Appendix B The Floating Point Instruction Set

Appendix A described the generic form which co-processor instructions take. In this appendix, we look at a specific use of these instructions: the floating point unit.

The definition given here does not describe any particular hardware. Rather, it gives a programmer's view of a system which could be implemented in a number of ways. In the first instance, the instruction set of the FPU has been implemented entirely in software. All of the instructions are trapped through the unimplemented instruction vector, and the floating point emulator package uses this facility to interpret the operation codes (using native ARM instructions) and perform the appropriate operations.

At the other end of the spectrum, a complete hardware implementation of the FPU would recognise the whole floating point instruction set, so the unimplemented instruction trap would never be used. The ARM's only involvement in dealing with the instructions would be in the generation of addresses for data transfers and providing or receiving values for register transfers.

The only difference, from the programmer's point of view, between the two implementations would be the speed at which the instructions are executed. The only reason for using a hardware solution is that this can provide a speed increase of hundreds of times over the interpreted implementation.

The FPU performs arithmetic to the IEEE 754 standard for floating point.

B.1 Some terminology

In describing the FPU's instruction set, we have to use certain terms which are specific to floating point arithmetic. These are explained briefly in this section. For a more detailed discussion, you should see the appropriate IEEE standard document (ANSI/IEEE 754-1985), or manufacturer's data on one of the current available FPUs (e.g. the Motorola MC68881 or Western Digital WE32206).

Precision

The instruction set includes three precisions to which calculations may be performed. The precision of a floating point number is the number of digits which can be stored with total accuracy. It is expressed in bits. In Chapter Five, we talked about the mantissa of a floating number being the part which stored the significant digits, and the exponent being the scaling factor applied to it.

When we say a given floating point representation has n bits of precision, this means that the mantissa part is stored in n bits. The three standard IEEE precisions are 24 bits, 53 bits and 65 bits. These are known as single, double, and extended precision respectively. The mantissa is stored in one fewer bits than the precision implies because numbers are held in normalised form (as in BBC BASIC) and the MSB is an implicit 1. The binary point comes *after* this.

There is a fixed correspondence between the precision of a number expressed in bits, and how many decimal digits that number can represent. Mathematically speaking, an n-bit number can hold n*LOG(2)/LOG(10) decimal digits, or more simply, n*0.3010. Thus the three IEEE floating points types can accurately represent the following number of decimal digits:

Type	Precision	Decimal digits
Single	24	7.2
Double	53	15.95
Extended	65	19.6

Obviously you cannot have fractional numbers of significant digits, so the values should be truncated to 7, 15 and 19 respectively. The fractional part means that one extra digit can be stored, but this cannot go all the way up to 9.

There is one further form of representation for floating point numbers: packed decimal. In this form, the decimal digits of the mantissa and exponent are represented in a coded form, with four bits representing each decimal digit (only the combinations 0000 to 1001 being used). In this form, called packed decimal, the mantissa is stored as 19 four-bit 'digits', and the exponent as 4 four-bit digits.

Just because a number can be represented exactly in decimal, do not assume that its binary representation is also exact. For example, the decimal fraction 0.1 is actually a recurring fraction in binary, and can never be stored exactly (though of course, when enough digits are used in the mantissa, the error is very small).

Dynamic range

The mantissa has an imaginary binary point before after its first digit, and the first digit is always a binary 1, never 0. This is known as 'normalised form'. Thus the mantissa stands for a fraction between 1.0 and 1.99999... The exponent provides the power of two by which the mantissa has to be multiplied to obtain the desired value.

To obtain numbers greater than 1.999999... a positive exponent is used, and to represent numbers smaller than 1.0, a negative exponent is used (meaning that the mantissa is divided by a power of two instead of multiplied). The exact range of numbers depends on the maximum power of two by which the mantissa may be multiplied or divided. This is known as the dynamic range of the representation.

Each of the precisions uses a different size of exponent, as follows:

Precision	Exponent bits	Smallest	Largest
Single	8	2^{-126}	2^{127}
Double	11	2^{-1022}	2^{1023}
Extended	15	2^{-16382}	2^{16383}

The table shows how many bits the exponent uses, and the smallest and largest factors by which the mantissa may be multiplied. The formula n*0.3010 can also be used to find out what power of ten these exponents correspond to. They are +/-39, +/-307 and +/-4931 respectively. Thus a single precision IEEE number could represent the number 1.234E12 but not 1.234E45. This would require a double precision number.

The exponent is held in an excess-n format, as with BBC BASIC. The excess number is 127, 1023 and 16383 for the three precisions respectively. Note that there is an exponent value at each end of the range (e.g. -127 and +128, stored as 0 and 255, in single precision) which is not used. These values are used to represent special numbers.

Rounding

When the FPU performs its calculations, more digits are used than are necessary to store the numbers involved. Calculations are performed in what is called 'full working precision'. This is so that any errors which accumulate due to non-exact representations of fractions do not affect the final result. When a result is presented, the FPU converts the special full working form into a value of the desired precision. The way in which this conversion is performed is called the 'rounding mode' of the calculation, and there are four of them to choose from.

'Round to nearest' means that the final result is the closest number in the selected precision to the internal version. This is used as the default mode for ARM FPU instructions.

'Round to zero' means that the extra bits of precision are effectively ignored, so the final result is the one which is closest to zero.

'Round to plus infinity' means that the final result is the first number which is greater than the 'exact' result which can be stored in the required precision.

'Round to minus infinity' means that the final result is the first number which is less than the 'exact' result which can be stored in the required precision.

These four modes can be illustrated by using decimal numbers. Suppose that the calculations are performed to nine digits precision, and the final precision is seven digits:

```
Mode             'Exact' result          Rounded result

Nearest           0.123456789             0.1234568
                 -0.123456789            -0.1234568

To zero           0.123456789             0.1234567
                 -0.123456789            -0.1234567

To +infinity      0.123456789             0.1234568
                 -0.123456789            -0.1234567

To -infinity      0.123456789             0.1234567
                 -0.123456789            -0.1234568
```

Special values

In addition to valid numeric values, the IEEE standard also defines representations for values which may arise from errors in calculations. These values are 'not a number' (NAN), plus infinity (+INF) and minus infinity (-INF). There are actually two types of NAN, trapping and non-trapping. Trapping is described in the next section. The ways in which these special values may arise are also described there.

Note that IEEE defines two representations for zero, positive and negative zero. Usually the distinction between them is not important, but sometimes a result will depend on the sign of the zero used (see the DVZ trap below for an example).

B.2 Programmer's model

To the programmer, the FPU looks like a cut-down version of the ARM CPU. There are eight general purpose registers, called F0 to F7, a status register and a control register. There is no program counter; the ARM controls all interaction between a co-processor and memory. It is responsible for generating addresses for instruction fetches and data transfers.

Whereas the ARM's own 16 registers are 32 bits wide, there is no definition about how wide the FPU's registers are. The IEEE standard specifies several levels of precision to which operations may be performed, as described above. All the programmer needs to know about the FPU registers is that they are wide enough to maintain numbers to the highest precision required by the standard.

However, when floating point numbers are transferred between the FPU and memory, their representation becomes important. The formats of the various types of floating number are described in section B.6.

The FPU status register

The FPU's status register is 32-bits wide, and contains three fields. These are the status flags, the interrupt masks, and a system id field. There are five flags, and each of these represents a specific error condition which can arise during floating point operations. Each flag has a corresponding interrupt mask. When an error condition arises, the FPU checks the state of the interrupt mask for that error. If the interrupt is enabled, a trap is generated, which causes the program to halt. The appropriate status flag is set, so that the trap handler can determine the cause of the error.

If the interrupt for the error condition is disabled, the status flag still gets set, but execution of the program is not affected. A special result, e.g. NAN or INF is returned.

Note that the way in which an error interrupt is implemented depends on the system in which the program is executing. Software and hardware FPUs will have different ways of stopping the program.

The flags are set if the appropriate condition has arisen, and cleared if not. The masks are set to enable the interrupt, and cleared to disable it. There is an FPU instruction which is used to initialise the status register to a known state.

Here is the layout of the status register:

bit 0	IVO flag	
bit 1	DVZ flag	
bit 2	OFL flag	
bit 3	UFL flag	
bit 4	INX flag	
bits 5 to 15	Unused	(These are read as zero)
bit 16	IVO mask	
bit 17	DVZ mask	
bit 18	OFL mask	
bit 19	UFL mask	
bit 20	INX mask	
bits 21 to 23	Unused	(These are read as zero)
bits 24 to 31	System id	(These are 'read-only')

The meanings of the three-letter abbreviations are as follows:

IVO - Invalid operation. There are several operations which are deemed 'invalid', and each of these sets the IVO flag, and causes the instruction to be trapped if enabled. If the trap is not enabled, an operation which causes the IVO flag to be set returns NAN as the result. Invalid operations are:

Any operation on a NAN
Trying to 'cancel' infinities, e.g. -INF plus +INF
Zero times +INF or -INF
0/0 or INF/INF
INF **REM** anything or anything **REM** 0
SQT(<0)
Converting INF, NAN or too large a number to integer
ACS(>1), **ASN**(>1)
SIN(INF),COS(INF), TAN(INF)
LOG(<=0), LGN(<=0)
Writing to the unused or system id bits of the status register

where >1 means 'a number greater than one' etc, and INF means either +INF or -INF. Note that in the case of converting an INF or too large a number to an integer, the result (if the error is not trapped) is the largest integer of the appropriate sign which can be represented in 32 bits two's complement.

DVZ - Divide by zero. This is caused by trying to divide a non-zero number by zero. The result, if the error is not trapped, is an infinity of the appropriate sign (e.g. -1/0 gives -INF, -32.5/-0 gives +INF).

OFL - Overflow. This occurs when a result is being rounded to the specified precision for the operation. If it is impossible to legally represent the number, a trap occurs. If the trap is disabled, the result depends on the rounding mode specified in the operation:

Nearest	Appropriately signed INF
To zero	Largest representable number of appropriate sign
To +INF	Negative overflows go to largest negative number Positive overflows go to +INF
To -INF	Negative overflows go to -INF Positive overflows go to largest positive number

UFL - Underflow. This occurs when a number becomes too small (i.e. too close to zero) to represent properly in the specified precision. If the error is not trapped, the result is affected by the rounding mode as follows:

Nearest	Appropriately signed zero
To zero	Appropriately signed zero
To +INF	Negative underflows go to -0 Positive underflows go to smallest positive number
To -INF	Negative underflows go to smallest negative number Positive underflows go to +0

INX - Inexact. This error occurs whenever a rounded result is obtained which is different from that which would have been calculated if 'infinite' working precision was used. Calculating the sine, cosine or tangent of an angle greater than 10E20 radians causes this error. Also, all OFL errors automatically cause this error, so if the OFL trap is disabled but the INX trap is enabled, an overflow will cause the inexact interrupt to be generated.

System id - These eight bits may be used to determine which type of FPU the system is running. The only one currently defined is bit 31:

bit 31 = 1	implies	hardware FPU
bit 31 = 0	implies	software FPU

The remaining eight bits are reserved by Acorn for further distinction between different FPU types. In the first systems they are all zero.

The FPU control register

The control register is a system dependent entity. It is present to enable programs to, for example, disable a hardware FPU. Its format is dependent on the characteristics of the FPU hardware; as of this writing, it is undefined. Note that the instructions which read and write the control register are privileged. An attempt to access it from a user mode program will cause an error.

B.3 The FPU data processing instructions

There are two classes of FPU data processing instructions, monadic and dyadic. These terms refer to whether the instruction operates on one or two operands respectively. The instruction formats are:

```
MNM{cond}<P>{R}    <dest>,<rhs>
MNM{cond}<P>{R}    <dest>,<lhs>,<rhs>
```

where:
MNM is one of the mnemonics listed below
{cond} is the optional condition
<P> is the required precision, being one of:

S	single precision
D	double precision
E	extended precision

Note that the precision is not optional - it must be given.

{R} is the required rounding mode for the stored result, being one of:

	Round to nearest
P	Round to +INF
M	Round to -INF
Z	Round to zero

This is optional, rounding to nearest being the default if it is omitted.

<dest> is the FPU register to hold the result. The standard names for the FPU registers are F0-F7, and assemblers which recognise the FPU instructions allow other names to be assigned, as for the normal ARM registers.

<lhs> (dyadic only) is the left hand side of the operation. It is an FPU register

<rhs> is the right hand side of a dyadic operation, or the argument of a monadic operation. It is an FPU register, or a small floating point constant. These are the allowed values:

```
0.0    1.0    2.0    3.0
4.0    5.0    0.5    10.0
```

As usual, the immediate operand is preceded by a # sign.

The available dyadic operations are:

ADF	add	<dest> = <lhs> + <rhs>
MUF	multiply	<dest> = <lhs> * <rhs>
SUF	subtract	<dest> = <lhs> - <rhs>
RSF	reverse sub.	<dest> = <rhs> - <lhs>
DVF	divide	<dest> = <lhs> / <rhs>
RDF	reverse div.	<dest> = <rhs> / <lhs>
POW	power	<dest> = <lhs> ^ <rhs>
RPW	reverse power	<dest> = <rhs> ^ <lhs>
RMF	remainder	<dest> = <lhs> REM <rhs>
FML	fast multiply	<dest> = <lhs> * <rhs>
FDV	fast divide	<dest> = <lhs> / <rhs>
FRD	fast rev. div.	<dest> = <rhs> / <lhs>
POL	polar angle	<dest> = arctan(<lhs>/<rhs>)

Notes:

Reverse power gives a base ten antilog function:

```
        RPWS     F0,F0,#10.0      ;F0 = 10^F0
```

REM <lhs>,<rhs> gives the remainder of the division of **<lhs>** by **<rhs>**. In other words, **a REM b = a-b*n**, where **n** is the nearest integer to **a/b**.

The 'fast' operations **FML**, **FDV** and **FRD** are performed in single precision, rather than full working precision used for the other calculations. This means that rounding to any precision will always yield a result which is only as accurate as a single precision number.

The **POL** operation is an alternative to the **ATN** operation described below. The two operands may be regarded as the 'opposite' and 'adjacent' lengths (signed) of the angle whose value is required. This function will return a result in the range -PI to +PI radians, whereas the standard **ATN** function only gives a result in the -PI/2 to +PI/2 range.

The monadic operations are:

MVF	move	<dest> = <rhs>
MNF	move negated	<dest> = -<rhs>
ABS	absolute value	<dest> = ABS(<dest>)
RND	integer value	<dest> = roundToInteger(<rhs>)
SQT	square root	<dest> = SQR(<rhs>)
LOG	log base 10	<dest> = LOG10(<rhs>)
LGN	log base e	<dest> = LN(<rhs>)

```
EXP    e to a power      <dest> = e ^ <rhs>
SIN    sine              <dest> = SIN(<rhs>)
COS    cosine            <dest> = COS(<rhs>)
TAN    tangent           <dest> = TAN(<rhs>)
ASN    arcsine           <dest> = ATN(<rhs>)
ACS    arccosine         <dest> = ACS(<rhs>)
ATN    arctangent        <dest> = ATN(<rhs>)
```

The argument of the **SIN**, **COS** and **TAN** is in radians, and **ASN**, **ACS** and **ATN** return a result in radians. The 'transcendental' functions, from **SQT** onwards, use 'to nearest' rounding for intermediate calculations performed in full working precision, only applying the specified rounding mode as the last operation.

Examples:

```
MUFS   F0,F1,#10.0    ;F0=F1*10, single precision,nearest
POWDZ  F2,F3,F4       ;F2=F3^F4, double precision,zero
SINE   F7,F0          ;F7=SIN(F0), extended, to nearest
EXPE   F0,#1.0        ;F0='e' as an extended constant
```

B.4 The FPU/ARM register transfer instructions

The FPU uses the **FRC**/**FCR** class of instructions for three purposes: to convert between integer and floating point values, to examine and change the FPU status and control registers, and to perform floating point comparisons.

FIX and **FLT** are the type-conversion instructions. They have the following forms:

```
FIX{cond}<P>{R}   <ARM dest>,<FPU srce>
FLT{cond}<P>{R}   <FPU dest>,<ARM srce>
```

<ARM dest> is R0-R14 (R15 not being useful; see Appendix A)
<ARM srce> is R0-R14 (R15 not being useful)
<FPU dest> is F0-F7
<FPU srce> is F0-F7

The **FIX** operation may result in an **IVO** trap occurring, as explained above.

Examples:

```
FIXNES R0,F2  ;Convert F2 to integer in R0 if NE
FLTSZ  F1,R0  ;Convert R0 to F1, round to zero
```

The instructions to read and write the FPU status register are:

```
RFS{cond} <ARM dest>
WFS{cond} <ARM srce>
```

`<ARM dest>` and **`<ARM srce>`** are the ARM register to or from which the 32-bit FPU status register is transferred. The format of this register is described above. After an **RFS** the ARM register will contain the state of the status flags and interrupt masks. The unused bits will be set to zero, and the system id part will be as described above. A program can clear the flags and set the desired mask bits using **WFS**. Zeros should be placed in the unused bits and the system id bits.

The instructions to read and write the FPU control register are:

```
RFC{cond} <ARM dest>
WFC{cond} <ARM srce>
```

The format of this register is system dependent. These are privileged instructions and will abort if an attempt is made to execute them in user mode.

Examples:

```
RFS R4
WFC R0
```

The FPU compare instructions have the form:

```
MNM{cond}<P>{R}   <lhs>,<rhs>
```

where all terms are as described in section B.3. The mnemonics are:

```
CMF    compare          <lhs> - <rhs>
CNF    compare negated  <lhs> + <rhs>
CMFE   compare          <lhs> - <rhs>
CNFE   compare negated  <lhs> + <rhs>
```

The versions with the **E** suffix generate an exception if the operands are 'unordered' (when at least one operand is a NAN) and thus can't be compared. After the instructions, the ARM flags are set as follows:

N	set if 'less than' else clear
Z	set if 'equal' else clear
C	set if 'greater or equal' else clear
V	set if 'unordered' else clear

Note that if **V**=1, then **N**=0 and **C**=0.

According to the IEEE standard, when testing for equality or for unorderedness, where the next instruction condition will be **EQ**, **NE**, **VS** or

VC you should use **CMF** (or **CNF**). To test for other relationships (**GT**, **LE** etc.) you should use **CMFE** (or **CNFE**).

Examples:

```
    CMFE   F0,#0.0    ;See if F0 is 0.0, extended precision
    CMFEE  F1,F2      ;Compare F1, F2 using extended
                      ;precision with disordered exception
    CNFS   F3,#1.0    ;Compare single precision F3 with -1
```

B.5 The FPU data transfer instructions

This final group is used to transfer floating point numbers, in the various formats, between main memory and the FPU. The instructions are **LDF** to load a floating point number, and **STF** to store one:

```
    LDF{cond}<P>  <FPU dest>,<address>
    STF{cond}<P>  <FPU srce>,<address>
```

where:
<P> is as already described, with the additional option of

P	packed decimal form

The FPU registers are F0-F7

the **<address>** can take any of the forms described in Appendix A, viz.

```
    <expression>        PC-relative offset calculated
    [<base>]
    [<base>,<offset>]{!}
    [<base>],<offset>
```

where:
<base> is R0-R14
<offset> is #**{+|-}<expression>**
and **{!}** specifies optional write-back.

When a value is stored using **STF**, it is rounded using 'to nearest' rounding to the precision specified in the instruction. (**E** and **P** will always store results without requiring rounding.) If some other rounding method is required, this can be done first using a **MVF** instruction from the register to itself, using the appropriate rounding mode.

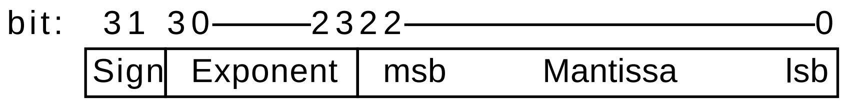
If an attempt is made to store a trapping NAN value, an exception will occur if the IVO trap is enabled, otherwise the value will be stored as a non-trapping NAN. OFL errors can occur on stores if the number is too large to be converted to the specified format.

Examples:

```
STFP   F0,[R0]    ;Store F0 in packed format at [R0]
LDFE   F0,pi      ;Load constant from label 'pi'
STFS   F1,[R2],#4 ;Store single prec. number, inc.  R2
```

B.6 Formats of numbers

Each of the four precisions has its own representation of legal numbers and special values. These are described in this section.



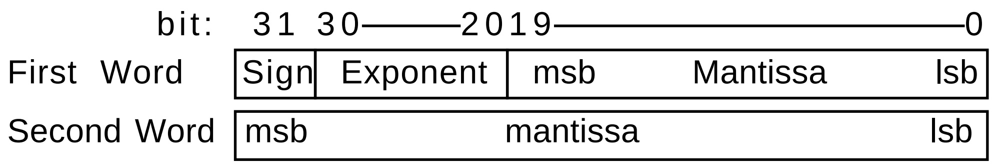
Single precision

Exponent = eight bits, excess-127
Mantissa = 23 bits, implicit 1. before bit 22

Formats of values

	Sign	Exponent	Mantissa
Non-trapping NAN	x	Maximum	1xxxxxxxxxxxxx...
Trapping NAN	x	Maximum	0<non zero>
INF	s	Maximum	00000000000000...
Zero	s	0	00000000000000...
Denormalised number	s	0	<non zero>
Normalised number	s	Not 0/Max	xxxxxxxxxxxxxx...

where:
x means 'don't care'
s means 1 for negative, 0 for positive (number, zero or INF)

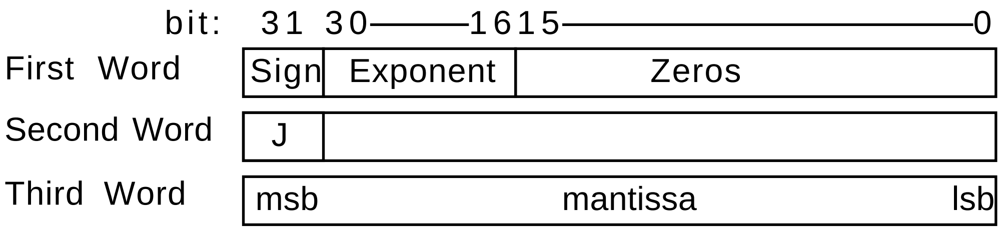


Maximum is 255 (for NANs etc)

Double precision

Exponent = 11 bits, excess-1023.
Mantissa = 52 bits, implied 1. before bit 19
Formats as above

Extended precision

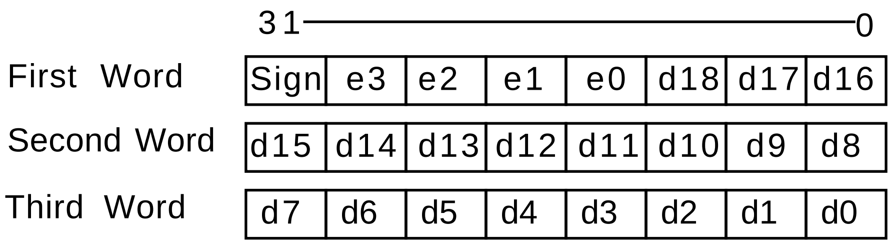


Exponent = 15 bits, excess-16383
Mantissa = 64 bits
Maximum exponent (for NANs etc) = 32767

NB The relative positions of the three parts of the first word had not been finalised as of this writing.

Format of values

	Sign	Exponent	J	Mantissa
Non-trapping NAN	x	Maximum	x	1xxxxxxxxxxxxx...
Trapping NAN	x	Maximum	x	0<non zero>
INF	s	Maximum	0	00000000000000...
Zero	s	0	0	00000000000000...
Denormalised number	s	0	0	<non zero>
Un-normalised numbr	s	Not 0/Max	0	xxxxxxxxxxxxxx...
Normalised number	s	Not 0/Max	1	xxxxxxxxxxxxxx...



Packed decimal

Each field is four bits
e0-e3 are the exponent digits
d0-d18 are the mantissa digits
Bit 31 of the first word is the sign of the number
Bit 30 of the first word is the sign of the exponent

Format of values

	bit 31	30	e3-e0	d18-d0
Non-trapping NAN	x	x	&FFFF	d18>7, rest non 0
Trapping NAN	x	x	&FFFF	d18<=7, rest non 0
INF	s	x	&FFFF	&00000000000000...
Zero	0	0	&0000	&00000000000000...
Number	s	s	&0000-&9999	&1-&99999999999...

Note that when -0 is stored in this format it is converted to +0.