8 The FPA Coprocessor Macrocell

This chapter gives an overview of the FPA coprocessor macrocell.

The FPA Coprocessor Macrocell

8.1 Overview

The FPA is a floating-point accelerator for the ARM family of CPUs. It has been designed to maximize the performance/power, performance/cost and performance/die size ratios while still providing a balanced floating-point versus integer performance for ARM-based systems.

Typical performance in the range 3 to 8 MFlops is expected at a clock frequency of 40 MHz; actual performance is dependent on the:

- precision selected
- system configuration
- the degree to which the floating-point code is scheduled and otherwise optimized

The FPA in the ARM7500FE is an on-chip floating-point coprocessor connected to the ARM processor core. It is a fully static design and its low power consumption, especially when in standby mode, makes it eminently suitable for portable and other power- and cost-sensitive applications. When used in conjunction with its support code, the FPA fully implements the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985).

The design of the FPA is based on an 81-bit internal datapath, with autonomous load/store and arithmetic units which can operate concurrently. Single, double and extended precision IEEE formats are all supported. The FPA achieves its high performance, whilst remaining a low cost and low power solution, by employing RISC and other advanced design techniques. It is interfaced to the ARM CPU over a simple, high-performance coprocessor bus. The ARM instruction pipeline is mirrored on the FPA so that floating-point instructions can be executed directly with minimal communication overhead. Pipelining, concurrent execution units and speculative execution are all employed to improve performance without having a great impact on power consumption.

A RISC approach has been taken in selecting between those floating-point instructions which are candidates for implementation in the FPA and those which are handled by software support. The FPA instruction repertoire includes only the basic operations plus compare, absolute value, round to integral value and floating-point to integer and integer to floating-point conversions. In addition, only normalized operands and zeros are handled in hardware; operations on denormalized numbers, infinities and NaNs are handled by the support code. Only the inexact exception is dealt with by hardware; all other exceptions cause the software support code to be called, whether or not the associated trap is enabled. This approach has helped to minimize the die size whilst having a negligible effect on performance in most applications.

8.2 FPA Functional Blocks

FPA consists of five main functional blocks:

- coprocessor interface
- instruction issuer
- load-store unit
- register bank
- arithmetic unit

These are described in the following sections

8.2.1 Coprocessor interface

This block is responsible for arbitrating instructions with the CPU and telling the Load-Store unit when to go ahead with data transfers.

Like ARM integer instructions, all ARM floating-point instructions are conditional, obviating the need for branches for many common constructs. If a failed condition causes an instruction already issued to the Load-Store or Arithmetic unit to be skipped, that instruction is cancelled and any results calculated thus far are discarded.

The same mechanism is used to cancel prefetched instructions if a branch is taken or if the ARM CPU gets interrupted before an FPA instruction has been arbitrated.

8.2.2 Instruction issuer

The instruction issuer is responsible for examining the incoming instruction stream and deciding whether any instructions are candidates for issuing to either the load-store unit or the arithmetic unit.

Instructions can be selected from the fetch, decode or execute stages of the ARM pipeline follower. Data anti-dependency hazards (write-after-write and write-after-read) are dealt with by this unit by preventing issue until the hazard has been cleared.

Instructions are issued strictly in order and only one can be issued per cycle.

8.2.3 The load-store unit

The load-store unit does the formatting and conversion necessary when moving data between the 32-bit ARM databus and the 81-bit internal register format. It is also responsible for checking all input operands and flagging any that are not normalized numbers or zero.

Most subsequent operations on flagged data cause the instruction to be passed to software which will then emulate the instruction. All internal operations are performed to the internal 81-bit format.

8.2.4 The register bank

The register bank contains eight 81-bit dual read-access, dual write-access registers.

Data dependency hazards (read-after-write) are handled by the register control logic; read requests from either unit are stalled until the hazard is cleared.

There is also a 33-bit temporary register, used by FIX, FLT and compare instructions to transfer intermediate results between the Load-Store Unit and the Arithmetic Unit.

The register bank also contains logic for register-forwarding, allowing the result of one calculation to be used directly as the source for the next.

8.2.5 The arithmetic unit

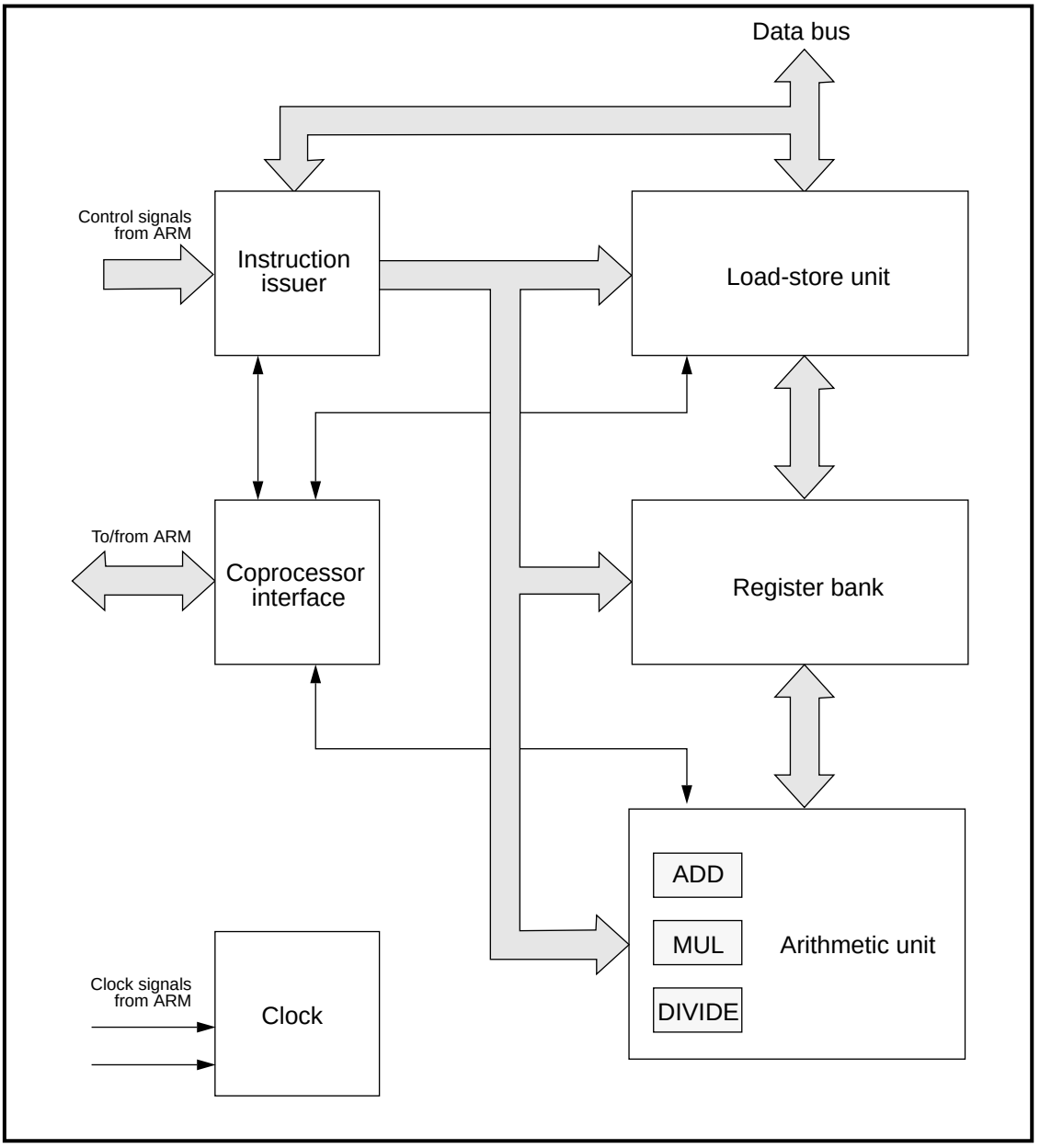
The arithmetic unit has a four-stage pipeline (Prepare, Calculate, Align and Round) and can speculatively execute instructions up to, but not including, register writeback. Writeback can only occur once the instruction has been arbitrated with the ARM CPU.

An unusual feature of the pipeline is that each of the pipeline stages is offset by one half-cycle from the previous stage, allowing some instructions to traverse the pipeline in 2 cycles.

The Calculate stage includes a 67-bit adder, iterative array multiplier and divide unit. Fast barrel shifters are used for pre-alignment and post-normalization.

Arithmetic operations are normally performed asynchronously to the ARM instruction stream so that an instruction is arbitrated with the CPU before the FPA has detected whether an exception will occur. Arithmetic exceptions are therefore normally imprecise. If precise exceptions are required (for example, in debugging), a mode bit (the **SO** bit in the FPSR) can be set. This forces arbitration to be delayed until the arithmetic operation has completed, at the expense of a reduction in performance.

8.3 FPA Block Diagram



9 Floating-Point Coprocessor Programmer's Model

This chapter details the floating-point coprocessor programmer's model

Floating-Point Coprocessor

9.1 Overview

The ARM IEEE floating-point system has:

- 8 high-precision floating-point registers, F0 to F7
- a working precision of 80 bits, comprising:
  - 64-bit mantissa
  - a 15-bit exponent
  - a sign bit

9.1.1 Floating-point status register

There is a floating-point status register (FPSR) which, like ARM's PSR, holds all the necessary status and control information for the floating-point system that an application should be able to access. It holds flags which indicate various error conditions, such as overflow and division by zero. Each flag has a corresponding trap enable bit, which can be used to enable or disable a trap associated with the error condition. Bits in the FPSR allow a client to distinguish different implementations of the floating-point system and to enable or disable special features of the system.

9.1.2 Floating-point control register

The FPA also contains a floating-point control register (FPCR). This is used to communicate status and control information between the FPA and the FPA support code.

Note: The definition of the FPCR may be different for other implementations of the ARM IEEE floating-point system; the FPCR may not even exist in some implementations. Software outside the floating-point system should therefore not use the FPCR directly.

9.2 Floating-Point Operation

All basic floating-point instructions operate as though the result were computed to infinite precision and then rounded to the length and in the way specified by the instruction. The rounding is selectable from:

- Round to nearest
- Round to +infinity (P)
- Round to -infinity (M)
- Round to zero (Z)

The default is **round to nearest**: as required by the IEEE, this rounds to **nearest even** for the tie case. If one of the other rounding modes is required it must be given in the instruction.

The floating-point system architecture is a load/store architecture (like the ARM CPU); the data-processing operations only refer to floating-point registers. Values may be stored into ARM memory in one of five formats (only four of which are visible at any one time since P and EP are mutually exclusive):

- IEEE Single Precision (S)
- IEEE Double Precision (D)
- IEEE Double Extended Precision (E)
- Packed Decimal (P)
- Expanded Packed Decimal (EP)

If it is required to preserve register contents exactly (including signalling NaNs), the LFM and SFM instructions should be used. Note however that LFM and SFM should only be used for register preservation within programs and not for data which is to be transferred between programs and/or systems. The format of data stored using SFM is implementation-dependent and can generally only be restored by an LFM instruction from the same implementation.

Floating-point systems may be built from software only, hardware only, or some combination of software and hardware and the results look the same to the programmer. However, the supervising operating system will need to be aware of which implementation is in use, in order to extract the best performance.

Similarly, compilers can be tuned to generate bunched FP instructions for the FPE and dispersed FP instructions for the FPA to improve overall performance. The manner in which exceptions are signalled is at the discretion of the surrounding operating system.

Note: In the case of the FPA system, an exception caused by a floating-point data operation or a FLT may be asynchronous (due to the nature of the ARM coprocessor interface.) Such an exception is raised some time after the instruction has started, by which time the ARM may have executed a number of instructions following the one that has failed. This means that the exact address of the instruction that caused the exception may not be identifiable. However, all the information about the exception that the IEEE Standard recommends is available.

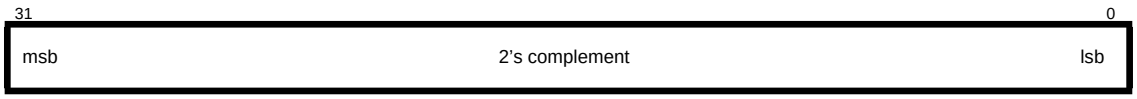
Furthermore, in the FPA a “fully synchronous, but slow” mode of operation is available that allows the address of the faulting instruction to be determined; this is described in *Bit 10 SO - Select Synchronous Operation of FPA* on page 9-9.

9.2.1 Additional information

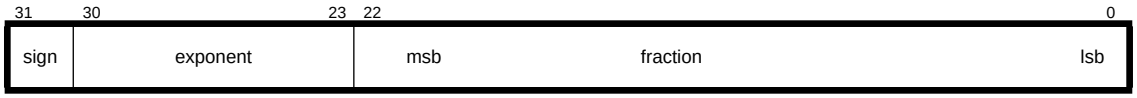
Familiarity with the *IEEE Standard for Binary Floating-point Arithmetic: ANSI/IEEE Std 754-1985* will be helpful in reading this datasheet.

9.3 ARM Integer and Floating-Point Number Formats

9.3.1 Integer

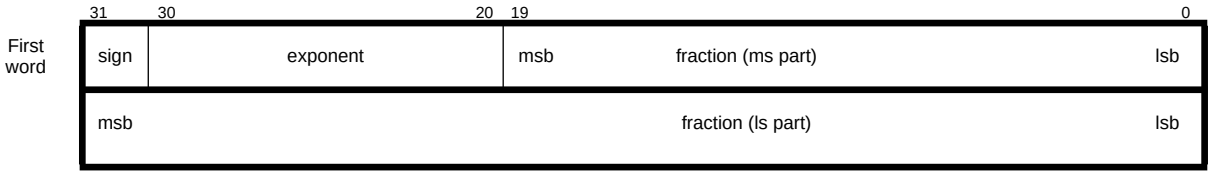


9.3.2 IEEE single precision (S)



127 Normalized number exponent bias
126 Denormalized number exponent bias

9.3.3 IEEE double precision (D)



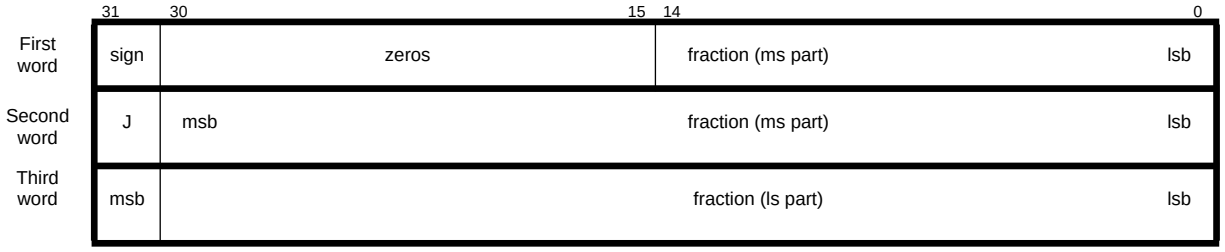
1023 Normalized number exponent bias
1022 Denormalized number exponent bias

Single and double values

	Sign	Exponent	Fraction	Value represented
Quiet NaN	x	maximum	1xxxxxxxxx	IEEE Quiet NaN
Signalling NaN	x	maximum	0 non-zero	IEEE Signalling NaN
Infinity	sign	maximum	0000000000	$(-1)^{sign}$ * infinity
Zero	sign	0	0000000000	$(-1)^{sign}$ * 0
Denormalized no	sign	0	non-zero	$(-1)^{sign}$ * 0.fraction * $2^{-(denorm.\ bias)}$
Normalized no.	sign	not 0 and not maximum	xxxxxxxxxx	$(-1)^{sign}$ * 1.fraction * $2^{(exponent\ -\ norm.\ bias)}$

Table 9-1: Single and double values

9.3.4 IEEE extended double precision (E)



J is the bit to the left of the binary point

16383 normalized and denormalized number exponent bias

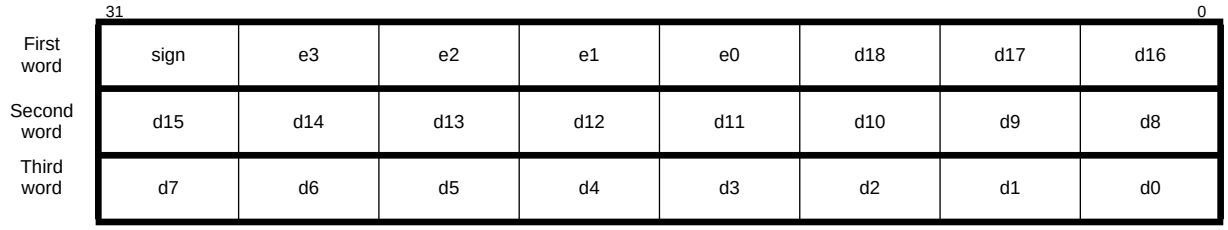
Extended values

	Sign	Exponent	J	Fraction	Value represented
Quiet NaN	x	maximum	x	1xxxxxxxxx	IEEE Quiet NaN
Signalling NaN	x	maximum	x	0 non-zero	IEEE Signalling NaN
Infinity	sign	maximum	0	0000000000	$(-1)^{sign}$ * infinity
Zero	sign	0	0	0000000000	$(-1)^{sign}$ * 0
Denormalized no	sign	0	0	non-zero	$(-1)^{sign}$ * 0.fraction * $2^{-(denorm.bias)}$
Normalized no.	sign	not max	1	xxxxxxxxxx	$(-1)^{sign}$ * 1.fraction * $2^{(exponent - norm.bias)}$
** Illegal value	x	not 0 and not max	0	xxxxxxxxxx	
** Illegal value	x	maximum	1	0000000000	

Table 9-2: Extended values

****** In general, illegal values must not be used, although specific floating-point implementations may use these bit patterns for internal purposes.

9.3.5 Packed decimal (P)



- the value is +/- d * 10^(+/- e)
- d18 and e3 are the most significant digits of d and e respectively
- sign contains both the number's sign (bit 31) and the exponent's sign (bit 30). The other bits (29,28) are 0
- the value of d is arranged with the decimal point between d18 and d17, and is normalized so that for an ordinary number 1<=d18<=9
- the guaranteed ranges for d and e are 17 and 3 digits respectively: e3 and d0, d1 may always be zero in a particular system.
- the result is undefined if any of the packed digits is hexadecimal A through F

Packed decimal values

	Sign (top bit)	Sign (next bit)	Exponent	Digit values
Quiet NaN	x	x	FFFF	d18>7, rest non-zero
Signalling NaN	x	x	FFFF	d18<8, rest non-zero
+/- Infinity	0,1	x	FFFF	all 0
+/- Zero	0,1	0	0000	all 0
Number	0,1	0,1	0000-9999	1-9.999999999999999999

Table 9-3: Packed decimal values

All other combinations are undefined.

9.3.6 Expanded packed decimal (EP)

	31							0
First word	sign	e6	e5	e4	e3	e12	e1	e0
Second word	d23	d22	d21	d20	d19	d18	d17	d16
Third word	d15	d14	d13	d12	d11	d10	d9	d8
	d7	d6	d5	d4	d3	d2	d1	d0

- Value is +/- d * 10^(+/- e).
- d23 and e6 are the most significant digits of d and e respectively.
- Sign contains both the number's sign (bit 31) and the exponent's sign (bit 30). The other bits (29,28) are 0.
- The value of d is arranged with the decimal point between d23 and d22, and is normalized so that for an ordinary number 1<=d23<=9.
- The guaranteed ranges for d and e are 21 and 4 digits respectively: e6, e5, e4 and d2, d1, d0 may always be zero in a particular system.
- The result is undefined if any of the packed digits is hexadecimal A through F.

Expanded packed decimal values

	Sign (top bit)	Sign (next bit)	Exponent	Digit values
Quiet NaN	x	x	FFFFFFF	d23>7, rest non-zero
Signalling NaN	x	x	FFFFFFF	d23<8, rest non-zero
+/- Infinity	0,1	x	FFFFFFF	all 0
+/- Zero	0,1	0	0000000	all 0
Number	0,1	0,1	0-9999999	1-9.99999999999999999999

Table 9-4: Expanded packed decimal values

All other combinations are undefined.

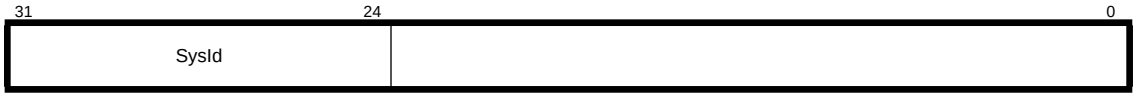
9.4 The Floating-Point Status Register (FPSR)

The floating-point status register (FPSR) consists of:

- a system ID byte
- an exception trap enable byte
- a system control byte
- a cumulative exception flags byte

Note: *The FPSR is not cleared on reset. It is typically cleared by the support code using an appropriate WFS.*

9.4.1 System ID byte



The 8-bit SysId allows a user or operating system to distinguish which floating-point system is in use. The top bit (bit 31) is:

set for HARDWARE (i.e. fast) systems

clear for SOFTWARE (i.e. slow) systems

Note: The SysId is read-only.

List of system IDs

The following system IDs are defined:

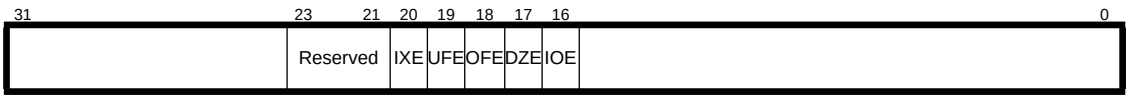
Floating-point Emulator	01 (HEX) (Software only)
FPA System	81 (HEX)

The following system IDs are also defined for backwards compatibility:

00(HEX) for pre-FPA software systems

80(HEX) for pre-FPA hardware systems

9.4.2 Exception trap enable byte

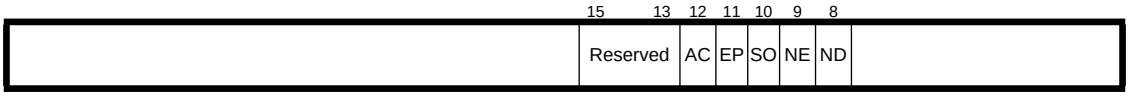


Each bit of the exception trap enable byte corresponds to one type of floating-point exception. The exception types (IX,UF,OF,DZ,IO) are described below.

A bit in the cumulative exception flags byte is set as a result of executing a floating-point instruction only if the corresponding bit *is not* set in the exception trap enable byte; if the corresponding bit in the exception trap enable byte *is* set, an exception trap will be taken instead of setting the exception flag. The trap handler code can then set the relevant cumulative exception bit if desired.

Normally, reserved FPSR bits should not be altered by user code. However, they may be initialized to zero.

9.4.3 System control byte



These control bits determine which features of the floating-point system are in use. Because these control bits are in the FPSR, their state will be preserved across context switches, allowing different processes to use different features if necessary. The following five control bits are defined for the FPA system:

Bit 8 ND - No Denormalized Numbers Bit

If this bit is set, the software forces all denormalized numbers to zero to reduce lengthy execution times when dealing with denormalized numbers. (Also known as abrupt underflow or flush to zero.) This mode is not IEEE-compatible but may be required by some programs for performance reasons. If this bit is clear, then denormalized numbers will be handled in the normal IEEE-conformant way.

Bit 9 NE - NaN Exception Bit

When this bit is clear, extended format is regarded as an internal format for conversions of signalling NaNs: only conversions between single and double-precision will produce an invalid operation exception because of a signalling NaN operand. This is required for compatibility with old programs which use STFE and LDFE to preserve register contents. When the NE bit is set, all conversions between single, double and extended precision will produce an invalid operation exception if the operand is a signalling NaN.

Bit 10 SO - Select Synchronous Operation of FPA

If this bit is set, all floating-point instructions will execute synchronously and ARM will be made to busy-wait until the instruction has completed. This will allow precise exceptions to be reported but at the expense of increased execution time. If this bit is clear, the class of floating-point instructions that can execute asynchronously to ARM will do so. Exceptions that occur as a result of these instructions may then be imprecise.

Bit 11 EP - Use Expanded Packed Decimal Format

If this bit is set, the expanded (four word) format will be used for Packed Decimal numbers. Use of this expanded format allows conversion from extended precision to packed decimal and back again to be carried out without loss of accuracy. If this bit is clear, standard (three word) format is used for Packed Decimal numbers.

Bit 12 AC - Use Alternative definition for C-flag on compare operations

If this bit is set, the ARM C-flag has the following interpretation after a compare:

```
C: Greater Than or Equal or Unordered
```

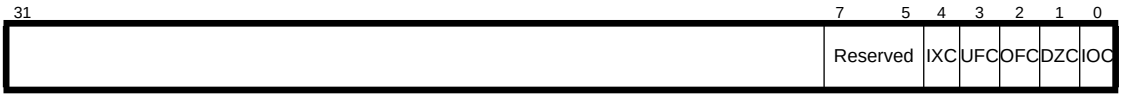
This interpretation of the C-flag allows more of the IEEE predicates to be tested by means of single ARM conditional instructions than is possible using the original interpretation of the C-flag as shown below.

If this bit is clear, the ARM C-flag has the following interpretation after a compare:

```
C: Greater Than or Equal
```

Normally, reserved FPSR bits should not be altered by user code. However, they may be initialized to zero.

9.4.4 Cumulative exception flags byte



Whenever an exception condition arises and the corresponding trap enable bit is not set, the appropriate cumulative exception flag in bits 0 to 4 will be set to 1.
If the relevant trap enable bit is set, an exception is delivered to the user's program in a manner specific to the operating system.

Note: In the case of underflow, the state of the trap enable bit determines under which conditions the underflow exception will arise.

These flags can only be cleared by a WFS instruction.

Normally, reserved FPSR bits should not be altered by user code. However, they may be initialized to zero.

IO - invalid operation

The invalid operation exception arises when an operand is invalid for the operation to be performed. The result (if the trap is not enabled) is a quiet NaN.

Invalid operations are:

- Any operation on a signalling NaN, except an LDF, LFM or SFM, or an MVF, MNF, ABS or STF without change of precision.
- Magnitude subtraction of infinities, e.g. +infinity + -infinity.
- Multiplication of 0 by an infinity.
- Division of 0/0 or infinity/infinity.
- x REM y where x is infinity or y is 0.
- Square root of any number less than zero (but SQT(-0) is -0).
- Conversion to integer when overflow, infinity or NaN make it impossible. If overflow makes a conversion to integer impossible, the largest positive or negative integer is produced (depending on the sign of the operand) and Invalid Operation is signalled.
- CMFE, CNFE when at least one operand is a NaN.

DZ - division by zero

The division-by-zero exception occurs if the divisor is zero and the dividend a finite, non-zero number. A correctly-signed infinity is returned if the trap is disabled.

OF - overflow

The OFC flag is set whenever the destination format's largest number is exceeded in magnitude by what would have been the rounded result if the exponent range were unbounded. The untrapped result returned is either:

- the correctly signed infinity
- the format's largest finite number

depending on the rounding mode.

UF - underflow

Two correlated events contribute to underflow:

1 Tininess
The creation of a tiny non-zero result smaller in magnitude than the format's smallest normalized number.

2 Loss of accuracy
A loss of accuracy due to denormalization that *may* be greater than would be caused by rounding alone.

If the underflow trap enable bit is set, the underflow exception occurs when tininess is detected, regardless of loss of accuracy. If the trap is disabled, then tininess and loss of accuracy must both be detected for the underflow flag to be set (in which case inexact will also be signalled).

IX - inexact

The inexact exception occurs if:

- the rounded result of an operation is not exact (different from the value computable with infinite precision)
- overflow has occurred while the OFE trap was disabled
- underflow has occurred while the UFE trap was disabled.

OFE or UFE traps take precedence over IXE.

9.5 The Floating-Point Control Register (FPCR)

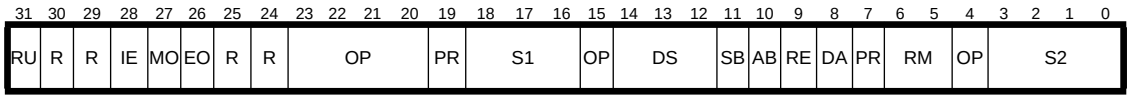
The floating-point control register (FPCR) is an implementation-specific register: it may not exist in some versions of the ARM floating-point system and, when it does exist, it may contain different information for different versions of the system.

When present, it is used for internal communication within the floating-point system and, in particular, to allow software and hardware components of the system to communicate with each other.

Use of the WFC and RFC instructions outside the floating-point system itself is strongly discouraged. In the case of User mode programs, it is actually prohibited: the WFC and RFC instructions will trap if executed in User mode.

The FPCR within the ARM7500FE has an FPCR. It is used to enable and disable the chip and to communicate information about instructions the hardware cannot handle to the support code.

The FPA FPCR bit allocation is as follows:



31	RU	Rounded-up bit
30		Reserved
29		Reserved
28	IE	Inexact bit
27	MO	Mantissa overflow
26	EO	Exponent overflow
25		Reserved
24		Reserved
23-20	OP	AU operation code
19;7	PR	AU precision
18-16	S1	AU source register 1
15	OP	AU operation code
14-12	DS	AU destination register
11	SB	Store bounce: decode (R14) to get opcode
10	AB	Arithmetic bounce: opcode supplied in rest of word
9	RE	Rounding Exception: Arithmetic bounce occurred during rounding stage and destination register was written
8	DA	Disable FPA
6-5	RM	AU rounding mode
4	OP	AU operation code
3-0	S2	AU source register 2 (bit 3 set denotes a constant)

All defined bits are cleared on reset, except bits 8, 10, and 11 (DA, AB, and SB) which are set.

Apart from by using the WFC instruction, the AB bit can only be set by the arithmetic unit and the SB bit can only be set by the load-store unit.

Only the arithmetic unit can write bits 31, 28:26, 23:12, 9, 7:0 of the FPCR.

The behavior of the FPCR when the RFC and WFC instructions are executed is as follows:

- A read of the FPCR by RFC clears the SB, AB and DA bits of the FPCR, and leaves the other bits of the FPCR unchanged.
- A write of the FPCR by WFC writes the SB, AB, & DA bits of the FPCR, and leaves the other bits of the FPCR unchanged.

Note: This information about the FPCR in the FPA is only supplied to aid with modifications to the FPA support code. Using it for any other purpose is likely to lead to compatibility problems and is strongly discouraged.

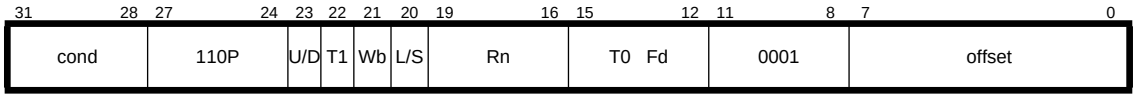
10 Floating-Point Instruction Set

This chapter lists the floating-point instruction set.

Note: Not all of the instructions detailed in this chapter are implemented in hardware on the FPA; the remainder are supported by software emulation.

10.1 Floating-Point Coprocessor Data Transfer (CPDT)

10.1.1 LDF/STF - load and store floating



Load or Store the high-precision value from or to memory, using one of the five memory formats.

On store, the value is rounded using the **round to nearest** rounding method to the destination precision, or is precise if the destination has sufficient precision. Thus, other rounding methods may be used by having applied a suitable floating-point data operation at some time before the store; this does not compromise the requirement of **rounding once only** since no additional rounding error is introduced by the store instruction.

Cond condition field

P pre/post-indexing bit:
- 0 post-indexing
- 1 pre-indexing

U/D up/down bit
- 0 down
- 1 up

T1 transfer length (see below)

Wb write-back bit

L/S load/store bit
- 0 store to memory
- 1 load from memory

Rn base register

T0 transfer length (see below)

Fd floating-point register number

offset unsigned 8-bit immediate offset

The length field is encoded into bits 22 (T1) and 15 (T0) as follows:

Precision		bit 22	bit 15	FPSR.EP	Data format size	Note
Single	S	0	0	x	1 memory word	
Double	D	0	1	x	2 memory words	
Extended	E	1	0	x	3 memory words	
Packed decimal	P	1	1	0	3 memory words	1
Expanded packed decimal	EP	1	1	1	4 memory words	1

Table 10-1: Length field

Note 1: LDFP and STFP are deprecated instructions and are intended for backwards compatibility only. These functions should be implemented by appropriate calls to a library.

The offset in bits [7:0] is specified in words and is added to (U/D=1) or subtracted from (U/D=0) a base register (Rn), either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be written back into the base register (Wb=1) or the old value of the base may be preserved (Wb=0).

Note: *Post-indexed addressing modes require explicit setting of the Wb bit, unlike LDR and STR which always write-back when post-indexed. The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.*

10.1.2 Assembler syntax

```
<LDF|STF>{cond}<S|D|E|P> Fd,[Rn]
                         [Rn, <#expression>]{!}
                         [Rn],<#expression>
```

Pre-indexed addressing specification

`[Rn]`	offset of zero
`[Rn, #<expression>]{!}`	offset of <expression> bytes
`{!}`	Write back the base register (set the Wb bit) if ! is present.

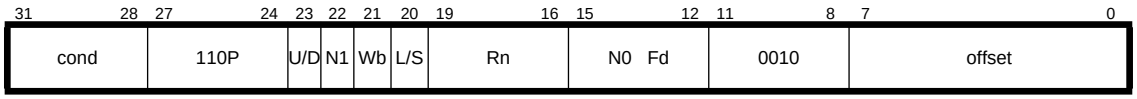
Note: *If Rn is R15, writeback should not be specified.*

Post-indexed addressing specification

`[Rn],#<expression>`	offset of <expression> bytes

Note: *The assembler automatically sets the Wb bit in this case.*
R15 should not be used as the base register where post-indexed addressing is used.
The `<expression>` *must be divisible by 4 and be in the range -1020 to 1020.*

10.1.3 Load and store multiple floating instructions (LFM/SFM)



The Load/Store Multiple Floating instructions allow between 1 and 4 floating-point registers to be transferred from/to memory in a single operation. These operations allow groups of registers to be saved and restored efficiently (e.g. across context switches).

Cond	Condition field
P	Pre/post-indexing bit:
	0 post-indexing
	1 pre-indexing
U/D	Up/down bit:
	0 down
	1 up)
N1	Register count (see below)
Wb	Write-back bit
L/S	Load/store bit
	0 store to memory
	1 load from memory
Rn	Base register
N0	Register count (see below)
Fd	Floating-point register number offset - unsigned 8-bit immediate offset

The values are transferred as three words of data for each register; the data format used is not defined (and may change in future implementations), and the only legal operation that can be performed on this data is to load it back into the FPA using the same implementation's LFM instruction. The data stored in memory by an SFM instruction should not be used or modified by any user process.

Note: *Coprocessor number 2 (bits 11-8 in the instruction field) rather than the usual FPA coprocessor number of 1 must be used for these instructions.*

The offset in bits [7:0] is specified in words and is added to (U/D=1) or subtracted from (U/D=0) a base register (Rn), either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be written back into the base register (Wb=1) or the old value of the base may be preserved (Wb=0). Note that post-indexed addressing modes require explicit setting of the Wb bit, unlike LDR and STR which always write-back when post-indexed. The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

10.1.4 Assembler syntax - form 1

```
<LFM|SFM>{cond} Fd,<count>,     [Rn]
                                [Rn, #<expression>]{!}
                                [Rn],#<expression>
```

The first register to transfer is specified as Fd.

The number of registers to transfer is specified in the <count> field and is encoded in bit 22 (N1) and bit 15 (N0) as follows:

bit 22	bit 15	No. of registers to transfer
0	1	1
1	0	2
1	1	3
0	0	4

Table 10-2: Count field

Registers are always transferred in ascending order and wrap around at register F7. For example:

```
SFM F6,4,[R0]
```

will transfer F6,F7,F0,F1 to memory starting at the address contained in register R0.

Pre-indexed addressing specification

`[Rn]`	offset of zero
`[Rn, #<expression>]{!}`	offset of <expression> bytes
`{!}`	Write back the base register (set the Wb bit) if ! is present.

Note: *If Rn is R15, writeback should not be specified.*

Post-indexed addressing specification

`[Rn],#<expression>`	offset of <expression> bytes

Note: *The assembler automatically sets the Wb bit in this case.*
R15 should not be used as the base register where post-indexed addressing is used.
The `<expression>` *must be divisible by 4 and be in the range -1020 to 1020.*

10.1.5 Assembler syntax - form 2

```
<LFM|SFM>{cond}<FD,EA> Fd,<count>,[Rn]{!}
```

This form of the instruction is intended for stacking type operations on the floating-point registers. The following table shows how the assembler mnemonics translate into bits in the instruction:

Name	Stack	L bit	P bit	U bit
post-increment load	LFMFD	1	0	1
pre-decrement load	LFMEA	1	1	0
post-increment store	SFMEA	0	0	1
pre-decrement store	SFMFD	0	1	0

Table 10-3: Assembler mnemonics

FD,EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a “full” or “empty” stack, i.e. whether a pre-index has to be done (full) before storing to the stack.

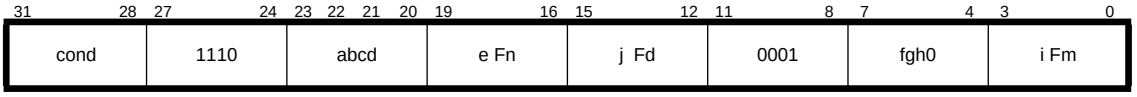
The A and D refer to whether the stack is ascending or descending. If ascending, an SFM will go up and LFM down; if descending, vice-versa.

Note: *Only EA and FD are permitted: the LFM/SFM instructions are not capable of supporting empty descending or full ascending stacks.*

`{!}` Write back the base register (set the Wb bit) if ! is present.

Note: *If Rn is R15, writeback should not be specified.*

10.2 Floating-Point Coprocessor Data Operations (CPDO)



where:

abcd opcode

j dyadic/monadic:

- 0 dyadic
- 1 monadic

ef destination size

gh rounding mode

i constant /Fm

10.2.1 Dyadic operations

```
<ADF|SUF|RSF|MUF|DVF|RDF|>{cond}<S|D|E>{P|M|Z} Fd, Fn, <Fm|#value>
<FML|FDV|FRD|RMF>
```

10.2.2 Monadic operations

```
<ABS|URD|NRM|MVF|MNF|SQT|RND>{cond}<S|D|E>{P|M|Z} Fd, <Fm|#value>
```

10.2.3 Library calls

It is recommended that the following floating-point operations are implemented with calls to an appropriate library (for example, the C library):

- power
- reverse power
- polar angle
- logarithm base 10
- logarithm base e
- exponent
- sine
- cosine
- tangent
- arc sine
- arc cosine
- arc tangent

However, for backwards compatibility with existing floating-point code, the following floating-point mnemonics are defined in the ARM floating-point instruction set. These opcodes are treated by the FPA as undefined instructions, and must be handled by support code, which is less efficient than using library calls.

```
<POW|RPW|POL> {cond} <S|D|E>{P|M|Z} Fd, Fn, <Fm|#value>
<LOG|LGN|EXP|SIN|COS|TAN|ASN|ACS|ATN> {cond}<S|D|E>{P|M|Z} Fd, <Fm|#value>
```

abcdj	Mnemonic	Description	Operation	Note
00000	ADF	Add	Fd := Fn + Fm	
00010	MUF	Multiply	Fd := Fn * Fm	
00100	SUF	Subtract	Fd := Fn - Fm	
00110	RSF	Reverse Subtract	Fd := Fm - Fn	
01000	DVF	Divide	Fd := Fn / Fm	
01010	RDF	Reverse Divide	Fd := Fm / Fn	
01100	POW	Power	Fd := Fn raised to the power of Fm	1
01110	RPW	Reverse Power	Fd := Fm raised to the power of Fn	1
10000	RMF	Remainder	Fd := IEEE remainder of Fn / Fm	
10010	FML	Fast Multiply	Fd := Fn * Fm	
10100	FDV	Fast Divide	Fd := Fn / Fm	
10110	FRD	Fast Reverse Divide	Fd := Fm / Fn	
11000	POL	Polar angle (ArcTan2)	Fd := polar angle of (Fn, Fm)	1
11010	---	trap: undefined instruction		
11100	---	trap: undefined instruction		
11110	---	trap: undefined instruction		
00001	MVF	Move	Fd := Fm	
00011	MNF	Move Negated	Fd := - Fm	
00101	ABS	Absolute value	Fd := ABS (Fm)	
00111	RND	Round to integral value	Fd := integer value of Fm	
01001	SQT	Square root	Fd := square root of Fm	
01011	LOG	Logarithm to base 10	Fd := $\log_{10}$ of Fm	1
01101	LGN	Logarithm to base e	Fd := $\log_{e}$ of Fm	1
01111	EXP	Exponent	Fd := e ** Fm	1
10001	SIN	Sine	Fd := sine of Fm	1
10011	COS	Cosine	Fd := cosine of Fm	1
10101	TAN	Tangent	Fd := tangent of Fm	1
10111	ASN	Arc Sine	Fd := arcsine of Fm	1

Table 10-4: Floating-point mnemonics

abcdj	Mnemonic	Description	Operation	Note
11001	ACS	Arc Cosine	Fd := arccosine of Fm	1
11011	ATN	Arc Tangent	Fd := arctangent of Fm	1
11101	URD	Unnormalized Round	Fd := integer value of Fm, possibly in abnormal form	
11111	NRM	Normalize	Fd := normalized form of Fm	

Table 10-4: Floating-point mnemonics

ef	suffix	Rounding precision	Note
00	S	IEEE Single precision	2
01	D	IEEE Double precision	2
10	E	IEEE Double Extended precision	2
11		trap: undefined instruction	

Table 10-5: Rounding precision

gh	suffix	Rounding Mode
00		Round to Nearest (default)
01	P	Round towards Plus Infinity
10	M	Round towards Minus Infinity
11	Z	Round towards Zero

Table 10-6: Rounding mode

Note 1: Deprecated instruction:
included for backwards compatibility only.

Note 2: The precision must be specified;
there is no default.

Note 3: These are specified when i=1.

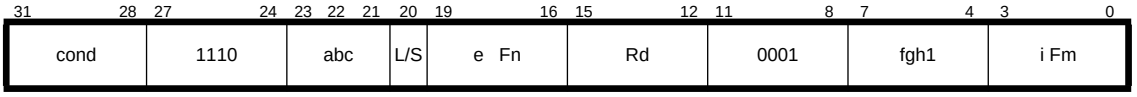
i Fm	Value assigned	Note
1000	0.0	3
1001	1.0	3
1010	2.0	3
1011	3.0	3
1100	4.0	3
1101	5.0	3
1110	0.5	3
1111	10.0	3

Table 10-7: Constants

Additional notes

- FML, FRD, FDV are only defined to work with single precision operands. It is not guaranteed that any particular implementation will execute the “fast” instructions any quicker than their respective “normal” versions (MUF, DVF, RDF).
- Directed rounding is done only at the last stage of a SIN, COS etc; the intermediate calculations to compute the value are done with round-to-nearest using the full working precision.
- The URD instruction performs the IEEE round-to-integer-value operation, but may leave its result in an abnormal unnormalized form. The NRM instruction converts this abnormal result into a proper floating-point value.
- Direct use of the result of a URD instruction by any instruction other than NRM may produce unexpected results and should therefore not be done. However, there is an exception to this rule, where a URD result may safely be preserved and restored by STFE/LDFE or SFM/LFM before being processed by NRM. So there is no need, for instance, to disable interrupts around a URD/NRM instruction sequence.
- Similarly, the NRM instruction should only be used on an URD result. Again, use of it on other values may produce unexpected results.

10.3 Floating-Point Coprocessor Register Transfer (CPRT)




```
FLT{cond}<S|D|E>{P|M|Z}  Fn, Rd
FIX{cond}{P|M|Z}         Rd, Fm
<WFS|RFS|WFC|RFC>{cond}  Rd
```

When L/S is:

1 the transfer is *to* an ARM register

0 the transfer is *from* an ARM register

abc L/S	Mnemonic	Description	Operation	Note
0000	**FLT**	Convert Integer to Floating-Point	Fn := Rd	
0001	**FIX**	Convert Floating-Point to Integer	Rd := Fm	
0010	**WFS**	Write Floating-Point Status Register	FPSR := Rd	
0011	**RFS**	Read Floating-Point Status Register	Rd := FPSR	
0100	**WFC**	Write Floating-Point Control Register	FPCR:= Rd	1
0101	**RFC**	Read Floating-Point Control Register	Rd := FPCR	1
011x		trap: undefined instruction		
1000		trap: undefined instruction		
1010		trap: undefined instruction		
1100		trap: undefined instruction		
1110		trap: undefined instruction		

Table 10-8: Coprocessor register transfer

Note 1: *Supervisor-only Instructions*

Definition of the efgh bits

The definition of the efgh bits is instruction-dependent:

FLT

ef destination size (*10.2 Floating-Point Coprocessor Data Operations (CPDO)* on page 10-7)

gh rounding mode (*10.2 Floating-Point Coprocessor Data Operations (CPDO)* on page 10-7)

FIX

ef these bits are reserved and should be zero.

gh rounding mode (*10.2 Floating-Point Coprocessor Data Operations (CPDO)* on page 10-7)

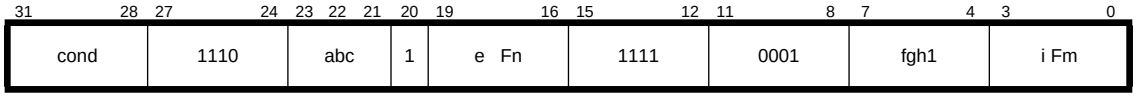
WFS,RFS,WFC,RFC

efgh these bits are reserved and should be zero.

Constants

Constants cannot be specified in the Fm field for the FIX instruction, as there is no point FIXing a known value into an ARM integer register; it would be quicker to use a MOV instruction.

10.3.1 Compare operations



Note: *These are special cases of the general CPRT instruction, with Rd = 15 and L/S = 1.*

`<CMF|CNF|CMFE|CNFE>{cond} Fn, Fm`

abc operation

i constant ROM/Fm
(see *10.2 Floating-Point Coprocessor Data Operations (CPDO)* on page 10-7)

efgh are reserved and should be zero

abc	Mnemonic	Description	Operation
100	**CMF**	Compare floating	compare Fn with Fm
101	**CNF**	Compare negated floating	compare Fn with -Fm
110	**CMFE**	Compare floating with exception	compare Fn with Fm
111	**CNFE**	Compare negated floating with exception	compare Fn with -Fm

Table 10-9: Compare operations

Compares

Compares are provided with and without the exception that could arise if the numbers are unordered. When testing IEEE predicates, the CMF instruction should be used to test for equality (i.e. when a BEQ or BNE will be used afterwards) or to test for unorderdness (in the V flag). The CMFE instruction should be used for all other tests (BGT, BGE, BLT, BLE afterwards). CMFE produces an exception if the numbers are unordered, i.e. whenever at least one operand is a NaN. CMF only produces an exception when at least one operand is a signalling NaN.

The ARM flags N, Z, C, V refer to the following after compares:

Flag	Description	Clarification
N	Less Than	Fn less than Fm (or -Fm)
Z	Equal	
C	Greater Than or Equal	Fn greater than or equal to Fm
V	Unordered	

Table 10-10: Flag settings when the AC bit in the FPSR is clear

Note: *That when two numbers are not equal N and C are not necessarily opposites: if the result is unordered they will both be false.*

Flag	Description
N	Less Than
Z	Equal
C	Greater Than or Equal or Unordered
V	Unordered

Table 10-11: Flag settings when the AC bit in the FPSR is set

Note: *In this case, N and C are necessarily opposites.*

10.4 FPA Instruction Set

The FPA and support software together implement the ARM floating-point instruction set as defined in the previous section. The FPA itself implements a subset of the instruction set.

The FPA will not however execute arithmetic instructions in *Table 10-12: Instructions implemented in FPA* on page 10-15 if one or more of the operands has one of the following exceptional values (also known as *uncommon values*):

- Infinity
- NaN (Not a Number)
- Denormalized
- Illegal extended precision bit patterns

In this case the instruction will be 'bounced' to the software support code for emulation.

10.4.1 Infinities and NaNs

Infinities and NaNs should occur very rarely in normal code. Although not common, there are a few 'normal' programs which frequently underflow and produce denormalized numbers, in which case handling of denormalized operands in software may cause a performance degradation. If necessary, this performance degradation can be minimized by setting a bit in the status register which disables support for denormalized numbers.

10.4.2 Exceptional conditions

Certain other exceptional conditions that arise during an operation will cause the FPA to transfer that operation to the support code. These conditions include all cases of the following IEEE exceptions:

- Invalid Operation
- Division by Zero
- Overflow
- Underflow

If the Inexact condition is detected, operation will only be transferred to the support code if the Inexact trap enable bit is set in the Floating-Point Status Register. Some other rare cases (such as mantissa overflow that occurs during the rounding stage of a Store Floating instruction) that do not in fact produce an IEEE exception will also trap to the support software.

Mnemonic	Instruction	IEEE Required
LDF(S/D/E)	Load (Single/Double/Extended)	*
STF(S/D/E)	Store (Single/Double/Extended)	*
ADF	Add	*
SUF	Subtract	*
RSF	Reverse Subtract	
MUF	Multiply	*
DVF	Divide	*
RDF	Reverse Divide	
FML	Fast Multiply	
FDV	Fast Divide	
FRD	Fast Reverse Divide	
ABS	Absolute	
URD	Round to Integral Value, possibly producing abnormal value	
NRM	Normalize result of URD	
MVF	Move	*
MNF	Move Negated	
FLT	Integer to floating point conversion	*
FIX	Floating-point to integer conversion	*
WFS	Write Floating-Point Status	*
RFS	Read Floating-Point Status	*
WFC	Write Floating-Point Control	
RFC	Read Floating-Point Control	
CMF	Compare Floating	*
CNF	Compare Negated Floating	
CMFE	Compare Floating with Exception	*
CNFE	Compare Negated Floating with Exception	
LFM	Load Floating Multiple (new to FPA)	
SFM	Store Floating Multiple (new to FPA)	

Table 10-12: Instructions implemented in FPA

Mnemonic	Instructions	IEEE Required
SQT	Square Root	*
RMF	Remainder	*
RND	Round to Integral Value	*

Table 10-13: Instructions supported by software support code (FPASC)

10.5 Floating-Point Support Code

Software support for the FPA includes the FPA support code (FPASC) and a software-only floating-point emulator (FPE).

The FPA system and the FPE produce identical results; both systems are fully IEEE-conformant. Both systems seamlessly implement the ARM floating-point instruction set.

The purpose of the FPASC is to:

1. Emulate in software those instructions rejected by the FPA because they involve uncommon values.
2. Provide support for exception conditions reported by the FPA.
3. Emulate in software those instructions in the floating point instruction set that are not implemented in the FPA (see list above).
4. Emulate in software any instructions that are included for backwards compatibility only; see *However, for backwards compatibility with existing floating-point code, the following floating-point mnemonics are defined in the ARM floating-point instruction set. These opcodes are treated by the FPA as undefined instructions, and must be handled by support code, which is less efficient than using library calls.* on page 10-7.

10.5.1 IEEE standard conformance

The full name of the IEEE Floating-Point Standard is as follows:

“IEEE Standard for Binary Floating-Point Arithmetic - ANSI/IEEE Std 754-1985”

This is referred to as the IEEE standard or merely as IEEE in this datasheet.

Note: *The FPA hardware on its own is not IEEE-conformant.*

Support software (the FPASC - FPA Support Code) is required to:

1. Implement the IEEE-required operations not provided by the FPA.
2. Handle operations on uncommon values which are bounced by the FPA.
3. Provide exception trap-handling capability.

10.6 Instruction Cycle Timing

The following table shows the number of cycles that the FPA takes in executing each instruction. Two numbers are given:

- the instruction latency
- the maximum instruction throughput

Instruction	Precision	No. registers	Throughput	Latency	Note
LDF/STF	S		2	3	
LDF/STF	D		3	4	
LDF/STF	E		4	5	
LFM/SFM		1	4	5	
LFM/SFM		2	7	8	
LFM/SFM		3	10	11	
LFM/SFM		4	13	14	
MVF/MNF/ABS	S/D/E		1	2	1
ADF/SUF/RSF/URD/NRM	S/D/E		2	4	
MUF	S/D/E		8	9	
FML	S/D/E		5	6	
DVF/RDF/FDV/FRD	S		30	31	2
DVF/RDF/FDV/FRD	D		58	59	2
DVF/RDF/FDV/FRD	E		70	71	2
FLT	S/D/E		6	8	
FIX			8	9	
CMF/CMFE/CNF/CNFE			5	6	
RFS/RFC			3	4	3
WFS/WFC			3	3	

Table 10-14: Instruction cycle timing

Notes:

1 Cannot be sustained for more than 2 cycles out of every 3 cycles.

2 May be less if the division comes out exactly, causing *early termination* of the division algorithm (minimum of 6 cycles throughput, 7 cycles latency).

3 The latency may be 2 or 3 cycles, depending on the previous instruction.

Throughput

Throughput is the number of cycles between the start of an instruction and the start of a succeeding instruction of the same type, both instructions occurring in a long sequence of instructions of the same type. To achieve the quoted throughput, register dependencies and anti-dependencies must be avoided.

Latency

Latency is the number of cycles between the start of instruction execution and its completion. The number of cycles taken by a sequence of floating point instructions, each of which depends on the result of the preceding instruction in the sequence, can generally be found by adding the latencies of the individual instructions. There may be minor discrepancies from this rule for particular sequences.

The exact definition is dependent on the type of instruction being executed:

Arithmetic instructions	From register read to register write.
LDF, LFM, FLT	From start of instruction arbitration to register write.
STF, SFM, CMF, FIX	From register read to start of next instruction arbitration.
WFS, WFC	From start of instruction arbitration until the next instruction would be deemed to start by these rules.
RFS, RFC	From the time that the previous instruction would be deemed to end by these rules to the start of the next instruction arbitration.

Note: *Speculative execution, concurrent execution between arithmetic and load/store instructions and concurrent execution between ARM integer instruction and FPA instructions can significantly reduce the effective timings shown.*

10.6.1 Instruction classification

Instructions can be classified into **arithmetic**, **load/store** and **joint** instructions:

Arithmetic	Those instructions that execute completely within the arithmetic unit. These include all the hardware-implemented coprocessor data operations (see *10.2 Floating-Point Coprocessor Data Operations (CPDO)* on page 10-7).
Load/store	Those instructions that execute completely within the load/store unit. These include LDF, STF, LFM and SFM.

Joint arithmetic and load/store instructions

FIX, CMF,CNF,CMFE,CNFE	Arithmetic followed by load/store.
FLT	Load/store followed by arithmetic.
WFS,RFS,WFC,RFC	Occupy both arithmetic and load/store units, since the arithmetic unit must be empty before any of these instructions may be executed.

10.6.2 Performance tuning

The FPA is capable of executing load/store and arithmetic instructions concurrently and is also capable of executing instructions speculatively - i.e. before they have been committed to execution by the ARM CPU. Both of these features can be exploited to maximize the performance of the FPA. The code fragment shown below is a good example of how this can be achieved:

```
1   SFM  F0,4,[R0],#48
2   DVFS F0,F1,#3
3   SFM  F4,4,[R0],#48
4   MOV  R1,R2
5   MOV  R3,R4
```

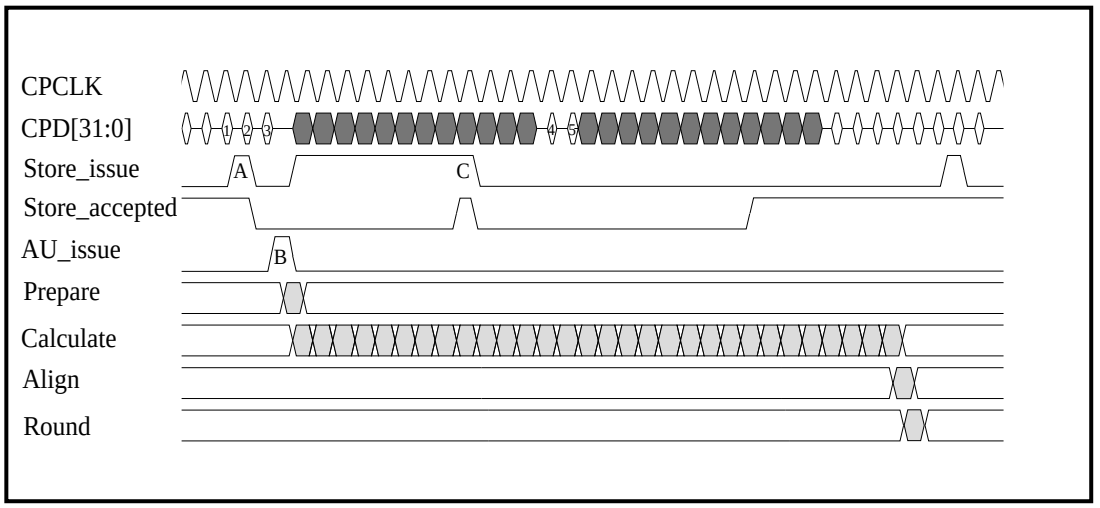


Figure 10-1: Performance tuning

The labels 1, 2, 3, 4 & 5 indicate the cycles in which these instructions are fetched on the CPD[31:0] bus, while A, B & C indicate the cycles in which the floating-point instructions are issued to their respective units in the FPA.

The first store multiple instruction (1) is issued (A) to the load/store unit, resulting in 12 words of data being transferred on CPD[31:0] as shown by the shaded boxes on the timing diagram. Meanwhile, the divide instruction (2) is issued (B) to the arithmetic unit (AU), which then begins execution speculatively; its progress through the Prepare, Calculate, Align and Round stages of the AU pipeline is shown by the shaded boxes on the timing diagram.

The second SFM instruction (3) is issued (C) to the load/store unit as soon as it is ready. This second SFM executes while the AU is still busy on the divide instruction; the second set of shaded boxes on the CPD[31:0] bus indicates the 12 words of data being transferred for the second SFM instruction. This example shows how the divide instruction's execution time can effectively be hidden by other instructions.

Note: *The concurrency between ARM integer unit execution and FPA execution can also be exploited. Contact ARM Ltd. for further details on optimizing floating-point code for the FPA.*