complete out of order with respect to load and store operations and divide or square root operations. This out-of-order completion might be visible to the user in the case of an exception generated by a short vector FMAC or DS operation, with a load or store operation initiated before the exception was detected. The destination registers or memory of the load or store operation will reflect the completion of a transfer while the destination registers of the exceptional FMAC or DS operation will retain their values before the operation was initiated. This is described in more detail in *Parallel execution of operations* on page 4-21.

The pipeline supports single-cycle throughput for all single-precision operations (excluding divide and square root) and most double-precision operations. Double-precision multiply and multiply-accumulate operations have a two-cycle throughput. The LS pipeline is capable of supplying two single-precision operands or one double-precision operand per cycle, balancing the data transfer capability with the operand requirements.

1.3.1 The FMAC pipeline

The FMAC pipeline is shown in Figure 1-1.

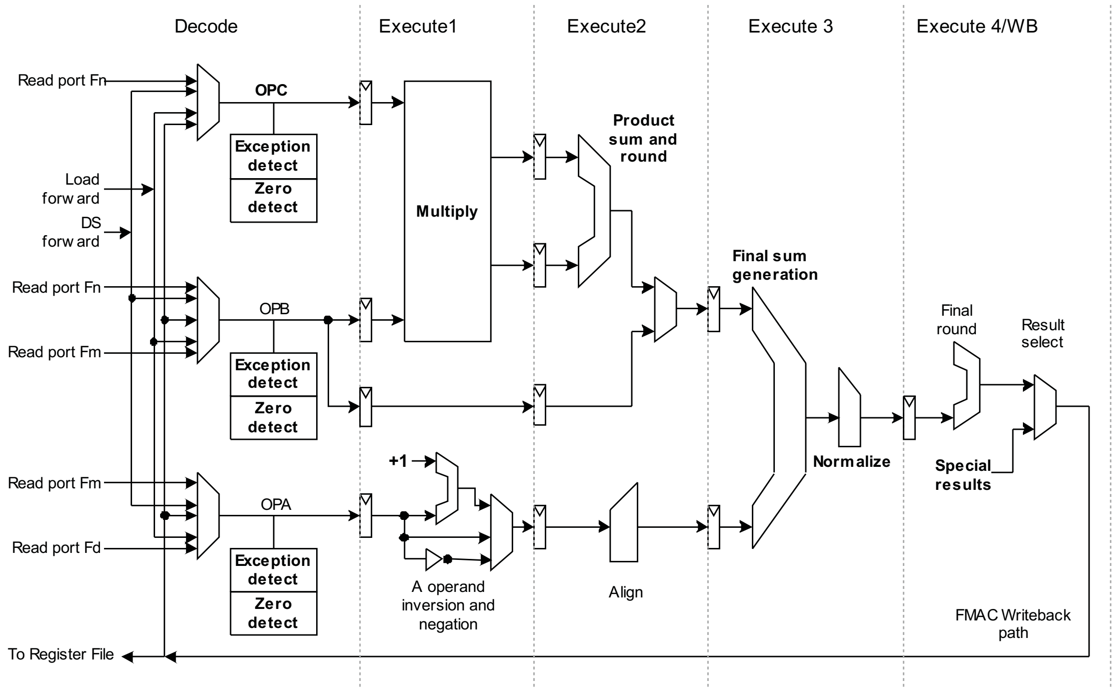


Figure 1-1 FMAC pipeline

1.3.2 FMAC pipeline execution

The FMAC pipeline executes the following instructions:

`FADD`	Addition.
`FSUB`	Subtraction.
`FMUL, FNMUL`	Multiply.
`FMAC,FNMAC,FMSC,FNMSC`	Multiply-accumulate.
`FABS`	Absolute value.
`FNEG`	Negation.
`FUITO, FTOUI`	Unsigned integer conversion.
`FSITO,FTOSI`	Signed integer conversion.

`FTOUIZ`, `FTOSIZ`	
	Floating-point to integer conversion with forced RZ rounding mode.
`FCMP`, `FCMPE`, `FCMPZ`,`FCMPEZ`	
	Comparison.
`FCVTSD,FCVTDS`	
	Format conversion.
`FCPY`	Copy register.

See *Execution timing* on page 4-23 for cycle counts.

The FMAC family of instructions (`FMAC`, `FNMAC`, `FMSC`, and `FNMSC`) perform a chained multiply and accumulate operation. The product is computed, rounded to the specified rounding mode and destination precision, and checked for exceptions before the accumulate operation is performed. The accumulate operation is also rounded to the specified rounding mode and destination precision, and checked for exceptions. The final result is identical to the equivalent sequence of operations executed in sequence. Exception processing and status reporting also reflect the independence of the components of the chained operations.

As an example, the `FMAC` instruction 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 and destination precision and checked for exceptions.
3. The result is summed with the operand in the Fd register.
4. The sum is rounded to the current rounding mode and destination precision and checked for exceptions. If no exception conditions that require support code are present, the result is written to the Fd register.

   For example, the instruction

   ```
   FMACS S0, S1, S2
   ```

   returns the same result as:

   ```
   FMULS TEMP, S1, S2
   FADDS S0, S1, TEMP
   ```

1.3.3 Divide and square root pipeline

The *divide and square root* (DS) pipeline is shown in Figure 1-2.

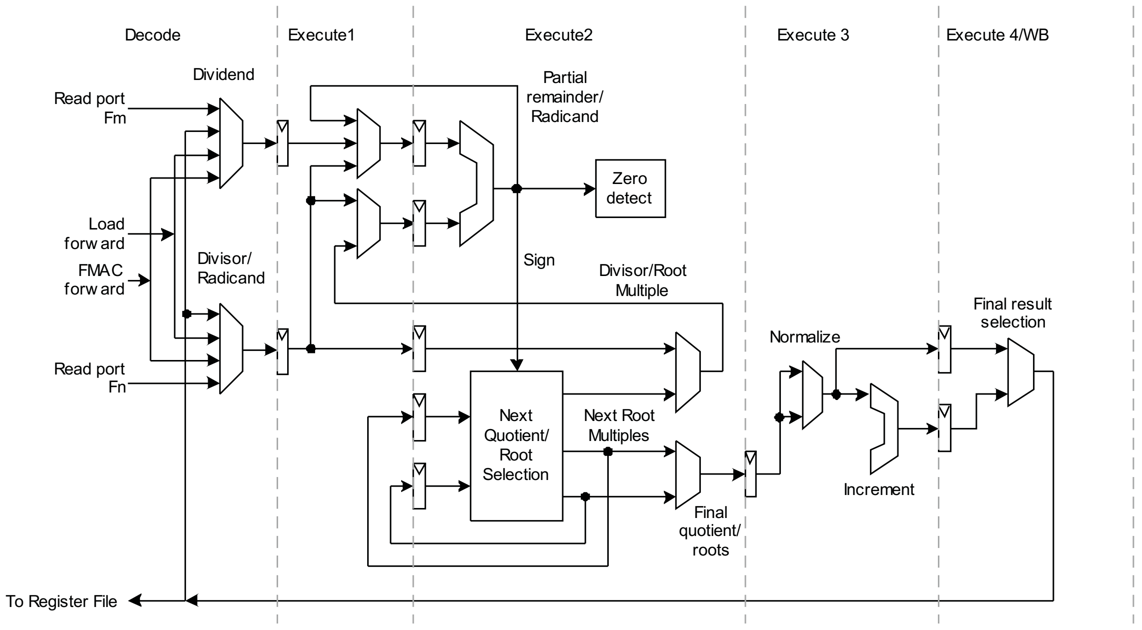


Figure 1-2 Divide and square root pipeline

The divide and square root pipeline executes the following instructions:

`FDIV`	Division
`FSQRT`	Square root

The VFP10 coprocessor executes divide and square root functions for both single-precision and double-precision operands with all IEEE 754 rounding modes supported. The DS unit uses a shared radix-4 algorithm that provides a good balance between speed and chip area. The DS operations have a latency of 17 cycles for single-precision operations and 31 cycles for double-precision operations. The throughput is 14 cycles for single-precision operations and 28 cycles for double-precision operations.

1.3.4 Load/store pipeline

The LS pipeline handles all of the instructions that involve data transfer to and from the ARM1020E macrocell, including loads (`LDC`) and (`LDM`), stores (`STC`) and (`STM`), moves to coprocessor register (`MCR`) and (`MRCC`), and moves from coprocessor register (`MRC`) and `MRRC`). It remains synchronized with the ARM1020E macrocell LS pipeline for the duration of the instruction.

Data written to the ARM1020E macrocell is read from the VFP10 coprocessor register file in the Decode (D) stage and transferred to the ARM1020E macrocell in the same cycle, and is latched on the ARM1020E macrocell Execute/Memory cycle boundary. The transfer is made on a dedicated 64-bit store data bus between all coprocessors and the ARM1020E macrocell.

Load data is written to the VFP10 coprocessor on a dedicated 64-bit load bus between the ARM1020E macrocell and all coprocessors. Data is received by the VFP10 coprocessor on the Memory (M)/Writeback (W) boundary. Data is written to the register file in the Writeback stage, and available for forwarding to `CDP` operations in the same cycle. Figure 1-3 on page 1-10 shows the LS pipeline.

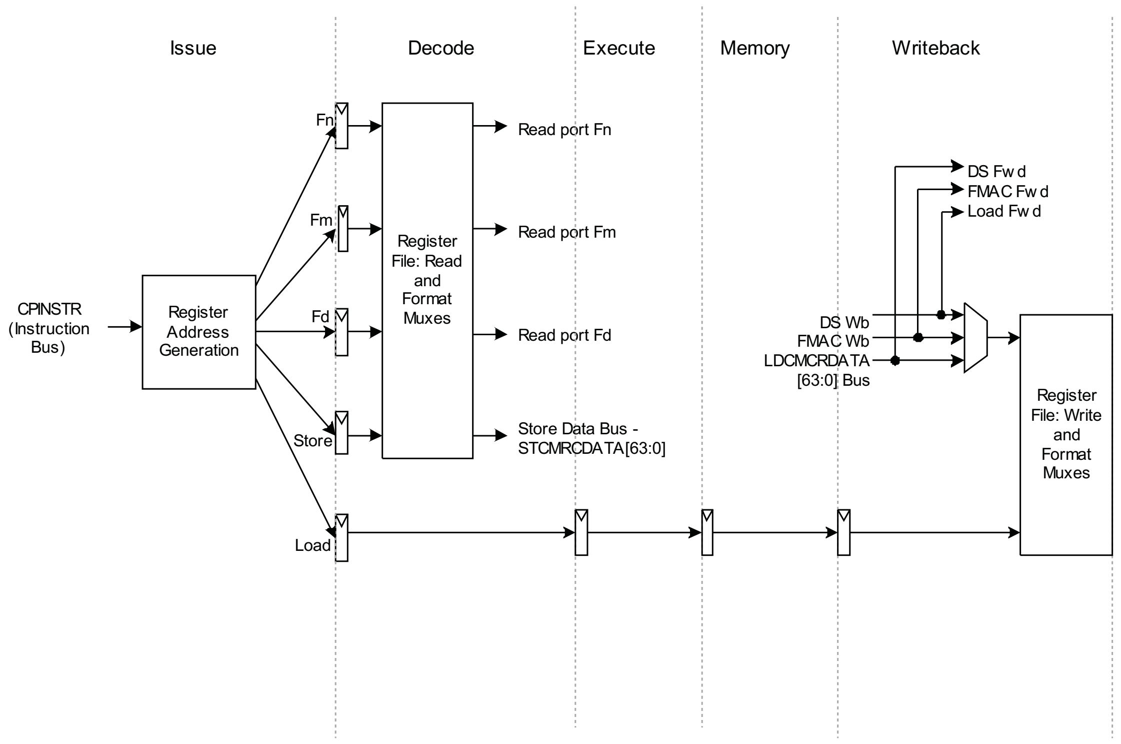

Issue
Decode
Execute
Memory
Writeback
CPINSTR (Instruction Bus)
Register Address Generation
Fn
Fm
Fd
Store
Load
Register File: Read and Format Muxes
Read port Fn
Read port Fm
Read port Fd
Store Data Bus - STCMRCDATA[63:0]
DS Fwd
FMAC Fwd
Load Fwd
DS Wb
FMAC Wb
LDCMCRDATA [63:0] Bus
Register File: Write and Format Muxes

Figure 1-3 Load/Store pipeline

1.3.5 Load/Store operations

The load/store pipeline executes the following instructions:

`FLD`	Load a single data value, either single-precision, double-precision, or 32-bit integer from memory to the VFP10 coprocessor register file.
`FLDM`	Load up to 32 single-precision or integer data values or 16 double-precision data values from memory to the VFP10 coprocessor register file.
`FST`	Store a single data value, either single-precision, double-precision, or 32-bit integer from the VFP10 coprocessor register file to memory.
`FSTM`	Store up to 32 single-precision or integer data values or 16 double-precision data values from the VFP10 coprocessor register file to memory.

`FMSR`	Transfer a single-precision or integer data value from a VFP10 coprocessor single-precision register to an ARM1020E macrocell register.
`FMDHR`	Transfer the upper-half of a double-precision data value from a VFP10 coprocessor double-precision register to an ARM1020E macrocell register.
`FMDLR`	Transfer the lower-half of a double-precision data value from a VFP10 coprocessor double-precision register to an ARM1020E macrocell register.
`FMRS`	Transfer a single-precision or integer data value from an ARM1020E macrocell register to a VFP10 coprocessor single-precision register.
`FMRDH`	Transfer the upper-half of a double-precision data value from a VFP10 coprocessor double-precision register.
`FMRDL`	Transfer the lower-half of a double-precision data value from an ARM1020E macrocell register to a VFP10 coprocessor double-precision register.
`FMDRR`	Transfer two ARM1020E macrocell registers to a double-precision register in the VFP10 coprocessor.
`FMRRD`	Transfer a double-precision register in the VFP10 coprocessor to two ARM1020E macrocell registers.
`FMRRS`	Transfer a pair of consecutively-numbered registers in the VFP10 coprocessor to two ARM1020E macrocell registers.
`FMXR`	Transfer an ARM1020E macrocell register value to a VFP10 coprocessor control register.
`FMRX`	Transfer a VFP10 coprocessor control register to an ARM1020E macrocell register value.

1.4 Modes of operation

The VFP10 coprocessor provides full IEEE 754 compatibility through a combination of hardware and software. Some of the rare cases in the IEEE 754 can require significant additional compute time to resolve correctly according to the requirements of the IEEE 754 specification. For instance, the VFP10 coprocessor does not process subnormal inputs directly. To provide correct handling of input subnormal according to the IEEE 754 specification, a trap is made to support code to process the operation. Using the support code for processing this operation can require hundreds of cycles. In some applications this is unavoidable, as compliance with the IEEE 754 specification is essential to proper operation of the program. In many other applications, especially in the embedded space, strict compliance to the IEEE 754 is unnecessary, while determinable runtime, low interrupt latency, and low power are of more importance. The VFP10 coprocessor provides both:

- the full compliance mode, referred to as non-RunFast, described in *Non-RunFast mode* on page 1-12
- limited compliance mode, referred to as RunFast, described in *RunFast Mode* on page 1-13.

1.4.1 Non-RunFast mode

When the VFP10 coprocessor is not in RunFast mode, all operations that cannot be processed according to the IEEE 754 specification utilize support code for assistance. The operations requiring support code are:

- any CDP operation involving a subnormal input when FTZ mode (FPSCR[24]) is not enabled
- any CDP operation involving a NaN input when DN mode (FPSCR[25]) is not enabled
- any CDP operation that has the potential of generating an underflow condition
- any CDP operation when the *Inexact Exception Enable* (IXE) bit is set
- any CDP operation when overflow is possible and the *Overflow Exception Enable* (OFE) FPSCR[10]) is set
- any CDP operation that involves an invalid combination as the result of a product overflow and the *Invalid Exception Enable* (IOE, FPSCR[8]) is set.

The VFP10 coprocessor properly signals valid exception conditions according to the IEEE 754 specification. The support code is utilized to determine the nature of the exception, whether processing is required to perform preliminary computation for an exception handler, and to call an installed exception handler or signal the termination of the process.

Arithmetic exceptions on page 5-23 describes in greater detail the conditions under which the VFP10 coprocessor traps to support code.

1.4.2 RunFast Mode

Although we refer to the behavior of the VFP10 coprocessor as being in *RunFast Mode*, RunFast is not a mode that is set specifically, but the behavior of the VFP10 coprocessor when the FTZ (Flush-to-Zero Mode, FPSCR[24]) and DN (Default NaN Mode, FPSCR[25]) bits are set, and all exception enable bits are clear, that is, no exceptions are enabled in the FPSCR bits [15], [12:8].

Specifically, in Run Fast mode the VFP10 coprocessor:

- processes an input subnormal operand and a tiny result before rounding as a positive zero
- processes an input NaN as a default NaN
- returns the IEEE specified default result for operations that overflow, operations which are considered as invalid, and for divide-by-zero cases, fully in hardware and without additional latency
- processes all operations in hardware without trapping to support code.

In the FTZ mode, the VFP10 coprocessor treats a subnormal input as a positive zero for computation. An operation that is determined to underflow the range of the destination precision before rounding returns a positive zero.

Two flags are available to provide visibility into the VFP10 coprocessor in FTZ mode

- the IDC bit in the FPSCR (FPSCR[7]) is set in a sticky manner to indicate the presence of a flushed input in the computations executed since this bit was last cleared
- the UFC bit in the FPSCR (FPSCR[11]) is set in a sticky manner to indicate the presence of a flushed result in the computations executed since this bit was last cleared.

These two bits provide visibility to the programmer of the behavior of the code in the presence of very small inputs or results.

The *Default NaN* (DN) mode specifies that the result of any operation that involves either input NaNs or generated a NaN result returns the default NaN. Propagation of the fraction bits is maintained only by `FABS`, `FNEG`, and `FCPY` operations, all other CDP operations ignore any information in the fraction bits of an input NaN.

RunFast mode enables the programmer to write code for the VFP10 coprocessor that runs in determinable time, regardless of the characteristics of the input data, without requiring the support code for assistance completing any operation. Within RunFast mode no user exception traps are available, although exception status bits in the FPSCR will be correct according to the IEEE754 for Inexact, Overflow, Invalid operation, and Divide-by-zero. The Underflow exception status bit has been modified for FTZ mode. All these bits are set by an exceptional condition and can only by cleared by a write to the FPSCR. See *Invalid operation* on page 5-13 and following for more detail on these exceptions.

Specifically, in Run Fast mode the VFP10 coprocessor:

- processes an input subnormal operand as a positive zero
- processes an input NaN as a default NaN
- returns the IEEE specified default result for operations that overflow, operations which are considered as invalid, and for divide-by-zero cases, fully in hardware and without additional latency.

1.5 Short vector instructions

The VFPv2 architecture provides a mechanism for execution of *short vectors* of up to 8 operations on single-precision data and up to 4 operations for double-precision data. Short vectors are most useful in graphics and signal-processing applications. They contribute to smaller code size, faster execution by supporting parallel operations, most notably multiple transfers, and simplify the generation of high data throughput algorithms.

Short vector operations issue the individual operations specified in the instruction in a serial fashion. A short vector does not begin execution until all the source registers are available and all destination registers are not the target of another operation (to eliminate write-after-write hazards).

See Chapter 4 *Instruction Execution in the VFP10 Coprocessor* for more information on instruction execution.

1.6 Parallel execution of instructions

The VFP10 coprocessor provides the ability to execute several floating-point operations in parallel, while the ARM1020E macrocell is executing ARM instructions. While a short vector operation will execute for a number of cycles in the VFP10 coprocessor, it will appear to the ARM1020E macrocell as a single-cycle instruction and be retired in the ARM1020E macrocell before it completes execution in the VFP10 coprocessor.

The three pipelines are designed to operate independently of one another once initial processing is completed. This makes it possible to issue a short vector operation and a load or store multiple operation in the next cycle, and have both executing at the same time, provided no data hazards exist between the two instructions. With this mechanism, algorithms which can be double-buffered can be written to hide much of the time to transfer data to and from the VFP10 coprocessor under the arithmetic operations, resulting in a significant improvement in performance.

The separate divide and square root pipeline allows for operations, both data transfer and CDPs (provided they are not to the DS pipeline) to execute in parallel with the divide. The DS block has a dedicated write port to the register file, and no special care is needed when executing operations in parallel with divide or square root instructions. This is only true for scalar divides; short vector divides will still support the parallel data transfer operations to execute in parallel in the LS pipeline, but the FMAC pipeline will be unavailable until the final iteration of the short vector divide or square root has completed the initial execute cycle. This is described further in *Parallel execution of operations* on page 4-21.

1.7 VFP10 coprocessor treatment of branch instructions

The VFP10 coprocessor does not directly 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 enables the ARM branch instructions and conditional execution capabilities to be used for executing conditional floating-point code. See section C5 of the *ARM Architecture Reference Manual* for information on the use of ARM conditional execution to test IEEE 754 predicates.

In many cases, in which full IEEE 754 comparisons are not needed, simple comparisons of single-precision data, such as comparisons to zero, or to a constant, can be done using a `FMRS` transfer and the ARM `CMP` and `CMN` instructions. This method is faster in many cases than using a `FCMP` followed by an `FMSTAT` instruction. For more information See *Compliance with IEEE-754* on page 3-4.

1.8 Writing optimal VFP10 coprocessor code

These guidelines provide significant performance increases for VFP10 coprocessor code:

- Schedule most scalar operations immediately following each other, provided there is no read-after-write hazard. Scalar double-precision multiply or multiply-accumulate instructions, or short vector instructions of length greater than 1, must be followed by either a single ARM or load/store instruction instead of an arithmetic `FMAC` VFP10 coprocessor instruction.
- Avoid short vector divides and square roots. The VFP10 coprocessor FMAC and DS pipelines are unavailable until the final iteration of the short vector divide or square root is issued from the D stage. If the short vector divide or square root can be separated, other VFP10 coprocessor instructions can be issued in the cycles immediately following the divide or square root. See Example 4-21 on page 4-22 for more information on parallel execution.
- The best performance for data-intensive applications requires double-buffering looped short vector instructions. The vector banks can be divided in half to provide two independent working areas. Arithmetic operations on one half of the bank must be followed by loads or stores to the other bank to take advantage of the simultaneous execution of data transfer operations with the arithmetic instructions.
- The first VFP10 coprocessor instruction following a branch mispredict is serialized and waits for all VFP10 coprocessor 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.
- In non-RunFast mode, avoid reading source operands in the next cycle (this generates a read-after-read hazard).
- Avoid using `FCMPZ`/`FCMPEZ` if fully compliant IEEE 754 comparisons are not required. The use of an `FMRS` instruction with an ARM `CMP` or `CMN` may be faster for simple comparisons.

1.9 Clocking

The VFP10 coprocessor is a fully static design, with a single clock input **GCLK** that can be stopped indefinitely without loss of state. **GCLK** has the same timing requirements as the ARM1020E **GCLK** and is in phase with it. The VFP **GCLK** must be implemented to avoid excessive skew between the ARM1020E **GCLK** clock to preserve signal integrity and timing on the coprocessor interface. Refer to *ARM1020E Technical Reference Manual* for more information on the coprocessor interface.

The clock generation within the VFP10 coprocessor is tightly integrated with the test functionality. Please see the next section on testing for more information on the impact on the clocking by the test logic.

1.10 Testing

The VFP10 coprocessor is a fully-scanned design, with full boundary scan capability allowing for independent testing. See Chapter 5 *Design for Test* for more information on testing.

1.11 Modifications from VFP10 coprocessor (Rev 0)

The VFP10 coprocessor described in this Technical Reference Manual is the second revision of the VFP10 coprocessor design. The first revision, VFP10 coprocessor (Rev 0), was designed as a prototype and not intended for product integration. Significant enhancements have been made to the VFP10 coprocessor as a result of continued development of the ARM floating-point products. These differences are as follows:

- 64-bit transfer instructions, implementations of the V5TE `MCRR` and `MRRC` operations, are included in the VFP10 coprocessor Rev1. These instructions transfer two ARM registers to and from a double-precision or two contiguous single-precision registers in the VFP10 coprocessor. These instructions are described in *ARM v5TE coprocessor extensions* on page 3-11.
- The DS is separate from the primary execution pipeline (FMAC) pipeline, enabling parallel execution of instructions in the FMAC pipeline with a divide or square root in the DS pipeline. Full hazard detection and register interlocking between the two pipelines is handled completely by hardware. This is discussed further in *Parallel execution of operations* on page 4-21.
- Conditions under which an instruction requires support code intervention have been significantly reduced. The VFP10 coprocessor (Rev 0) requires support code to process arithmetic operations involving infinities or which could potentially overflow, and divide-by-zero cases. The VFP10 coprocessor (Rev1) handles infinity inputs, overflow conditions, and divide-by-zero cases according to the IEEE 754 for the case of the exception not enabled. Support code is utilized for any arithmetic operation for which overflow is possible when the overflow trap is enabled. This is discussed further in Chapter 5 *Exception Handling*.
- The VFP10 coprocessor (Rev1) introduces a new mode which simplifies and significantly increases the performance for programs that use NaNs but do not require propagation of the fraction bits of the NaN. This mode, referred to as Default NaN (DN) mode, when enabled causes the VFP10 coprocessor (Rev 1) to process any arithmetic operation involving a NaN in accordance with the IEEE 754 specification. Any arithmetic operation involving a quiet NaN returns the default NaN without trapping to support code. Any arithmetic operation involving a signaling NaN will set the Invalid Operation Exception status bit, and, if the Invalid Operation exception is enabled, a trap is taken and the user trap handler is called. This is described further in *IEEE-754 implementation choices* on page 3-4.
- A further enhancement is made to the performance of high data throughput code when the code is capable of executing in FTZ mode and Default NaN mode, and when no exceptions are enabled. This condition, referred to as RunFast mode, enables the VFP10 coprocessor to remove certain hazard conditions which are

related to the pipeline, namely between short vector operations and loads involving the short vector source registers. In VFP10 coprocessor (Rev 0) the load operation would be required to stall until the scoreboard locks on the source registers were removed by each iteration of the short vector operation. This requirement was present to preserve the source registers in the event of an exception detected on one of the short vector iterations. When executing in RunFast mode the source registers are not required to be preserved, and the load does not stall. This is discussed further in *RunFast Mode* on page 1-13, and in *Hazard and resource stall conditions* on page 4-11.

- A new exception is introduced in Rev1 which identifies cases of an input subnormal when in Flush-to-zero mode. The VFP10 (Rev 0) coprocessor did not report the instance of a subnormal input when flushed to zero in FTZ mode. The new exception status flag is called IDC. A corresponding enable, IDE, allows for trapping on this case to a user trap handler. This is discussed further in *Input subnormal* on page 5-12.
- The functionality of the UFC bit is modified in FTZ mode to identify the flushing of a tiny result. The exception does not cause a trap even if UFE is enabled. This is discussed further in *Underflow* on page 5-19.
- The VFP10 coprocessor Rev0 was implemented in a fully-synthesized methodology, while the Rev1 is a semi-custom design. The VFP10 coprocessor (Rev1) supports full scan testing, with boundary scan for isolation of the VFP10 coprocessor from other modules for testing purposes.

Chapter 2
VFP10 Register File

This chapter describes implementation-specific features of the VFP10 coprocessor that are useful to programmers. It contains the following sections:

- *About the register file* on page 2-2
- *Register file internal formats* on page 2-3
- *Decoding the register file* on page 2-5
- *Loading operands from ARM registers* on page 2-7
- *Maintaining consistency in register precisions* on page 2-9
- *Data transfer between memory and VFP10 coprocessor registers* on page 2-10
- *Access to register banks in CDP operations* on page 2-12.

2.1 About the register file

The ARM VFP10 coprocessor uses a register file that 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 ($R_{(even+1)}$,$R_{(even)}$) can be used to store a double-precision item. The registers in the VFP10 coprocessor can also be used as secondary data storage by a non floating-point application, because no modification of the data is performed on a load or store operation.

The register file addressing is circular within each of the banks for most operations. Load and store operations do not circulate, allowing for multiple banks, up to the entire register file, to be loaded or stored in a single instruction. Short vector operations obey certain rules specifying in what conditions the registers in the argument list specify circular buffers or scalar registers. The LEN and STRIDE fields within the FPSCR specify the number of operations performed by the short vector instructions. Further information and examples are in the ARM *Architecture Reference Manual*, Section C5. The banked approach to the register file supports the use of circular buffers by short vector instructions for applications requiring high data throughput, such as filtering and graphics transforms.

2.2 Register file internal formats

The VFPv2 architecture provides the option of an internal data format that is different from some or all of the external formats. For the VFP10 coprocessor, 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 for single-precision, double-precision, or 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 context saving and restoring VFP data registers you use the `FLDMX`/`FSTMX` instructions for compatibility with future implementations.

Attempting to access a register that has not been initialized or loaded with valid data is `UNPREDICTABLE`. A means to detect access to an uninitialized register is to load all registers with signaling NaNs in the precision of the initial access of the register and enable the invalid exception to detect access to an uninitialized register.

2.2.1 Integer data format

The VFP10 coprocessor supports signed and unsigned 32-bit integers. Signed integers are treated as two's complement values. Figure 2-1 shows the integer format for signed and unsigned integers.

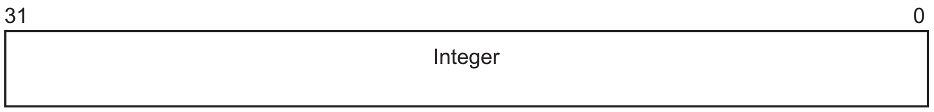


Figure 2-1 Integer format

No modification to the data is implicit in a load, store, or transfer operation on integer data. The format of integer data within the register file is identical to the format in memory or in an ARM general-purpose register.

2.2.2 Single-precision data format

The single-precision data format used in the VFP10 coprocessor is defined in the ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-point Arithmetic. Refer to this for details about:

- the exponent bias
- special formats
- numerical ranges.

Figure 2-2 shows the single-precision bit fields.

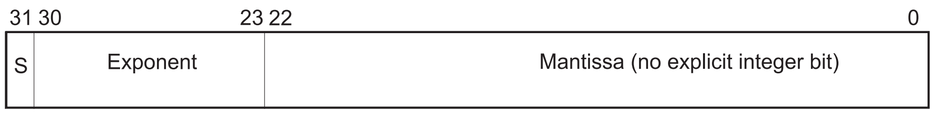


Figure 2-2 Single-precision data format

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].

2.2.3 Double-precision data format

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

- the exponent bias
- special formats
- numerical ranges.

Double-precision format comprises the *Most Significant Word* (MSW) and the *Least Significant Word* (LSW). Figure 2-3 shows the bit fields of the two words in double-precision format.

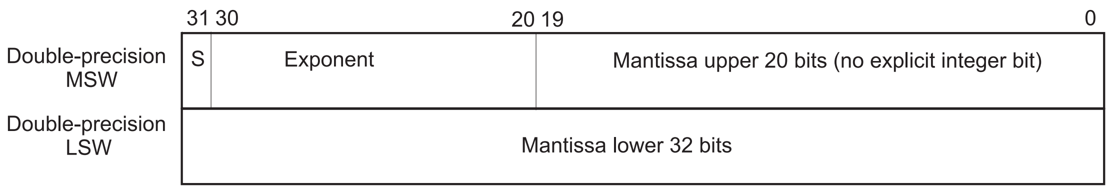


Figure 2-3 Register data formats

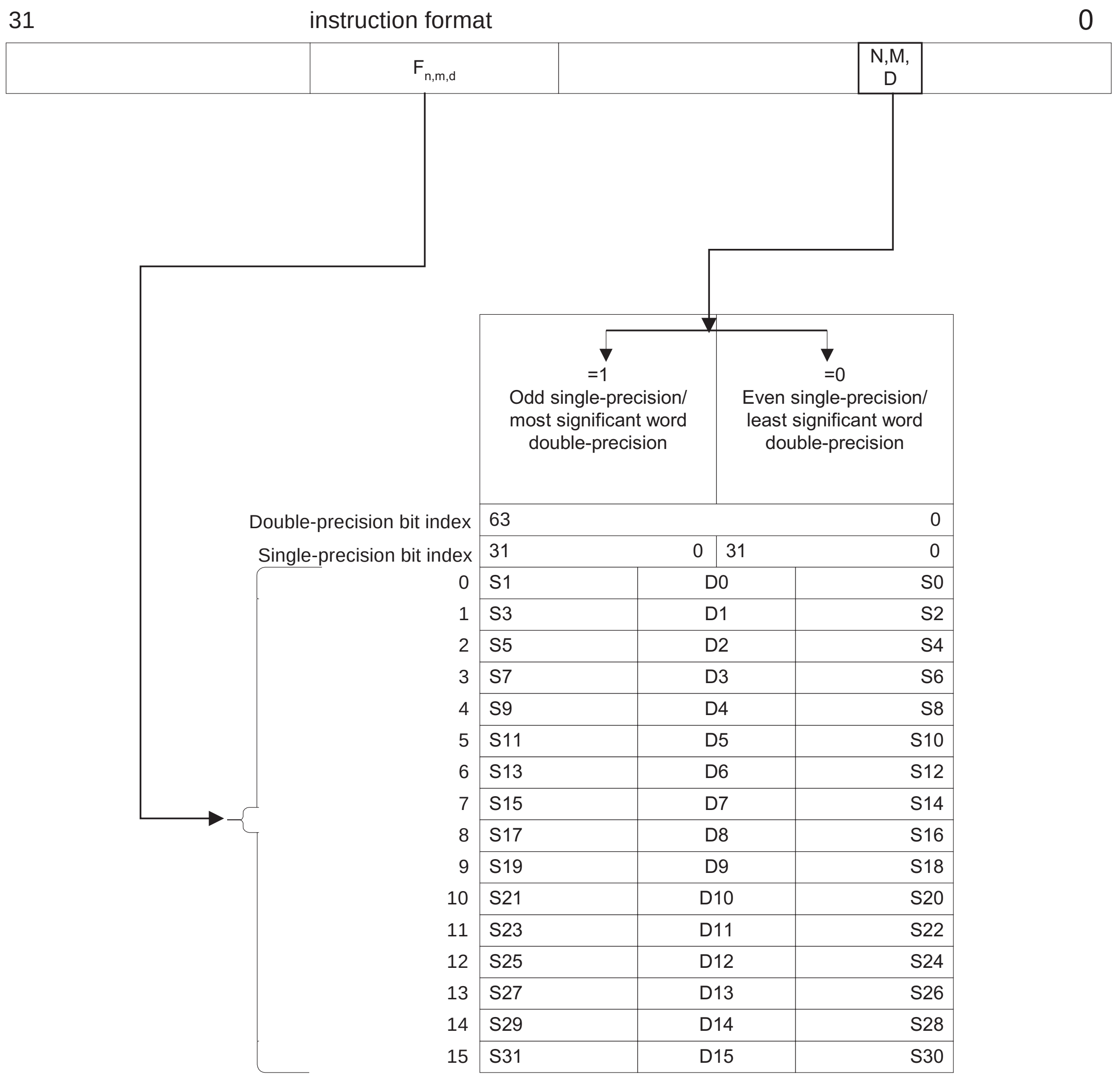
MSW comprises:

- 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.

2.3 Decoding the register file

Register file access involves the most significant four bits of the register number in the instruction word. 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 and integer accesses the most significant four bits is in the Fx bit positions (where x is, m, n, or d) and the least significant bit in the M, N, or D bits respectively for each instruction format. Figure 2-4 on page 2-6 shows the register file encoding. See the *ARM Architecture Reference Manual* for instruction formats and the position of these bits.


31
instruction format
0
$F_{n,m,d}$
N,M, D
=1
Odd single-precision/ most significant word double-precision
=0
Even single-precision/ least significant word double-precision

Double-precision bit index	63		0
Single-precision bit index	31 0		31 0
0	S1	D0	S0
1	S3	D1	S2
2	S5	D2	S4
3	S7	D3	S6
4	S9	D4	S8
5	S11	D5	S10
6	S13	D6	S12
7	S15	D7	S14
8	S17	D8	S16
9	S19	D9	S18
10	S21	D10	S20
11	S23	D11	S22
12	S25	D12	S24
13	S27	D13	S26
14	S29	D14	S28
15	S31	D15	S30

Figure 2-4 Register file format

2.4 Loading operands from ARM registers

Floating-point data can be transferred between ARM registers and VFP10 coprocessor registers using the `MCR`, `MRC`, `MCRR`, and `MRCC` coprocessor data transfer instructions. Single-precision and integer data can be transferred to the ARM1020E macrocell and manipulated in a single ARM register, while double-precision data requires two ARM registers. No exceptions are possible on these transfer instructions.

`MCR and MRC` instructions transfer 32-bit quantities between ARM and VFP10 coprocessor registers. Table 2-1 describes `MCR` transfersr.

Table 2-1 MCR transfers

Instruction	Operation	Description
`FMXR`	VFP System Reg = Rd	System register transfer. Register may be any of FPSID, FPSCR, FPEXC, FPINST, or FPINST2.
`FMDLR`	Dn[31:0] = Rd	Transfer of the lower half of a double-precision data item.
`FMDHR`	Dn[63:32] = Rd	Transfer of the upper half of a double-precision data item.
`FMSR`	Sn = Rd	Transfer of a single-precision or integer data item.

Table 2-2 describes `MRC` transfers.

Table 2-2 MRC transfers

Instruction	Operation	Description
`FMRX`	Rd = VFP System Reg	System register transfer. Register may be any of FPSID, FPSCR, FPEXC, FPINST, or FPINST2.
`FMRDL`	Rd = Dn[31:0]	Transfer of the lower half of a double-precision data item.
`FMRDH`	Rd = Dn[63:32]	Transfer of the upper half of a double-precision data item.
`FMRS`	Rd = Sn	Transfer of a single-precision or integer data item.

`MCRR` and `MRRC` instructions transfer 64-bit quantities between ARM and VFP10 coprocessor registers. Table 2-3 describes `MCRR` transfers.

Table 2-3 MCRR transfers

Instruction	Operation	Description
`FMDRR`	Dm[lower half] = Rd Dm = [upper half]Rn	Transfer the concatenation of Rn:Rd to VFP double-precision register Dm.
`FMSRR`	Sm = Rd Sm+1 = Rn	Transfer the pair of ARM registers {Rn, Rd} to a contiguous pair of VFP single-precision registers {Sm+1, Sm}.

Table 2-4 describes `MRRC` transfers

Table 2-4 MRRC transfers

Instruction	Operation	Description

`FMRRD`	Rd = Dm[lower half]
	Rn = Dm[upper half]
		Transfer the VFP double-precision register Dm to the concatenation of Rn:Rd.

`FMRRS`	Rd = Sm
	Rn = Sm+1
		Transfer the contiguous pair of VFP single-precision registers {Sm+1, Sm} to a pair of ARM registers {Rn, Rd}.

2.5 Maintaining consistency in register precisions

The VFP10 coprocessor register file stores single-precision, double-precision, and integer 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 that 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.

2.6 Data transfer between memory and VFP10 coprocessor registers

The format for accessing data stored in memory is determined by the CP15 control register B bit. The ARM1020E macrocell supports both little-endian and big-endian access formats in memory.

The ARM1020E macrocell 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 LSB bits 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 coprocessor register you must set the lower two bits of the target address to 00.

For single-precision data, Table 2-5 on page 2-10 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 2-5 on page 2-10 and Table 2-6 on page 2-11 the target address is `0x40000000`.

Table 2-5 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 that comprise the data are stored in different locations for little-endian and big-endian data access formats. Table 2-6 shows the data storage in memory and the address to access each byte in little-endian and big-endian access modes.

Table 2-6 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`	`0x40000003`	`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 to provide both little-endian and big-endian addressing to the programmer.

2.7 Access to register banks in CDP operations

The register file is especially suited for short 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*.

2.7.1 About register banks

The register file is divided into 4 banks with 8 registers in each bank for single-precision operations and 4 registers per bank for double-precision operations. The banks are accessed in a circular manner by CDP instructions. Load and store multiple instructions do not access the registers in a circular manner but will treat the register file as a linearly ordered structure.

Table 2-7 shows how the register banks are defined.

Table 2-7 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

A short vector CDP operation that has a source or destination vector crossing a bank boundary accesses the registers within the bank as if the last register in the bank was followed in a linear order by the first register in the bank.

Example 2-1 on page 2-12 shows a short vector operation crossing bank boundaries.

Example 2-1 Register access example

```
For instance, the add operation:
FADDS S11, S22, S31

if treated as a vector of length 6, would access the registers in the following
manner:

FADDS S11, S22, S31      ; the first iteration
FADDS S12, S23, S24      ; the second iteration. The second source vector has
                         ; wrapped around and is accessing the first register in
                         ; the 4th bank
FADDS S13, S16, S25      ; the third iteration. The first source vector has
                         ; wrapped around and is accessing the first register in
                         ; the 3rd bank
FADDS S14, S17, S26      ; the fourth iteration
FADDS S15, S18, S27      ; the fifth iteration
FADDS S8, S19, S28       ; the sixth and last iteration The destination vector
                         ; has wrapped around and is writing to the first;
                         ; register in the second bank
```

2.7.2 Operations using register banks

The register file organization supports four types of operations described in the following sections:

- *Scalar-only operations* on page 2-13
- *Vector-only operations* on page 2-14
- *Vector-only operation with scalar source* on page 2-14
- *Scalar operations in short vector mode* on page 2-15.

See *FPSCR register* on page 3-23 for details of LEN and STRIDE fields and the FPSCR.

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.

- If the LEN field of the FPSCR is not 0, the operation is 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 short vector mode* on page 2-15 shows an example where scalar and short 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 are:

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

Vector-only operations

Vector-only operations require the LEN field to be nonzero, and the destination and Fm registers 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 instruction:

```
FMACS S16, S0, S8
```

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

```
FMACS S16, S0, S8
FMACS S17, S1, S9
FMACS S18, S2, S10
FMACS S19, S3, S11.
```

Vector-only operation with scalar source

The VFPv2 architecture enables 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.

Scalar operations in short vector mode

You can intermix scalar and short vector operations by carefully selecting the source and destination registers. 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 to perform scalar operations.

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

```
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 2-8 to Table 2-11 on page 2-16 summarize the four types of operations possible in the VFPv2 architecture. *Any* refers to the availability of all registers in the precision for the specified operand. The VFP10 coprocessor 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 also for Fm. Table 2-8 describes single-precision three-operand register usage.

Table 2-8 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 2-9 describes single-precision two-operand register usage.

Table 2-9 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 2-10 describes double-precision three-operand register usage.

Table 2-10 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 2-10 describes double-precision two-operand register usage.

Table 2-11 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

Chapter 3
VFP10 Programmer's Model

This chapter describes implementation-specific features of the VFP10 coprocessor that are useful to programmers. It contains the following sections:

- *About the programmer's model* on page 3-2
- *Compliance with IEEE-754* on page 3-4
- *ARM v5TE coprocessor extensions* on page 3-11
- *Summary of VFP coprocessor system control registers* on page 3-17
- *FPSCR register* on page 3-23.

3.1 About the programmer's model

This section gives a general introduction to the VFP10 coprocessor implementation of the VFPv2 floating-point architecture.

ARM Architecture Reference Manual deals with Architecture aspects of VFPv1.

VFP10 implements all the instructions and modes of the VFPv2 architecture. The VFPv2 adds the following features and enhancements to the VFPv1 architecture:

- The ARM v5TE instruction set, which includes `MRRC` and `MCRR` 64-bit ARM to coprocessor transfer instructions. These instructions allow the transfer of a double-precision register, or two consecutively numbered single-precision registers, to or from a pair of ARM registers. See *Loading operands from ARM registers* on page 2-7 for syntax and usage of VFP `MRRC` and `MCRR` instructions.
- The Default NaN operating mode. In this mode, any operation that involves one or more NaNs as operands produces the default NaN as a result, rather than return the NaN or one of the NaNs involved in the operation. This mode is compatible with the IEEE-754 specification but not with current industry handling of NaNs.
- Addition of the subnormal Input exception flag (IDC). This flag is set whenever an operation has as an operand a subnormal value. It remains set until cleared through a write to the FPSCR. A separate trap enable bit is also added (IDE). When set, the VFP10 coprocessor traps to the `UNDEFINED` trap upon an assertion of IDC.
- Modification of the functionality of the UFC bit when FTZ modes are enabled. In this mode, the UFC bit is set whenever a result is below the threshold for normal numbers before rounding, and is flushed to zero. UFC remains set until cleared through a write to the FPSCR. The underflow trap enable bit, UFE, does not cause a trap to the `UNDEFINED` trap handler on an assertion of UFC.
- Modification of the invalid trap functionality when FTZ and DN modes are enabled. In this mode, the IOC bit is set on any operation that would normally have asserted IOC with the exception of certain cases of floating-point to integer conversions. If the conversion is performed in a rounding mode other than round-to-zero (truncate), and the result overflows the destination integer format due to rounding, IOC does not cause a trap to be taken if IOE is enabled, but does set the IOC bit. If IOE is set and a floating-point to integer conversion overflows the destination integer format before rounding, IOC is set, and the VFP10 coprocessor does trap to the `UNDEFINED` trap handler.

- Modification of the functionality of the IXC bit in FTZ mode. In the VFPv1 architecture specification the IXC bit was set when an input or result was flushed to zero. In VFPv2 the IDC and UFC bits provide this information. See *Inexact result* on page 5-21 for more information.

3.2 Compliance with IEEE-754

This section introduces issues connected with IEEE-754 compliance:

- why compliance is important
- hardware and software components
- software-based components and their availability.

3.2.1 An IEEE-754-compliant implementation

The VFP10 coprocessor and support code together provide IEEE-754-compliant implementations of all the floating-point operations supplied by the VFPv2 architecture. Unless a floating-point exception occurs and the enable bit of the exception in the FPSCR is set, it appears to the program that the floating-point instruction was executed by the hardware. However, if in the execution of the instruction an exceptional condition is detected which requires software to complete the operation, the instruction is processed, taking significantly more cycles than normal to produce the result. This only happens for cases whose incidence is typically very low, and is a common practice in the industry.

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

3.2.2 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.

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. See *AFS Firmware Suite Version 1.3 Reference Guide* for details of support code.

3.2.3 IEEE-754 implementation choices

The VFPv2 architecture specifies how various implementation choices allowed by the IEEE-754 standard are made. Full details are in the *ARM Architecture Reference Manual Section C1 1.3*.

Further implementation choices are made within the VFP10 coprocessor about which cases are handled by the VFP10 coprocessor hardware and which cases are bounced to the support code.

To execute frequently encountered operations as fast as possible and minimize silicon area, handling of infrequently occurring values and some exceptions is relegated to the support code. The VFP10 coprocessor supports two modes for handling infrequently occurring values:

- non-RunFast, which is fully-IEEE 754 compliant with support code assistance
- RunFast, which is near fully-IEEE 754 compliant in hardware alone.Non-RunFast requires the floating-point support code to handle certain operands and exceptional conditions not supported in the hardware. Although fully compliant with the IEEE 754, the support code can increase the runtime of an application and increase the size of kernel code.

When the flush-to-zero (FTZ) and default NaN (DN) modes are enabled, and all exceptions are disabled, the VFP10 coprocessor operates in RunFast mode. While the potential loss of accuracy for very small values is present, the use of the RunFast mode removes a significant number of performance-limiting stall conditions, allowing for increased performance of typical and optimized code, and a reduction in the size of kernel code by not requiring the floating-point support code to be present.

Supported formats

The supported formats are:

- Single-precision and double-precision. No extended format is supported.
- Integer formats:
  - unsigned 32-bit integers
  - two's complement signed 32-bit integers.

NaN handling

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.

Table 3-1 Default NaN values

	Single-precision	Double-precision
Sign	`0`	`0`
Exponent	`FF`	`7FF`
Fraction	[22] - 1 [21:0] - all 0	[51] - 1 [50:0] - all 0

Any signaling NaN passed as input to an operation causes an Invalid Operation exception, which is passed to a user handler if present, and if not, then a default quiet NaN is created. The rules for cases involving multiple NaN operands may be found in the *ARM Architecture Reference Manual*.

In the absence of any signalling NaNs, any quiet NaNs passed as input to an operation cause a default quiet NaN to be returned. The return NaN is guaranteed to be one of the input NaNs.

The default NaN for ARM floating-point processors and libraries is defined as follows:

- In non-RunFast mode, NaNs are handled according to the description in the ARM Architecture Reference Manual. The hardware does not process the NaNs directly for arithmetic CDP instructions, but traps to the support code for all NaN processing. For data transfer operations, NaNs are transferred without raising the Invalid Operation Exception or trapping to support code. For the non-arithmetic CDP instructions, `FABS`, `FNEG`, and `FCPY`, NaNs are copied, with change of sign if specified in the instructions, without setting the Invalid Operation Exception or trapping to support code.
- In RunFast mode, NaNs are handled completely within the hardware without support code assistance. Signaling NaNs set the IOC bit when encountered in an arithmetic CDP operation. NaN handling by data transfer and non-arithmetic CDP instructions is the same as in non-RunFast mode. Arithmetic CDP instructions involving NaN operands return the default NaN regardless of the fractions of the NaN operands. Although this is a departure from the behavior of most hardware floating-point units in the industry, it is compliant with the IEEE 754 specification.

Comparisons

Comparison results set condition codes in the FPSCR. The `FMSTAT` instruction transfers the current condition codes in the FPSCR to the ARM CPSR. Refer to the *ARM Architecture Reference Manual* for mapping of IEEE predicates to ARM conditions. The condition codes used are chosen so that subsequent conditional execution of ARM instructions can test the predicates defined in the standard.

The VFP10 coprocessor 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*.

The VFP10 coprocessor supports:

- compare operations `FCMPS`,`FCMPZS`,`FCMPD`,and `FCMPDZS`
- compare with exception operations `FCMPES`,`FCMPEZS`,`FCMPED`,and `FCMPEDZ`.

In the compare family the presence of a signaling NaN compares as unordered and generates an Invalid Operation exception. If the Invalid Operation exception enable is set (IOE, FPSCR[8]) the user trap handler is called. A quiet NaN compares as unordered but does not generate an Invalid Operation exception.

In the compare with exception family the invalid exception is signaled when one or both operands to the compare are NaNs, either signaling or quiet and the comparison is unordered.

Some simple comparisons on single-precision data may be computed directly by the ARM1020E core. If only equality or comparison to zero is needed, and NaNs are not an issue, performing the comparison in ARM registers using `CMP` or `CMN` instructions may be faster.

If comparison to zero is needed, the ARM comparison instructions may be faster. The following instructions set the Z flag for positive values:

```
FMRS    Rx,Sn
CMP     Rx,#0
BEQ     label
```

If the input values might include negative numbers, including negative zero, the following code sets the Z flag correctly:

```
FMRS    Rx, Sn
CMP     Rx, #0x80000000
CMPNE   Rx, #0
BEQ     label
```

Using a temporary register is even faster:

```
FMRS    Rx,Sn
MOVS    Rt,Rx,LSL #1
BEQ     label
```

Comparisons with particular values are also possible. For example, to check if a positive value is greater or equal to +1.0, use:

```
FMRS    Rx,Sn
CMP     Rx,#0x3F800000
BGE     label
```

Magnitude comparisons are possible for single-precision values using the following code.

Note

NaNs compare equal when all bits of the NaN are identical

```
FMRS    Rx,Sn
FMRS    Ry,SM
CMP     Rx,Ry
ORRNE   Rt,Rx,Ry
MOVNES  Rt,Rt,LSL #1
```

The Z flag is set correctly and this makes unsigned comparisons easier.

When comparisons are required for double-precision values or when IEEE comparisons are required, it is safer to use the VFP `FCMP` and `FCMPE` instructions with `FMSTAT`.

Underflow

Note

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

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

In FTZ mode (see part C section 2-4, page C2-13 of the ARM Architecture Reference Manual for information on FTZ mode) results which are tiny before rounding are flushed to a positive zero and the UFC bit in the FPSCR (FPSCR[3]) is set. Support code is not involved.

When the VFP10 coprocessor is not in FTZ mode, any operation for which there exists a risk of tininess occurring bounces to support code. If the operation does not result in a tiny result, the computed result is returned and the UFC bit in the FPSCR (FPSCR[3]) is not set. However, IXC might be set if the operation was inexact. If tininess does occur, the rules given above govern what actions are taken as a result. See *Exception disabled* on page 5-14 for more information on underflow handling.

Exceptions

Exceptions are taken in the VFP10 coprocessor in an imprecise manner. The state of the ARM and of the VFP is not guaranteed to be the state at the point in the program flow at which the exception occurred. Rather, exceptional instructions causes the VFP10 coprocessor to enter an exceptional state, and the next floating-point instruction issued to the VFP10 coprocessor triggers exception processing. It is possible that a number of non-VFP10 instructions and some VFP10 coprocessor instructions may have been executed after the exceptional instruction was issued and before exception processing begins. Any source registers involved in the exceptional instruction are preserved, and the destination register is not overwritten on entry to the support code. Once the support code has processed the exception it returns to the program flow at the point of the trigger instruction, if the detected exception enable is not set, or passes control to a user trap handler if the detected exception enable is set and a trap handler has been installed. If the exception is overflow or underflow, the IEEE 754 specified intermediate result is written to the destination register in the VFP10 coprocessor before the user trap handler is called.

Note

The precise set of facilities available are system-dependent.

3.2.4 Non-IEEE 754 operation modes

The VFP10 coprocessor provides two non-IEEE 754 modes:

- *Flush-to-zero (FTZ) mode* on page 3-9
- *Default NaN mode* on page 3-10.

Flush-to-zero (FTZ) mode

The VFP10 coprocessor provides a *Flush-To-Zero* (FTZ) mode to increase performance on very small inputs and results. FTZ mode is enabled by setting the FZ bit in the FPSCR (FPSCR[24]). When the VFP10 coprocessor is in FTZ mode all input subnormal operands to arithmetic CDP operations are treated as positive zeros in the operation. Exceptions that result from a zero operand are signaled appropriately. `FABS`, `FCMP`, and `FNEG` are not considered arithmetic CDP operations, and are not affected by FTZ mode. Results that are tiny for the destination precision (that is, smaller in magnitude than the minimum normal value) *before rounding* are replaced with a positive zero. Two exception status bits, IDC (FPSCR[15]) and UFC (FPSCR[3]), are used to identify when an input flush or a result flush occurred, respectively.

Default NaN mode

Default NaN mode is selected by setting the DN bit in the (FPSCR [25]). The default for this bit is disabled, or 0. This mode specifies a behavior that is consistent with the IEEE 754 but not with contemporary general purpose or embedded offerings. The IEEE 754 specifies the result of an operation involving a NaN returns a QNaN but suggests the QNaN be of one of the source NaNs. In most contemporary floating-point implementations the fraction bits returned are the fraction bits of the input NaN or one of the input NaNs in a case of more than one, and which input NaN is specified in the architecture. When Default NaN mode is not enabled, the VFPv2 architecture behaves as described in the *ARM Architecture Reference Manual*.

In Default NaN mode any operation involving one or more input NaNs, quiet or signaling, returns the default NaN. The IOC bit is set in any arithmetic CDP instruction with a signalling NaN operand.

The exception to this are data transfer operations and the non-arithmetic operations `FCPY`, `FABS`, and `FNEG`. These operations continue to process NaNs retaining the fraction bits. As in the case when the DN mode is not enabled, no exception status bits can be set for these instructions when a NaN is involved.

3.3 ARM v5TE coprocessor extensions

This section describes the syntax and usage of the four v5TE architecture coprocessor extension instructions:

- *FMDRR* on page 3-11
- *FMRRD* on page 3-12
- *FMSRR* on page 3-13
- *FMRRS* on page 3-15.

3.3.1 FMDRR

The `FMDRR` operation transfers data in two ARM registers to a double-precision register in the VFP10 coprocessor. The ARM registers are not required to be contiguous. Figure 3-1 shows the bit fields for the FMDRR instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				1	1	0	0	0	1	0	0	Rn				Rd				1	0	1	1	0	R	R	1	Dm			

Figure 3-1 FMDRR bit fields

Syntax

```
FMDRR {<cond>} <Dm>, <Rd>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Dm>`	Specifies the destination double-precision VFP coprocessor register.
`<Rd>`	Specifies the source ARM register for the lower half of the 64-bit operand.
`<Rn>`	Specifies the source ARM register for the upper half of the 64-bit operand.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
        Dm[upper half] = Rn
        Dm[lower half] = Rd
```

Notes

Conversions In the programmer's model, FMDRR does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer, whereas most VFP instructions treat the Dm value as a double-precision floating-point number.

3.3.2 FMRRD

The FMRRD operation transfers data in a double-precision register in the VFP to two ARM registers. The ARM registers are not required to be contiguous. Figure 3-2 shows the bit fields for the FMRRD instruction.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11	10	9	8	7	6	5	4	3 2 1 0
Cond	1	1	0	0	0	1	0	1	Rn	Rd	1	0	1	1	0	R	R	1	Dm

Figure 3-2 FMRRD bit fields

Syntax

```
FMRRD {<cond>} <Rd>, <Rn>, <Dm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination ARM register for the lower half of the 64-bit operand.
`<Rn>`	Specifies the destination ARM register for the upper half of the 64-bit operand.

`<Dm>` Specifies the source double-precision VFP coprocessor register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
        Rn = Dm[upper half]
        Rd = Dm[lower half]
```

Notes

Use of R15	If `R15` is specified for `<Rd>` or `<Rn>`, the results are `UNPREDICTABLE`.
Conversions	In the programmer's model, FMRRD does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer, whereas most VFP instructions treat the `Dm` value as a double-precision floating-point number.

3.3.3 FMSRR

The `FMSRR` operation transfers data in two ARM registers to two consecutively numbered single-precision registers `Sm` and `Sm+1` in the VFP10 coprocessor. The ARM registers are not required to be contiguous. Figure 3-3 shows the bit fields of the FMSRR instruction.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11	10	9	8	7	6	5	4	3 2 1 0
Cond	1	1	0	0	0	1	0	0	Rn	Rd	1	0	1	0	0	R	M	1	Sm

Figure 3-3 FMSRR bit fields

Syntax

```
FMSRR {<cond>} <Rd>, <Rn>, <registers>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the source ARM register for the `Sm+1` VFP coprocessor single-precision register.
`<Rn>`	Specifies the source ARM register for the `Sm` VFP coprocessor single-precision register.
`<registers>`	Specifies the pair of consecutively numbered single-precision destination VFP coprocessor registers, separated by a comma and surrounded by brackets. If m is the number of the first register in the list, the list is encoded in the instruction by setting `Sm` and `M` to the top 4 bits and the bottom bit respectively of `m`. For example, if <registers> is `{S1, S2}`, the `Sm` field of the instruction is `0b0000` and the `M` bit is 1.

Architecture version

All

Exceptions

None

Operation

```
If ConditionPassed(cond) then
        Sm = Rd
        Sm+1 = Rn
```

Notes

Conversions In the programmer's model, `FMSRR` does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer, whereas most VFP instructions treat the `Sm` and `Sm+1` values as a single-precision floating-point numbers.

Invalid register lists

If `Sm` is `0b1111` and `M` is 1 (an encoding of `S31`) the instruction is `UNPREDICTABLE`.

3.3.4 FMRRS

The `FMRRS` operation transfers data in two consecutively numbered single-precision registers in the VFP to two ARM registers. The ARM registers are not required to be contiguous. Figure 3-4 shows the bit fields for `FMRRS`.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11	10	9	8	7	6	5	4	3 2 1 0
Cond	1	1	0	0	0	1	0	1	Rn	Rd	1	0	1	0	0	R	M	1	Sm

Figure 3-4 FMRRS bit fields

Syntax

```
FMRRS {<cond>} <Rd>, <Rn>, <registers>
```

where:

`<cond>`	Is the condition under which the instruction is executed. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination ARM register for the `Sm+1` VFP coprocessor single-precision value.
`<Rn>`	Specifies the destination ARM register for the `Sm` VFP coprocessor single-precision value.
`<registers>`	Specifies the pair of consecutively numbered single-precision source VFP coprocessor registers, separated by a comma and surrounded by brackets. If `m` is the number of the first register in the list, the list is encoded in the instruction by setting `Sm` and `M` to the top 4 bits and the bottom bit respectively of `m`. For example, if `<registers>` is `{S16, S17}`, the `Sm` field of the instruction is `0b1000` and the `M` bit is `0`.

Architecture version

All

Exceptions

None

Operation

```
If ConditionPassed(cond) then
        Rd = Sm
```

```
Rn = Sm+1
```

Notes

Conversions In the programmer's model, FMRRS does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer, whereas most VFP instructions treat the Sm and Sm+1 values as a single-precision floating-point numbers.

Invalid register lists

If `Sm` is `0b1111` and `M` is 1 (an encoding of `S31`) the instruction is `UNPREDICTABLE`

Use of R15 If `R15` is specified for `<Rd>` or `<Rn>`, the results are `UNPREDICTABLE`.

3.4 Summary of VFP coprocessor system control registers

The VFP10 coprocessor provides sufficient information for processing of all exception conditions encountered by the hardware. In the event of an exceptional situation, the hardware provides the instruction word, exception status information, such as the detected exceptional condition and in the case of vector operations, the iteration count of the exceptional iteration. These registers are designed to be used with the support code software available from ARM Ltd. As a result, this document does not fully specify exception handling in all cases.

Support for exceptional conditions is provided in hardware through three exception registers:

- FPINST
- FPINST2
- FPEXC.

In addition, the source data registers for an exceptional instruction is available to the support code. However, it is possible that some or all of the other data registers will have been modified and not in the state at the time the exceptional instruction was issued.

Access to the FPEXC, FPINST, and FPINST2 registers is available only in a Privileged mode, and access does not trigger exceptions. The `FMXR` and `FMRX` instructions are used to store and load these registers, respectively. Table 3-2 describes access to these registers.

Table 3-2 Access to control registers

Register	FMXR/FMRX <reg> field encoding	Trigger exception processing?	Legal modes
FPINST	b1001	No	Privileged
FPINST2	b1010	No	Privileged
FPEXC	b1000	No	Privileged

The FPEXC must be saved and restored whenever the context is changed. If the VFP10 coprocessor is in the exceptional state (EX, FPEXC[31], is set) the FPINST and FPINST2 registers must also be saved and restored. The context switch code can be written to consider the EX bit in the determination of which registers to save and restore, or it might choose to save all three.

3.4.1 Instruction word registers (FPINST and FPINST2)

In an exceptional condition, the VFP10 coprocessor provides two exception status registers. The first, FPINST, contains the exceptional instruction, while the second, FPINST2, contains an instruction which was issued and acknowledged by the VFP10 coprocessor before the exception was detected. This instruction has been retired in the ARM1020E processor and cannot be reissued, and must be executed by support code.

The instruction in the FPINST register is in the same format as the issued instruction but is modified in several ways. The condition code bits ([31:28]) have been forced to `1110`, the AL (always) condition. If the instruction is a short vector, the source and destination registers which reference short vectors are updated to point to the source and destination registers of the first exceptional iteration. See *Exception processing for CDP short vector instructions* on page 5-8 for more information.

The instruction in the FPINST2 register is in the same format as the issued instruction and is modified only by the forcing of the condition code bits ([31:28]) to `1110`, the AL (always) condition.

Both the FPINST and FPINST2 registers must be saved and restored in a context switch if the EX bit in the FPSCR (FPSCR[31)] is set. If EX is clear, these registers are not required to be saved and restored. They may be saved and restored to simplify context switch code.

3.4.2 The support code exception status word FPEXC

The FPEXC register contains the VFP enable bit (FPEXC(30). Access to the FPEXC with the `FMRX` and `FMXR` instructions does not cause the `UNDEFINED` instruction trap to be taken if the VFP10 coprocessor is disabled.

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 3-5 on page 3-19.

The exception signals in the FPEXC identify potential exceptional conditions. For two of the bits, INV and UFC, an instruction that sets one of these bits signals a condition that cannot in every situation be completed by the hardware and requires assistance from the support code. These bits do not always signify a true exceptional condition. For example, the UFC flag is set whenever an operation has the potential to generate a result that is below the minimum threshold for the destination precision, which is not known conclusively until the final normalization and rounding in the last stage. The INV bit always represents a condition in which one or more input operands cannot be processed according to the architectural specifications by the hardware. This includes subnormalized inputs when the VFP10 coprocessor is not in FTZ mode and NaNs when the VFP10 coprocessor is not in DN mode. Table 3-3 shows the function of the status and exception bits in the FPEXC.

For the OFC and IOC bits, the conditions identified by these bits being set cause a bounce only when the corresponding trap enable bit in the FPSCR is set. They represent potential exceptional conditions that will be handled by the hardware but could produce a true exceptional condition, but this is not known conclusively until the last stage. If the user wants to take a trap on one of these conditions the bounce must occur based on information known only in the first stage. Support code is required to complete the operation to the point of determination of the exceptional state. If a true exception exists, the user-provided trap handler is called. If not, the result is returned and no exception is signalled. Figure 3-5 on page 3-19 shows the FPEXC bit fields

Note

The support code must clear the EX bit immediately on entry to avoid a recursive exception trap situation. All exception status bits must be cleared before returning from exception code to user code. The FPEXC must be saved and restored in a context switch.

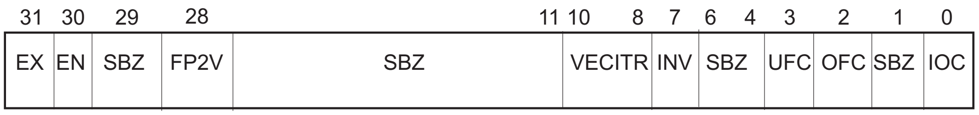


Figure 3-5 FPEXC register format

Table 3-3 shows the FPEXC bit fields.

Table 3-3 FPEXC bit field descriptions

Bit	Name	Description
31	EX	Exception status bit. If set, the VFP is in exception mode and causes all following VFP instructions (except `FMRX` and `FMXR` of the FPEXC, FPINST, FPINST2, or FPSID registers in a Privileged Mode) to assert **CPBOUNCEE**.
30	EN	Enable VFP: 0 = disabled (default) 1 = enabled.
29	SBZ	Should be zero.

Table 3-3 FPEXC bit field descriptions (continued)

Bit	Name	Description
28	FP2V	Set if the FPINST2 register contains a valid instruction.
Bits [27:11]	SBZ	Should be zero.
Bits[10:8]	VECITR	Vector iteration count. This field contains the number of iterations remaining in a short vector operation in which an iteration was exceptional. Details of the counts are given in Table 3-4.
7	INV	Set if the VFP10 coprocessor is not in FTZ mode and an operand is a subnormal or if the VFP10 coprocessor is not in DN mode and an operand is a NaN.
Bits[6:4]	SBZ	Should be zero.
3	UFC	Set if the VFP10 coprocessor is not in FTZ mode and a potential underflow condition exists.
2	OFC	Set if the OFE bit in the FPSCR is set and the VFP10 coprocessor is not in RunFast mode and a potential overflow condition exists.
1	DNM	Do not modify.
0	IOC	Set if the IOE bit in the FPSCR is set and the VFP10 coprocessor is not in RunFast mode and a potential invalid operation condition exists.

Table 3-4 lists the iterations for short vector operations in FPEXC.

Table 3-4 Vector iteration count bit values

Bit values for FPEXC[10:8]	Iterations
`000`	1
`001`	2
`010`	3
`011`	4
`100`	5

Table 3-4 Vector iteration count bit values (continued)

Bit values for FPEXC[10:8]	Iterations
`101`	6
`110`	7
`111`	0

3.4.3 The FPSID register

Figure 3-6 shows the bit fields in the FPSID register.

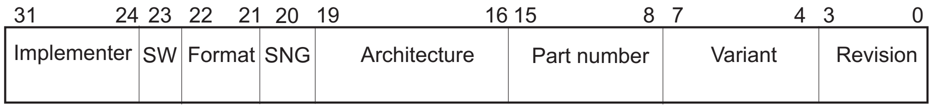


Figure 3-6 FPSID register format

The value for of the FPSID register for the VFP10 coprocessor (Rev1) is `0x410101A0`.

Table 3-5 gives the meanings of the bit fields in FPSID.

Table 3-5 FPSID bit fields

Bit	Meaning	Value
Bits[31:24]	Implementer	`0x41 = A` (ARM Limited)
Bit[23]	Hardware/Software	`0b0`: Hardware implementation
Bits[22:21]	`FSTMX`/`FLDMX` format	`0b00`: Format 1
Bit[20]	Precisions supported	`0b0`: Both single-precision and double-precision data are supported
Bits[19:16]	Architecture version	`0b0001`: VFPv2 architecture

Table 3-5 FPSID bit fields

Bit	Meaning	Value
Bits[15:8]	Part number	`0x10`: VFP10 (Rev 1)
Bits[7:4]	Variant	`0xA`: ARM10 coprocessor interface
Bits[3:0]	Revision	`0x0`: First version

Access to the FPSID register with the `FMRX` and `FMXR` instructions does not trigger exception processing in any ARM processor mode. The FPSID may be read when the VFP10 coprocessor is disabled without causing an `UNDEFINED` instruction trap to be taken.

3.5 FPSCR register

All FPSCR bits can be read and written, and can be accessed in both privileged and unprivileged modes. All bits described as *SBZ (Should be Zero)* in Figure 3-7 are reserved for future expansion. They are initialized to zeros. Non-initialization code must use read/modify/write techniques when handling the FPSCR, to ensure that these bits are not modified. Failure to observe this rule can result in code which has unexpected side effects on future systems. Figure 3-7 shows the bit fields for the FPSCR register.

31	30	29	28	27 26	25	24	23 22	21 20	19	18 16	15	14 13	12	11	10	9	8	7	6 5	4	3	2	1	0
N	Z	C	V	SBZ	DN	FZ	RMode	Stride	SBZ	LEN	IDE	SBZ	IXE	UFE	OFE	DZE	IOE	IDC	SBZ	IXC	UFC	OFC	DZC	IOC

Figure 3-7 User status and control bit fields summary

3.5.1 FPSCR bit descriptions

The FPSCR bits are described in the following subsections:

- *Condition flags* on page 3-23
- *Default NaN mode control* on page 3-24
- *FTZ mode control* on page 3-24
- *Rounding mode control* on page 3-24
- *Vector length/stride control* on page 3-25
- *Exception status and control* on page 3-26.

Condition flags

Bits[31:28] of the FPSCR contain the results of the most recent floating-point comparison:

N	Is 1 if the comparison produced a *less than* result.
Z	Is 1 if the comparison produced an *equal* result.
C	Is 1 if the comparison produced an *equal*, *greater than* or *unordered* result.
V	Is 1 if the comparison produced an *unordered* result.

These condition flags do not directly affect conditional execution, either of ARM instructions or of VFP instructions. A comparison instruction is normally followed by an `FMSTAT` instruction. This transfers the FPSCR condition flags to the ARM CPSR flags, after which they can affect conditional execution.

Default NaN mode control

Bit[25] of the FPSCR is the DN bit and controls default NaN mode.

DN == 0	Default NaN mode is disabled and the behavior of the floating-point system is fully compliant with the IEEE 754 standard.
DN == 1	Default NaN mode is enabled.

FTZ mode control

Bit[24] of the FPSCR is the FZ bit and controls *flush-to-zero mode*.

FZ == 0	Flush-to-zero mode is disabled and the behavior of the floating-point system is fully compliant with the IEEE 754 standard.
FZ == 1	Flush-to-zero mode is enabled.

Rounding mode control

Bits[23:22] of the FPSCR select the current rounding mode. This rounding mode is used for almost all floating-point instructions. The only floating-point instructions which do not use it are `FTOSIZD`, `FTOSIZS`, `FTOUIZD` and `FTOUIZS`, which always use RZ mode.

The rounding modes are encoded as follows:

0b00	Indicates *Round to Nearest* (RN) mode.
0b01	Indicates *Round towards Plus Infinity* (RP) mode.
0b10	Indicates *Round towards Minus Infinity* (RM) mode.
0b11	Indicates *Round towards Zero* (RZ) mode.

Vector length/stride control

The LEN field (bits [18:16]) of the FPSCR controls the vector length for VFP instructions that operate on short vectors, that is, how many registers are in a vector operand. Similarly, the STRIDE field (bits[21:20]) controls the vector stride, that is, how far apart the registers in a vector lie in the register bank. The allowed combinations of LEN and STRIDE are shown in Table 3-6.

All other combinations of LEN and STRIDE produce UNPREDICTABLE results.

The combination LEN == 0b000, STRIDE == 0b00 is sometimes called *scalar mode*. When it is in effect, all arithmetic instructions specify simple scalar operations. Otherwise, most arithmetic instructions specify a scalar operation if their destination lies in the range S0-S7 (for single precision) or D0-D3 (for double precision). The full rules used to determine which operands are vectors and full details of how vector operands are specified can be found in *The ARM Architecture Reference Manual*.

The rules for vector operands do not allow the same register to appear twice or more in a vector. The allowed LEN/STRIDE combinations listed in Table 3-6 never cause this to happen for single-precision instructions, so single-precision scalar and vector instructions can be used with all of these LEN/STRIDE combinations.

For double-precision vector instructions, some of the allowed LEN/STRIDE combinations would cause the same register to appear twice in a vector. If a double-precision vector instruction is executed with such a LEN/STRIDE combination in effect, the instruction is UNPREDICTABLE. The last column of Table 3-6 indicates which LEN/STRIDE combinations this applies to. Double-precision scalar instructions work normally with all of the allowed LEN/STRIDE combinations.

Table 3-6 Vector length/stride combinations

LEN	STRIDE	Vector length	Vector stride	Double-precision vector instructions
0b000	0b00	1	-	All instructions are scalar
0b001	0b00	2	1	Work normally
0b001	0b11	2	2	Work normally
0b010	0b00	3	1	Work normally
0b010	0b11	3	2	UNPREDICTABLE
0b011	0b00	4	1	Work normally

Table 3-6 Vector length/stride combinations (continued)

LEN	STRIDE	Vector length	Vector stride	Double-precision vector instructions
0b011	0b11	4	2	UNPREDICTABLE
0b100	0b00	5	1	UNPREDICTABLE
0b101	0b00	6	1	UNPREDICTABLE
0b110	0b00	7	1	UNPREDICTABLE
0b111	0b00	8	1	UNPREDICTABLE

Exception status and control

Bits [12:8] and bits [4:0] of the FPSCR are the trap enable bits and cumulative exception bits respectively for the six types of exception.

Table 3-7 shows which bits are associated with each exception.

Table 3-7 Exception status and control bits

Exception type	Trap enable bit	Cumulative exception bit
Invalid Operation	IOE (bit[8])	IOC (bit[0])
Division by Zero	DZE (bit[9])	DZC (bit[1])
Overflow	OFE (bit[10])	OFC (bit[2])
Underflow	UFE (bit[11])	UFC (bit[3])
Inexact	IXE (bit[12])	IXC (bit[4])
Input Denormal	IDE (bit[15])	IOC (bit[7])

Chapter 4
Instruction Execution in the VFP10 Coprocessor

This chapter contains detailed information about the ARM VFP10 coprocessor instruction execution. It contains the following sections:

- *About instruction execution in the VFP10 coprocessor* on page 4-2
- *Serializing instructions* on page 4-4
- *Interrupting VFP10 coprocessor instructions* on page 4-5
- *Hazard and resource stall conditions* on page 4-11
- *Parallel execution of operations* on page 4-21
- *Execution timing* on page 4-23.

4.1 About instruction execution in the VFP10 coprocessor

The VFP10 coprocessor supports in hardware all addressing modes described in section C5 of the *ARM Architecture Reference Manual*.

The advanced features of the VFP10 coprocessor, specifically the short vector operations and the recirculating register file, are further enhanced in the VFP10 coprocessor through a high-performance interface that allows the VFP10 coprocessor to execute several operations in parallel. To the ARM1020E processor, a short vector operation appears as a single-cycle operation. The short vector operation issues in a single cycle and, once clear of hazards, proceeds through the ARM pipeline one stage per cycle, while iterating in the VFP10 coprocessor pipeline for numerous cycles.

The appearance of a short vector operation to the ARM1020E processor as a single-cycle instruction permits the ARM1020E processor to continue execution of both ARM1020E processor and coprocessor instructions without waiting for the short vector operation to retire. In addition, the VFP10 coprocessor, with a separate LS pipeline, can execute load or store operations while processing short vector operations. This allows for very efficient processing of high data throughput operations such as filters and matrix computations. With the large register set, most operations can be double buffered, with one data buffer processed in the arithmetic pipeline while the other buffer is stored or loaded. A more detailed description of the parallel execution capabilities of the VFP10 coprocessor is given in *An example of parallel execution* on page 4-21.

4.1.1 Interrupting serializing instructions

The overlapping execution of instructions can be interrupted by serializing instructions. These instructions stall both the VFP10 coprocessor and ARM instruction Issue stages until the VFP10 coprocessor pipelines are past the point of updating either the condition codes or exception status or when a write to a system register can no longer affect the operation of a current or pending instruction. Serializing instructions may be used to capture condition codes and exception status, or to delineate a block of instructions for execution with the ability to capture the exception status of that block of instructions.

Serializing instructions are the `FMRX` and `FMXR` operations, including the `FMSTAT` instruction. These operations are also used to modify the mode of operation of subsequent instructions, such as the rounding mode or vector length. See the *ARM Architecture Reference Manual* for more information on serializing instructions.

4.1.2 Hazard detection

The VFP10 coprocessor detects and processes hazards completely in hardware. A *hazard* is a condition in which a prior instruction may change the contents of a register required by a subsequent instruction after the contents of the register have been read (this is a read-after-write, or RAW, hazard) or a later instruction may write a register before an earlier instruction will write it, causing the register to contain the data written by the earlier instruction rather than the later one (this is a write-after-write, or WAW, hazard), or a later instruction writes a register before an earlier instruction can read the prior contents (this is a write-after-read, or WAR, hazard.)

A fourth hazard exists, although not a data hazard, in which a later instruction is reading a register before an earlier instruction has read the register (this is a read-after-read, or RAR, hazard). This last hazard is a control hazard, and can cause disruption in the register scoreboarding logic, allowing one of the first 3 hazards to occur.

4.2 Serializing instructions

The following instructions behave as serializing operations until the information in the read, for `FMRX` operations, is valid up to this instruction, or the impact of the write, for `FMRX` operations, cannot affect current or pending operations.

For read operations, such as a read of the FPSCR, the instructions currently in the pipeline, in most cases, could cause a change in the condition codes, as in the case of a `FCMP` instruction, or exception status flags, such as INEXACT. A read of the FPSCR, FPEXC, FPINST, or FPINST2, is stalled until it is no longer possible for these registers to be changed by any instruction executing or awaiting execution in any of the VFP10 coprocessor pipelines.

A write to the FPSCR stalls until modification of any of the control bits cannot affect any operation currently executing or awaiting execution. Writing the FPEXC, FPINST, or FPINST2 registers will stall until the pipeline is completely clear before executing.

The FPSID register is a unique case. While the contents are unchangeable by any instruction, accessing this register may be used as a general purpose serializing operation or to create a exception boundary.

Note

`FMXR`, `FMRX`, and `FPSTAT` instructions are valid trigger instructions, and cause exception processing if a pending exception has caused the VFP10 coprocessor to be in the exceptional state. The instruction that causes the trigger is executed on the return of the exception processing routine.

4.3 Interrupting VFP10 coprocessor instructions

VFP instructions are issued by the core and maintain a lockstep between the core and the VFP10 coprocessor until the instruction completes, for load and store operations, or completes the Execute stage in the core. While VFP instructions can be short vectors with long execution times, the core sees only a single-cycle instruction and retires the instruction in the core many cycles before it is retires in the VFP10 coprocessor. When the core takes an interrupt any instruction which is flushed from the core pipeline will also be flushed from the VFP10 pipeline. Any instructions which are stalled by either the core or the VFP10 coprocessor will be flushed.

If the interrupt is the result of a data abort condition, the load or store operation which caused the abort will be restarted once the interrupt condition has been handled. Load and store multiples are idempotent, allowing for load and store multiple operations to detect some exception conditions after transfer has begun, and interrupt the operation after the initial transfer. Once the interrupt has been processed, the load or store may restart from the beginning; the source data is guaranteed to be unchanged and no operations depending on the load or store data will have executed until the load or store operation is completed.

Once the interrupting condition has begun processing, the VFP10 coprocessor may not be available to the interrupt routine until any short vector operations which were begun before the interrupt was processed and passed the core Execute stage has passed the VFP10 coprocessor Execute 1 stage. In other words, the VFP10 coprocessor can still have resource and data hazards which could impact the execution of a context switch of the VFP10 coprocessor after the interrupt has been taken. The maximum delay the VFP10 coprocessor may be unavailable is the time to process a short vector of 8 single-precision divide or square root operation, or 114 cycles after the divide or square root has entered the Execute 1 stage of the VFP10 coprocessor.

4.4 Forwarding

The VFP10 coprocessor forwards data from load operations and CDP operations to CDP operations. In general, any forwarding operation reduces by one the number of cycles a dependent operation that would have stalled waiting on the forwarded data. The VFP10 coprocessor does not forward in the following cases:

- To or from an operation involving integer data, either as a producer or consumer
- To a store operation (`FST`,`FSTM`, `MRC`, `MRRC`)
- To any operation with a different source precision than the precision of the writeback data.

In Example 4-1 no forward from D2 to the `FADDS` operation occurs even though S5 is the upper half of D2.

Example 4-1 No forwarding with different precisions

```
FMULD   D2,D0,D1
FADDS   S12,S13,S5
```

In the following examples, Example 4-2 to Example 4-12 on page 4-9, the stall counts listed are given assuming that all transfers to and from memory hit in the cache and are aligned in memory according to the size of the transfer (8-byte aligned for FLDM and FSTM operations.) Memory access timings directly impact final cycle counts and should be taken into account when predicting performance.

In Example 4-2 the load data is not forwarded to the float-to-integer conversion operation. The `FTOUIS` instruction stalls for 3 cycles until the data has been loaded into the register file before reading the operand for the conversion. No forwarding is done to or from integer operations.

Example 4-2 Load data not forwarded

```
FLDS    S1,[Rx]
FTOUIS  S2,S1
```

Example 4-3 shows a store of the destination register of the double-precision `FMULD`. The `FSTD` stalls for 5 cycles (the `FMULD` requires two cycles in the Execute 1 stage) and there is no forwarding path to store instructions in the VFP10 coprocessor.

Example 4-3 Store of a destination register of a double-precision FMULD

```
FMULD   D1,D2,D3
FSTD    D1,[Rx]
```

Example 4-4 shows a single-precision case of Example 4-3. Again, no forwarding is done from the `FADDS` to the `FSTS`, and the `FSTS` stalls for 4 cycles.

Example 4-4 Store of a destination register of a single-precision FMULD

```
FADDS   S1,S2,S3
FSTS    S1,[Rx]
```

In Example 4-5 the second `FADDS` instruction is dependent on the result of the first `FADDS` instruction. The result of the first `FADDS` is forwarded to the second `FADDS` instruction, reducing the stall from 4 cycles to 3 cycles.

Example 4-5 Second FADD dependent on result of first FADD

```
FADDS   S1,S2,S3
FADDS   S8,S9,S1
```

Example 4-6 shows a double-precision case of Example 4-5. The result of the first `FMACS` is forwarded to the second `FMACS`, reducing the stall from 5 cycles to 4 cycles.

Example 4-6 Reducing stall cycles

```
FMACS   D1,D2,D3
FMACS   D8,D9,D1
```

Example 4-7 is similar to Example 4-5 because the result of the `FADDS` is not forwarded to the `FTOUIS`. The `FTOUIS` stalls for 4 cycles.

Example 4-7 FADDS not forwarded

```
FADDS   S1,S2,S3
FTOUIS  S12,S1
```

In Example 4-8 the result of the compare is loaded into the ARM CPSR and a conditional branch is performed based on the condition codes from the FCMPS. In this case, the FMSTAT stalls for 3 cycles until the condition codes from the FCMPS are known. 2 cycles later the CPSR in the ARM1020E is updated with the condition codes, and a branch decision can be made based on the result of the FCMPS.

Example 4-8 Condition codes and branches

```
FCMPS   S1,S2
FMSTAT
Bxx     label
```

Example 4-9 illustrates the use of the ARM10E core to perform simple comparisons on single-precision data. The CMP stalls for 1 cycle until the data is in the ARM register.

Example 4-9 Using the ARM10E core for comparisons

```
FMRS    Rx,S1
CMP     Rx,Constant
Bxx     label
```

In Example 4-10, the FADDS requires S15, which is being loaded in the FLDM, to be valid. The FLDM attempts to load the data in order of lowest register number to highest register number, making the loading of S15 the last load performed. The VFP10 coprocessor interface to the ARM1020E core is 64-bits, allowing two single-precision data values to be loaded in a single cycle. S15 is loaded in the 4th transfer of the FLDM. The FADDS executes after a stall of 2 + Nt cycles, where N is the transfer iteration, beginning with 0 for S8 and S9, and t is the number of cycles between transfers in the FLDM. For Example 4-10, if data is transferred every two cycles, the stall for the FADDS is 2 + 3 * 2, for 8 cycles.

Example 4-10 Last load causing a stall

```
FLDM    [Rx],{S8-S15}
FADDS   S1,S2,S15
```

In Example 4-11 on page 4-9 the FADDS is stalled by the divide for 16 cycles. If the operations were in double-precision rather than single-precision, the stall for the FADDD would be 30 cycles.

Example 4-11 FDIVS stall

```
FDIVS S1, S2, S3
FADDS S4, S5, S1
```

Example 4-12 shows a resource conflict for the DS pipeline. The second `FSQRT` stalls for 13 cycles without a data conflict, and 16 cycles if the destination of the first `FSQRT` is a source operand for the second. If the operations were in double-precision rather than single-precision, the stall counts would be 27 and 30, respectively.

Example 4-12 Resource conflict in the DS pipeline

```
FSQRT S1, S2
FSQRT S3, S4
```

4.4.1 Operation of the scoreboard

The scoreboard contains a single bit for each register which will not be available to an instruction in the next cycle. Note that no distinction is made between source and destination register. As a result, Read-after-Read hazards are detected as valid hazards by the VFP10 coprocessor. Clearing of bits in the scoreboard lock register is done at two points in the pipelines. Source registers for store operations are locked, if the operation is a store multiple, and cleared in the E stage of the LS pipeline. Source registers for CDP operations are cleared in the E1 stage of the pipeline for scalar operations, and for short vector operations the registers involved in the iteration in the E1 stage are cleared.

Destination registers are cleared in the cycle before they are written back to the register file or available for forwarding. Destination registers are cleared in the scoreboard in the E3 stage of either the FMAC or DS pipeline and in the Memory stage of the LS pipeline.

The registers involved in an operation, both as source and destination, are determined in the I stage of the VFP10 coprocessor pipeline and a lock mask is generated. Registers involved in each iteration of a short vector operation are included in the lock mask. As described in the next several sections, the determination of the which source registers are included in the lock mask is a function also of the RunFast mode. A check is made in this cycle on the scoreboard lock register, and if the lock mask and the lock register do not contain any of the same registers, the lock mask is ORed with the lock register to form the new scoreboard register. If a hazard is detected, the lock register is not updated and the instruction stalls in the I stage.

Note

The clearing of registers and the check are performed in parallel, with the cleared registers not available to the check operation until the following cycle. The clearing is done in the cycle before the data is available, and will not stall an operation unless the data is not available in the next cycle.

4.5 Hazard and resource stall conditions

The VFP10 coprocessor incorporates full hazard detection and implements a fully-interlocked pipeline protocol. No scheduling is required by the compiler to guarantee that the instructions execute in what appears to be a serial order and with the same results as if each instruction were allowed to complete fully before the subsequent instruction was allowed to begin. The VFP10 coprocessor uses a scoreboard mechanism to process interlocks caused by either source or destination registers unavailable to the instructions or unavailable data and to stall the instruction until all data operands or registers are available when required.

The determination of hazards and interlock conditions is different in non-RunFast mode and RunFast mode. RunFast mode, with its guarantee of no bounce conditions, implements a less strict hazard detection mechanism, allowing, in some cases, for instructions to begin execution earlier in time than in non-RunFast mode. *Interlock determination in non-RunFast mode* on page 4-11 and *Interlock determination in RunFast mode* on page 4-13 describe these differences.

4.5.1 Interlock determination in non-RunFast mode

The possibility of a bounce condition on any operation requires all source registers for that operation, and for any iterations remaining after the bounced iteration, to be unchanged by subsequent instructions. This causes the typical read-after-write (RAW) and write-after-read (WAR) hazards, as well as the read-after-read (RAR) hazard, to introduce stalls in the pipeline. The nature of the scoreboard does not support a distinction between source registers and destination registers, and continues to detect a hazard on any register involved in a computation until the lock on that register is cleared. Source registers, which are not also the destination, have their locks cleared in the first execute stage (E1) and destination register locks are cleared in the next to last execute stage (E3).

Vector operations are not allowed to begin execution until all registers involved in the operation are not locked. When a short vector operation is allowed to proceed in the pipeline beyond the Decode (D) stage, all registers involved in the operation are locked. Each iteration clears its source register locks (provided they are not also the destination register) in the E1 stage and the destination register in the next to last execute stage.

Table 4-1 and Table 4-2 show the registers that are locked and the cycle in which they are cleared for both scalar (VECITR set to 0) and short vector operations.An *L* in Table 4-1 and Table 4-2 denotes the source registers for that iteration are locked in the scoreboard.

Table 4-1 Single-precision source register locking and clearing in non-RunFast mode

	Iteration source registers locked in D stage								Cycle iteration source registers cleared in E1							
VECITR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	L	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-
1	L	L	-	-	-	-	-	-	-	2	-	-	-	-	-	-
2	L	L	L	-	-	-	-	-	-	-	3	-	-	-	-	-
3	L	L	L	L	-	-	-	-	-	-	-	4	-	-	-	-
4	L	L	L	L	L	-	-	-	-	-	-	-	5	-	-	-
5	L	L	L	L	L	L	-	-	-	-	-	-	-	6	-	-
6	L	L	L	L	L	L	L	-	-	-	-	-	-	-	7	-
7	L	L	L	L	L	L	L	L	-	-	-	-	-	-	-	8

For double-precision operations the source register is cleared in the first E1 cycle, with operations involving a multiplication requiring 2 cycles in the E1 stage as shown in Table 4-2. For example, for a 2 iteration `FMULD` instruction, the source registers for the second iteration are cleared in cycle 3.

Table 4-2 Double-precision source register locking and clearing in non-RunFast mode

	Iteration source registers locked in D stage				Cycle iteration source registers cleared in E1			
VECITR	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
0	L	-	-	-	1/1	-	-	-
1	L	L	-	-	-	2/3	-	-
2	L	L	L	-	-	-	3/5	-
3	L	L	L	L	-	-	-	4/7

4.5.2 Interlock determination in RunFast mode

RunFast mode guarantees that no bouncing is possible when all exceptions are disabled, removing the requirement to preserve source registers. For all scalar operations and non-multiple store operations no source registers are locked. For short vector operations, the length of the vector dictates which source registers are locked. Table 4-3 shows the source registers that are locked for a short vector operation and in which cycle they are cleared.

Table 4-3 Single-precision source register locking and clearing in RunFast mode

	Iteration source registers locked in D stage								Cycle iteration source registers cleared in E1							
VECITR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	L	-	-	-	-	-	-	-	1	-	-	-
5	-	-	-	-	L	L	-	-	-	-	-	-	-	2	-	-
6	-	-	-	-	L	L	L	-	-	-	-	-	-	-	3	-
7	-	-	-	-	L	L	L	L	-	-	-	-	-	-	-	4

Table 4-4 shows the source registers that are locked for a short vector operation and in which cycle they are cleared.

Table 4-4 Double-precision source register locking and clearing in RunFast mode

	Iteration source registers locked in D stage				Cycle iteration source registers cleared in E1			
VECITR	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	L	-	-	-	1/1	-
3	--	-	L	L	-	-	-	2/3

4.5.3 Examples of hazard conditions

Source registers must be protected in the event of an exceptional condition on the instruction or an iteration if it is a short vector operation. Read-after-read hazards are respected, that is, a read of a locked source register will stall until the source register is released by the prior operation.

Source registers are cleared in the first E1 cycle of an operation. Destination registers are cleared in the 2nd to last cycle (to enable forwarding to a subsequent instruction.)

Read after write example 1

Example 4-13 is a load of a single-precision data item followed by an arithmetic operation on that data.

Example 4-13 Read after write example 1

```
        FLDS    S4, [r0]
        FADDS   S5, S4, S3
```

In cycle 4 the data is written from the ARM1020E processor core and forwarded in cycle 5 to the first D stage of the `FADDS`.

Table 4-5 Instruction cycles for example 1

	Instruction cycle number								
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`FLDS`	I	D	E	M	W	-	-	-	-
`FADDS`	-	I	D	D	D	E1	E2	E3	E4

Read after write example 2

Example 4-14 is a load multiple of single-precision data, with a vector FADDS following. The only register shared is the first loaded by the FLDM, and the stall ends after that register has been received by the VFP10 coprocessor.

Example 4-14 Read after write example 2

```
        FLDM    [r2], {s7-s14}
        FADDS   S16, S7, S25
```

In Example 4-14 the LEN field is 3, for a vector length of 4, and the STRIDE field is 0, for +1 striding. This is another example of a RAW hazard case. The operand data referenced by S7 is forwarded to the FADDS in cycle 5.

Table 4-6 Instruction cycles for example 2

	Instruction cycle number								
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
FLDM	I	D	E	M	W	W	W	W	-
FADDS	-	I	D	D	D	E1	E2	E3	E4

Example 3

Example 4-15 is a vector FMULS of length 4 (LEN is set to 3, with STRIDE set to 0) with a store of source register S25.

Example 4-15 Vector FMULS example

```
        FMULS    S8,S16,S24
        FSTS     S25,[r2]
```

The VFP10 coprocessor in non-RunFast mode stalls until the source register has been cleared by the FMULS before allowing the store to begin execution. Register S25 is released in cycle 4, and the FSTS moves from Decode (D) to Execute (E) in the next cycle.

Table 4-7 Instruction cycles for short vector MULS example

	Instruction cycle number								
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
FMULS	I	D	E1	E1	E1	E1	E2	E3	E4
FSTS	-	I	D	D	D	E	M	W	-

Example 4 load of all source registers

Example 4-16 is a short vector FMULS of length 4 (LEN is set to 3, with STRIDE set to 0) with a load of all of the source registers.

Example 4-16 Vector FMULS example

```
        FMULS   S8,S16,S24
        FLDMS   [r2], {S16-S27}
```

The VFP10 coprocessor in non-RunFast mode stalls until all the source registers, S16-S19 and S24-S27, have been cleared by the FMULS before allowing the load to begin execution. Table 4-8 shows the instruction cycles for the short vector FMULS example

Table 4-8 Instruction cycles for short vector FMULS example

	Instruction cycle number								
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
FMULS	I	D	E1	E1	E1	E1	E2	E3	E4
FLDMS	-	I	D	D	D	D	E	M	W

Hazards in RunFast mode

In RunFast mode source registers are locked only for vectors that are larger than half-vectors, that is, when the vector length exceeds 4 for single-precision operations or 2 for double-precision operations. When the vectors are sufficiently short, no hazards exist involving the source registers. Repeating the last two examples above illustrates the advantage of RunFast mode for these cases.

Vector FMULS example in RunFast mode

Example 4-17 is a vector `FMULS` of length 4 (LEN is set to 3, with STRIDE set to 0) with a store of one of the last source registers.

Example 4-17 Vector FMULS RunFast mode example

```
        FMULS   S8,S16,S24
        FSTS    S25,[r2]
```

The VFP10 coprocessor in RunFast mode does not stall the store, which can begin execution in the next cycle. Table 4-9 shows the instruction cycle for Example 4-17 in RunFast mode.

Table 4-9 Instruction cycles for example in Run Fast mode

	Instruction cycle number								
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`FMULS`	I	D	E1	E1	E1	E1	E2	E3	E4
`FSTS`	-	I	D	E	M	W	W	-	-

In Example 4-18 a vector `FMULS` of length 4 (LEN is set to 3, with STRIDE set to 0) with a load of all of the source registers. The VFP10 coprocessor in RunFast mode does not stall until the `FLDM` operation.

Example 4-18 Vector FMULS with a load of all registers in RunFast mode

```
        FMULS   S8, S16, S24
        FLDM    [r2], {S16-S27}
```

Table 4-10 shows the instruction cycle progression for Example 4-18.

Table 4-10 Instruction cycles for vector MULS example

	Instruction cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
`FMULS`	I	D	E1	E1	E1	E1	E2	E3	E4	-
`FLDM`	-	I	D	E	M	W	W	W	W	W

4.5.4 Resource hazards

The VFP10 coprocessor has three pipelines:

- the L/S pipeline
- the FMAC pipeline
- the DS pipeline.

The L/S pipeline is completely separate from the other two, and no resource hazards exist between arithmetic instructions and data transfer instructions. However, the first E1 stage instruction is shared between the FMAC and the DS pipelines, creating a resource stall for a short vector CDP operation for subsequent CDP operations. Resource stalls in the VFP are possible in the following cases:

- a data transfer operation following an incomplete data transfer operation. Each data transfer may be stalled by the core due to unavailable data, for example, memory latency or a cache miss.
- an arithmetic operation following either a short vector arithmetic operation or a double-precision multiply or multiply-accumulate operation. The latency on double-precision multiply and multiply-accumulate operations is 2 cycles, causing a single cycle stall for immediately following arithmetic operations.
- a divide or square root will stall the DS pipeline for 13 or 27 cycles, for single-precision or double-precision operations, respectively. A subsequent divide or square root operation will stall until this number of cycles has passed.

4.5.5 Resource hazard examples

The following examples illustrate the resource hazards present in the VFP10 coprocessor.

Load multiple, single load followed by FADDS

In Example 4-19 a load multiple is followed by a single `FMULS` and a `FADDS`. The single load stalls the VFP10 coprocessor and the ARM core until the load multiple is completed. The `FADDS` is stalled in the core as a result.

Example 4-19 Load multiple followed by a single FMULS and FADDS

```
FLDM    [r2],{S8-S12}
FLDS    [r4] S16
FADDS   S2, S3, S4
```

Table 4-11 shows the pipeline stages for the 3 instructions in Example 4-19 on page 4-19.

Table 4-11 Instruction cycles for example 2

	Instruction cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
`FLDM`	I	D	E	M	W	W	W	-	-	-
`FLDS`	-	I	D	D	D	E	M	W	-	-
`FADDS`	-	-	I	I	I	D	E1	E2	E3	E4

Load multiple, vector FMULS followed by scalar FADDS

In Example 4-20 a load multiple is followed by a vector `FMULS` (assume LEN is set to 3 and STRIDE is set to 0), followed by a scalar `FADDS`. No register conflicts exist between the `FLDM` and the `FMULS`. Notice that the destination of the `FADDS` is in bank 0, forcing scalar operation.

Example 4-20 Load multiple, vector FMULS followed by scalar FADDS

```
FLDM    [r2], {S8-S12}
FMULS   S16, S24, S4
FADDS   S1, S20, S21
```

Table 4-12 shows the pipeline stages for the 3 instructions in Example 4-20.

Table 4-12 Pipeline stages for load multiple, vector MULS, scalar FADDS

	Instruction cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`FLDM`	I	D	E	M	W	W	W	-	-	-	
`FMULS`	-	I	D	E1	E1	E1	E1	E2	E3	E4	-
`FADDS`	-	-	I	D	D	D	D	E1	E2	E3	E4

4.6 Parallel execution of operations

An instruction may begin execution when no register or resource conflicts exist (including read-after-read hazards) and the respective pipeline or pipelines (Load/Store or CDP) are not executing a vector or multiple operation with pending iterations. The following further outlines these rules.

A load or store operation begins execution if:

- No data hazards exist with any currently executing operations (including read-after-read hazards)
- The LS pipeline is not currently stalled by the ARM or busy with a load or store multiple.

A CDP may be issued to the FMAC pipeline if:

- No data hazards exist with any currently executing operations (including read-after-read hazards)
- The arithmetic pipeline is available (it may be unavailable if a vector CDP is executing or a double multiply is in the first cycle of the multiply operation)
- No vector operation is currently executing in either the arithmetic or DS pipeline.

A divide or square root instruction may be issued to the DS pipeline if:

- No data hazards exist (including read-after-read hazards)
- The DS pipeline is available (no current divide or square root is executing in the DS pipeline E1 stage)
- No vector operation is executing in the arithmetic pipeline.

4.6.1 An example of parallel execution

The VFP10 coprocessor is capable of execution in each of the three pipelines independently of the others and without blocking issue or writeback from any pipeline. Example 4-21 on page 4-22 shows a case of the VFP10 coprocessor using the 3 pipelines in parallel:

- a load multiple in the L/S pipeline
- a short vector add in the FMAC pipeline
- a divide in the DS pipeline.

Assume the LEN field in the FPSCR is set to 3, for a vector length of 4, and the STRIDE field is set to 0, for a stride of +1.

Example 4-21 Parallel execution in all 3 pipelines

```
FLDM    [r4], {S4-S13}
FDIVS   S0, S1, S2
FADDS   S16, S20, S24
```

Table 4-13 shows the pipeline progression for the 3 instructions

Table 4-13 Parallel execution in three pipelines

	Instruction cycle number															
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
`FLDM`	I	D	E	M	W	W	W	W	W	-	-	-	-	-	-	-
`FDIVS`	-	I	D	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E2
`FADDS`	-	-	I	D	E1	E1	E1	E1	E2	E3	E4	-	-	-	-	-

In Example 4-21, no data hazards exist between any of the three instructions. The load multiple is able to begin execution immediately, and data is transferred to the register file beginning in cycle 5. The `FDIVS` is a scalar operation (the destination is in Bank 0) and requires one cycle in the FMAC E1 stage. If the divide was a short vector operation the `FADDS` would not begin execution until the last iteration had passed the E1 stage. The `FADDS` is a short vector operation and requires the FMAC E1 stage for cycles 5-8. In cycle 9 another arithmetic operation could begin provided it was not a divide or square root. This instruction stalls only if it requires the destination register of the divide or any of the destination registers of the last 3 iterations of the `FADDS` (the result of the first iteration is available from the register file in cycle 9, and all of the registers updated by the `FLDM` are valid).

4.7 Execution timing

These VFP10 coprocessor instruction timing computations are provided as a guide, and not a substitute for running the code on a system or cycle-accurate simulator. Also, the execution of VFP10 coprocessor instructions is also dependent on the execution of the instruction in the ARM1020E macrocell, and stall and memory access issues will directly impact performance of VFP10 coprocessor code. See the *ARM1020E Technical Reference Manual* for information on instruction timing within the ARM core.

In Table 4-14 throughput is defined as the cycle after issue in which another instruction, without a data hazard, could begin execution. Instruction latency is the number of cycles after which the data is available for another operation. Forwarding reduces the latency by one cycle for dependent operations on floating-point data when the destination precision of the first or the precision of the load data is the same as the source precision of the second.

Note

`FMXR` and `FMRX` are serializing instructions. The latency depends on the register transferred and the current activity in the VFP10 coprocessor when the instruction is issued.

Table 4-14 shows the throughput and latency for all CDP operations in the VFP10 coprocessor.

Table 4-14 Throughput and latency cycle timings for VFP10 CDP operations

Instructions	Single-precision		Double-precision	
	Throughput	Latency	Throughput	Latency
`FADD`, `FSUB`, `FABS`, `FNEG`, `FCVT`, `FCPY`	1	4	1	4
`FCMP`, `FCMPE`, `FCMPZ`, `FCMPEZ`	1	4	1	4
`FSITO`, `FUITO`, `FTOSI`, `FTOUI`, `FTOUIZ`, `FTOSIZ`	1	4	1	4
`FMUL`, `FNMUL`	1	4	2	5
`FMAC`, `FNMAC`, `FMSC`, `FNMSC`	1	4	2	5
`FDIV`, `FSQRT`	14	17	28	31
`FLD`	1[a]	2	1	2

Table 4-14 Throughput and latency cycle timings for VFP10 CDP operations (continued)

Instructions	Single-precision		Double-precision	
	Throughput	Latency	Throughput	Latency
`FST`	1[a]	1	1	1
`FLDM`	X[b]	X[b]+2	N	N+2
`FSTM`	X[b]	X[b]+1	N	N+1
`FMSTAT`	1	2	-	-
`FMSR`	1	2	-	-
`FMDHR/DLR`	-	-	1	2
`FMRS`	1	1	-	-
`FMRDH/RDL`	1	1	-	-
`FMXR`[c]	1	2	-	-
`FMRX`[c]	1	1	-	-

a. Two single-precision data values can be loaded or stored in a single cycle.

b. The number of cycles represented by X is ceiling (N/2). The data for load and store multiples maybe in most cases used when it is loaded or stored, not waiting until the instruction is completed.

c. FMXR and FMRX are serializing instructions. The latency will depend on the register transferred and the current activity in the VFP10 when the instruction is issued.

Chapter 5
Exception Handling

This chapter describes VFP10 coprocessor exception processing. It contains the following sections:

- *About exception processing* on page 5-2
- *Support code* on page 5-3
- *Invalid operation* on page 5-13
- *Division by zero* on page 5-16
- *Overflow* on page 5-17
- *Underflow* on page 5-19
- *Inexact result* on page 5-21
- *Arithmetic exceptions* on page 5-23.

5.1 About exception processing

The VFP10 coprocessor processes exceptions imprecisely with respect to both the ARM state and VFP10 state. Exceptions are detected after the instruction has passed the point in the ARM for exception processing. The VFP10 coprocessor enters an exceptional state after the exceptional operation has been detected, and signals the presence of an exception by refusing to accept a subsequent VFP10 coprocessor instruction. The instruction that triggers the exception processing is said to *bounce* to the ARM1020E. The instruction that bounces is always a subsequent instruction but not necessarily the instruction immediately following the exceptional instruction. In many cases, a VFP10 coprocessor instruction following the exceptional instruction will bounce, although, depending on the nature of the exceptional instruction, it can be several instructions following before a bounce occurs.

VFP10 generated exceptions are only possible on arithmetic operations and not on data transfer operations. Another class of instructions, involving copying data between VFP10 coprocessor registers, is considered to be non-arithmetic and is not capable of producing exceptions. These are `FCPY`, `FABS`, and `FNEG`. The `FCPY` instruction can be used to copy SNaNs between VFP10 coprocessor registers, without setting the IOC bit, and subnormals between VFP10 coprocessor registers, without flushing the subnormal to positive zero or taking an input exception. The `FABS` and `FNEG` instructions maybe viewed as copying with sign changing, and behave in the same manner as `FCPY` with regards to exceptions.

In both non-RunFast and RunFast modes the VFP10 coprocessor, with support code, processes exceptions according to the IEEE 754 specification, including the calling of user trap handlers with IEEE 754 specified intermediate operands.

Note

In RunFast mode the VFP10 coprocessor modifies the definition of the underflow exception flag to provide additional information in cases in which the result has been flushed to positive zero.

Complete descriptions of each of the exception flags and their bounce characteristics are given in sections *Invalid operation* on page 5-13 to *Arithmetic exceptions* on page 5-23.

5.2 Support code

The VFP10 coprocessor provides floating-point functionality through a combination of hardware and software support. Floating-point instructions are normally executed by the VFP hardware. However, the VFP10 coprocessor may use the interface signals between it and the ARM1020E core to refuse to accept a floating-point instruction, causing the ARM1020E undefined instruction exception. This is known as *bouncing* the instruction. When an instruction is bounced, software installed on the ARM1020E undefined instruction vector determines why the VFP10 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 that perform floating-point arithmetic functions
- a set of exception handlers that process exception conditions.

There are two main reasons for bouncing an instruction:

- potential floating-point arithmetic exceptions
- illegal instructions.

See *AFS Firmware Suite Version 1.3 Reference Guide* for details of support code.

5.2.1 Bounced instructions

When a bounce occurs, the hardware sets the EX bit in the FPEXC register and loads FPINST with a copy of the potentially exceptional instruction. This condition in the VFP10 coprocessor is referred to as the *exceptional state*. The instruction that is bounced as a result of the exceptional state is referred to as the *trigger* instruction. Any trigger instruction currently in the VFP10 coprocessor Decode (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 an enabled floating-point exception but also occurs in some rare cases when there is no floating-point exception present, only a potential for an exception detected in the E1 stage.

The remedial action is performed as follows:

1. The support code starts with reading the FPEXC register. If the EX bit (FPEXC[31]) is set, a potential exception is present. If not, an illegal instruction is detected. See *Illegal instructions* on page 5-6.

2. The FPEXC register is written to clear the EX bit (failure to do this can result in an infinite loop of exception traps when the support code next accesses the VFP hardware).

3. The FPINST register is read to determine the instruction that caused the potential exception.

4. The support code then decodes the instruction in the FPINST register, reads its operands (including implicit ones such as the FPSCR rounding mode and vector length), executed the operation, and determines whether a floating-point exception occurred.

5. If no floating-point exception occurred, the support code writes the correct result of the operation, and sets any appropriate status flags in the FPSCR.

   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 computes the IEEE-754 specified intermediate result, if required, and calls the user-provided trap handler for that exception. The user's trap handler can provide a result for the instruction and continue program execution, generate a signal or message to the operating system or the user, or simply terminate the program.

6. If the potentially exceptional instruction specified a short 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 *Exception processing for CDP short vector instructions* on page 5-8 for more details.

7. If the FPv2 bit is set in the FPEXC (FPEXC Bit [28]), the FPINST2 register contains another VFP instruction that has been issued between the potentially exceptional instruction and the trigger instruction. This instruction is executed by the support code in the same manner described above. See *Instruction word registers (FPINST and FPINST2)* on page 318 for more on FPINST2.

   Steps 1-7 imply that the support code must be capable of performing steps 4 and 5 for any operation/operands combination, not just for those combinations that the VFP10 hardware treats as potentially exceptional.

8. 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 executing.

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 coprocessor. 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 use the VFP10 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.

5.3 Illegal instructions

If there is not a 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 EX bit in the FPEXC (FPEXC[31]) is 0. 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 occur 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 instructions are architecturally required to be treated as illegal instructions:

- instructions with opcode bit combinations defined as Reserved in the Architecture 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 may 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 by some implementations of the VFP, 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 short vector operation that has a combination of precision, length, and stride that would cause the vector to wrap around more than once (more than one access to the same register). In this implementation this case is bounced.
- A short 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`.

5.4 Determination of the trigger instruction

The ARM1020E coprocessor interface specifies an exceptional instruction that bounces to support code must signal on a subsequent coprocessor instruction. This is known as *imprecise exception handling* and has the characteristic that the user state of the VFP10 coprocessor as well as the ARM and any other coprocessors or processors available when the exception is processed may not represent the state at the time of the exceptional instruction execution or the state that is expected in a serial execution of the code stream. The VFP10 coprocessor parallel execution of Load/Store operations and CDP operations allows for the VFP10 coprocessor and ARM1020E core register files and memory to be modified outside program order in normal operation.

The determination of what is the trigger instruction is a matter of instruction issue timing. A CDP instruction is not determined potentially exceptional until the E1 Execute cycle. Another VFP10 instruction issued immediately following this instruction will have completed processing by the ARM1020E and could no longer cause an undefined instruction exception to be taken. In this case, this instruction is in what is referred to as the *pre-trigger slot* and must be retained for the support code in the FPINST2 register.

When the exceptional condition is detected on a short vector operation the rules change. Because the short vector operation appears to the ARM1020E as a single-cycle operation, other VFP10 instructions can be issued, execute and retire before the short vector operation retires. Several rules determine what is the trigger instruction:

- accessing the exception registers (FPEXC, FPINST, and FPINST2) or FPSID, is not a trigger instruction in a Privileged mode
- any instruction which is stalled in the Decode stage due to register or resource hazard id the trigger instruction
- the first instruction issued at least two cycles after the exceptional condition has been detected is the trigger instruction
- a load or store instruction which reaches the Execute stage is not the trigger instruction (there can be several of these if the short vector is sufficiently long and the exception is detected on a later iteration).

5.4.1 Exception processing for 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 might exist in the following situation:

- If there is not a floating-point instruction (CDP or Load/Store) in the VFP10 Decode stage, the VFP10 coprocessor waits until one is issued. The next trigger instruction is bounced.
- If there is a trigger instruction in the VFP10 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 that was in the VFP10 Decode stage is retried by the ARM core when the ARM core returns from exception processing.

5.4.2 Exception processing for CDP short vector instructions

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

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

1. The EX bit in the FPEXC register is set.
2. The FPINST register is loaded with the operation instruction word.
3. The source and destination register addresses are modified to point to the source and destination registers of the offending iteration.
4. The VECITR field is written with the coded number of the offending iteration.

5.4.3 Examples of exception detection for short vector instructions

In Example 5-1 on page 5-9 to Example 5-4 on page 5-11 code fragments illustrate the exception detection mechanism of the VFP10 coprocessor for short vector operations. The LEN field in the FPSCR is set to `0b011`, for a vector length of four.

In Example 5-1, assume the LEN field in the FPSCR is set to 0 (scalar operations). The `FLDMD` (Inst A) issues and retires regardless of the exceptional status of the `FMULD` in Inst B. The `FSTMD` in Inst C is stalled waiting on the `FLDMD` to complete, and will be the trigger instruction, and retried upon the return from exception processing. The FPINST register contains the `FMULD` (with the condition codes set to AL) and the FPINST2 register is invalid and FPV2 is set to 0.

Example 5-1 FLDMD completes regardless of a subsequent exceptional CDP

```
FLDMD R2, {D0-D5} ; Inst A load multiple of 6 double-precision words
FMULD D8, D12, D8 ; Inst B scalar double-precision multiply
FSTMD R3, {D6-D7} ; Inst C store multiple of 2 double-precision words
FMULS S0, S1, S1  ; Inst D scalar single-precision multiply S0 = S1*S1
```

In Example 5-2, the `FMULD` is a vector operation of length 4 (LEN set to 3 in the FPSCR) and a potential underflow exception is detected on the second iteration. The load in Inst B and the store in Inst C both issue before the exception is detected on Inst A. (A double multiply requires 2 cycles in the E1 stage, with exceptions detected in the first of the two cycles. The exception on the 4th and last iteration is detected in the 3rd cycle after the issue of the `FMULD` to the E1 stage.) The first load issues in the 2nd cycle after the `FMULD` and requires one cycle. The following store issues in the 3rd cycle after the `FMULD` but before the exception is detected, and is allowed to complete and retire. The `FLDS` (Inst D) is stalled in the D stage due to a resource conflict with Inst C and is the trigger instruction. It will be retried upon the return from exception processing. FPINST2 is invalid and FPV2 is set to 0.

Example 5-2 Exceptional vector CDP followed by several load/store operations

```
FMULD D8, D12, D8   ; Inst A short vector double-precision multiply of len 4
FLDDD D0, {R5}      ; Inst B load of a single double-precision data
FSTMS R3, {S2-S9}   ; Inst C store multiple of 8 single-precision data
FLDS S8, {R9}       ; Inst D load of a single double-precision data
```

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

```
EX:         1    (Signaling the VFP10 coprocessor is exceptional)
EN:         1
VECITR:   001    (VECITR reports 2 iterations remain after exceptional iteration
IDC:        0
INV:        0
UFC:        1    (The exception detected is a potential underflow)
OFC :       0
IOC:        0
```

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

```
Op:      0100 (multiply)
Fd/D:    1001/0 (Destination is D9 for the exceptional iteration)
Fn/N:    1001/0 (Fn source is D9 for the exceptional iteration)
```

```
Fm/M:    1101/0 (Fm source is D13 for the exceptional iteration)
CpID:    1011 (operation is double-precision)
```

In Example 5-3 Inst A is a scalar operation (the destination is in bank 0) and has a potential invalid exception. Inst B has progressed into the D stage and is captured into the FPINST2 register (with the conditional bits forced to AL) and is not the trigger. Inst C is 2 cycles behind the exceptional instruction and is the trigger instruction. It will be retried upon the return from exception processing.

Example 5-3 Exceptional CDP with CDP in the pre-trigger slot

```
FADDS S0, S1, S2         ; Inst A scalar single-precision add
FADDS S3, S4, S5         ; Inst B scalar single-precision add
FMULS S12, S16, S16      ; Inst C short vector single-precision multiply
```

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

```
EX:         1   (Signaling the VFP10 coprocessor is exceptional)
EN:         1
FPV2:       1   (FPINST2 contains a valid instruction)
VECITR:   111   (no iterations remaining after exceptional iteration)
IDC:        0
INV:        0
UFC:        0
OFC:        0
IOC:        1   (exception is a potential invalid)
```

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

```
Op:      0110   (add)
Fd/D:    0000/0 (Destination is S0)
Fn/N:    0000/1 (Fn source is S1)
Fm/M:    0001/0 (Fm source is S2)
CpID:    1010   (operation is single-precision)
```

FPINST2 contains the instruction word for the FADDS in Inst B.

In Example 5-4 on page 5-11 an exceptional short vector of length 4 (LEN set to 3) with a potential overflow exception in the first iteration is followed by a CDP with a register conflict. The second CDP (Inst B) is stalled in the D stage waiting on Inst A to exit the E1 stage. Inst B is the trigger instruction and will be retried upon the return from exception processing. FPINST2 is invalid and FPV2 is set to 0.

Example 5-4 Exceptional vector CDP followed by scalar CDP with register conflict

```
FABSD D4, D4, D12   ;Inst A short vector double-precision absolute value of
                    ; length 4
FMACS S0, S3, S2    ;Inst B scalar single-precision mac
```

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

```
EX:      1   (Signaling the VFP10 coprocessor is exceptional)
EN:      1
FPV2:    0   (FPINST2 does not contain a valid instruction
VECITR:010   (VECITR reports 3 iterations remain)
IDC:     0
INV:     0
UFC:     0
OFC:     1   (The exception detected is a potential overflow)
IOC:     0
```

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

```
Op:     1111    (extend)
Fd/D:   0100/0  (Destination is D4)
Fn/N:   0000/1  (Fn specifies FABS instruction)
Fm/M:   1100/0  (Fm source is D12)
CpID:   1011    (operation is double-precision
```

FPINST2 contains invalid data.

5.5 Input subnormal

The IDC bit in the FPSCR (FPSCR[7]) is set whenever an input operand is a subnormal and the operation is not a floating-point to integer conversion. The behavior of the VFP10 coprocessor with a subnormal input operand is a function of the FZ bit in the FPSCR. If the FZ bit is 0, the VFP10 coprocessor bounces on the presence of an input subnormal. If the FZ bit is 1, the IDE bit in the FPSCR (FPSCR[15]) determines whether a bounce occurs.

5.5.1 Exception enabled

If the IDE bit in the FPSCR (FPSCR[15]) is set, the EX bit in the FPEXC ([31]) and the IDC bit in the FPSCR (bit [7]) is set. The source and destination registers for the instruction will be valid in the VFP10 coprocessor register file.

5.5.2 Exception disabled

If the VFP10 coprocessor is not in FTZ mode, the result of the operation, with the input subnormal replaced with a positive zero, is completed and written to the register file. All appropriate status bits in the FPSCR are set accordingly.

5.6 Invalid operation

An operation is *invalid* if there does not exist a representation for the result, or if the result is not defined. An example is adding a positive infinity to a negative infinity, or trying to represent a floating-point number greater than 2^{32} as a 32-bit integer. The VFP10 coprocessor in RunFast mode handles all invalid cases in hardware without support code intervention. In non-RunFast mode, only cases involving signaling NaNs require support code intervention.

Table 5-1 shows the operand combinations that produce invalid operation exceptions. In addition to the conditions in Table 5-1, 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 3-1).

Table 5-1 Possible IEEE 754 invalid operation exceptions

Instruction	Invalid operation exceptions
`FMAC/FNMAC`	Any of the conditions that 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 that 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 In FTZ mode a subnormal input is treated as a positive zero for INVALID exception determination.
`FMUL/FNMUL`	0 * ± infinity or ± infinity*0
`FSQRT`	Source is < 0
`FFTOUI`	Rounded result would lie outside the range $0 <= \text{result} < 2^{32}$
`FFTOSI`	Rounded result would lie outside the range $-2^{31} <= \text{result} < 2^{31}$

5.6.1 Exception enabled

The VFP10 coprocessor detects most invalid conditions correctly but some are detected pessimistically. The pessimistically detected cases are:

- `FTOUI` with a negative input. A small negative input may round to a zero, which is not a invalid condition
- Float-to-integer with a maximum exponent for the destination precision and any rounding mode other than RZ. The impact of rounding is unknown in the E1 stage
- A FMAC-family operation with an infinity for the A operand and a potential product overflow to an infinity that can result in an invalid condition.

When the VFP10 coprocessor detects a pessimistic case, the EX bit in the FPEXC ([31]) and the IOC bit in the FPEXC (bit [0]) will be set. The IOC bit in the FPSCR doesnot have been set by the hardware, and must be set by the support code before calling the user-provided trap handler.

The support code determines the exception status of the pessimistically bounced cases, and if an invalid condition exists, the invalid exception trap handler you created is called. The source and destination registers for the instruction will be valid in the VFP10 coprocessor register file.

5.6.2 Exception disabled

If the IOE bit is clear, the VFP10 coprocessor processes all invalid cases according to the IEEE-754 specification. The value written into the destination register for all operations except integer conversion operations will be the default NaN.

Conversion of a floating-point value that is outside the range of the destination integer is an invalid condition rather than an overflow condition. When an invalid condition exists for a floating-point to integer conversion, the VFP10 coprocessor delivers a default result to the destination register and sets the IOC bit in the FPSCR. The default results are given below in Table 5-2 on page 5-15.

Note

A negative input to an unsigned conversion, which does not round to a true zero in the conversion process, will set the IOC bit in the FPEXC.

Table 5-2Table 5-2

Table 5-2 Default results for positive invalid inputs

Input value	FTUOI(Z)		FTOSI(Z)	
	Result	IOC set?	Result	IOC set?
$x \geq 2^{32}$	`FFFFFFFF`	Yes	`7FFFFFFF`	Yes
$2^{31} \leq x < 2^{31}$	Integer	No	`7FFFFFFF`	Yes
$0 \leq x < 2^{31}$	Integer	No	Integer	No
$0 > x \geq -2^{31}$	`00000000`	Yes	Integer	No
$x < -2^{31}$	`00000000`	Yes	`80000000`	Yes

5.7 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. In FTZ mode a subnormal input is treated as a positive zero for divide-by-zero determination. What happens depends on whether the invalid operation exception is enabled.

5.7.1 Exception enabled

If the DZE bit of the FPSCR (FPSCR[9]) is 1, the divide-by-zero user trap handler is called. The source and destination registers for the instruction will be valid in the VFP10 coprocessor register file.

5.7.2 Exception disabled

A correctly signed infinity is written to the destination register and the DZC bit is set in the FPSCR (FPSCR[1]).

5.8 Overflow

When OFE is set in the FPSCR (FPSCR[10]) 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.

5.8.1 Exception enabled

The VFP10 coprocessor detects most overflow conditions conclusively but some are detected pessimistically. Specifically, when the initial computation of the result exponent is the maximum exponent or one less than the maximum exponent of the destination precision, the possibility of overflow due to mantissa overflow or rounding exists, but cannot be known in the first Execute stage. The VFP10 coprocessor bounces on such cases and uses the support code to determine the exceptional status of the operation. If it does not overflow, the support code writes the computed result to the destination register and returns without setting OFC (FPSCR[2]). If it does overflow, the intermediate result is written to the destination register, OFC is set, and the user overflow trap handler is called.

When the VFP10 coprocessor detects a pessimistic case, the EX bit in the FPEXC ([31]) and the OFC bit in the FPEXC (bit [2]) will be set. The OFC bit in the FPSCR will not have been set by the hardware, and must be set by the support code before calling the user's trap handler. The source and destination registers for the instruction will be valid in the VFP10 coprocessor register file. See *Arithmetic exceptions* on page 5-23 for the conditions which will cause an overflow bounce.

5.8.2 Exception disabled

A correctly signed infinity or largest finite number for the destination precision is written to the destination register according to Table 5-3. The OFC bit and the IXC bit are set in theFPSCR.

Table 5-3 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.

5.9 Underflow

Underflow is detected pessimistically in non-RunFast mode. 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 VFP10 coprocessor is in Flush-to-zero mode.

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

5.9.1 Exception enabled

The VFP10 coprocessor detects most underflow conditions conclusively but some are detected pessimistically. Specifically, when the initial computation of the result exponent is below a threshold for the destination precision, the possibility of underflow due to massive cancellation exists, but cannot be known in the first Execute stage. The VFP10 coprocessor will bounce on such cases and utilize the support code to determine the exceptional status of the operation. If it does not underflow, either catastrophically or to a subnormal result, the support code will write the computed result to the destination register and return without setting UFC. If it does underflow, regardless of any accuracy loss, the intermediate result will be written to the destination register, UFC will be set, and the trap handler you created will be called. 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 user underflow trap handler is called.

When the VFP10 coprocessor detects a pessimistic case, the EX bit in the FPEXC ([31]) and the UFC bit in the FPEXC (bit [3]) will be set. The UFC bit in the FPSCR will not have been set by the hardware, and must be set by the support code before calling the user's trap handler. The source and destination registers for the instruction will be valid in the VFP10 coprocessor register file. See section *Arithmetic exceptions* on page 5-23 for the conditions that will cause an underflow bounce.

5.9.2 Exception disabled

When the FZ bit in the FPSCR is not set, the VFP10 coprocessor will bounce on potential underflow cases in the same fashion as detailed above for the exception enabled case. The correct result will be written to the destination register, and any exception status bits set accordingly.

When the FZ bit in the FPSCR is set, the VFP10 coprocessor will make the determination of underflow before rounding and flush any result that underflows, returning a positive zero to the destination register and setting the UFC and IXC bits in the FPSCR.

Note

The determination of an underflow condition is made before rounding rather than after. This can result in an intermediate value, with the minimum exponent for the destination precision (00 for single-precision and 000 for double-precision), fraction of all ones, and a round increment, to be flushed to zero rather than the minimum normal value to be returned. If the intermediate value was the minimum normal value before the underflow condition test is made, it will not be flushed to zero.

5.10 Inexact result

Floating-point arithmetic inherently has limited precision and typically the result of an arithmetic operation on two floating-point values has more significant bits than the destination register can contain. 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 not equal to the computed result before rounding
- an untrapped overflow exception occurs
- an untrapped underflow exception occurs, and there is loss of accuracy.

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. Enabling the inexact exception in the FPSCR can significantly reduce the performance of the VFP10 coprocessor.

The VFP10 coprocessor 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 0).

5.10.1 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 user trap handler.

Note

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

5.10.2 Exception disabled

If the IXE bit in the FPSCR is 0, the VFP10 coprocessor writes the result to the destination register and sets the IXC bit in the FPSCR.

5.11 Input exceptions

The VFP10 coprocessor processes most input operands completely in hardware. However, the hardware is incapable of processing some operands and will bounce to support code to process the instruction. The inputs which are bounced are:

- NaNs operands, when the DN mode is not enabled
- subnormal operands, when the FTZ mode is not enabled.

5.12 Arithmetic exceptions

This section details the conditions under which the VFP10 coprocessor will bounce an arithmetic operation pessimistically. It is the task of the support code to determine the actual exception status of the instruction, and return either the result and appropriate exception status bits, or the intermediate result and a call to the user's trap handler.

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 -121
- FMAC/FMSC/FNMAC/FNMSC on page -138
- FDIV on page -138
- FSQRT on page -145
- FCPY/FABS/FNEG on page -146
- FCVTDS/FCVTSD on page -146
- FUITO/FSITO on page -147
- FTOUI/FTOUIZ/FTOSI/FTOSIZ on page -147.

5.12.1 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 5-4 specifies how this division is made. + refers to a positive operand and - refers to a negative operand.

Table 5-4 LSA and USA determination

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

Table 5-4 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 5-26).

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) to reduce the number of logic terms. Table 5-5 lists the USA and LSA values and conditions. All exponent values are in hexadecimal, 11 bits for double-precision, and 8 bits for single-precision.

Table 5-5 USA and LSA values and conditions

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
>7FF	-	DP Ovfl	-	Bounce
7FF	-	DP Ovfl, NaN, Inf	-	Bounce
7FE	-	DP Ovfl Det	-	Bounce
7FD	-	DP Ovfl Det	-	Bounce
7FC	-	DP Norm	-	Norm
>47F	>FF	SP Ovfl	Bounce	Norm
47F	FF	SP NaN, Inf	Bounce	Norm
47E	FE	SP Ovfl Det	Bounce	Norm
47D	FD	SP Ovfl Det	Bounce	Norm
47C	FC	SP Norm	Norm	Norm

Table 5-5 USA and LSA values and conditions (continued)

Double-precision	Single-precision	Value	Condition (non-FZ mode)	
			SP	DP
`3FF`	`7F`	e=0 bias value	Norm	Norm
`3A0`	`20`	SP Norm (LSA)	MIN (USA)	Norm
`39F`	`1F`	SP Unfl (USA)	Bounce (USA) Norm (LSA)	Norm
`381`	`01`	SP Norm (LSA)	MIN (LSA)	Norm
`380`	`00`	SP subnormal	Bounce	Norm
`<380`	`<00`	SP Unfl	Bounce	Norm
`040`	-	DP Norm (USA)	-	Norm (LSA) MIN (USA)
`03F`	-	DP Unfl (USA)	-	Norm (LSA) Bounce (USA)
`001`	-	DP Norm (LSA)	-	MIN (LSA) Bounce (USA)
`000`	-	DP subnormal	-	Bounce
`<000`	-	DP Unfl	-	Bounce

5.12.2 FMUL/FNMUL

The determination for potential exceptional conditions is made based on the initial product exponent, the sum of the multiplicand and multiplier exponents. *FMUL family bounce and exceptional thresholds* on page 5-26 lists the VFP10 coprocessor 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 5-23.

.

Table 5-6 FMUL family bounce and exceptional thresholds

Double-precision	Single-precision	Value	Condition (non-RunFast 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
`>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 subnormal	Bounce	Norm
`<380`	`<00`	SP Unfl	Bounce	Norm
`001`	-	DP Norm	-	Norm
`000`	-	DP subnormal	-	Bounce
`<000`	-	DP Unfl	-	Bounce

5.12.3 FMAC/FMSC/FNMAC/FNMSC

The FMAC family of operations adds to the potential overflow range by generating final values 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.

5.12.4 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 5-23 (again, to reduce logic complexity). The underflow ranges include the minimum normal exponent (`0x01` for single-precision and `0x001` for double-precision). The complete table is shown in Table 5-7.

Table 5-7 FDIV bounce and exceptional thresholds

Double-precision	Single-precision	Value	Condition (non-RunFast 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
`>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
`382`	`02`	SP Norm	Norm	Norm
`381`	`01`	SP Norm	Bounce	Norm
`380`	`00`	SP subnormal	Bounce	Norm

Table 5-7 FDIV bounce and exceptional thresholds (continued)

Double-precision	Single-precision	Value	Condition (non-RunFast mode)	
			SP	DP
<380	<00	SP Unfl	Bounce	Norm
002	-	DP Norm	-	Norm
001	-	DP Norm	-	Bounce
000	-	DP subnormal	-	Bounce
<000	-	DP Unfl	-	Bounce

5.12.5 FSQRT

It is not possible for `FSQRT` to overflow or underflow.

5.12.6 FCPY/FABS/FNEG

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

5.12.7 FCVTDS/FCVTSD

Only the `FCVTSD` operation is capable of overflow or underflow. Table 5-8 lists the `FCVTSD` bounce conditions. The overflow ranges are the same as the LSA ranges. This is to reduce logic complexity. Table 5-8 lists the `FCVTSD` bounce conditions.

Table 5-8 FCVTSD bounce conditions

DP	Value	Condition (non-RunFast mode) FCVTSD
>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

Table 5-8 FCVTSD bounce conditions (continued)

DP	Value	Condition (non-RunFast mode) FCVTSD
`381`	SP Norm	Norm
`380`	SP subnormal	Bounce
`<380`	SP Unfl	Bounce

5.12.8 FUITO/FSITO

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

5.12.9 FTOUI/FTOUIZ/FTOSI/FTOSIZ

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

Table 5-9 on page 5-30 and Table 5-10 on page 5-32 use the following notation. *Ex* stands for *Exception generated*:

I	Invalid
None	Operation is valid

In the *VFP Response* column:

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.
None	All values are valid

In the *Unsigned results* and *Signed results* column:

N	Round-to-nearest rounding mode.
P	Round-to-Plus-Infinity rounding mode.
M	Round-to-Minus-Infinity rounding mode.
Z	Round-to-Zero mode.

Table 5-9 shows the single-precision float-to-integer bounce range and the results returned for exceptional conditions.

Table 5-9 SP Float-to-integer bounce thresholds and stored results

Float value	Value	Unsigned result	Ex	Signed result	Ex	VFP response
NaN	-	00000000	I	00000000	I	Bounce All
7F800000	+Inf	FFFFFFFF	I	7FFFFFFF	I	Bounce All
7F7FFFFF to 4F800000	+Max Sp to 2^{32}	FFFFFFFF	I	7FFFFFFF	I	Bounce All
4F7FFFFF to 4F000000	$(2^{32} - 2^{8})$ to 2^{31}	FFFFFF00 to 80000000	I	7FFFFFFF	I	Bounce S UnZ
4EFFFFFF to 4E800000	$(2^{31} - 2^{7})$ to 2^{30}	7FFFFF80 to 40000000	V	7FFFFF80 to 40000000	V	Bounce SnZ
4E7FFFFF to 00000000	$(2^{30} - 2^{6})$ to +0	3FFFFFC0 to 00000000	V	3FFFFFC0 to 00000000	V	Bounce None
80000000 to CE7FFFFF	-0 to $(-2^{30} + 2^{6})$	00000000	I	00000000 to C0000040	V	Bounce U

Table 5-9 SP Float-to-integer bounce thresholds and stored results (continued)

Float value	Value	Unsigned result	Ex	Signed result	Ex	VFP response
CE800000 to CEFFFFFF	-2^{30} to $(-2^{31} + 2^{7})$	00000000	I	C0000000 to 80000080	V	Bounce U Bounce U SnZ
CF000000 to FF7FFFFF	-2^{31} to -Max Sp	00000000	I	80000000	I	Bounce All
FF800000	-Inf	00000000	I	80000000	I	Bounce All

Table 5-10 shows the double-precision float-to-integer bounce range and the results returned for exceptional conditions.

Table 5-10 DP Float-to-integer bounce thresholds and stored results

Float value	Value	Unsigned result	Ex	Signed result	Ex	VFP response
NaN	-	00000000	I	00000000	I	Bounce All
7FF00000_00000000	+Inf	FFFFFFFF	I	7FFFFFFF	I	Bounce All
7FEFFFFF_FFFFFFFF	+Max DP					
to	to	FFFFFFFF	I	7FFFFFFF	I	Bounce All
41F00000_00000000	2^{32}					
41EFFFFF_FFFFFFFF	$(2^{32} - 2^{-21})$	FFFFFFFF (NP)	I			
to	to	FFFFFFFF (ZM)	V	7FFFFFFF	I	
41EFFFFF_FFF00000	$(2^{32} - 2^{-1})$					
41EFFFFF_FFEFFFFF	$(2^{32} -2^{-1}-2^{-21})$	FFFFFFFF (P)	I			
to	to	FFFFFFFF (NZM)	V	7FFFFFFF	I	Bounce S UnZ
41EFFFFF_FFE000001	$2^{32} - 2^{-1}+ 2^{-21}$					
41EFFFFF_FFE000000	$2^{32} - 2^{0}$	FFFFFFFF	V			
to	to	to		7FFFFFFF	I	
41E00000_00000000	2^{31}	80000000	V			
41DFFFFF_FFFFFFFF	$(2^{31}- 2^{-22})$	80000000 (NP)	V	7FFFFFFF (NP)	I	
to	to	7FFFFFFF (ZM)	V	7FFFFFFF (ZM)	V	
41DFFFFF_FFE000000	$(2^{32} - 2^{-1})$	to				
41DFFFFF_FFDFFFFF	$2^{32} - 2^{-1}- 2^{-22}$	80000000 (P)		7FFFFFFF (P)	I	
to	to	7FFFFFFF (NZM)	V	7FFFFFFF (NZM)	V	Bounce SnZ
41D00000_FFC00001	$2^{32} - 2^{-1}+ 2^{-21}$	to				
41D00000_FFC00000	$2^{32} - 2^{0}$	7FFFFFFF	V	7FFFFFFF	V	
to	to	to		to		
41D00000_00000000	2^{31}	40000000	V	40000000	V	
41CFFFFF_FFFFFFFF	$(2^{30} - 2^{-23})$	40000000 (NP)	V	40000000 (NP)	V	Bounce none
		3FFFFFFF (ZM)	V	3FFFFFFF (ZM)	V	
to	to	to		to		
00000000_00000000	+0	00000000	V	00000000	V	
80000000_00000000	-0			00000000	V	
to	to	00000000	I	to		Bounce U
C1CFFFFF_FFFFFFFF	$(-2^{30} +2^{-23})$			C00000001 (ZP)	V	
				C00000000 (NM)	V	

Table 5-10 DP Float-to-integer bounce thresholds and stored results (continued)

Float value	Value	Unsigned result	Ex	Signed result	Ex	VFP response
C1D00000_00000000	-2^{30}			C0000000	V	
to	to	00000000	I	to		Bounce U SnZ
				80000001 (ZP)	I	
C1DFFFFF_FFFFFFFF	$(-2^{31} +2^{-22})$			80000000 (NM)	I	
C1E00000_00000000	-2^{-31}	00000000	I	80000000	V	
C1E00000_00000001	$-2^{-31} -2^{-21}$			80000000 (NZP)	V	
to	to	00000000	I	80000000 (M)	I	
C1E00000_00100000	$-2^{-31} -2^{-1}$					
C1E00000_00100001	$-2^{-31} -2^{-1}-2^{-21}$					Bounce All
to	to	00000000	I	80000000 (ZP)	V	
C1E00000_00200000	$2^{-31} -2^{0}$			80000000 (NM)	I	
C1E00000_00200001	$2^{-31} -2^{0}-2^{-21}$				I	
to	to	00000000	I	80000000		
FFEFFFFF_FFFFFFFF	-Max DP					
FFF00000_00000000	-Inf	00000000	I	00000000	I	Bounce All

Chapter 6
Design for Test

This chapter describes the *Design For Test* (DFT) features of the VFP10 coprocessor and describes how best to integrate the DFT features into an *System on a Chip* (SoC). This chapter contains the following sections:

- *About DFT* on page 6-2
- *VFP10 DFT* on page 6-3
- *VFP10 Core* on page 6-4
- *VFP10 test wrapper* on page 6-6
- *VFP10 clocking* on page 6-10
- *Test Pins* on page 6-11.

6.1 About DFT

Using DFT techniques during the design and implementation phase of a chip produces the hardware hooks in the design unit to enable a tester to apply vectors, or control stimulus to achieve a high quality measurement. This is especially important if the design unit is to be embedded within other design units or chip logic.

If the proper mix of DFT techniques and logic are used, the resulting design:

- is easier to integrate
- is easier to generate vectors for
- has more efficient test vectors (in terms of size and tester time)
- has more cost-effective vectors with higher defect coverage per clock cycle.

Ultimately, the vectors that are generated for the design are easier to apply to the embedded core by the tester.

6.2 VFP10 DFT

The VFP10 coprocessor is a full scan Mux Dflip-flop core, with the exception of the latch-based Register file module. It contains one internal clock domain, **GCLK**.

The VFP10 coprocessor has a test wrapper to allow for test control and observation of the core from the ports as well as control and observation of the external logic surrounding the core. The test wrapper provides a single serial scan ring around the entire periphery of the core. The ultimate goal of adding a wrapper is to allow a tester to apply vectors, or control stimulus, to achieve a high quality measurement with a minimal amount of external pin control. This is extremely important if the design unit is to be embedded or buried within other design units or chip logic. The test wrapper can have dedicated wrapper cells or shared wrapper cells. The VFP10 coprocessor contains only dedicated wrapper cells that are clocked by a dedicated wrapper clock, **VFP10WCLK**.

VFP10WCLK is not perfectly delay matched with **GCLK** and care must be taken to prevent hold time errors. In the case of the VFP10 coprocessor hard core, the patterns are created with **VFP10WCLK** 180 degrees out of phase, with **GCLK**.

In addition, any asynchronous signals must be directly controlled by the *Automated Test Pattern Generator* (ATPG) tool. The asynchronous reset signals on the VFP10 coprocessor are directly controlled during scan mode by the **VFP10DFTRESET** signal. This port must be controlled directly by a pin in scan mode.

6.3 VFP10 Core

The VFP10 coprocessor core contains two different configurations of scan chains. These configurations are twelve, or six internal scan chains. The scan chains are shorter if there are more parallel scan chains in a design. The total vector count becomes smaller as the scan chains become shorter which saves tester memory. However, the final package or test environment may not have the pin bandwidth to handle the highest number of chains attainable on the VFP10 coprocessor, so other options are made available.

6.3.1 Scan chains

The VFP10 coprocessor is comprised of twelve individual scan chains. These scan chains are concatenated with the control signals **SCANMUX6** and **SCANMUX12** to allow another configuration of the scan chains. The other option is six internal scan chains. Table 6-1 illustrates the how the scan chains are concatenated.

Table 6-1 Scan chain configuration

Mode	Scan chains concatenated	Scan-in	Scan-out
SCANMUX6	11, 5	SCANIN[5]	VFP10SCANOUT[5]
SCANMUX6	10, 4	SCANIN[4]	VFP10SCANOUT[4]
SCANMUX6	9, 3	SCANIN[3]	VFP10SCANOUT[3]
SCANMUX6	8, 2	SCANIN[2]	VFP10SCANOUT[2]
SCANMUX6	7, 1	SCANIN[1]	VFP10SCANOUT[1]
SCANMUX6	6, 0	SCANIN[0]	VFP10SCANOUT[0]

There are two signals labelled **SCANMUX6** and **SCANMUX12** for the internal scan chains. These signals are tied HIGH or LOW to obtain the desired configuration as shown in Table 6-2.

Table 6-2 Internal scan chain configuration

Configuration	SCANMUX12 value	SCANMUX6 value
1 scan chain, legal with **SCORETEST** asserted	0	0
6 internal scan chains, 1 wrapper chain	0	1
12 internal scan chains, 3 wrapper chains	1	0
Restricted	1	1

6.4 VFP10 test wrapper

The VFP10 coprocessor test wrapper contains one configuration of the wrapper scan chain. It is important that the wrapper chain is not the longest scan chain so that it does not control the ultimate length of each scan pattern. This test wrapper chain is the shortest scan chain regardless of which internal scan chain mode is chosen. This wrapper scan chain consists of only dedicated test wrapper cells shown in Figure 6-1and Figure 6-2 on page 6-7. There is a wrapper cell connected to every input and output functional port with the exception of the clock ports. The test wrapper cells can be used for control and observation of the ports during testing of theVFP10 coprocessor and the testing of logic external to the VFP10 coprocessor. Figure 6-1 shows a dedicated input wrapper cell.

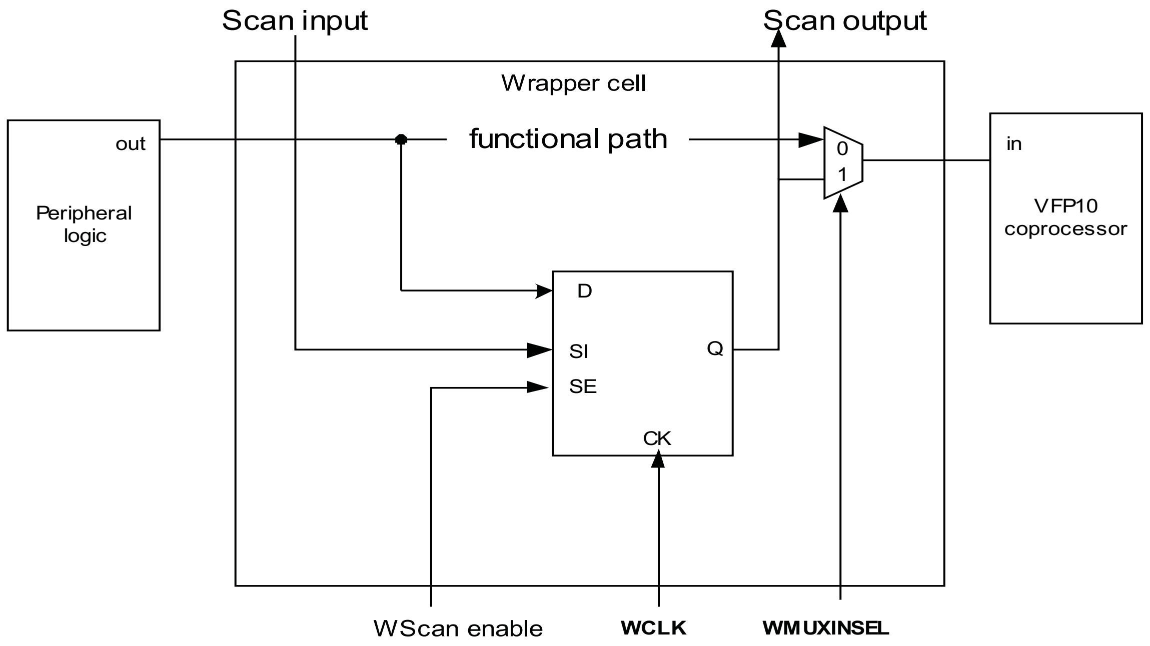


Figure 6-1 Dedicated input wrapper cell

Figure 6-2 on page 6-7 shows a dedicated output wrapper cell. The dedicated output cell has a safe gate.

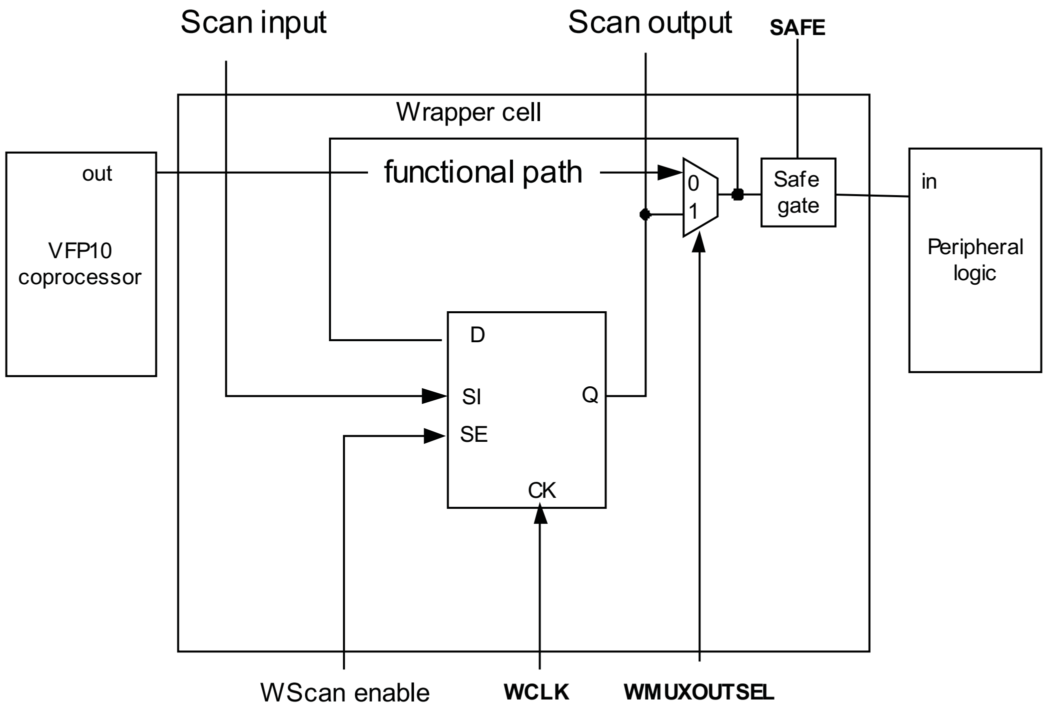


Figure 6-2 Dedicated output wrapper cell

6.4.1 Reset dedicated wrapper cell

There is a third type of wrapper cell designed for asynchronous reset input. Figure 6-3 on page 6-8 shows the elements of the reset dedicated wrapper cell.

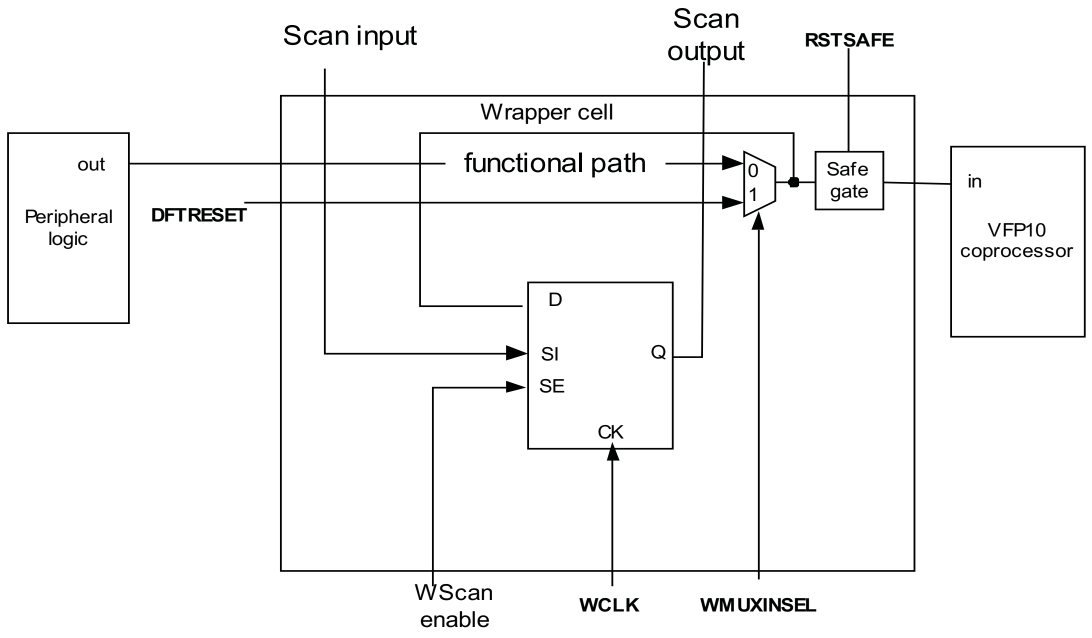


Figure 6-3 Reset dedicated wrapper cell

During external test mode, the safe gate on the reset wrapper cells can enable the reset of the core to reduce power and to keep the core safe. In addition, all asynchronous resets are directly controllable during scan mode. The **VFP10DFTRESET** port is a separate port that must be directly connected to a pin to have direct control of the reset during ATPG testing.

6.4.2 Wrapper cell control and observation configurations

The dedicated test cells require some control signals to differentiate:

- core testing
- external testing
- functional mode.

When **VFP10WMUXINSEL** is selected all of the input wrapper cells are in inward facing mode to allow for control of the core inputs during test. When this signal is negated, the wrapper input cells can observe data from logic peripheral to the core. This is also the state for functional mode. **VFP10WMUXSELOUT** is connected to the wrapper cells adjacent to the output ports of the core. When **VFP10WMUXSELOUT** is negated, it enables observation of the core logic. This is also the state during functional mode. When **VFP10WMUXSELOUT** is asserted, the wrapper cells can control data to the logic peripheral to the core. Table 6-3 describes the wrapper cell control and observation configurations.

Table 6-3 Wrapper cell control and observation configurations

Mode	Wrapper Mux Control Pins	
	VFP10WMUXINSEL	VFP10WMUXSELOUT
Core test	1	0
External test	0	1
Functional	0	0

6.4.3 Serial core test clocking

There is a serial core test mode enabled by the **SCORETEST** signal. In SCORETEST mode, all of the scan chains are connected serially in the VFP10 coprocessor macrocell. The last cell in the serial chain is a lock-up latch so that this output can connect to another clock domain and retain safe shift properties. Care must be taken to make sure the chain shifts safely. **VFP10WCLK** must be in the same phase as **GCLK** during this mode. Capture cycles cannot occur safely because of probable delay differences between the clock domains.

6.4.4 Clock gating

The clock gating signals are **VFP10DFTCKEN** and **VFP10DFTWCKEN**. These signals enable the gating of:

- the core clocks
- the wrapper clock
- both.

While the clock gating signals are enabled, **GCLK** and **VFP10WCLK** are enabled.

Note

In functional mode, **VFP10DFTCKEN** must be enabled. You are advised to disable **VFP10DFTWCKEN**.

6.5 VFP10 clocking

The VFP10 coprocessor wrapper clock **VFP10WCLK** is 180 degrees out of phase with **GCLK** during production scan mode as shown in Figure 6-4.

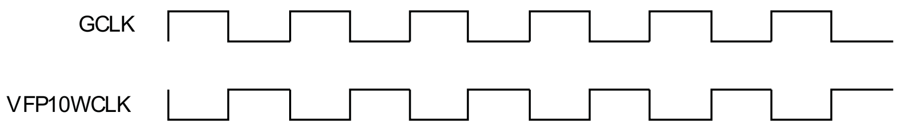


Figure 6-4 VFP10 production scan mode clocking

This prevents hold timing issues because **GCLK** and **VFP10WCLK** are not perfectly delay-matched within the VFP10 coprocessor macrocell. **VFP10WCLK** can be created by inverting **GCLK**, but the timing from the package pins to the ports of these two signals on the VFP10 coprocessor macrocell should be closely delay-matched.

In Serial Core Test (SCORETEST) all scan enables must remain enabled. All clocks are coincident as shown in Figure 6-5.

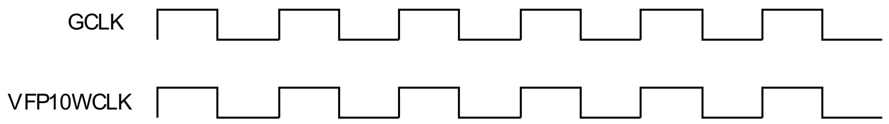


Figure 6-5 VFP10 serial core test clocking requirement

The scan chains in the VFP10 coprocessor are concatenated into one scan chain. There is a lock-up latch attached to the end of the wrapper scan chain.

6.5.1 VFP10 serial core test clocking requirement in safe mode

The wrapper cells connected to the outputs of the VFP10 coprocessor core all have safe state logic. In core test mode, **VFP10SAFE** can be asserted so that the values at the output of the core are held in a steady state. The reset also has a safe gate attached to it. In external test mode, the **VFP10RSTSAFE** signal can be asserted. This puts the core into reset during external test mode. If the state of the core is to be frozen for iddq testing. **VFP10RSTSAFE** should be disabled along with the clock enable signals after set-up of the core to hold state.

6.6 Test Pins

The dedicated test ports on this core must be instantiated in a specific manner for the test of the core to operate properly. Some of the signals are static and some are dynamic. In the case of the VFP10 coprocessor scan patterns, a dynamic signal must make it from the pin of the chip to the first flip-flop in the core, that is, the head flip-flop of a scan chain, within a cycle of the test pattern. The timing of the test patterns is such that at time 0, the inputs change and at mid-point through the cycle, the clock becomes active (except in the case of **VFP10WCLK**, as shown in Figure 6-4 on page 6-10). Table 6-4 describes the VFP10 coprocessor macrocell test ports.

Table 6-4 VFP10 macrocell test ports

Port Name	Direction	Type	Description
VFP10SCANMODE	Input	Static	Puts the device into scan mode
VFP10SCANEN	Input	Dynamic	Scan enable for all internal clock domains HIGH= shift
SCORETEST	Input	Static	Serialize all of the scan chains (internal and wrapper)
SCANMUX6	Input	Static	Enables accessibility to 6 separate internal scan chains
SCANMUX12	Input	Static	Enables accessibility to 12 separate internal scan chains
SCANIN[23:0]	Input	Dynamic	Scan input ports
VFP10SCANOUT[23:0]	Output	Dynamic	Scan output ports
VFP10DFTGCKEN	Input	Static	Enables the internal core clock
VFP10DFTRESET	Input	Dynamic	Direct control over asynchronous reset during scan mode
VFP10DFTWCKEN	Input	Static	Enables the wrapper clock to the dedicated test cells
VFP10WSCANEN	Input	Dynamic	Scan enable for all dedicated test cells in the wrapper HIGH = shift
WSCANIN[1:0]	Input	Dynamic	Input ports for the wrapper scan chains
VFP10WSCANOUT[1:0]	Output	Dynamic	Output ports for the wrapper scan chains

Table 6-4 VFP10 macrocell test ports

Port Name	Direction	Type	Description
VFP10WMUXINSEL	Input	Static	Configures the wrapper cells into core test mode
VFP10WMUXOUTSEL	Input	Static	Configures the wrapper cells in external test mode
VFP10SAFE	Input	Static	Forces safe values onto the outputs of the core Used during core test.
VFP10RSTSAFE	Input	Static	Enables the Reset to the core
VFP10WCLK	Input	Dynamic	Wrapper clock for dedicated wrapper cells

Table 6-5 shows the configuration of the VFP10 coprocessor test ports during core testing. A test control module can be created to control the states of these signals.

Table 6-5 VFP10 test signals during core scan test

Signal	Value
VFP10SCANMODE	1
VFP10DFTGCKEN	1
VFP10DFTWCKEN	1
VFP10SCANEN	Connect to an external pin
VFP10WSCANEN	Connect to an external pin
VFP10DFTRESET	Connect to an external pin
VFP10WMUXINSEL	1
VFP10WMUXOUTSEL	0
VFP10SAFE	1 (recommendation)
VFP10RSTSAFE	0
SCANIN	Connect to external pins
VFP10SCANOUT	Connect to external pins

6.6.1 Additional test pin configurations

Additional test pin configurations are described in:

- *VFP10 coprocessor test signals in functional mode* on page 6-13
- *VFP10 test pins in VFP10 coprocessor external test wrapper mode* on page 6-13.

Table 6-6 describes VFP10 coprocessor test signals in functional mode

Table 6-6 VFP10 coprocessor test signals in functional mode

VFP10 Test Pins	Connection
VFP10SCANMODE	0
VFP10DFTGCKEN	1
VFP10DFTWCKEN	0 (recommended)
VFP10SCANEN	0
VFP10WSCANEN	0
VFP10DFTRESET	0 (recommended)
VFP10MUXINSEL	0
VFP10MUXOUTSEL	0
VFP10SAFE	0
VFP10RSTSAFE	0
SCANIN	0 (recommended)
VFP10SCANOUT	gated 0 (recommended)

Table 6-7 describes VFP10 test pins in VFP10 coprocessor external test wrapper mode

Table 6-7 VFP10 test pins in VFP10 coprocessor external test wrapper mode

VFP10TestMode	Connection
VFP10SCANMODE	0
VFP10DFTGCKEN	0 (recommended)
VFP10DFTWCKEN	1
VFP10SCANEN	0

Table 6-7 VFP10 test pins in VFP10 coprocessor external test wrapper mode

VFP10TestMode	Connection
VFP10WSCANEN	Connected to a pin
VFP10DFTRESET	Connected to a pin if **VFP10RSTSAFE** disabled
VFP10MUXINSEL	0
VFP10MUXOUTSEL	1
VFP10SAFE	0
VFP10RSTSAFE	1 (recommended)
SCANIN	0
VFP10SCANOUT	Not needed, gated 0 (recommended)
VFP10WSCANOUT	Connected to a pin or another scan chain
WSCANIN	Connected to a pin or another scan chain

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

An instruction is said to be *bounced* by the VFP10 coprocessor if it is valid for the VFP10 coprocessor but not acknowledged to the ARM. This action initiates exception processing through the undefined instruction trap. The VFP10 coprocessor bounces an instruction by asserting CPBOUNCEE in the D stage of a trigger instruction.

See also Trigger instruction, Potentially exceptional instruction, and Exceptional state.

Coprocessor Data Processing (CDP)

For the VFP10 coprocessor, CDP operations are arithmetic operations rather than load or store operations.

Default NaN Mode

A mode enabled by setting the DN bit in the FPSCR (FPSCR bit 25). In this mode, all operations, which result in a NaN, will return the default NaN, regardless of the cause of the NaN result. This mode is compliant with the IEEE 754 specification, but implies that all information contained in any input NaNs to an operation will be lost.

Disabled exception

An exception that has its associated exception enable bit in the FPCSR set to 0 is referred to as *disabled*. For these exceptions the IEEE-754 specification defines the 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 in the FPSCR 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 short 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.

See also 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

A mode enabled by setting the FZ bit in the FPSCR (FPSCR bit 24). In this mode all inputs to arithmetic operations which are in the subnormal range for the input precision ($-2^{Emin} < x < 2^{Emin}$) and all results which are in the given range, before rounding, are treated as positive zero, rather than interpreted as, or converted to, a subnormalized value.

Half vector

A short vector operation in which the length is 4 or less for single-precision and 2 or less for double-precision. In RunFast mode these half-vector operations do not lock their source registers, and a load immediately following will not have a stall introduced due to a write-after-read hazard on the source registers. Half-vectors are only CDP operations which are vectorizable, and do not include DIV or SQRT instructions.

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/MCRR

A class of data transfer instructions which transfer 32-bit or 64-bit quantities from an ARM register or registers to a VFP10 coprocessor register or registers.

MRC/MRRC

A class of data transfer instructions which transfer 32-bit or 64-bit quantities from an VFP10 coprocessor register or registers to an ARM register or registers.

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 exponent field will be maximum 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 that 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.

See also Bouncing, Trigger instruction, and Exceptional state.

Register banks

A bank of registers is defined for use in vector operations. For the VFPv2 architecture, the register banks are defined as:

Table G-8 Register banks in single-precision and double-precision registers

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

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 - *Round to Nearest* (RN) 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*. *Round to Zero*, or chop (RZ) 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) and *Round to Minus Infinity* (RM) are used in interval arithmetic.

RunFast Mode

RunFast mode specifies hardware support for the handling of IEEE-754 exceptional conditions and special operands. RunFast mode is enabled by enabling the Default NaN mode (FPSCR[25] set), Flush-to-Zero mode (FPSCR[24] set), and disabling all exceptions (FPSCR[12:8] all clear). In RunFast mode the VFP10 coprocessor will not bounce to the ARM for any legal operation or any operand, but will supply a result to the destination. This result will be what is specified by the IEEE-754 for all inexact and overflow results, and all invalid operations that result from operations not involving NaNs. For operations involving NaNs, the Default NaN mode specifies the result to be the default NaN.

Scalar operation

An operation involving a single destination register.

Short vector operation

An operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination.

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.

Stride

The stride field in the FPSCR (FPSCR[21:20]) specifies the increment applied to register addresses in short vector operations. A stride of `00`, specifying an increment of +1, will cause a short vector operation to increment each vector register by 1 for each iteration, while a stride of 11 will specify an increment of +2.

For example, with a LEN of 011 (for an effective short vector length of 4 iterations) and a stride of 00, the instruction:

```
FADDS S8, S16, S24
```

executes the scalar operations:

```
FADDS S8, S16, S24
FADDS S9, S17, S25

FADDS S10, S18, S26

FADDS S8, S19, S27
```

If the stride was changed to 11, the same instruction would execute the following scalar operations. Notice the change in registers for the 2nd through 4th iterations:

```
FADDS S8, S16, S24

FADDS S10, S18, S26

FADDS S12, S20, S28

FADDS S14, S22, S30
```

See the *ARM Architecture Reference Manual* for a listing of combinations of precision, short vector length, and stride which are UNPREDICTABLE.

Subnormalized value (subnormal)
A representation of a value in the range ($-2^{Emin} < x < 2^{Emin}$). In the IEEE-754 format for single and double precision operands, a subnormalized value, or subnormal, 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 subnormalized operands be performed with the same precision as with normal operands.

Support code
Software that must be used to complement the hardware to provide compatibility with the IEEE-754 standard. The support code is intended to have two components:

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 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 (subnormal values or decimal data types). The routines can be written to utilize the VFP10 coprocessor in their intermediate calculations if care is taken to restore the user state at the exit of the routine.

Trap

An exceptional condition that has the respective exception enable bit set in the FPSCR. The user provided trap handler is executed.

Trigger instruction

The instruction that 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.

See also Bouncing, Potentially exceptional instruction, and Exceptional state.

UNDEFINED

Indicates an instruction that generates an undefined instruction trap. See the *ARM Architecture Reference Manual* 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, subnormal 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

See Short vector operation.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

Addition
- like-signed 5-23
- unlike-signed 5-23

Applications
- data-intensive 1-18
- examples 1-3, 2-2
- filtering 2-2

Architecture
- VFPv2 1-2

B

Bouncing
- FPEXC register 3-18
- FPINST register 5-4
- FPINST2 register 3-18, 5-4
- in RunFast mode 1-14, 4-13, 5-13
- instructions 5-3
- pipelines 5-8

Bouncing *see also* CPBOUNCEE, FPEXC

C

CDP instruction 1-9, 1-12, 1-16, 2-12, 3-6, 3-9, 3-10, 4-6, 4-11, 4-18, 4-21, 4-23, 5-7, 5-10, 5-21

Code
- examples 2-10, 2-12, 2-15, 3-8, 4-6, 4-14, 4-15, 4-17, 4-19, 5-8, 5-10

Comparisons
- floating-point 3-23
- FPINST2 register 5-10
- FPSCR register 3-7
- use of FMSTAT 1-17

CPBOUNCEE
- signal 3-19

Cumulative exception bits
- DZC 3-26, 5-16
- IDC 1-13, 1-22, 3-2, 5-12
- IOC 3-2, 3-6
- IXC 3-3, 3-9, 3-26, 5-18, 5-19, 5-20, 5-21
- OFC 3-19, 3-26, 5-17, 5-18
- UFC 1-13, 1-22, 3-2, 3-3, 3-8, 3-10, 3-18, 5-19

Cumulative exception bits *see also* Exceptions, Trap enable bits

D

Data storage
- endianness 2-10
- in memory 2-10
- target address values 2-10

Default NaN mode 1-14
- bits 1-13
- default NaNs 3-10
- description 3-10
- DN bit 3-24
- modifications 1-21
- NaN results 1-14
- removal of hazards 1-21

Division by zero
  exceptions 5-16
DZC
  cumulative exception bit 3-26, 5-16
DZE
  trap enable bit 3-26, 5-16, 5-17, 5-27

E

EX
  FPEXC bit field 3-17, 3-18, 3-19, 5-3, 5-6, 5-8, 5-10, 5-14, 5-17
Exception processing
  description 5-2
  FPINST register 5-8
  procedure 5-3
  support code 5-3
Exceptions
  arithmetic 5-3
  checking 1-7
  detection 1-5
  division by zero 5-16
  handlers 5-3
  illegal instructions 5-3
  imprecise 3-9
  inexact 5-21
  like-signed addition 5-23
  overflow 5-17
  trap enable bits 1-13
  underflow 5-19
  unlike-signed addition 5-23
Exceptions *see also* Trap enable bits, Cumulative exception bits
Exponent 2-3, 2-4

F

FABS instruction 1-6, 5-13
FADD instruction 1-6, 4-7, 4-19
FCMP instruction 1-7, 3-7, 4-4, 5-28
FCPY instruction 1-7, 5-13, 5-28
FCVTSD instruction 1-7
FDIV instruction 1-8, 2-7, 4-6, 4-8, 4-9, 4-22, 4-23, 5-27, 5-28
FLD instruction 1-10, 2-3, 4-6, 4-8, 4-15, 4-16, 4-17, 4-19, 4-22
FLDM instruction 1-10
Floating-point comparisons
  condition flags 3-23
Flush-to-zero mode
  controlling 3-24
FMAC instruction 1-6, 1-7, 1-18, 4-23, 5-14, 5-24, 5-26
FMAC pipeline
  diagram 1-6
FMDHR instruction 1-11, 2-7
FMDLR instruction 1-11, 3-17, 4-23, 5-6, 5-8
FMDRR instruction 1-11, 2-8, 3-11
FMRDH instruction 1-11, 1-17, 1-18, 2-7, 3-7, 4-8
FMRDL instruction 1-11, 2-7
FMRRD instruction 1-11, 2-8, 3-12
FMRRS instruction 1-11, 2-8, 3-15
FMRS instruction 1-11, 1-17
FMRX instruction 1-11, 3-17, 3-19, 4-2, 4-4
FMSR instruction 1-11, 1-18, 2-7, 3-7
FMUL instruction 1-6, 2-14, 4-23, 5-13, 5-25, 5-27
FMXR instruction 1-11, 2-7, 3-17, 4-4, 4-23, 5-6, 5-8
FNEG instruction 1-6
Forwarding
  examples 4-6
FPEXC register
  bouncing 3-18
  EX 3-17, 3-18, 3-19, 5-3, 5-6, 5-8, 5-10, 5-14, 5-17
  exception bits 3-19
  format 3-18
  FPv2 bit 5-4
  FTZ mode 3-19
  illegal instructions 5-6
  not a trigger instruction 5-7
  pipeline stalls 4-4
  support code 3-17, 5-3
  system control register 3-17
FPINST register
  bouncing 5-4
  exception processing 5-8
  not a trigger instructions 5-8
  potentially exceptional instruction 5-3
  serializing operations 4-4
  support code 5-4
  system control register 3-17
  trigger instructions 5-8
FPINST2 register
  AL in condition code 5-10
  bouncing 3-18, 5-4
  load and store instructions 3-17
  pre-trigger slot 5-7
  RunFast mode 1-13
  serializing operations 4-4
  system control register 3-17
  with invalid data 5-11
FPSCR register
  comparisons 3-7
  controlling flush-to zero mode 3-24
  controlling the rounding mode 3-24
  DZE bit 3-26, 5-16, 5-17, 5-27
  format 3-23
  FTZ mode 3-24
  FZ bit 1-12, 3-9, 3-24, 5-12, 5-19
  IDC bit 3-3, 3-10
  IXC bit set 5-18
  LEN field settings 2-2, 2-13
  OFC bit set 5-18
  potentially exceptional conditions 3-19, 5-2, 5-8
  serializing operations 4-4
  STRIDE 2-2, 3-25
  subnormal input exception bit 3-2
  system control register 3-17
  UFE bit 3-2
  using ARM comparison operations 1-17, 3-7
FPSID register
  format 3-21
FPv2 bit
  FPEXC register 5-4
FSITO instruction 1-6
FSQRT instruction 1-8
FST instruction 1-10, 2-3, 4-6, 4-15, 4-16
FSTM instruction 1-10, 2-3, 3-13, 4-6, 5-8
FSUB instruction 1-6, 5-23
FTOSIZD instruction 3-24
FTOSIZS instruction 3-24
FTOUI instruction 1-6
FTOUIZD instruction 3-24
FTOUIZS instruction 3-24

FTZ 3-3
FTZ mode
  controlling 3-24
  description 3-9
  FPEXC 3-19
  in Non-RunFast mode 1-12
  in RunFast mode 1-13, 3-5
  modification to Invalid trap 3-2
  modifications from Rev 0 1-21
  modifications to IXC bit 3-3
  subnormal input 5-16
  subnormal operands 5-22
  underflowed results 3-8
FZ bit
  FPSCR register 1-12, 3-9, 3-24, 5-12, 5-19

G

GCLK
  signal 6-10
  timing 6-10

H

Hazard detection
  Non-RunFast mode 4-11
  RunFast mode 4-11
Hazards
  defined 4-3
  examples 4-14
  read-after-read 1-18
  read-after-write 1-18

I

ID numbers
  VFP10 coprocessor 1-4
IDC
  cumulative exception bit 1-13, 1-22, 3-2, 5-12
  FPSCR register 3-3, 3-10
  subnormal input exception bit 3-2
IDE
  trap enable bit 1-22, 3-2, 5-12
Illegal instructions
  FPEXC register 5-6
Inexact
  exceptions 5-21
Input exceptions
  NaNs 5-22
Instructions
  bouncing 5-3
  branch 1-17
  CDP 1-9, 1-12, 1-16, 2-12, 3-6, 3-9, 3-10, 4-6, 4-11, 4-18, 4-21, 4-23, 5-7, 5-10, 5-21
  executed in hardware 1-6, 1-8
  FABS 1-6, 5-13
  FADD 1-6, 4-7, 4-19
  FCMP 1-7, 3-7, 4-4, 5-28
  FCPY 1-7, 5-13, 5-28
  FCVTSD 1-7
  FDIV 1-8, 2-7, 4-6, 4-8, 4-9, 4-22, 4-23, 5-27, 5-28
  FLD 1-10, 2-3, 4-6, 4-8, 4-15, 4-16, 4-17, 4-19, 4-22
  FLDM 1-10
  FLMDMX 2-3
  FMAC 1-6, 1-7, 1-18, 4-23, 5-14, 5-24, 5-26
  FMDHR 1-11, 2-7
  FMDLR 1-11, 3-17, 4-23, 5-6, 5-8
  FMDRR 1-11, 2-8, 3-11
  FMRDH 1-11, 1-17, 1-18, 2-7, 3-7, 4-8
  FMRDL 1-11, 2-7
  FMRRD 1-11, 2-8, 3-12
  FMRRS 1-11, 2-8, 3-15
  FMRS 1-11, 1-17
  FMRX 1-11, 3-17, 3-19, 4-2, 4-4
  FMSR 1-11, 1-18, 2-7, 3-7
  FMSTAT 1-17
  FMUL 1-6, 2-14, 4-23, 5-13, 5-25, 5-27
  FMXR 1-11, 2-7, 3-17, 4-4, 4-23, 5-6, 5-8
  FNEG 1-6
  FSITO 1-6
  FSQRT 1-8
  FST 1-10, 2-3, 4-6, 4-15, 4-16
  FSTM 1-10, 2-3, 3-13, 4-6, 5-8
  FSTMX 2-3
  FSUB 1-6, 5-23
  FTOSIZD 3-24
  FTOSIZS 3-24
  FTOUI 1-6
  FTOUIZD 3-24
  FTOUIZS 3-24
  illegal 5-6
  MCR 1-9, 2-7
  MRC 1-9, 2-7, 4-6
  MRCC 1-9, 2-7
  MRRC 1-21, 2-8, 3-2
Invalid operations
  NaNs 5-13
IOC
  cumulative exception bit 3-2, 3-6
IOE
  trap enable bit 1-12, 3-2, 3-7, 3-26, 5-14
IXC
  cumulative exception bit 3-3, 3-9, 3-26, 5-18, 5-19, 5-20, 5-21
  modifications 3-3
IXE
  trap enable bit 1-12, 5-21

L

LEN
  FPSCR bits 2-13
Like-signed addition 5-23

M

Mantissa 2-4
  overflow 5-24
MCR instruction 1-9, 2-7
Modes
  Default NaN 1-13, 1-14, 1-21, 3-10, 3-24
  Non-RunFast 1-12, 1-18, 3-5, 3-6, 4-11, 4-12, 4-16, 5-2, 5-13, 5-19
  RunFast 1-2, 1-13, 1-22, 3-5, 3-6, 4-9, 4-17, 5-2, 5-13
Modifications
  Default NaN mode 1-21
  IXC 3-3
  UFC 3-2
MRC instruction 1-9, 2-7, 4-6
MRCC instruction 1-9, 2-7

MRRC instruction 1-21, 2-8, 3-2

N

NaNs
 compare with exception operands 3-7
 default definition 3-6
 handling in RunFast mode 3-6
 input exceptions 5-22
 invalid operations 5-13
 IOE bit set 3-7
 quiet 3-6
 signaling 2-3, 3-6, 5-13
Non-RunFast mode 1-12, 1-18, 3-5, 3-6, 4-11, 4-12, 4-16, 5-2, 5-13, 5-19
 FTZ mode 1-12
 hazard detection 4-11
 read-after-read hazard 1-18

O

OFC
 cumulative exception bit 3-19, 3-26, 5-17, 5-18
OFE
 trap enable bit 1-12, 3-26, 5-17
Operations
 scalar-only 2-13
 serializing 4-4
 vector-only 2-13
Overflow
 exceptions 5-17

P

Pipelines
 bouncing 5-8
 DS 1-8, 1-16, 1-18, 1-21, 4-9, 4-18, 4-21
 FMAC 1-5, 1-6, 1-16, 1-21, 4-21
 FMAC diagram 1-6
 FPEXC register and stalls 4-4
 LS 1-5, 1-9, 1-16, 4-2, 4-9, 4-21
 operating in parallel 1-5, 4-21
 stages 1-5, 5-8
Potentially exceptional conditions
 description 5-2
 detailed conditions 5-8
 FPSCR 3-19
Potentially exceptional instructions
 FPINST register 5-3
Pre-trigger slot
 FPINST2 register 5-7

R

Read-after-read
 hazard 1-18
 Non-RunFast mode 1-18
Read-after-write
 hazards 1-18
Register file
 banks 2-12
 consistency in usage 2-9
 data formats 2-3
 details 2-2
 double-precision items 2-2
 instructions 2-2
 internal formats 2-2
 single-precision items 2-2
 suitability for vector operations 2-12
Rounding modes
 controlling 3-24
RunFast
 mode 4-9, 5-13
RunFast mode 1-2, 1-13, 1-22, 3-5, 3-6, 4-17, 5-2
 bouncing 4-13
 FTZ mode 1-13, 3-5
 hazard detection 4-11

S

SCANMUX12
 signal 6-4
SCANMUX6
 signals 6-4
Scoreboard
 description 4-9
SCORETEST
 signal 6-9
Serializing operations 4-4
Short vectors
 defined 1-15
Signals
 CPBOUNCEE 3-19
 GCLK 6-10
 SCANMUX12 6-4
 SCANMUX6 6-4
 SCORETEST 6-9
 VFP10DFTCKEN 6-9
 VFP10DFTRESET 6-3, 6-8
 VFP10RSTSAFE 6-10
 VFP10SAFE 6-10
 VFP10WCLK 6-3, 6-4, 6-9
 VFP10WMUXINSEL 6-8
 VFP10WMUXSELOUT 6-8, 6-9
STRIDE
 2-15
 field settings 3-25, 4-15, 4-16, 4-17, 4-19
 FPSCR bits 3-25
 scalar mode 3-25
Subnormal input
 FTZ mode 5-16
Subnormal operands
 FTZ mode 5-22
Support code
 arithmetic exceptions 5-23
 components 1-2, 1-7, 1-12, 1-13, 5-3
 determining exceptional state 3-19, 5-4
 determining trigger instructions 5-7
 exception processing 5-3
 FPEXC register 3-17, 5-3
 FPINST register 5-4
 IEEE 754 compliance 3-4
 library functions 3-4
 Non-RunFast mode operation 5-13
 overflow 5-17
 RunFast mode operation 5-13
 system control registers 3-17
 UFC bit 5-19
 underflow 5-19
System control registers
 FPEXC 3-17
 FPINST 3-17
 FPINST2 1-13, 3-17

FPINST2 register 5-11
FPSCR 3-17
FPSID 3-17, 3-21, 4-4, 5-7

T

Test pins
configurations 6-13
list 6-11
Timing
GCLK 6-10
Trap enable bits
DZE 3-26, 5-16
IDE 1-22, 3-2, 5-12
IOE 1-12, 3-2, 3-7, 3-26, 5-14
IXE 1-12, 5-21
OFE 1-12, 3-26, 5-17
UFE 1-22, 3-2
Trap enable bits *see also* Exceptions, Cumulative exception bits

U

UFC
cumulative exception bit 1-13, 1-22, 3-2, 3-3, 3-8, 3-10, 3-18, 5-19
modifications to 3-2
set in support code 5-19
UFE
trap enable bit 1-22, 3-2
Underflow
exceptions 5-2, 5-19
pessimistic determination 5-19
Underflowed results
FTZ mode 3-8
Unlike-signed addition
exceptions 5-23

V

VECITR
FPEXC bit field 1-17, 3-20, 5-4, 5-8, 5-9, 5-10, 5-16, 5-19
Vectors
short 1-5, 1-15
VFPv2
vector floating-point architecture 1-2
VFP10
vector floating-point coprocessor ix
VFP10 coprocessor
ID numbers 1-4
VFP10 pipeline
stages 5-8
VFP10DFTCKEN
clock gating signal 6-9
signals 6-9
VFP10DFTRESET
signal 6-3, 6-8
VFP10DFTWCKEN
clock gating signal 6-9
signals 6-9
VFP10RSTSAFE
signal 6-10
VFP10SAFE
signal 6-10
VFP10WCLK
signal 6-3, 6-4, 6-9
VFP10WMUXINSEL
signal 6-8
VFP10WMUXSELOUT
signal 6-8, 6-9

W

Wrapper cells
dedicated input wrapper cells 6-6
dedicated output wrapper cells 6-6
reset dedicated output wrapper cells 6-7