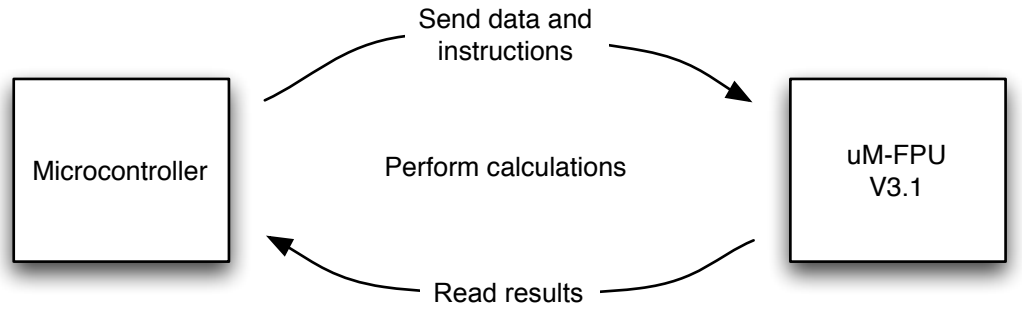
Micromega Corporation

uM-FPU V3.1 Instruction Set

32-bit Floating Point Coprocessor

Introduction

The uM-FPU V3.1 floating point coprocessor provides instructions for working with 32-bit IEEE 754 compatible floating point numbers and 32-bit long integer. A typical calculation involves sending instructions and data from the microcontroller to the uM-FPU, performing the calculation, and transferring the result back to the microcontroller.



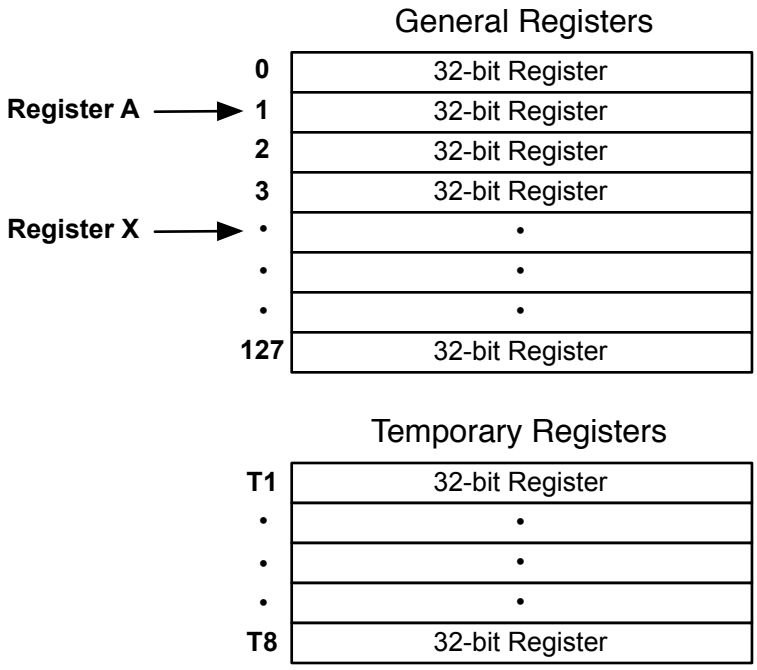
Instructions and data are sent to the uM-FPU using either a SPI or I^2C interface. The uM-FPU V3.1 chip has a 256 byte instruction buffer which allows for multiple instructions to sent. This improves the transfer times and allows the microcontroller to perform other tasks while the uM-FPU is performing a series of calculations. Prior to issuing any instruction that reads data from the uM-FPU, the Busy/Ready status must be checked to ensure that all instructions have been executed. If more than 256 bytes are required to specify a sequence of operations, the Busy/Ready status must be checked at least every 256 bytes to ensure that the instruction buffer does not overflow. See the datasheet for more detail regarding the SPI or I^2C interfaces.

Instructions consist of an single opcode byte, optionally followed by addition data bytes. A detailed description of each instruction is provided later in this document, and a summary table is provided in Appendix A.

For instruction timing, see Appendix B of the uM-FPU V3.1 Datasheet.

uM-FPU Registers

The uM-FPU V3.1 contains 128 general purpose registers, and 8 temporary registers. All registers are 32-bits and can be used to store either floating point or long integer values. The general purpose registers are numbered 0 to 127, and can be directly accessed by the instruction set. The eight temporary registers are used by the `LEFT` and `RIGHT` parenthesis instructions to store temporary results and can't be accessed directly. Register 0 is normally only used to store temporary values, since it is modified by many instructions.



Register A

To perform arithmetic operations, one of the uM-FPU registers is selected as register A. Register A can be regarded as the accumulator or working register. Arithmetic instructions use the value in register A as an operand and store the results of an operation in register A. Any register can be selected as register A using the `SELECTA` instruction. For example,

```
SELECTA,5     select register 5 as register A
```

Arithmetic instructions that only involve one register implicitly refer to register A. For example,

```
FNEG          negate the value in register A
```

Arithmetic instructions that use two registers will specify the second register as part of the instruction. For example,

```
FADD,4        add the value of register 4 to register A
```

Register X

Register X is used to reference a series of sequential registers. The register X selection is automatically incremented to the next register in sequence by all instructions that use register X. Any register can be selected as register X using the `SELECTX` instruction. For example,

```
SELECTX,16   select register 16 as register X
CLRX         clear register 16 (and increment register X)
CLRX         clear register 17 (and increment register X)
CLRX         clear register 18 (and increment register X)
```

Another example would be to use the `FWRITEX` and `READX` instructions to store and retrieve blocks of data.

In this document the following abbreviations are used to refer to registers:

reg[0]	register 0
reg[A]	register A
reg[X]	register X
reg[nn]	any one of the 128 general purpose registers

Floating Point Instructions

The following descriptions provide a quick summary of the floating point instructions. Detailed descriptions are provided in the next section.

Basic Floating Point Instructions

Each of the basic floating point arithmetic instructions are provided in three different forms as shown in the table below. The FADD instruction will be used as an example to describe the three different forms of the instructions. The FADD,nn instruction allows any general purpose register to be added to register A. The register to be added to register A is specified by the byte following the opcode. The FADD0 instruction adds register 0 to register A and only requires the opcode. The FADDB instruction adds a small integer value the register A. The signed byte (-128 to 127) following the opcode is converted to floating point and added to register A. The FADD,nn instruction is most general, but the FADD0 and FADDI,bb instructions are more efficient for many common operations.

Register nn	Register 0	Immediate value	Description
FSET,nn	FSET0	FSETI,bb	Set
FADD,nn	FADD0	FADDI,bb	Add
FSUB,nn	FSUB0	FSUBI,bb	Subtract
FSUBR,nn	FSUBR0	FSUBRI,bb	Subtract Reverse
FMUL,nn	FMUL0	FMULI,bb	Multiply
FDIV,nn	FDIV0	FDIVI,bb	Divide
FDIVR,nn	FDIVR0	FDIVRI,bb	Divide Reverse
FPOW,nn	FPOW0	FPOWI,bb	Power
FCMP,nn	FCMP0	FCMPI,bb	Compare

Loading Floating Point Values

The following instructions are used to load data from the microprocessor and store it on the uM-FPU as 32-bit floating point values.

FWRITE,nn,b1,b2,b3,b4	Write 32-bit floating point value to reg[nn]
FWRITEA,b1,b2,b3,b4	Write 32-bit floating point value to reg[A]
FWRITEX,b1,b2,b3,b4	Write 32-bit floating point value to reg[X]
FWRITE0,b1,b2,b3,b4	Write 32-bit floating point value to reg[0]
WRBLK, tc, t1…tn	Write multiple 32-bit values
ATOF,aa…00	Convert ASCII string to floating point value and store in reg[0]
LOADBYTE,bb	Convert signed byte to floating point and store in reg[0]
LOADUBYTE,bb	Convert unsigned byte to floating point and store in reg[0]
LOADWORD,b1,b2	Convert signed 16-bit value to floating point and store in reg[0]
LOADUWORD,b1,b2	Convert unsigned 16-bit value to floating point and store in reg[0]
LOADUWORD,b1,b2	Convert unsigned 16-bit value to floating point and store in reg[0]
LOADE	Load the value of e (2.7182818) to reg[0]
LOADPI	Load the value of pi (3.1415927) to reg[0]

Reading Floating Point Values

The following instructions are used to read floating point values from the uM-FPU.

`FREAD,nn [b1,b2,b3,b4]`	Return 32-bit floating point value from reg[nn]
`FREADA [b1,b2,b3,b4]`	Return 32-bit floating point value from reg[A]
`FREADX [b1,b2,b3,b4]`	Return 32-bit floating point value from reg[X]
`FREAD0 [b1,b2,b3,b4]`	Return 32-bit floating point value from reg[0]
`RDBLK,tc [t1…tn]`	Read multiple 32-bit values
`FTOA,bb`	Convert floating point to ASCII string (use `READSTR` to read string)

Additional Floating Point Instructions

`FSTATUS,nn`	`LOG`	`ACOS`	`ROUND`
`FSTATUSA`	`LOG10`	`ATAN`	`FMIN,nn`
`FCMP2,nn,mm`	`EXP`	`ATAN2,nn`	`FMAX,nn`
`FNEG`	`EXP10`	`DEGREES`	`FCNV,bb`
`FABS`	`SIN`	`RADIANS`	`FMAC,nn,mm`
`FINV`	`COS`	`FMOD`	`FMSC,nn,mm`
`SQRT`	`TAN`	`FLOOR`	`FRACTION`
`ROOT,nn`	`ASIN`	`CEIL`	

Matrix Instructions

`SELECTMA,nn,b1,b2`	select matrix A at register nn of size b1 rows x b2 columns
`SELECTMB,nn,b1,b2`	select matrix B at register nn of size b1 rows x b2 columns
`SELECTMC,nn,b1,b2`	select matrix C at register nn of size b1 rows x b2 columns
`LOADMA,b1,b2`	load reg[0] with value from matrix A row b1, column b2
`LOADMB,b1,b2`	load reg[0] with value from matrix B row b1, column b2
`LOADMC,b1,b2`	load reg[0] with value from matrix C r row b1, column b2
`SAVEMA,b1,b2`	store reg[A] value to matrix A row b1, column b2
`SAVEMB,b1,b2`	store reg[A] value to matrix A row b1, column b2
`SAVEMC,b1,b2`	store reg[A] value to matrix A row b1, column b2
`MOP,bb`	perform matrix operation

Fast Fourier Transform Instruction

`FFT`	perform Fast Fourier Transform operation

Conversion Instructions

`FLOAT`	convert reg[A] from long integer to floating point
`FIX`	convert reg[A] from floating point to long integer
`FIXR`	convert reg[A] from floating point to long integer (with rounding)
`FSPLIT`	reg[A] = integer value, reg[0] = fractional value

Long Integer Instructions

The following descriptions provide a quick summary of the long integer instructions. Detailed descriptions are provided in the next section.

Basic Long Integer Instructions

Each of the basic long integer arithmetic instructions are provided in three different forms as shown in the table below. The LADD instruction will be used as an example to descibe the three different forms of the instructions. The LADD,nn instruction allows any general purpose register to be added to register A. The register to be added to register A is specified by the byte following the opcode. The LADD0 instruction adds register 0 to register A and only requires the opcode. The LADDB instruction adds a small integer value the register A. The signed byte (-128 to 127) following the opcode is converted to a long integer and added to register A. The LADD,nn instruction is most general, but the LADD0 and LADDB,bb instructions are more efficient for many common operations.

Register nn	Register 0	Immediate value	Description
LSET,nn	LSET0	LSETI,bb	Set
LADD,nn	LADD0	LADDI,bb	Add
LSUB,nn	LSUB0	LSUBI,bb	Subtract
LMUL,nn	LMUL0	LMULI,bb	Multiply
LDIV,nn	LDIV0	LDIVI,bb	Divide
LCMP,nn	LCMP0	LCMPI,bb	Compare
LUDIV,nn	LUDIV0	LUDIVI,bb	Unsigned Divide
LUCMP,nn	LUCMP0	LUCMPI,bb	Unsigned Compare
LTST,nn	LTST0	LTSTI,bb	Test Bits

Loading Long Integer Values

The following instructions are used to load data from the microprocessor and store it on the uM-FPU as 32-bit long integer values.

LWRITE,nn,b1,b2,b3,b4	Write 32-bit long integer value to reg[nn]
LWRITEA,b1,b2,b3,b4	Write 32-bit long integer value to reg[A]
LWRITEX,b1,b2,b3,b4	Write 32-bit long integer value to reg[X]
LWRITE0,b1,b2,b3,b4	Write 32-bit long integer value to reg[0]
WRBLK, tc, t1…tn	Write multiple 32-bit values
ATOL,aa…00	Convert ASCII string to long integer value and store in reg[0]
LONGBYTE,bb	Convert signed byte to long integer and store in reg[0]
LONGUBYTE,bb	Convert unsigned byte to long integer and store in reg[0]
LONGWORD,b1,b2	Convert signed 16-bit value to long integer and store in reg[0]
LONGUWORD,b1,b2	Convert unsigned 16-bit value to long integer and store in reg[0]

Reading Long Integer Values

The following instructions are used to read long integer values from the uM-FPU.

`LREAD,nn [b1,b2,b3,b4]`	returns 32-bit long integer value from reg[nn]
`LREADA [b1,b2,b3,b4]`	returns 32-bit long integer value from reg[A]
`LREADX [b1,b2,b3,b4]`	returns 32-bit long integer value from reg[X]
`LREAD0 [b1,b2,b3,b4]`	returns 32-bit long integer value from reg[0]
`RDBLK,tc [t1…tn]`	Read multiple 32-bit values
`LREADBYTE [b1]`	returns 8-bit byte from reg[A]
`LREADWORD [b1,b2]`	returns 16-bit value from reg[A]
`LTOA,bb`	convert long integer to ASCII string (use `READSTR` to read string)

Additional Long Integer Instructions

`LSTATUS,nn`	`LNEG`	`LNOT`	`LSHIFT,nn`
`LSTATUSA`	`LABS`	`LAND,nn`	`LMIN,nn`
`LCMP2,nn,mm`	`LINC,nn`	`LOR,nn`	`LMAX,nn`
`LUCMP2,nn,mm`	`LDEC,nn`	`LXOR,nn`	

General Purpose Instructions

RESET	COPYI,bb,nn	LOADIND,nn	SYNC
NOP	COPYA,nn	SAVEIND,nn	READSTATUS
SELECTA,nn	COPYX,nn	INDA	READSTR
SELECTX,nn	LOAD,nn	INDX	VERSION
CLR,nn	LOADA	SWAP,nn,mm	IEEEMODE
CLRA	LOADX	SWAPA,nn	PICMODE
CLRX	ALOADX	LEFT	CHECKSUM
COPY,mm,nn	XSAVE,nn	RIGHT	READVAR,bb
COPY0,nn	XSAVEA	SETOUT,bb	SETSTATUS,bb

Special Purpose Instructions

Stored Function Instructions

FCALL,fn	Call Flash user-defined function
EECALL,fn	Call EPROM user-defined function
RET	Return from user-defined function
RET,cc	Conditional return from user-defined function
BRA,bb	Unconditional branch inside user-defined function
BRA,cc,bb	Conditional branch inside user-defined function
JMP,b1,b2	Unconditional jump inside user-defined function
JMP,cc,b1,b2	Conditional jump inside user-defined function
GOTO,nn	Computed goto
TABLE,tc,t1…tn	Table lookup
FTABLE,cc,tc,t1…tn	Floating point reverse table lookup
LTABLE,cc,tc,t1…tn	Long integer reverse table lookup
POLY,tc,t1…tn	N^{th} order polynomial

Analog to Digital Conversion Instructions

ADCMODE,bb	Select A/D trigger mode
ADCTRIG	Manual A/D trigger
ADCSCALE,bb	Set A/D floating point scale factor
ADCLONG,bb	Get raw long integer A/D reading
ADCLOAD,bb	Get scaled floating point A/D reading
ADCWAIT	Wait for A/D conversion to complete

Timer Instructions

TIMESET	Set timers
TIMELONG	Get time in seconds
TICKLONG	Get time in milliseconds

EEPROM Instructions

EESAVE,mm,nn	Save reg[nn] value to EEPROM
EESAVEA,nn	Save reg[A] to EEPROM
EELOAD,mm,nn	Load reg[nn] with EEPROM value
EELOADA,nn	Load reg[A] with EEPROM value
EEWRITE,nn,bc,b1..bn	Write byte string to EEPROM

External Input Instructions

EXTSET	Set external input counter
EXTLONG	Get external input counter
EXTWAIT	Wait for next external input pulse

String Manipulation Instructions

STRSET,aa…00	Copy string to string buffer
STRSEL,bb,bb	Set string selection point
STRINC	Increment string selection point
STRDEC	Decrement string selection point
STRINS,aa…00	Insert string at selection point
STRBYTE	Insert byte at selection point
STRCMP,aa…00	Compare string with string selection
STRFIND,aa…00	Find string
STRFCHR,aa…00	Set field delimiters
STRFIELD,bb	Find field
STRTOF	Convert string selection to floating point
STRTOL	Convert string selection to long integer
FTOA,bb	Convert floating point value to string
LTOA,bb	Convert long integer value to string
READSTR	Read entire string buffer
READSEL	Read string selection

Serial Input/Output

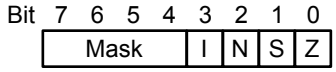
SEROUT,bb	Serial Output
SEROUT,bb,bd	Serial Output
SEROUT,bb,aa…00	Serial Output
SERIN,bb	Serial Input

Debugging Instructions

BREAK	Debug breakpoint
TRACEOFF	Turn debug trace off
TRACEON	Turn debug trace on
TRACESTR,aa…00	Display string in debug trace
TRACEREG,nn	Display contents of register in debug trace

Test Conditions

Several of the stored function instructions use a test condition byte. The test condition is an 8-bit byte that defines the expected state of the internal status byte. The upper nibble is used as a mask to determine which status bits to check. A status bit will only be checked if the corresponding mask bit is set to 1. The lower nibble specifies the expected value for each of the corresponding status bits in the internal status byte. A test condition is considered to be true if all of the masked test bits have the same value as the corresponding bits in the internal status byte. There are two special cases: `0x60` evaluates as greater than or equal, and `0x62` evaluates as less than or equal.



Bits 4-7 Mask bits
- Bit 7 Mask bit for Infinity
- Bit 6 Mask bit for NaN
- Bit 5 Mask bit for Sign
- Bit 4 Mask bit for Zero

Bits 0-3 Test bits
- Bit 3 Expected state of Infinity status bit
- Bit 2 Expected state of NaN status bit
- Bit 1 Expected state of Sign status bit
- Bit 0 Expected state of Zero status bit

The uM-FPU V3 IDE assembler has built-in symbols for the most common test conditions. They are as follows:

Assembler Symbol	Test Condition	Description
`Z`	`0x51`	Zero
`EQ`	`0x51`	Equal
`NZ`	`0x50`	Not Zero
`NE`	`0x50`	Not Equal
`LT`	`0x72`	Less Than
`LE`	`0x62`	Less Than or Equal
`GT`	`0x70`	Greater Than
`GE`	`0x60`	Greater Than or Equal
`PZ`	`0x71`	Positive Zero
`MZ`	`0x73`	Negative Zero
`INF`	`0xC8`	Infinity
`FIN`	`0xC0`	Finite
`PINF`	`0xE8`	Positive Infinity
`MINF`	`0xEA`	Minus infinity
`NAN`	`0x44`	Not-a-Number (NaN)
`TRUE`	`0x00`	True
`FALSE`	`0xFF`	False

ACOS **Arc Cosine**

Opcode: `4B`

Description: reg[A] = acos(reg[A])

Calculates the arc cosine of an angle in the range 0.0 through pi. The initial value is contained in register A, and the result is returned in register A.

Special Cases: • if reg[A] is NaN or its absolute value is greater than 1, then the result is NaN

ADCLOAD **Load scaled A/D value**

Opcode: `D5 nn` where: `nn` is the A/D channel number

Description: reg[0] = float(ADCchannel[nn]) * ADCscale[nn])

Wait until the A/D conversion is complete, then load register 0 with the reading from channel nn of the A/D converter. The 12-bit value is converted to floating point, multiplied by a scale value, and stored in register 0. The instruction buffer should be empty when this instruction is executed. If there are other instructions in the instruction buffer, or another instruction is sent before the `ADCLOAD` instruction has been completed, the wait will terminate and the previous value for the selected channel will be used.

ADCLONG **Load raw A/D value**

Opcode: `D4 nn` where: `nn` is the A/D channel number

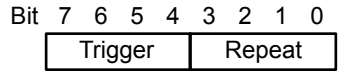
Description: reg[0] = ADCchannel[nn]

Wait until the A/D conversion is complete, then load register 0 with the reading from channel `nn` of the A/D converter. The 12-bit value is converted to a long integer and stored in register 0. The instruction buffer should be empty when this instruction is executed. If there are other instructions in the instruction buffer, or another instruction is sent before the `ADCLONG` instruction has been completed, the wait will terminate and the previous value for the selected channel will be used.

ADCMODE **Set A/D trigger mode**

Opcode: `D1 nn` where: `nn` is the trigger mode

Description: Set the trigger mode of the A/D converter. The value nn is interpreted as follows:



Bits 4-7 Trigger Type (high nibble)
- 0 - disable A/D conversions
- 1 - manual trigger
- 2 - external input trigger
- 3 - timer trigger, the value in register 0 specifies the period in microseconds (the minimum period is 100 microseconds)

Bits 0-3 Repeat Count (low nibble)

The number of samples taken for each trigger is equal to the repeat count plus one. (e.g. a value of 0 will result in one sample per trigger)

Examples:

```
ADCMODE,0x10          set manual trigger with 1 sample per trigger

ADCMODE,0x24          set external trigger with 5 samples per trigger

LOADWORD,1000         set timer trigger every 1000 usec, with 1 sample per trigger
ADCMODE,0x30

ADCMODE,0             disable A/D conversions
```

ADCSCALE — Set scale multiplier for A/D

Opcode: `D3 nn` where: `nn` is the A/D channel number

Description: ADCscale[nn] = reg[0]
Set the scale value for channel nn to the floating point value in register 0. The scale value for all channels is set to 1.0 at device reset or when the `ADCMODE` mode is set to disable A/D conversions.

ADCTRIG — Trigger an A/D conversion

Opcode: `D2`

Description: Trigger an A/D conversion. If a conversion is already in progress the trigger is ignored. This is normally used only when the `ADCMODE` is set for manual trigger.

ADCWAIT — Wait for next A/D sample

Opcode: `D6`

Description: Wait until the next A/D sample is ready. When `ADCMODE` is set for manual trigger, this instruction can be used to wait until the conversion started by the last `ADCTRIG` is done. `ADCLONG` and `ADCLOAD` automatically wait until the next sample is ready. If the `ADCMODE` is set for timer trigger or external input trigger, this instruction will wait until the next full conversion is completed. The instruction buffer should be empty when this instruction is executed. If there are other instructions in the instruction buffer, or another instruction is sent before the `ADCWAIT` instruction has been completed, the wait will terminate.

ALOADX — Load register A from register X

Opcode: `0D nn` where: nn is a register number

Description: reg[A] = reg[X] , X = X + 1
Set register A to the value of register X, and increment X to select the next register in sequence.

Special Cases: • the X register will not increment past the maximum register value of 127

ASIN — Arc Sine

Opcode: `4A`

Description: reg[A] = asin(reg[A])
Calculates the arc sine of an angle in the range of –pi/2 through pi/2. The initial value is contained

in register A, and the result in returned in register A.

Special Cases:
- if reg[A] is NaN or its absolute value is greater than 1, then the result is NaN
- if reg[A] is 0.0, then the result is a 0.0
- if reg[A] is –0.0, then the result is –0.0

ATAN **Arc Tangent**

Opcode: `4C`

Description: reg[A] = atan(reg[A])
Calculates the arc tangent of an angle in the range of –pi/2 through pi/2. The initial value is contained in register A, and the result in returned in register A.

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if reg[A] is 0.0, then the result is a 0.0
- if reg[A] is –0.0, then the result is –0.0

ATAN2 **Arc Tangent (two arguments)**

Opcode: `4D nn` where: `nn` is a register number

Description: reg[A] = atan(reg[A] / reg[nn])
Calculates the arc tangent of an angle in the range of –pi/2 through pi/2. The initial value is determined by dividing the value in register A by the value in register `nn`, and the result in returned in register A. This instruction is used to convert rectangular coordinates (reg[A], reg[nn]) to polar coordinates (r, theta). The value of theta is returned in register A.

Special Cases:
- if reg[A] or reg[nn] is NaN, then the result is NaN
- if reg[A] is 0.0 and reg[nn] > 0, then the result is 0.0
- if reg[A] > 0 and finite, and reg[nn] is +inf, then the result is 0.0
- if reg[A] is –0.0 and reg[nn] > 0, then the result is –0.0
- if reg[A] < 0 and finite, and reg[nn] is +inf, then the result is –0.0
- if reg[A] is 0.0 and reg[nn] < 0, then the result is pi
- if reg[A] > 0 and finite, and reg[nn] is –inf, then the result is pi
- if reg[A] is –0.0, and reg[nn] < 0, then the result is –pi
- if reg[A] < 0 and finite, and reg[nn] is –inf, then the result is –pi
- if reg[A] > 0, and reg[nn] is 0.0 or –0.0, then the result is pi/2
- if reg[A] is +inf, and reg[nn] is finite, then the result is pi/2
- if reg[A] < 0, and reg[nn] is 0.0 or –0.0, then the result is –pi/2
- if reg[A] is –inf, and reg[nn] is finite, then the result is –pi/2
- if reg[A] is +inf, and reg[nn] is +inf, then the result is pi/4
- if reg[A] is +inf, and reg[nn] is –inf, then the result is 3*pi/4
- if reg[A] is –inf, and reg[nn] is +inf, then the result is –pi/4
- if reg[A] is –inf, and reg[nn] is –inf, then the result is –3*pi/4

ATOF **Convert ASCII string to floating point**

Opcode: `1E aa…00` where: `aa…00` is a zero-terminated ASCII string

Description: Converts a zero terminated ASCII string to a 32-bit floating point number and stores the result in register 0. The string to convert is sent immediately following the opcode. The string can be

normal number format (e.g. 1.56, -0.5) or exponential format (e.g. 10E6). Conversion will stop at the first invalid character, but data will continue to be read until a zero terminator is encountered.

Examples: `1E 32 2E 35 34 00` (string 2.54) stores the value 2.54 in register 0
`1E 31 46 33 00` (string 1E3) stores the value 1000.0 in register 0

ATOL **Convert ASCII string to long integer**

Opcode: `9A aa…00` where: `aa...00` is a zero-terminated ASCII string

Description: Converts a zero terminated ASCII string to a 32-bit long integer and stores the result in register 0. The string to convert is sent immediately following the opcode. Conversion will stop at the first invalid character, but data will continue to be read until a zero terminator is encountered.

Examples: `9A 35 30 30 30 30 30 00` (string 500000) stores the value 500000 in register 0
`9A 2D 35 00` (string -5) stores the value -5 in register 0

BRA **Unconditional branch**

Opcode: `81 bb` where: bb is the relative address in bytes (-128 to +127)

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. Function execution will continue at the address determined by adding the signed byte value to the address of the byte immediately following the instruction. It has a range of -128 to 127 bytes. The JMP instruction can be used for addresses that are outside this range. If the new address is outside the address range of the function, a function return occurs.

BRA,cc **Conditional branch**

Opcode: `82 cc, bb` where: cc is the test condition
bb is the relative address in bytes (-128 to +127)

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. If the test condition is true, then function execution will continue at the address determined by adding the signed byte value to the address of the byte immediately following the instruction. It has a range of -128 to 127 bytes. The JMP instruction can be used for addresses that are outside this range. If the new address is outside the address range of the function, a function return occurs.

BREAK **Debug breakpoint**

Opcode: `F7`

Description: Used in conjunction with the built-in debugger. If the debugger is enabled, a breakpoint occurs and the debug monitor is entered. If debug mode is not selected, this instruction is ignored.

CEIL **Ceiling**

Opcode: `52`

Description: reg[A] = ceil(reg[A])
Calculates the floating point value equal to the nearest integer that is greater than or equal to the floating point value in register A. The result is stored in register A.

Special Cases:
- if is NaN, then the result is NaN
- if reg[A] is +infinity or -infinity, then the result is +infinity or -infinity
- if reg[A] is 0.0 or –0.0, then the result is 0.0 or –0.0
- if reg[A] is less than zero but greater than –1.0, then the result is –0.0

CHECKSUM **Calculate checksum for uM-FPU code**

Opcode: `F6`

Description: A checksum is calculated for the uM-FPU code and user-defined functions stored in Flash. The checksum value is stored in register 0. This can be used as a diagnostic test for confirming the state of a uM-FPU chip.

CLR **Clear register**

Opcode: `03 nn` where: `nn` is a register number

Description: reg[nn] = 0
Set the value of register `nn` to zero.

CLR0 **Clear register 0**

Opcode: `06`

Description: reg[0] = 0
Set the value of register 0 to zero.

CLRA **Clear register A**

Opcode: `04`

Description: reg[A] = 0
Set the value of register A to zero.

CLRX **Clear register X**

Opcode: `05`

Description: reg[X] = 0, X = X + 1
Set the value of register A to zero, and increment X to select the next register in sequence.

Special Cases:
- the X register will not increment past the maximum register value of 127

COPY **Copy registers**

Opcode: `07 mm nn` where: `mm` and `nn` are register numbers

Description: reg[nn] = reg[mm]
The value of register `mm` is copied to register nn.

COPYA **Copy register A**

Opcode: `08 nn` where: `nn` is a register number

Description: reg[nn] = reg[A]
Set register `nn` to the value of register A.

COPY0 **Copy register 0**

Opcode: `10 nn` where: `nn` is a register number

Description: reg[nn] = reg[0]
Set register `nn` to the value of register 0.

COPYI **Copy Immediate value**

Opcode: `11 bb nn` where: `bb` is an unsigned byte value (0 to 255)
`nn` is a register number

Description: reg[nn] = long(unsigned bb)
The 8-bit unsigned value is converted to a long integer and stored in register nn.

COPYX **Copy register X**

Opcode: `09 nn` where: `nn` is a register number

Description: reg[nn] = reg[X], X = X + 1
Set register `nn` to the value of register X, and increment X to select the next register in sequence.

Special Cases: • the X register will not increment past the maximum register value of 127

COS **Cosine**

Opcode: `48`

Description: reg[A] = cosine(reg[A])
Calculates the cosine of the angle (in radians) in register A and stored the result in register A.

Special Cases: • if reg[A] is NaN or an infinity, then the result is NaN

DEGREES **Convert radians to degrees**

Opcode: `4E`

Description: The floating point value in register A is converted from radians to degrees and the result is stored in register A.

Special Cases: • if reg[A] is NaN, then the result is NaN

EECALL **Call EEPROM memory user defined function**

Opcode: `7F fn` where: `fn` is the function number

Description: The user defined function `nn`, stored in EEPROM memory, is executed. Up to 16 levels of nesting is supported for function calls. The EEPROM functions can be stored at run-time using the `EEWRITE` instruction.

Special Cases: If the selected user function is not defined, register 0 is set to NaN, and execution continues.

EELOAD **Load register nn with value from EEPROM**

Opcode: `DC nn ee` where: `nn` is a register number
`ee` is the EEPROM address slot.

Description: reg[nn] = EEPROM[ee]
Register `nn` is set to the value in EEPROM at the address slot specified by `ee`. EEPROM address slots are 4 bytes in length (32-bits).

EELOADA **Load register A with value from EEPROM**

Opcode: `DD ee` where: `ee` is the EEPROM address slot

Description: reg[A] = EEPROM[ee]
Register A is set to the value in EEPROM at the address slot specified by `ee` . EEPROM address slots are 4 bytes in length (32-bits).

EESAVE **Save register nn to EEPROM**

Opcode: `DA nn ee` where: `nn` is a register number
`ee` is the EEPROM address slot

Description: EEPROM[ee] = reg[nn]
The value in register nn is stored in EEPROM at the address slot specified by `ee`. EEPROM address slots are 4 bytes in length (32-bits).

EESAVEA **Save register A to EEPROM**

Opcode: `DB ee` where: `ee` is the EEPROM address slot

Description: EEPROM[ee] = reg[A]
The value in register A is stored in EEPROM at the address slot specified by `ee`. EEPROM address slots are 4 bytes in length (32-bits).

EEWRITE **Write bytes to EEPROM**

Opcode: `DE ee bc bb...bb`
where: `ee` is the EEPROM address slot
`bc` is the number of bytes
`bb...bb` is a string of bytes

Description: Bytes are stored sequentially in EEPROM starting at the EEPROM[ee] address slot
The number of bytes specified by `bc` are copied to the EEPROM starting at address slot `ee`. Address slots are 4 bytes in length (32-bits). Consecutive address slots are used to store the specified number of bytes. This instruction can be used to store multiple values to the EEPROM address slots or to dynamically store a user-defined function.

EXP **The value e raised to a power**

Opcode: `45`

Description: reg[A] = exp(reg[A])
Calculates the value of e (2.7182818) raised to the power of the floating point value in register A. The result is stored in register A.

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if reg[A] is +infinity or greater than 88, then the result is +infinity

• if reg[A] is –infinity or less than -88, then the result is 0.0

EXP10 **The value 10 raised to a power**

Opcode: `46`

Description: reg[A] = exp10(reg[A])
Calculates the value of 10 raised to the power of the floating point value in register A. The result is stored in A.

Special Cases:
• if reg[A] is NaN, then the result is NaN
• if reg[A] is +infinity or greater than 38, then the result is +infinity
• if reg[A] is –infinity or less than -38, then the result is 0.0

EXTLONG **Load value of external input counter**

Opcode: `E1`

Description: reg[0] = external input count
Load register 0 with the external input count.

EXTSET **Set value of external input counter**

Opcode: `E0`

Description: external input count = reg[0]
The external input count is set to the value in register 0.

EXTWAIT **Wait for next external input pulse**

Opcode: `E2`

Description: Wait for the next external input to occur.

FABS **Floating point absolute value**

Opcode: `3F`

Description: reg[A] = | reg[A] |
Sets the floating value in register A to the absolute value.

Special Cases:
• if reg[A] is NaN, then the result is NaN

FADD **Floating point add**

Opcode: `21 nn` where: `nn` is a register number

Description: reg[A] = reg[A] + reg[nn]
The floating point value in register `nn` is added to the floating point value in register A and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if one value is +infinity and the other is –infinity, then the result is NaN
• if one value is +infinity and the other is not –infinity, then the result is +infinity
• if one value is -infinity and the other is not +infinity, then the result is -infinity

FADD0 **Floating point add register 0**

Opcode: `2A`

Description: reg[A] = reg[A] + reg[0]

The floating point value in register 0 is added to the floating point value in register A and the result is stored in register A.

Special Cases:
- if either value is NaN, then the result is NaN
- if one value is +infinity and the other is –infinity, then the result is NaN
- if one value is +infinity and the other is not –infinity, then the result is +infinity
- if one value is -infinity and the other is not +infinity, then the result is -infinity

---

FADDI **Floating point add immediate value**

Opcode: `33 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] + float(bb)

The signed byte value is converted to floating point and added to the value in register A and the result is stored in register A.

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if reg[A] is +infinity, then the result is +infinity
- if reg[A] is -infinity, then the result is -infinity

FCALL **Call Flash memory user defined function**

Opcode: `7E fn` where: `fn` is the function number

Description: The user defined function `nn`, stored in Flash memory, is executed. Up to 16 levels of nesting is supported for function calls. The uM-FPU IDE provides support for programming user defined functions in Flash memory using the serial debug monitor (see datasheet).

Special Cases: If the selected user function is not defined, register 0 is set to NaN, and execution continues.

FCMP **Floating point compare**

Opcode: `28 nn` where: `nn` is a register number

Description: status = compare(reg[A] - reg[nn])

Compares the floating point value in register A with the value in register `nn` and sets the internal status byte. The status byte can be read with the READSTATUS instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	N	S	Z

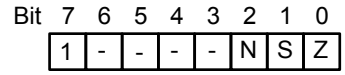
Bit 2	Not-a-Number	Set if either value is not a valid number
Bit 1	Sign	Set if reg[A] < reg[nn]
Bit 0	Zero	Set if reg[A] = reg[nn]
		If neither Bit 0 or Bit 1 is set, reg[A] > reg[nn]

FCMP0 **Floating point compare register 0**

Opcode: `31`

Description: status = compare(reg[A] - reg[0])

Compares the floating point value in register A with the value in register 0 and sets the internal status byte. The status byte can be read with the READSTATUS instruction. It is set as follows:



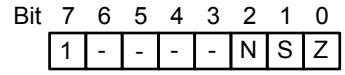
Bit 2	Not-a-Number	Set if either value is not a valid number
Bit 1	Sign	Set if reg[A] < reg[0]
Bit 0	Zero	Set if reg[A] = reg[0]
		If neither Bit 0 or Bit 1 is set, reg[A] > reg[0]

FCMP2 **Floating point compare**

Opcode: `3D nn mm` where: `nn` and mm are register numbers

Description: status = compare(reg[nn] - reg[mm])

Compares the floating point value in register `nn` with the value in register mm and sets the internal status byte. The status byte can be read with the READSTATUS instruction. It is set as follows:



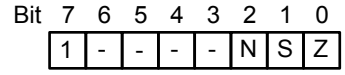
Bit 2	Not-a-Number	Set if either value is not a valid number
Bit 1	Sign	Set if reg[mm] < reg[nn]
Bit 0	Zero	Set if reg[mm] = reg[nn]
		If neither Bit 0 or Bit 1 is set, reg[mm] > reg[nn]

FCMPI **Floating point compare immediate value**

Opcode: `3A bb` where: `bb` is a signed byte value (-128 to 127)

Description: status = compare(reg[A] - float(bb))

The signed byte value is converted to floating point and compared to the floating point value in register A. The status byte can be read with the READSTATUS instruction. It is set as follows:



Bit 2	Not-a-Number	Set if either value is not a valid number
Bit 1	Sign	Set if reg[A] < float(bb)
Bit 0	Zero	Set if reg[A] = float(bb)
		If neither Bit 0 or Bit 1 is set, reg[A] > float(bb)

FCNV **Floating point conversion**

Opcode: `56 bb` where: `bb` is an unsigned byte value (0 to 255)

Description: reg[A] = the converted value of reg[A]

Convert the value in register A using the conversion specified by the byte `bb` and store the fresult in register A. The conversions are as follows:

0	Fahrenheit to Celsius
1	Celsius to Fahrenheit
2	inches to millimeters

3	millimeters to inches
4	inches to centimeters
5	centimeters to inches
6	inches to meters
7	meters to inches
8	feet to meters
9	meters to feet
10	yards to meters
11	meters to yards
12	miles to kilometers
13	kilometers to miles
14	nautical miles to meters
15	meters to nautical miles
16	acres to meters^2
17	meters ^{2} to acres
18	ounces to grams
19	grams to ounces
20	pounds to kilograms
21	kilograms to pounds
22	US gallons to liters
23	liters to US gallons
24	UK gallons to liters
25	liters to UK gallons
26	US fluid ounces to milliliters
27	milliliters to US fluid ounces
28	UK fluid ounces to milliliters
29	milliliters to UK fluid ounces
30	calories to Joules
31	Joules to calories
32	horsepower to watts
33	watts to horsepower
34	atmospheres to kilopascals
35	kilopascals to atmospheres
36	mmHg to kilopascals
37	kilopascals to mmHg
38	degrees to radians
39	radians to degrees

Special Cases: • if the byte value bb is greater than 39, the value of register A is unchanged.

FDIV **Floating point divide**

Opcode: `25 nn` where: `nn` is a register number

Description: reg[A] = reg[A] / reg[nn]

The floating point value in register A is divided by the floating point value in register `nn` and the result is stored in register A.

Special Cases: • if either value is NaN, then the result is NaN
• if both values are zero or both values are infinity, then the result is NaN

• if reg[nn] is zero and reg[A] is not zero, then the result is infinity
• if reg[nn] is infinity, then the result is zero

FDIV0 **Floating point divide by register 0**

Opcode: `2E`

Description: reg[A] = reg[A] / reg[0]
The floating point value in register A is divided by the floating point value in register 0 and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if both values are zero or both values are infinity, then the result is NaN
• if reg[nn] is zero and reg[A] is not zero, then the result is infinity
• if reg[nn] is infinity, then the result is zero

FDIVI **Floating point divide by immediate value**

Opcode: `37 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] / float(bb)
The signed byte value is converted to floating point and the value in register A is divided by the converted value and the result is stored in register A.

Special Cases:
• if reg[A] is NaN, then the result is NaN
• if both values are zero, then the result is NaN
• if the value bb is zero and reg[A] is not zero, then the result is infinity

---

FDIVR **Floating point divide (reversed)**

Opcode: `26 nn` where: `nn` is a register number

Description: reg[A] = reg[nn] / reg[A]
The floating point value in register `nn` is divided by the floating point value in register A and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if both values are zero or both values are infinity, then the result is NaN
• if reg[A] is zero and reg[nn] is not zero, then the result is infinity
• if reg[A] is infinity, then the result is zero

FDIVR0 **Floating point divide register 0 (reversed)**

Opcode: `2F`

Description: reg[A] = reg[0] / reg[A]
The floating point value in register 0 is divided by the floating point value in register A and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if both values are zero or both values are infinity, then the result is NaN
• if reg[A] is zero and reg[0] is not zero, then the result is infinity
• if reg[A] is infinity, then the result is zero

FDIVRI **Floating point divide by immediate value (reversed)**

Opcode: `38 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = float(bb) / reg[A]
The signed byte value is converted to floating point and divided by the value in register A. The result is stored in register A.

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if both values are zero, then the result is NaN
- if the value reg[A] is zero and float(bb) is not zero, then the result is infinity

---

FFT **Fast Fourier Transform**

Opcode: `6F bb` where: `bb` specifies the type of operation

Description: The type of operation is specified as follows:

0	first stage
1	next stage
2	next level
3	next block
+4	pre-processing bit reverse sort
+8	pre-processing for inverse FFT
+16	post-processing for inverse FFT

The data for the `FFT` instruction is stored in matrix A as a Nx2 matrix, where N must be a power of two. The data points are specified as complex numbers, with the real part stored in the first column and the imaginary part stored in the second column. If all data points can be stored in the matrix (maximum of 64 points if all 128 registers are used), the Fast Fourier Transform can be calculated with a single instruction. If more data points are required than will fit in the matrix, the calculation must be done in blocks. The algorithm iteratively writes the next block of data, executes the `FFT` instruction for the appropriate stage of the FFT calculation, and reads the data back to the microcontroller. This proceeds in stages until all data points have been processed. See application notes for more details.

FINV **Floating point inverse**

Opcode: `40`

Description: reg[A] = 1 / reg[A]
The inverse of the floating point value in register A is stored in register A.

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if reg[A] is zero, then the result is infinity
- if reg[A] is infinity, then the result is zero

FIX **Convert floating point to long integer**

Opcode: `61`

Description: reg[A] = fix(reg[A])
Converts the floating point value in register A to a long integer value.

Special Cases: • if reg[A] is NaN, then the result is zero
• if reg[A] is +infinity or greater than the maximum signed long integer, then the result is the maximum signed long integer (decimal: 2147483647, hex: $7FFFFFFF)
• if reg[A] is –infinity or less than the minimum signed long integer, then the result is the minimum signed long integer (decimal: -2147483648, hex: $80000000)

FIXR **Convert floating point to long integer with rounding**

Opcode: `62`

Description: reg[A] = fix(round(reg[A]))
Converts the floating point value in register A to a long integer value with rounding.

Special Cases: • if reg[A] is NaN, then the result is zero
• if reg[A] is +infinity or greater than the maximum signed long integer, then the result is the maximum signed long integer (decimal: 2147483647, hex: $7FFFFFFF)
• if reg[A] is –infinity or less than the minimum signed long integer, then the result is the minimum signed long integer (decimal: -2147483648, hex: $80000000)

FLOAT **Convert long integer to floating point**

Opcode: `60`

Description: reg[A] = float(reg[A])
Converts the long integer value in register A to a floating point value.

FLOOR **Floor**

Opcode: `51`

Description: reg[A] = floor(reg[A])
Calculates the floating point value equal to the nearest integer that is less than or equal to the floating point value in register A. The result is stored in register A.

Special Cases: • if reg[A] is NaN, then the result is NaN
• if reg[A] is +infinity or -infinity, then the result is +infinity or -infinity
• if reg[A] is 0.0 or –0.0, then the result is 0.0 or –0.0

FMAC **Multiply and add to accumulator**

Opcode: `57 nn mm` where: `nn` and `mm` are a register numbers

Description: reg[A] = reg[A] + (reg[nn] * reg[mm])
The floating point value in register `nn` is multiplied by the value in register mm and the result is added to register A.

Special Cases: • if either value is NaN, or one value is zero and the other is infinity, then the result is NaN
• if either values is infinity and the other is nonzero, then the result is infinity

---

FMAX **Floating point maximum**

Opcode: `55 nn` where: `nn` is a register number

Description: reg[A] = max(reg[A], reg[nn])
The maximum floating point value of registers A and register `nn` is stored in register A.

Special Cases: • if either value is NaN, then the result is NaN

FMIN **Floating point minimum**

Opcode: `54 nn` where: `nn` is a register number

Description: reg[A] = min(reg[A], reg[nn])
The minimum floating point value of registers A and register `nn` is stored in register A.

Special Cases: • if either value is NaN, then the result is NaN

FMOD **Floating point remainder**

Opcode: `50 nn` where: `nn` is a register number

Description: reg[A] = remainder of reg[A] / (reg[nn]
The floating point remainder of the floating point value in register A divided by register `nn` is stored in register A.

FMSC **Multiply and subtract from accumulator**

Opcode: `58 nn mm` where: `nn` and `mm` are a register numbers

Description: reg[A] = reg[A] - (reg[nn] * reg[mm])
The floating point value in register `nn` is multiplied by the value in register mm and the result is subtracted from register A.

Special Cases: • if either value is NaN, or one value is zero and the other is infinity, then the result is NaN
• if either values is infinity and the other is nonzero, then the result is infinity

---

FMUL **Floating point multiply**

Opcode: `24 nn` where: `nn` is a register number

Description: reg[A] = reg[A] * reg[nn]
The floating point value in register A is multiplied by the value in register `nn` and the result is stored in register A.

Special Cases: • if either value is NaN, or one value is zero and the other is infinity, then the result is NaN
• if either values is infinity and the other is nonzero, then the result is infinity

---

FMUL0 **Floating point multiply by register 0**

Opcode: `2D`

Description: reg[A] = reg[A] * reg[0]
The floating point value in register 0 is multiplied by the value in register `nn` and the result is stored in register A.

Special Cases: • if either value is NaN, or one value is zero and the other is infinity, then the result is NaN
• if either values is infinity and the other is nonzero, then the result is infinity

---

FMULI **Floating point multiply by immediate value**

Opcode: `36 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] * float[bb]

The signed byte value is converted to floating point and the value in register A is multiplied by the converted value and the result is stored in reg[A].

Special Cases:
- if reg[A] is NaN, then the result is NaN
- if the signed byte is zero and reg[A] is infinity, then the result is NaN

FNEG **Floating point negate**

Opcode: `3E`

Description: reg[A] = -reg[A]

The negative of the floating point value in register A is stored in register A.

Special Cases:
- if the value is NaN, then the result is NaN

FPOW **Floating point power**

Opcode: `27 nn` where: `nn` is a register number

Description: reg[A] = reg[A] ** reg[nn]

The floating point value in register A is raised to the power of the floating point value in register `nn` and stored in register A.

Special Cases:
- if reg[nn] is 0.0 or –0.0, then the result is 1.0
- if reg[nn] is 1.0, then the result is the same as the A value
- if reg[nn] is NaN, then the result is Nan
- if reg[A] is NaN and reg[nn] is nonzero, then the result is NaN
- if | reg[A] | > 1 and reg[nn] is +infinite, then the result is +infinity
- if | reg[A] | < 1 and reg[nn] is -infinite, then the result is +infinity
- if | reg[A] | > 1 and reg[nn] is -infinite, then the result is 0.0
- if | reg[A] | < 1 and reg[nn] is +infinite, then the result is 0.0
- if | reg[A] | = 1 and reg[nn] is infinite, then the result is NaN
- if reg[A] is 0.0 and reg[nn] > 0, then the result is 0.0
- if reg[A] is +infinity and reg[nn] < 0, then the result is 0.0
- if reg[A] is 0.0 and reg[nn] < 0, then the result is +infinity
- if reg[A] is +infinity and reg[nn] > 0, then the result is +infinity
- if reg[A] is -0.0 and reg[nn] > 0 but not a finite odd integer, then the result is 0.0
- if the reg[A] is -infinity and reg[nn] < 0 but not a finite odd integer, then the result is 0.0
- if reg[A] is -0.0 and the reg[nn] is a positive finite odd integer, then the result is –0.0
- if reg[A] is -infinity and reg[nn] is a negative finite odd integer, then the result is –0.0
- if reg[A] is -0.0 and reg[nn] < 0 but not a finite odd integer, then the result is +infinity
- if reg[A] is -infinity and reg[nn] > 0 but not a finite odd integer, then the result is +infinity
- if reg[A] is -0.0 and reg[nn] is a negative finite odd integer, then the result is –infinity
- if reg[A] is -infinity and reg[nn] is a positive finite odd integer, then the result is –infinity
- if reg[A] < 0 and reg[nn] is a finite even integer,

then the result is equal to | reg[A] | to the power of reg[nn]
• if reg[A] < 0 and reg[nn] is a finite odd integer,
then the result is equal to the negative of | reg[A] | to the power of reg[nn]
• if reg[A] < 0 and finite and reg[nn] is finite and not an integer, then the result is NaN

FPOW0 **Floating point power by register 0**

Opcode: `30 nn` where: `nn` is a register number

Description: reg[A] = reg[A] ** reg[0]
The floating point value in register A is raised to the power of the floating point value in register 0 and stored in register A.

Special Cases:
• if reg[0] is 0.0 or –0.0, then the result is 1.0
• if reg[0] is 1.0, then the result is the same as the A value
• if reg[0] is NaN, then the result is Nan
• if reg[A] is NaN and reg[0] is nonzero, then the result is NaN
• if | reg[A] | > 1 and reg[0] is +infinite, then the result is +infinity
• if | reg[A] | < 1 and reg[0] is -infinite, then the result is +infinity
• if | reg[A] | > 1 and reg[0] is -infinite, then the result is 0.0
• if | reg[A] | < 1 and reg[0] is +infinite, then the result is 0.0
• if | reg[A] | = 1 and reg[0] is infinite, then the result is NaN
• if reg[A] is 0.0 and reg[0] > 0, then the result is 0.0
• if reg[A] is +infinity and reg[0] < 0, then the result is 0.0
• if reg[A] is 0.0 and reg[0] < 0, then the result is +infinity
• if reg[A] is +infinity and reg[0] > 0, then the result is +infinity
• if reg[A] is -0.0 and reg[0] > 0 but not a finite odd integer, then the result is 0.0
• if the reg[A] is -infinity and reg[0] < 0 but not a finite odd integer, then the result is 0.0
• if reg[A] is -0.0 and the reg[0] is a positive finite odd integer, then the result is –0.0
• if reg[A] is -infinity and reg[0] is a negative finite odd integer, then the result is –0.0
• if reg[A] is -0.0 and reg[0] < 0 but not a finite odd integer, then the result is +infinity
• if reg[A] is -infinity and reg[0] > 0 but not a finite odd integer,
then the result is +infinity
• if reg[A] is -0.0 and reg[0] is a negative finite odd integer, then the result is –infinity
• if reg[A] is -infinity and reg[0] is a positive finite odd integer,
then the result is –infinity
• if reg[A] < 0 and reg[0] is a finite even integer,
then the result is equal to | reg[A] | to the power of reg[0]
• if reg[A] < 0 and reg[0] is a finite odd integer,
then the result is equal to the negative of | reg[A] | to the power of reg[0]
• if reg[A] < 0 and finite and reg[0] is finite and not an integer, then the result is NaN

FPOWI **Floating point power by immediate value**

Opcode: `39 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] ** float[bb]
The signed byte value is converted to floating point and the value in register A is raised to the power of the converted value. The result is stored in register A.

Special Cases:
- if bb is 0, then the result is 1.0
- if bb is 1, then the result is the same as the A value
- if reg[A] is NaN and bb is nonzero, then the result is NaN
- if reg[A] is 0.0 and bb > 0, then the result is 0.0
- if reg[A] is +infinity and bb < 0, then the result is 0.0
- if reg[A] is 0.0 and bb < 0, then the result is +infinity
- if reg[A] is +infinity and bb > 0, then the result is +infinity
- if reg[A] is -0.0 and bb > 0 but not an odd integer, then the result is 0.0
- if the reg[A] is -infinity and bb < 0 but not an odd integer, then the result is 0.0
- if reg[A] is -0.0 and bb is a positive odd integer, then the result is –0.0
- if reg[A] is -infinity and bb is a negative odd integer, then the result is –0.0
- if reg[A] is -0.0 and bb < 0 but not an odd integer, then the result is +infinity
- if reg[A] is -infinity and bb > 0 but not an odd integer, then the result is +infinity
- if reg[A] is -0.0 and bb is a negative odd integer, then the result is –infinity
- if reg[A] is -infinity and bb is a positive odd integer, then the result is –infinity
- if reg[A] < 0 and bb is an even integer,

then the result is equal to | reg[A] | to the power of bb
- if reg[A] < 0 and bb is an odd integer,

then the result is equal to the negative of | reg[A] | to the power of bb

---

FRAC **Get fractional part of floating point value**

Opcode: `63`

Description: Register A is loaded with the fractional part the floating point value in register A. The sign of the fraction is the same as the sign of the original value.

Special Cases:
- if register A is NaN or infinity, then the result is NaN

FREAD **Read floating point value**

Opcode: `1A nn` where: `nn` is a register number

Returns: `b1,b2,b3,b4` where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[nn]

The floating point value of register `nn` is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction. If the PIC data format has been selected (using the `PICMODE` instruction), the IEEE 754 format floating point value is converted to PIC format before being sent.

FREAD0 **Read floating point value from register 0**

Opcode: `1D`

Returns: `b1,b2,b3,b4` where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[0]

The floating point value from register 0 is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction. If the PIC data format has been selected (using the `PICMODE` instruction), the IEEE 754 format floating point value is converted to PIC format before being sent.

FREADA **Read floating point value from register A**

Opcode: `1B`

Returns: `b1,b2,b3,b4` where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[A]

The floating point value of register A is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction. If the PIC data format has been selected (using the `PICMODE` instruction), the IEEE 754 format floating point value is converted to PIC format before being sent.

FREADX **Read floating point value from register X**

Opcode: `1C`

Returns: `b1,b2,b3,b4` where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[X], X = X + 1

The floating point value from register X is returned, and X is incremented to the next register. The four bytes of the 32-bit floating point value must be read immediately following this instruction. If the PIC data format has been selected (using the `PICMODE` instruction), the IEEE 754 format floating point value is converted to PIC format before being sent.

FSET **Set register A**

Opcode: `20 nn` where: `nn` is a register number

Description: reg[A] = reg[nn]

Set register A to the value of register `nn`.

FSET0 **Set register A from register 0**

Opcode: `29`

Description: reg[A] = reg[0]

Set register A to the value of register 0.

FSETI **Set register from immediate value**

Opcode: `32 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = float(bb)

The signed byte value is converted to floating point and stored in register A.

FSPLIT **Split integer and fractional portions of floating point value**

Opcode: `64`

Description: reg[A] = integer portion of reg[A], reg[0] = fractional portion of reg[A]

The integer portion of the original value in register A is stored in register A, and the fractional portion is stored in register 0. Both values are stored as floating point values.

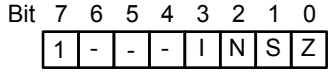
Special Cases: • if the original value is NaN or Infinity, reg[A] is set to zero and reg[0] is set to NaN

FSTATUS **Get floating point status**

Opcode: `3B nn` where: `nn` is a register number

Description: status = status(reg[nn])

Set the internal status byte to the floating point status of the value in register nn. The status byte can be read with the READSTATUS instruction. It is set as follows:



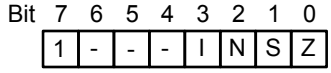
Bit 3	Infinity	Set if the value is an infinity
Bit 2	Not-a-Number	Set if the value is not a valid number
Bit 1	Sign	Set if the value is negative
Bit 0	Zero	Set if the value is zero

FSTATUSA **Get floating point status of register A**

Opcode: `3C`

Description: status = status(reg[A])

Set the internal status byte to the floating point status of the value in register A. The status byte can be read with the READSTATUS instruction. It is set as follows:



Bit 3	Infinity	Set if the value is an infinity
Bit 2	Not-a-Number	Set if the value is not a valid number
Bit 1	Sign	Set if the value is negative
Bit 0	Zero	Set if the value is zero

FSUB **Floating point subtract**

Opcode: `22 nn` where: `nn` is a register number

Description: reg[A] = reg[A] - reg[nn]

The floating point value in register `nn` is subtracted from the floating point value in register A.

Special Cases:
- if either value is NaN, then the result is NaN
- if both values are infinity and the same sign, then the result is NaN
- if reg[A] is +infinity and reg[nn] is not +infinity, then the result is +infinity
- if reg[A] is -infinity and reg[nn] is not -infinity, then the result is -infinity
- if reg[A] is not an infinity and reg[nn] is an infinity, then the result is an infinity of the opposite sign as reg[nn]

FSUB0 **Floating point subtract register 0**

Opcode: `2B`

Description: reg[A] = reg[A] - reg[0]

The floating point value in register 0 is subtracted from the floating point value in register A.

Special Cases:
- if either value is NaN, then the result is NaN
- if both values are infinity and the same sign, then the result is NaN
- if reg[A] is +infinity and reg[0] is not +infinity, then the result is +infinity
- if reg[A] is -infinity and reg[0] is not -infinity, then the result is -infinity

• if reg[A] is not an infinity and reg[0] is an infinity, then the result is an infinity of the opposite sign as reg[0]

FSUBI **Floating point subtract immediate value**

Opcode: `34 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] - float[bb]
The signed byte value is converted to floating point and subtracted from the value in register A.

Special Cases:
• if reg[A] is NaN, then the result is NaN
• if reg[A] is +infinity, then the result is +infinity
• if reg[A] is -infinity, then the result is -infinity

FSUBR **Floating point subtract (reversed)**

Opcode: `23 nn` where: `nn` is a register number

Description: reg[A] = reg[nn] - reg[A]
The floating point value in register A is subtracted from the floating point value in register nn and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if both values are infinity and the same sign, then the result is NaN
• if reg[nn] is +infinity and reg[A] is not +infinity, then the result is +infinity
• if reg[nn] is -infinity and reg[A] is not -infinity, then the result is -infinity
• if reg[nn] is not an infinity and reg[A] is an infinity, then the result is an infinity of the opposite sign as reg[A]

FSUBR0 **Floating point subtract register 0 (reversed)**

Opcode: `2C`

Description: reg[A] = reg[0] - reg[A]
The floating point value in register A is subtracted from the floating point value in register 0 and the result is stored in register A.

Special Cases:
• if either value is NaN, then the result is NaN
• if both values are infinity and the same sign, then the result is NaN
• if reg[nn] is +infinity and reg[0] is not +infinity, then the result is +infinity
• if reg[nn] is -infinity and reg[A] is not -infinity, then the result is -infinity
• if reg[nn] is not an infinity and reg[A] is an infinity, then the result is an infinity of the opposite sign as reg[A]

FSUBRI **Floating point subtract immediate value (reversed)**

Opcode: `35 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = float[bb] - reg[A]
The signed byte value is converted to floating point and the value in reg[A] is subtracted from it and stored in reg[A].

Special Cases:
• if reg[A] is NaN, then the result is NaN
• if reg[A] is +infinity, then the result is +infinity

• if reg[A] is -infinity, then the result is -infinity

FTABLE **Floating point reverse table lookup**

Opcode: `85 cc tc t1…tn` where: `cc` is the test condition
`tc` is the size of the table
`t1…tn` are 32-bit floating point values

Description: reg[0] = index of table entry that matches the test condition for reg[A]
This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. It performs a reverse table lookup on a floating point value. The value in register A is compared to the values in the table using the test condition. The index number of the first table entry that satisfies the test condition is returned in register 0. If no entry is found, register 0 is unchanged. The index number for the first table entry is zero.

FTOA **Convert floating point value to ASCII string**

Opcode: `1F bb` where: `bb` is the format byte

Description: The floating point value in register A is converted to an ASCII string and stored in the string buffer at the current selection point. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended. The byte immediately following the `FTOA` opcode is the format byte and determines the format of the converted value.

If the format byte is zero, as many digits as necessary will be used to represent the number with up to eight significant digits. Very large or very small numbers are represented in exponential notation. The length of the displayed value is variable and can be from 3 to 12 characters in length. The special cases of NaN (Not a Number), +infinity, -infinity, and -0.0 are handled. Examples of the ASCII strings produced are as follows:

```
1.0             NaN             0.0
10e20           Infinity        -0.0
3.1415927       -Infinity       1.0
-52.333334      -3.5e-5         0.01
```

If the format byte is non-zero, it is interpreted as a decimal number. The tens digit specifies the maximum length of the converted string, and the ones digit specifies the number of decimal points. The maximum number of digits for the formatted conversion is 9, and the maximum number of decimal points is 6. If the floating point value is too large for the format specified, asterisks will be stored. If the number of decimal points is zero, no decimal point will be displayed. Examples of the display format are as follows: (note: leading spaces are shown where applicable)

Value in register A	Format byte	Display format
123.567	61 (6.1)	123.6
123.567	62 (6.2)	123.57
123.567	42 (4.2)	*.**
0.9999	20 (2.0)	1
0.9999	31 (3.1)	1.0

This instruction is usually followed by a READSTR instruction to read the string.

FWRITE **Write floating point value**

Opcode: `16 nn b1...b4` where: nn is register number
b1...b4 is floating point value (b1 is MSB)

Description: reg[nn] = 32-bit floating point value
The floating point value is stored in register `nn`. If the PIC data format has been selected (using the `PICMODE` instruction), the PIC format floating point value is converted to IEEE 754 format before being stored in the register.

FWRITE0 **Write floating point value to register 0**

Opcode: `19 b1...b4` where: b1...b4 is floating point value (b1 is MSB)

Description: reg[0] = 32-bit floating point value
The floating point value is stored in register A. If the PIC data format has been selected (using the `PICMODE` instruction), the PIC format floating point value is converted to IEEE 754 format before being stored in register A.

FWRITEA **Write floating point value to register A**

Opcode: `17 b1...b4` where: b1...b4 is floating point value (b1 is MSB)

Description: reg[A] = 32-bit floating point value
The floating point value is stored in register A. If the PIC data format has been selected (using the `PICMODE` instruction), the PIC format floating point value is converted to IEEE 754 format before being stored in register A.

FWRITEX **Write floating point value to register X**

Opcode: `18 b1...b4` where: b1...b4 is floating point value (b1 is MSB)

Description: reg[A] = 32-bit floating point value, X = X + 1
The floating point value is stored in register X, and X is incremented to the next register. If the PIC data format has been selected (using the `PICMODE` instruction), the PIC format floating point value is converted to IEEE 754 format before being stored in register A.

Special Cases: • the X register will not increment past the maximum register value of 127

GOTO **Computed GOTO**

Opcode: `89 nn` where: `nn` is a register number

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. Function execution will continue at the address determined by adding the register value to the current function address. If the register value is negative, or the new address is outside the address range of the function, a function return occurs.

IEEEMODE **Select IEEE floating point format**

Opcode: `F4`

Description: Selects the IEEE 754 floating point format for the `FREAD`, `FREADA`, `FREADX`, `FWRITE`, `FWRITEA`, and `FWRITEX` instructions. This is the default mode on reset and only needs to be changed if the `PICMODE` instruction has been used.

INDA **Select A using value in register**

Opcode: `7C nn` where: nn is a register number

Description: A = reg[nn]
Select register A using the value contained in register `nn`

INDX **Select X using value in register**

Opcode: `7D nn` where: nn is a register number

Description: X = reg[nn]
Select register X using the value contained in register `nn`.

JMP **Unconditional jump**

Opcode: `83 b1 b2` where: b1,b2 is the function address

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. Function execution will continue at the address specified. The `BRA` instruction can be used for addresses that are within -128 to 127 bytes of the current address. If the new address is outside the address range of the function, a function return occurs.

JMP,cc **Conditional jump**

Opcode: `84 cc, bb` where: cc is the test condition
b1,b2 is the function address

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. If the test condition is true, then function execution will continue at the address specified. The BRA instruction can be used for addresses that are within -128 to 127 bytes of the current address. If the new address is outside the address range of the function, a function return occurs.

LABS **Long Integer absolute value**

Opcode: `BC`

Description: reg[A] = | reg[A] |, status = status(reg[A])
The absolute value of the long integer value in register A is stored in register A.

LADD **Long integer add**

Opcode: `9B nn` where: `nn` is a register number

Description: reg[A] = reg[A] + reg[nn], status = status(reg[A])
The long integer value in register `nn` is added to register A.

LADD0 **Long integer add register 0**

Opcode: `A6`

Description: reg[A] = reg[A] + reg[0], status = status(reg[A])
The long integer value in register 0 is added to register A.

LADDI **Long integer add immediate value**

Opcode: `AF bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] + long(bb), status = status(reg[A])
The signed byte value is converted to a long integer and added to register A.

LAND **Long integer AND**

Opcode: `C0 nn` where: `nn` is a register number

Description: reg[A] = reg[A] AND reg[nn], status = status(reg[A])
The bitwise AND of the values in register A and register `nn` is stored in register A.

LCMP **Long integer compare**

Opcode: `A1 nn` where: `nn` is a register number

Description: status = compare(reg[A] - reg[nn])
Compares the signed long integer value in register A with the value in register `nn` and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if reg[A] < reg[nn]
Bit 0	Zero	Set if reg[A] = reg[nn]
		If neither Bit 0 or Bit 1 is set, reg[A] > reg[nn]

LCMP0 **Long integer compare register 0**

Opcode: `AA`

Description: status = compare(reg[A] - reg[0])
Compares the signed long integer value in register A with the value in register 0 and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if reg[A] < reg[0]
Bit 0	Zero	Set if reg[A] = reg[0]
		If neither Bit 0 or Bit 1 is set, reg[A] > reg[0]

LCMP2 **Long integer compare**

Opcode: `B9 nn mm` where: `nn` and `mm` are register numbers

Description: status = compare(reg[nn] - reg[mm])
Compares the signed long integer value in register `nn` with the value in register mm and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if reg[nn] < reg[mm]

Bit 0 Zero — Set if reg[nn] = reg[mm]
If neither Bit 0 or Bit 1 is set, reg[nn] > reg[mm]

LCMPI **Long integer compare immediate value**

Opcode: `B3 bb` where: `bb` is a signed byte value (-128 to 127)

Description: status = compare(reg[A] - long(bb))
The signed byte value is converted to long integer and compared to the signed long integer value in register A. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign — Set if reg[A] < long(bb)
Bit 0 Zero — Set if reg[A] = long(bb)
If neither Bit 0 or Bit 1 is set, reg[A] > long(bb)

LDEC **Long integer decrement**

Opcode: `BE nn` where: `nn` is a register number

Description: reg[nn] = reg[nn] - 1, status = status(reg[nn])
The long integer value in register `nn` is decremented by one. The long integer status is stored in the status byte.

LDIV **Long integer divide**

Opcode: `A0 nn` where: `nn` is a register number

Description: regA] = reg[A] / reg[nn], reg[0] = remainder, status = status(reg[A])
The long integer value in register A is divided by the signed value in register `nn`, and the result is stored in register A. The remainder is stored in register 0.

Special Cases: • if reg[nn] is zero, the result is the largest positive long integer ($7FFFFFFF)

LDIV0 **Long integer divide by register 0**

Opcode: `A9`

Description: reg[A] = reg[A] / reg[0], reg[0] = remainder, status = status(reg[A])
The long integer value in register A is divided by the signed value in register 0, and the result is stored in register A. The remainder is stored in register 0.

Special Cases: • if reg[0] is zero, the result is the largest positive long integer ($7FFFFFFF)

LDIVI **Long integer divide by immediate value**

Opcode: `B2 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] / long(bb), reg[0] = remainder, status = status(reg[A])
The signed byte value is converted to a long integer and register A is divided by the converted value. The result is stored in register A. The remainder is stored in register 0.

Special Cases: • if the signed byte value is zero, the result is the largest positive long integer ($7FFFFFFF)

LEFT **Left Parenthesis** *(modified V3.1)*

Opcode: `14`

Description: The `LEFT` instruction saves the current register A selection, allocates the next temporary register, sets the value of the temporary register to the current register A value, then selects the temporary register as register A. The `RIGHT` instruction is used to restore previous values. When used together, these instruction are like parentheses in an equation, and can be used to allocate temporary registers, and change the order of a calculation. Parentheses can be nested up to eight levels.

Special Cases: • If the maximum number of temporary register is exceeded, the value of register A is set to NaN ($7FFFFFFF).

LINC **Long integer increment**

Opcode: `BD nn` where: `nn` is a register number

Description: reg[nn] = reg[nn] + 1, status = status(reg[nn])
The long integer value in register `nn` is incremented by one. The long integer status is stored in the status byte.

LMAX **Floating point maximum**

Opcode: `C5 nn` where: `nn` is a register number

Description: reg[A] = max(reg[A], reg[nn]), status = status(reg[A])
The maximum signed long integer value of registers A and register `nn` is stored in register A.

Special Cases: • if either value is NaN, then the result is NaN

LMIN **Floating point minimum**

Opcode: `C4 nn` where: `nn` is a register number

Description: reg[A] = min(reg[A], reg[nn]), status = status(reg[A])
The minimum signed long integer value of registers A and register `nn` is stored in register A.

Special Cases: • if either value is NaN, then the result is NaN

LMUL **Long integer multiply**

Opcode: `9F nn` where: `nn` is a register number

Description: reg[A] = reg[A] * reg[nn], status = status(reg[A])
The long integer value in register A is multiplied by register `nn` and the result is stored in register A.

LMUL0 **Long integer multiply by register 0**

Opcode: `A8`

Description: reg[A] = reg[A] * reg[0], status = status(reg[A])
The long integer value in register A is multiplied by register 0 and the result is stored in register A.

LMULI **Long integer multiply by immediate value**

Opcode: `B1 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] * long(bb), status = status(reg[A])

The signed byte value is converted to a long integer and the long integer value in register A is multiplied by the converted value. The result is stored in register A.

LNEG **Long integer negate**

Opcode: `BB`

Description: reg[A] = -reg[A], status = status(reg[A])

The negative of the long integer value in register A is stored in register A.

LNOT **A = NOT A**

Opcode: `BF`

Description: reg[A] = NOT reg[A], status = status(reg[A])

The bitwise complement of the value in register A is stored in register A.

LOAD **reg[0] = reg[nn]**

Opcode: `0A nn` where: `nn` is a register number

Description: reg[0] = reg[nn]

Load register 0 with the value in register `nn`.

LOADA **Load register 0 with the value of register A**

Opcode: `0B`

Description: reg[0] = reg[A]

Load register 0 with the value of register A.

LOADBYTE **Load register 0 with 8-bit signed value**

Opcode: `59 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[0] = float(signed bb)

Loads register 0 with the 8-bit signed integer value converted to floating point value.

LOADCON **Load register 0 with floating point constant**

Opcode: `5F bb` where: `bb` selects the constant

Description: This instruction is defined for version 3.0.0 to V3.1.3 of the uM-FPU V3 chip, but will be removed in future versions. Use of this instruction is not recommended. Constant values can easily be loaded using the `FWRITE0` instruction.

reg[0] = constant[bb]

Loads register 0 with the floating point constant specified by bb as follows:

0	1.0	10^0
1	10.0	10^1
2	100.0	10^2
3	1000.0	10^3

4	10000.0	10^4
5	100000.0	10^5
6	1000000.0	10^6
7	10000000.0	10^7
8	100000000.0	10^8
9	1000000000.0	10^9
10	$\approx 3.4028235\times10^{38}$	largest positive finite 32-bit floating point value
11	$\approx 1.4012985\times10^{-45}$	smallest positive non-zero 32-bit floating point value
12	299792458.0	speed of light in vacuum (m/s)
13	6.6742e-11	Newtonian constant of gravitation ($m^3/kg*s^2$)
14	9.80665	acceleration of gravity
15	9.1093826e-31	electron mass (kg)
16	1.67262171e-27	proton mass (kg)
17	1.67492728e-27	neutron mass (kg)
18	6.0221415e23	Avogadro constant (/mol)
19	1.60217653e-19	elementary charge, electron volt
20	101.325	standard atmosphere (kPa)

Special Cases: • if the byte value bb is greater than 20, register A is set to NaN.

LOADE **Load register 0 with floating point value of e (2.7182818)**

Opcode: `5D`

Description: reg[0] = 2.7182818

Loads register 0 with the floating point value of e (2.7182818).

LOADIND **Load Indirect**

Opcode: `7A nn` where: `nn` is a register number

Description: reg[0] = reg[reg[nn]]

Load register 0 with the value of the register number contained in register `nn`. The value in register nn is assumed to be a long integer value.

Special Cases: If the value in register nn > 127, register 127 is used.

LOADMA **Load register 0 with the value from matrix A**

Opcode: `68 bb bb` where: bb, bb selects the row, column of matrix A

Description: reg[0] = matrix A [bb, bb]

Load register 0 with a value from matrix A.

Special Cases: If the row or column is out of range, NaN is returned.

LOADMB **Load register 0 with the value from matrix A**

Opcode: `69 bb bb` where: bb, bb selects the row, column of matrix B

Description: reg[0] = matrix B [bb, bb]

Load register 0 with a value from matrix B.

Special Cases: If the row or column is out of range, NaN is returned.

LOADMC **Load register 0 with the value from matrix A**

Opcode: `6A bb bb` where: bb, bb selects the row, column of matrix C

Description: reg[0] = matrix C [bb, bb]
Load register 0 with a value from matrix C.

Special Cases: If the row or column is out of range, NaN is returned.

LOADPI **Load register 0 with value of Pi**

Opcode: `5E`

Description: reg[0] = 3.1415927
Loads register 0 with the floating point value of pi (3.1415927).

LOADUBYTE **Load register 0 with 8-bit unsigned value**

Opcode: `5A bb` where: `bb` is an unsigned byte value (0 to 255)

Description: reg[0] = float(unsigned bb)
The 8-bit unsigned value is converted to floating point and stored in register 0.

LOADUWORD **Load register 0 with 16-bit unsigned value**

Opcode: `5C b1,b2` where: `b1,b2` is an unsigned word value (0 to 65535)

Description: reg[0] = float(unsigned (b1*256 + b2))
The 16-bit unsigned value is converted to floating point and stored in register 0.

LOADWORD **Load register 0 with 16-bit signed value**

Opcode: `5B b1,b2` where: `b1,b2` is a signed word value (-32768 to 32767)

Description: reg[0] = float (signed(b1*256 + b2))
The 16-bit signed value is converted to floating point and stored in register 0.

LOADX **Load register 0 with the value of register X**

Opcode: `0C`

Description: reg[0] = reg[X], X = X + 1
Load register 0 with the value of register X, and increment X to select the next register in sequence.

Special Cases: • the X register will not increment past the maximum register value of 127

LOG **Logarithm (base e)**

Opcode: `43`

Description: reg[A] = log(reg[A])
Calculates the natural log of the floating point value in register A. The result is stored in register A. The number e (2.7182818) is the base of the natural system of logarithms.

Special Cases:
- if the value is NaN or less than zero, then the result is NaN
- if the value is +infinity, then the result is +infinity
- if the value is 0.0 or –0.0, then the result is -infinity

LOG10 **Logarithm (base 10)**

Opcode: `44`

Description: reg[A] = log10(reg[A])
Calculates the base 10 logarithm of the floating point value in register A. The result is stored in register A.

Special Cases:
- if the value is NaN or less than zero, then the result is NaN
- if the value is +infinity, then the result is +infinity
- if the value is 0.0 or –0.0, then the result is -infinity

LONGBYTE **Load register 0 with 8-bit signed value**

Opcode: `C6 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[0] = long(signed (bb)), status = status(reg[0])
The 8-bit signed value is converted to a long integer and stored in register 0.

LONGUBYTE **Load register 0 with 8-bit unsigned value**

Opcode: `C7 bb` where: `bb` is an unsigned byte value (0 to 255)

Description: reg[0] = long(unsigned (bb)), status = status(reg[0])
The 8-bit unsigned value is converted to a long integer and stored in register 0.

LONGUWORD **Load register 0 with 16-bit unsigned value**

Opcode: `C9 b1,b2` where: `b1,b2` is an unsigned word value (0 to 65535)

Description: reg[0] = long(unsigned (b1*256 + b2)), status = status(reg[0])
The 16-bit unsigned value is converted to a long integer and stored in register 0.

LONGWORD **Load register 0 with 16-bit signed value**

Opcode: `C8 b1,b2` where: `b1,b2` is a signed word value (-32768 to 32767)

Description: reg[0] = long(signed (b1*256 + b2)), status = status(reg[0])
The 16-bit signed value is converted to a long integer and stored in register 0.

LOR **Long integer OR**

Opcode: `C1 nn` where: `nn` is a register number

Description: reg[A] = reg[A] OR reg[nn], status = status(reg[A])
The bitwise OR of the values in register A and register nn is stored in register A.

LREAD **Read long integer value**

Opcode: `94 nn` where: `nn` is a register number
Returns: `b1,b2,b3,b4` where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[nn]

The long integer value of register nn is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

LREAD0 **Read long integer value from register 0**

Opcode: 97

Returns: b1,b2,b3,b4 where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[0]

The long integer value of register 0 is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

LREADA **Read long integer value from register A**

Opcode: 95

Returns: b1,b2,b3,b4 where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[A], status = status(reg[A])

The long integer value of register A is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

LREADBYTE **Read the lower 8-bits of register A**

Opcode: 98

Returns: bb where: bb is 8-bit value

Description: Return 8-bit value from reg[A]

Returns the lower 8 bits of register A. The byte containing the 8-bit long integer value must be read immediately following the instruction.

LREADWORD **Read the lower 16-bits of register A**

Opcode: 99

Returns: b1,b2 where: b1, b2 is 16-bit value (b1 is MSB)

Description: Return 16-bit value from reg[A]

Returns the lower 16 bits of register A. The two bytes containing the 16-bit long integer value must be read immediately following this instruction.

LREADX **Read long integer value from register X**

Opcode: 96

Returns: b1,b2,b3,b4 where: b1, b2, b3, b4 is floating point value (b1 is MSB)

Description: Return 32-bit value from reg[X], X = X + 1

The long integer value from register X is returned, and X is incremented to the next register. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

LSET **Set register A**

Opcode: 9C nn where: nn is a register number

Description: reg[A] = reg[nn], status = status(reg[A])

Set register A to the value of register nn.

LSET0 **Set register A from register 0**

Opcode: A5

Description: reg[A] = reg[0], status = status(reg[A])
Set register A to the value of register 0.

LSETI **Set register from immediate value**

Opcode: AE bb where: bb is a signed byte value (-128 to 127)

Description: reg[A] = long(bb), status = status(reg[A])
The signed byte value is converted to a long integer and stored in register A.

LSHIFT **A = A shifted by B bit positions**

Opcode: C3 nn where: nn is a register number

Description: if reg[nn] > 0, then reg[A] = reg[A] shifted left by bb bits
if reg[nn]< 0, then reg[A] = reg[A] shifted right by bb bits
status = status(reg[nn])
The value in register A is shifted by the number of bit positions specified by the long integer value in register nn. Register A is shifted left if the value in register nn is positive, and right if the value is negative.

Special Cases:
- if reg[nn] = 0, no shift occurs
- if reg[nn] > 32 or reg[nn] < –32, then reg[A] = 0

LSTATUS **Get long integer status**

Opcode: B7 nn where: nn is a register number

Description: status = status(reg[nn])
Set the internal status byte to the long integer status of the value in register nn. The status byte can be read with the READSTATUS instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if the value is negative
Bit 0 Zero Set if the value is zero

LSTATUSA **Get long integer status of register A**

Opcode: B8

Description: status = status(reg[A])
Set the internal status byte to the long integer status of the value in register A. The status byte can be read with the READSTATUS instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if the value is negative
Bit 0 Zero Set if the value is zero

LSUB **Long integer subtract**

Opcode: `9E nn` where: `nn` is a register number

Description: reg[A] = reg[A] - reg[nn], status = status(reg[A])
The long integer value in register `nn` is subtracted from register A.

LSUB0 **Long integer subtract register 0**

Opcode: `A7`

Description: reg[A] = reg[A] - reg[0], status = status(reg[A])
The long integer value in register 0 is subtracted from register A.

LSUBI **Long integer subtract immediate value**

Opcode: `B0 bb` where: `bb` is a signed byte value (-128 to 127)

Description: reg[A] = reg[A] - long(bb), status = status(reg[A])
The signed byte value is converted to a long integer and subtracted from register A.

LTABLE **Long integer reverse table lookup**

Opcode: `87 cc tc t1…tn` where: `cc` is the test condition
`tc` is the size of the table
`t1…tn` are 32-bit long integer values

Description: reg[0] = index of table entry that matches the test condition for reg[A]
This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. It performs a reverse table lookup on a long integer value. The value in register A is compared to the values in the table using the specified test condition. The index number of the first table entry that satisfied the test condition is returned in register 0. If no entry is found, register 0 is unchanged. The index number for the first table entry is zero.

LTOA **Convert long integer value to ASCII string and store in string buffer**

Opcode: `9B bb` where: `bb` is the format byte

Description: stringbuffer = converted string, status = status(reg[A])
The long integer value in register A is converted to an ASCII string and stored in the string buffer at the current selection point. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended. The byte immediately following the `LTOA` opcode is the format byte and determines the format of the converted value.

If the format byte is zero, the length of the converted string is variable and can range from 1 to 11 characters in length. Examples of the converted string are as follows:

1
500000
-3598390

If the format byte is non-zero, it is interpreted as a decimal number. A value between 0 and 15 specifies the length of the converted string. The converted string is right justified. If 100 is added to the format value the value is converted as an unsigned long integer, otherwise it is converted as an signed long integer. If the value is larger than the specified width, asterisks are stored. If the length is specified as zero, the string will be as long as necessary to represent the number.

Examples of the converted string are as follows: (note: leading spaces are shown where applicable)

Value in register A	Format byte	Display format
-1	10	(signed 10)	-1
-1	110	(unsigned 10)	4294967295
-1	4	(signed 4)	-1
-1	104	(unsigned 4)	****
0	4	(signed 4)	0
0	0	(unformatted)	0
1000	6	(signed 6)	1000

The maximum length of the string is 15. This instruction is usually followed by a READSTR instruction to read the string.

LTST **Long integer bit test**

Opcode: `A4 nn` where: `nn` is a register number

Description: status = status(reg[A] AND reg[nn])

Sets the internal status byte based on the result of a bitwise `AND` of the values in register A and register `nn`. The values of register A and register `nn` are not changed. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if the MSB of the result is set
Bit 0	Zero	Set the result is zero

LTST0 **Long integer bit test register 0**

Opcode: `AD`

Description: status = status(reg[A] AND reg[0])

Sets the internal status byte based on the result of a bitwise AND of the value in register A and register 0. The values of register A and register `0` are not changed. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if the MSB of the result is set
Bit 0	Zero	Set the result is zero

LTSTI **Long integer bit test using immediate value**

Opcode: `B6 bb` where: `bb` is a signed byte value (0 to 255)

Description: status = status(reg[A] AND long(bb))

The unsigned byte value is converted to long integer and the internal status byte is set based on the result of a bitwise `AND` of the converted value and register A. The value of register A is not changed. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	-	Z

Bit 0	Zero	Set if the result is zero

LUCMP **Unsigned long integer compare**

Opcode: `A3 nn` where: `nn` is a register number

Description: status = compare(reg[A] - reg[nn])

Compares the unsigned long integer value in register A with register `nn` and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if reg[A] < reg[nn]

Bit 0 Zero Set if reg[A] = reg[nn]

If neither Bit 0 or Bit 1 is set, reg[A] > reg[nn]

LUCMP0 **Unsigned long integer compare register 0**

Opcode: `AC`

Description: status = compare(reg[A] - reg[0])

Compares the unsigned long integer value in register A with register 0 and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if reg[A] < reg[0]

Bit 0 Zero Set if reg[A] = reg[0]

If neither Bit 0 or Bit 1 is set, reg[A] > reg[0]

LUCMP2 **Unsigned long integer compare**

Opcode: `BA nn mm` where: `nn` and mm are register numbers

Description: status = compare(reg[nn] - reg[mm])

Compares the signed long integer value in register `nn` with the signed long integer value in register `mm` and sets the internal status byte. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if reg[nn] < reg[mm]

Bit 0 Zero Set if reg[nn] = reg[mm]

If neither Bit 0 or Bit 1 is set, reg[nn] > reg[mm]

LUCMPI **Unsigned long integer compare immediate value**

Opcode: `B5 bb` where: `bb` is an unsigned byte value (0 to 255)

Description: status = compare(reg[A] - long(bb))

The unsigned byte value is converted to long integer and compared to register A. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1 Sign Set if reg[A] < long(bb)

Bit 0 Zero Set if reg[A] = long(bb)

If neither Bit 0 or Bit 1 is set, reg[A] > long(bb)

LUDIV **Unsigned long integer divide**

Opcode: `A2 nn` where: `nn` is a register number

Description: reg[A] = reg[A] / reg[nn], reg[0] = remainder, status = status(reg[A])
The unsigned long integer value in register A is divided by register nn, and the result is stored in register A. The remainder is stored in register 0.

Special Cases: • if register nn is zero, the result is the largest unsigned long integer ($FFFFFFFF)

LUDIV0 **Unsigned long integer divide by register 0**

Opcode: `AB`

Description: reg[A] = reg[A] / reg[0] , reg[0] = remainder, status = status(reg[A])
The unsigned long integer value in register A is divided by the signed value in register 0, and the result is stored in register A. The remainder is stored in register 0.

Special Cases: • if register 0 is zero, the result is the largest unsigned long integer ($FFFFFFFF)

LUDIVI **Unsigned long integer divide by immediate value**

Opcode: `B4 bb` where: `bb` is a signed byte value (0 to 255)

Description: reg[A] = reg[A] / long(bb) , reg[0] = remainder, status = status(reg[A])
The unsigned byte value is converted to a long integer and register A is divided by the converted value. The result is stored in register A. The remainder is stored in register 0.

Special Cases: • if the signed byte value is zero, the result is the largest unsigned long integer ($FFFFFFFF)

LWRITE **Write long integer value**

Opcode: `90 nn b1,b2,b3,b4` where: nn is register number
b1, b2, b3, b4 is long integer value (b1 is MSB)

Description: reg[nn] = 32-bit long integer value, status = status(reg[nn])
The long integer value is stored in register `nn`.

LWRITE0 **Write long integer value to register0**

Opcode: `93 b1,b2,b3,b4` where: b1, b2, b3, b4 is long integer value (b1 is MSB)

Description: reg[0] = 32-bit long integer value, status = status(reg[0])
The long integer value is stored in register 0.

LWRITEA **Write long integer value to register A**

Opcode: `91 b1,b2,b3,b4` where: b1, b2, b3, b4 is long integer value (b1 is MSB)

Description: reg[A] = 32-bit long integer value, status = status(reg[A])
The long integer value is stored in register A.

LWRITEX **Write long integer value to register X**

Opcode: `92 b1,b2,b3,b4` where: b1, b2, b3, b4 is long integer value (b1 is MSB)

Description: reg[X] = 32-bit long integer value, status = status(reg[X]), X = X + 1
The long integer value is stored in register X, and X is incremented to the next register.

LXOR **Long integer XOR**

Opcode: `C2 nn` where: `nn` is a register number

Description: reg[A] = reg[A] XOR reg[nn], status = status(reg[A])
The bitwise XOR of the values in register A and register `nn` is stored in register A.

MOP **Matrix Operation**

Opcode: `6E bb` where: `bb` is the operation code
`6E bb ic, i1…in` `ic` is the index count
`i1…in` are the index values

Description: The operation code nn selects one of the following operations:

0 Scalar Set. Each element: MA[r,c] = reg[0]
1 Scalar Add. For each element: MA[r,c] = MA[r,c] + reg[0]
2 Scalar Subtract. For each element: MA[r,c] = MA[r,c] + reg[0]
3 Scalar Subtract (reverse). For each element: MA[r,c] = reg[0] - MA[r,c]
4 Scalar Multiply. For each element: MA[r,c] = MA[r,c] * reg[0]
5 Scalar Divide. For each element: MA[r,c] = MA[r,c] / reg[0]
6 Scalar Divide (reverse). For each element: MA[r,c] = reg[0] / MA[r,c]
7 Scalar Power. For each element: MA[r,c] = MA[r,c] ** reg[0]
8 Element-wise Set. Each element: MA[r,c] = MB[r,c]
9 Element-wise Add. For each element: MA[r,c] = MA[r,c] + MB[r,c]
10 Element-wise Subtract. For each element: MA[r,c] = MA[r,c] + MB[r,c]
11 Element-wise Subtract (reverse). For each element: MA[r,c] = MB[r,c] - MA[r,c]
12 Element-wise Multiply. For each element: MA[r,c] = MA[r,c] * MB[r,c]
13 Element-wise Divide. For each element: MA[r,c] = MA[r,c] / MB[r,c]
14 Element-wise Divide (reverse). For each element: MA[r,c] = MB[r,c] / MA[r,c]
15 Element-wise Power. For each element: MA[r,c] = MA[r,c] ** MB[r,c]
16 Matrix Multiply. Calculate: MA = MB * MC
17 Identity matrix. Set: MA = identity matrix
18 Diagonal matrix. Set: MA = diagonal matrix (reg[0] value stored on diagonal)
19 Transpose. Set: MA = transpose MB
20 Count. Set: reg[0] = count of all elements in MA
21 Sum. Set: reg[0] = sum of all elements in MA
22 Average. Set: reg[0] = average of all elements in MA
23 Minimum. Set: reg[0] = minimum of all elements in MA
24 Maximum Set: reg[0] = maximum of all elements in MA
25 Copy matrix A to matrix B
26 Copy matrix A to matrix C
27 Copy matrix B to matrix A
28 Copy matrix B to matrix C
29 Copy matrix C to matrix A
30 Copy matrix C to matrix B
31 Matrix Determinant: reg[0] = determinant of MA (2x2 and 3x3 matrices only)
32 Matrix Inverse: MA = inverse of MB (2x2 and 3x3 matrices only)
33 Indexed Load Registers to Matrix A : MOP,33,ic,i1...in

34 Indexed Load Registers to Matrix B : MOP,34,ic,i1...in
35 Indexed Load Registers to Matrix C : MOP,35,ic,i1...in
36 Indexed Load Matrix B to Matrix A: MOP,36,ic,i1...in
37 Indexed Load Matrix C to Matrix A: MOP,37,ic,i1...in
38 Indexed Save Matrix A to Register: MOP,38,ic,i1...in
39 Indexed Save Matrix A to Matrix B: MOP,39,ic,i1...in
40 Indexed Save Matrix A to Matrix C: MOP,40,ic,i1...in

The Indexed Load Registers operations take a list of register numbers and sequentially copy the indexed register values to the matrix specified. The Indexed Load Matrix operations take a list of matrix indexes and sequentially copy the indexed matrix values to Matrix A. The Indexed Save operations take a list of register numbers or matrix indices and sequentially copy the values from matrix A to registers, matrix B, or matrix C. These operations can be used to quickly load matrices and save results, or to extract and save matrix subsets.

Special Cases:
- Indexed Load Register: register 0 is cleared to zero before the indexed values are copied, to provide an easy way to load zero values to a matrix.
- Indexed Load Register: if index is negative, the absolute value is used as an index, and the negative of the indexed value is copied.
- Indexed Load Matrix: an index of 0x80 is used to copy the negative of the value at index 0.
- Indexed Save Matrix: if index value is negative, the matrix A value for that index position is not stored.

NOP **No operation**

Opcode: `00`

Description: No operation.

PICMODE **Select PIC floating point format**

Opcode: `F5`

Description: Selects the alternate PIC floating point mode using by many PIC compilers. All internal data on the uM-FPU is stored in IEEE 754 format, but when the uM-FPU is in PIC mode an automatic conversion is done by the FREAD, FREADA, FREADX, FWRITE, FWRITEA, and FWRITEX instructions so the PIC program can use floating point data in the alternate format. Normally this instruction would be issued immediately after the reset as part of the initialization code. The IEEEMODE instruction can be used to revert to standard IEEE 754 floating point mode.

POLY **A = nth order polynomial**

Opcode: `88 tc t1…tn` where: `tc` is the number of coefficient values
`t1…tn` are 32-bit floating point values

Description: reg[A] = result of nth order polynomial calculation
This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. The value of the specified polynomial is calculated and stored in register A. The general form of the polynomial is:

$$y = A_0 + A_1x^1 + A_2x^2 + \ldots A_nx^n$$

The value of x is the initial value of register A. An n^{th} order polynomial will have n+1 coefficients

stored in the table. The coefficient values A_0, A_1, A_2, … are stored as a series of 32-bit floating point values (4 bytes) stored in order from A_n to A_0. If a given term in the polynomial is not needed, a zero must be is stored for that value.

Example: The polynomial 3x + 5 would be represented as follows:

```
88 02 40 A0 00 00 40 40 00 00
```

Where:		
	`88`	opcode
	`02`	size of the table (order of the polynomial + 1)
	`40 40 00 00`	floating point constant 3.0
	`40 A0 00 00`	floating point constant 5.0

RADIANS **Convert degrees to radians**

Opcode: `4F`

Description: reg[A] = radians(reg[A])
The floating point value in register A is converted from degrees to radians and the result is stored in register A.

Special Cases: • if the value is NaN, then the result is NaN

RDBLK **Read multiple 32-bit point values** *(new V3.1)*

Opcode: `71 tc` where: `tc` is the number of 32-bit values to read

Description: Return tc 32-bit values from reg[X], X = X+1
This instruction is used to read multiple 32-bit values from the uM-FPU registers. The byte immediately following the opcode is the transfer count, and bits 6:0 specify the number of 32-bit values that follow (a value of zero specifies a transfer count of 128). If bit 7 of the transfer count is set, the bytes are reversed for each 32-bit value that follows. This allows for efficient data transfers when the native storage format of the microcontroller is the reverse of the uM-FPU format. The X register specifies the register to read from, and it is incremented after each 32-bit value is read.

Special Cases: • the X register will not increment past the maximum register value of 127
• if PICMODE is enabled, the 32-bit values are assumed to be floating point values

---

READSEL **Read string selection**

Opcode: `EC`
Returns: `aa...00` where: `aa...00` is a zero-terminated string

Description: Returns the current string selection. Data bytes must be read immediately following this instruction and continue until a zero byte is read. This instruction is typically used after `STRSEL` or `STRFIELD` instructions.

READSTATUS **Return the last status byte**

Opcode: `F1`
Returns: `ss` where: `ss` is the status byte

Description: The 8-bit internal status byte is returned.

READSTR **Read string**

Opcode: `F2`

Returns: `aa...00` where: `aa...00` is a zero-terminated string

Description: Returns the zero terminated string in the string buffer. Data bytes must be read immediately following this instruction and continue until a zero byte is read. This instruction is used after instructions that load the string buffer (e.g. `FTOA`, `LTOA`, `VERSION`). On completion of the `READSTR` instruction the string selection is set to select the entire string.

READVAR **Read internal variable** *(modified V3.1)*

Opcode: `FC bb` where: `bb` is index of internal register

Description: reg[0] = internal register value, status = status(reg[0])

Sets register 0 to the current value of one of the internal registers (based on index value passed).

0	A register
1	X register
2	Matrix A register
3	Matrix A rows
4	Matrix A columns
5	Matrix B register
6	Matrix B rows
7	Matrix B columns
8	Matrix C register
9	Matrix C rows
10	Matrix C columns
11	internal mode word
12	last status byte
13	clock ticks per millisecond
14	current length of string buffer
15	string selection starting point
16	string selection length
17	8-bit character at string selection point
18	number of bytes in instruction buffer

RESET **Reset**

Opcode: `FF`

Description: Nine consecutive FF bytes will cause the uM-FPU to reset. If less then nine consecutive FF bytes are received, they are treated as NOPs.

RET **Return from user-defined function**

Opcode: `80`

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. It causes a return from the current function. Execution will continue with the instruction following the last function call. This instruction is required as the last instruction of a user-defined function in EEPROM memory.

RET,cc **Conditional return from user-defined function** *(new V3.1)*

Opcode: `8A cc` where: `cc` is the test condition

Description: This instruction is only valid in a user-defined function in Flash memory or EEPROM memory. If the test condition is true, it causes a return from the current function, and execution will continue with the instruction following the last function call. If the test condition is false, execution continues with the next instruction.

RIGHT **Right Parenthesis**

Opcode: `15`

Description: The right parenthesis command copies the value of register A (the current temporary register) to register 0. If the right parenthesis is the outermost parenthesis, the register A selection from before the first left parenthesis is restored, otherwise the previous temporary register is selected as register. Used together with the left parenthesis command to allocate temporary registers, and to change the order of a calculation. Parentheses can be nested up to eight levels.

Special Cases:
- if no left parenthesis is currently outstanding, then register 0 is set to NaN. ($7FFFFFFF).

ROOT **Calculate n^{th} root**

Opcode: `42 nn` where: `nn` is a register number

Description: reg[A] = reg[A] ** (1 / reg[nn])
Calculates the n^{th} root of the floating point value in register A and stores the result in register A. Where the value n is equal to the floating point value in register `nn`. It is equivalent to raising A to the power of (1 / `nn`).

Special Cases:
- see the description of the `POWER` instruction for the special cases of (1/reg[nn])
- if reg[nn] is infinity, then (1 / reg[nn]) is zero
- if reg[nn] is zero, then (1 / reg[nn]) is infinity

ROUND **Floating point Rounding**

Opcode: `53`

Description: reg[A] = round(reg[A])
The floating point value equal to the nearest integer to the floating point value in register A is stored in register A.

Special Cases:
- if the value is NaN, then the result is NaN
- if the value is +infinity or -infinity, then the result is +infinity or -infinity
- if the value is 0.0 or –0.0, then the result is 0.0 or –0.0

SAVEIND **Save Indirect**

Opcode: `7B nn` where: `nn` is a register number

Description: reg[reg[nn]] = reg[A]
The value of register A is stored in the register whose register number is contained in register `nn`. The value in register `nn` is assumed to be long integer.

Special Cases: If the value in register nn > 127, register 127 is used.

SAVEMA **Save register A value to matrix A**

Opcode: `6B b1 b2` where: b1 selects the row and b2 selects the column of matrix A

Description: matrix A [b1, b2] = reg[A]
Store the register A value to matrix A at the row, column specified.

Special Cases: If the row or column is out of range, no value is stored

---

SAVEMB **Save register A value to matrix B**

Opcode: `6C b1 b2` where: b1 selects the row and b2 selects the column of matrix B

Description: matrix A [b1, b2] = reg[A]
Store the register A value to matrix B at the row, column specified.

Special Cases: If the row or column is out of range, no value is stored

---

SAVEMC **Save register A value to matrix C**

Opcode: `6D b1 b2` where: b1 selects the row and b2 selects the column of matrix C

Description: matrix A [b1, b2] = reg[A]
Store the register A value to matrix C at the row, column specified.

Special Cases: If the row or column is out of range, no value is stored

---

SELECTA **Select A**

Opcode: `01 nn` where: `nn` is a register number

Description: A = nn
The value `nn` is used to select register A.

SELECTMA **Select matrix A**

Opcode: `65 nn b1 b2` where: `nn` is a register number
b1 is the number of rows, b2 is number of columns

Description: Select matrix A, X = nn
The value `nn` is used to select a register that is the start of matrix A. Matrix values are stored in sequential registers (rows * columns). The upper four bits of the `rc` value specify the number of rows, and the lower four bits specify the number of columns (a row or column value of zero is interpreted as 16). The X register is also set to the first element of the matrix so that the `FREADX, FWRITEX, LREADX, LWRITEX, SAVEX, SETX, LOADX` instructions can be immediately used to store values to or retrieve vales from the matrix.

SELECTMB **Select matrix B**

Opcode: `66 nn b1 b2` where: `nn` is a register number
b1 is the number of rows, b2 is number of columns

Description: Select matrix B, X = nn
The value `nn` is used to select a register that is the start of matrix B. Matrix values are stored in

sequential registers (rows * columns). The upper four bits of the `rc` value specify the number of rows, and the lower four bits specify the number of columns (a row or column value of zero is interpreted as 16). The X register is also set to the first element of the matrix so that the `FREADX, FWRITEX, LREADX, LWRITEX, SAVEX, SETX, LOADX` instructions can be immediately used to store values to or retrieve vales from the matrix.

SELECTMC **Select matrix C**

Opcode: `67 nn b1 b2` where: `nn` is a register number
`b1` is the number of rows, `b2` is number of columns

Description: Select matrix C, X = nn

The value `nn` is used to select a register that is the start of matrix B. Matrix values are stored in sequential registers (rows * columns). The upper four bits of the `rc` value specify the number of rows, and the lower four bits specify the number of columns (a row or column value of zero is interpreted as 16). The X register is also set to the first element of the matrix so that the `FREADX, FWRITEX, LREADX, LWRITEX, SAVEX, SETX, LOADX` instructions can be immediately used to store values to or retrieve vales from the matrix.

SELECTX **Select register X**

Opcode: `02 nn` where: `nn` is a register number

Description: X = nn

The value `nn` is used to select register X.

SERIN **Serial input** *(new V3.1)*

Opcode: `CF bb` where: `bb` specifies the type of operation

Description: This instruction is used to read serial data from the SERIN pin. The instruction is ignored if Debug Mode is enabled. The baud rate for serial input is the same as the baud rate for serial output, and is set with the `SEROUT,0` instruction. The operation to be performed is specified by the byte immediately following the opcode:

0	Disable serial input
1	Enable character mode serial input
2	Get character mode serial input status
3	Get serial input character
4	Enable NMEA serial input
5	Get NMEA input status
6	Transfer NMEA sentence to string buffer

`SERIN,0`

Disable serial input. This can be used to save interrupt processing time if serial input is not used continuously.

`SERIN,1`

Enable character mode serial input. Serial input is enabled, and incoming characters are stored in a 160 byte buffer. The serial input status can be checked with the `SERIN,2` instruction and input characters can be read using the `SERIN,3` instruction.

`SERIN,2`

Get character mode serial input status. The status byte is set to zero (Z) if the input buffer is empty,

or non-zero (NZ) if the input buffer is not empty.

`SERIN,3`
Get serial input character. The serial input character is returned in register 0. If this instruction is the last instruction in the instruction buffer, it will wait for the next available input character. It there are other instructions in the instruction buffer, or another instruction is sent before the `SERIN,3` instruction has completed, it will terminate and return a zero value.

`SERIN,4`
Enable NMEA serial input. Serial input is enabled, and the serial input data is scanned for NMEA sentences which are then stored in a 200 byte buffer. Additional NMEA sentences can be buffered while the current sentence is being processed. The sentence prefix character ($), trailing checksum characters (if specified), and the terminator (CR,LF) are not stored in the buffer. NMEA sentences are transferred to the string buffer for processing using the `SERIN,6` instruction, and the NMEA input status can be checked with the `SERIN,5` instruction.

`SERIN,5`
Get the NMEA input status. The status byte is set to zero (Z) if the buffer is empty, or non-zero (NZ) if at least one NMEA sentence is available in the buffer.

`SERIN,6`
Transfer NMEA sentence to string buffer. This instruction transfers the next NMEA sentence to the string buffer, and selects the first field of the string so that a `STRCMP` instruction can be used to check the sentence type. If the sentence is valid, the status byte is set to `0x80` and the greater-than (GT) test condition will be true. If an error occurs, the status byte will be set to `0x82`, `0x92`, `0xA2`, or `0xB2`. Bit 4 of the status byte is set if an overrun error occurred. Bit 5 of the status byte is set if a checksum error occurred. The less-than (LT) test condition will be true for all errors. If this instruction is the last instruction in the instruction buffer, it will wait for the next available NMEA sentence. It there are other instructions in the instruction buffer, or another instruction is sent before the `SERIN,6` instruction has completed, it will terminate and return an empty sentence.

SEROUT **Serial Output** *(new V3.1)*

Opcode:
`CE bb`
`CE bb bd`
`CE bb aa…00`

where: `bb` specifies the type of operation
`bd` specifies the I/O mode and baud rate
`aa…00` is a zero-terminated string

Description: This instruction is used to set the serial input/output mode and baud rate, and to send serial data to the SEROUT pin. The operation to be performed is specified by the byte immediately following the opcode:

0	Set serial I/O mode and baud rate
1	Send text string to serial output
2	Send string buffer to serial output
3	Send string selection to serial output
4	Send lower 8 bits of register 0 to serial output
5	Send text string and zero terminator to serial output

`SEROUT,0,bb`
This instruction sets the baud rate for serial input/output, and enables or disables Debug Mode.

The mode is specified by the byte immediately following the operation code:

0	57,600 baud, Debug Mode enabled
1	300 baud, Debug Mode disabled
2	600 baud, Debug Mode disabled
3	1200 baud, Debug Mode disabled
4	2400 baud, Debug Mode disabled
5	4800 baud, Debug Mode disabled
6	9600 baud, Debug Mode disabled
7	19200 baud, Debug Mode disabled
8	38400 baud, Debug Mode disabled
9	57600 baud, Debug Mode disabled
10	115200 baud, Debug Mode disabled

For mode 0, a `{DEBUG ON}` message is sent to the serial output and the baud rate is changed. For modes 1 to 10, if the debug mode is enabled, a `{DEBUG OFF}` message is sent to the serial output before the baud rate is changed.

`SEROUT,1,aa..00`
The text string specified by the instruction (not including the zero-terminator) is sent to the serial output. The instruction is ignored if Debug Mode is enabled.

`SEROUT,2`
The contents of the string buffer are sent to the serial output. The instruction is ignored if Debug Mode is enabled.

`SEROUT,3`
The current string selection is sent to the serial port. The instruction is ignored if Debug Mode is enabled.

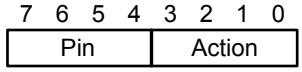
`SEROUT,4`
The lower 8 bits of register 0 are sent to the serial port as an 8-bit character. The instruction is ignored if Debug Mode is enabled.

`SEROUT,5,aa..00`
The text string specified by the instruction (including the zero-terminator) is sent to the serial output. The instruction is ignored if Debug Mode is enabled.

SETOUT **Set output**

Opcode: `D0 nn` where: `nn` is a command byte

Description: Set the OUT0 or OUT1 output pin according to the command byte `nn` as follows:



Bits 4-7 Output pin (upper nibble)
0 - OUT 0
1 - OUT 1
Bits 0-3 Action (lower nibble)
0 - set output low

1 - set output high
2 - toggle the output to opposite level
3 - set output to high impedance

SETSTATUS **Set status byte** *(new V3.1)*

Opcode: `CD bb` where: `ss` is status value

Description: The internal status byte is set to the 8-bit value specified.

SIN **Sine**

Opcode: `47`

Description: reg[A] = sin(reg[A])
Calculates the sine of the angle (in radians) in register A and stored the result in register A.

Special Cases:
- if A is NaN or an infinity, then the result is NaN
- if A is 0.0, then the result is 0.0
- if A is –0.0, then the result is –0.0

SQRT **Square root**

Opcode: `41`

Description: reg[A] = sqrt(reg[A])
Calculates the square root of the floating point value in register A and stored the result in register A.

Special Cases:
- if the value is NaN or less than zero, then the result is NaN
- if the value is +infinity, then the result is +infinity
- if the value is 0.0 or –0.0, then the result is 0.0 or –0.0

STRBYTE **Insert byte at string selection** *(new V3.1)*

Opcode: `ED`

Description: The lower 8 bits of register 0 are stored as an 8-bit character in the string buffer at the current selection point. The selection point is updated to point immediately after the stored byte, so multiple bytes can be appended.

STRCMP **Compare string with string selection**

Opcode: `E6 aa...00` where: `aa...00` is a zero-terminated string

Description: The string is compared with the string at the current selection point and the internal status byte is set. The status byte can be read with the `READSTATUS` instruction. It is set as follows:

Bit	7	6	5	4	3	2	1	0
	1	-	-	-	-	-	S	Z

Bit 1	Sign	Set if string selection < specified string
Bit 0	Zero	Set if string selection = specified string
		If neither Bit 0 or Bit 1 is set, string selection > specified string

---

STRDEC **Decrement string selection point** *(new V3.1)*

Opcode: `EF`

Description: The string selection point is incremented and the selection length is set to zero.

Special Cases: • the selection point will not decrement past the beginning of the string

---

STRFCHR **Set field separator characters**

Opcode: `E8 aa...00` where: `aa...00` is a zero-terminated string

Description: The string specifies a list of characters to be used as field separators. The default field separator is a comma.

STRFIELD **Find field in string** *(modified V3.1)*

Opcode: `E9 bb` where: `bb` is the field number

Description: The selection point is set to the specified field. Fields are numbered from 1 to n, and are separated by the characters specified by the last `STRFCHR` instruction. If no `STRFCHR` instruction has been executed, the default field separator is a comma. If bit 7 of `bb` is set, then bits 6:0 of `bb` specify a register number, and the lower 8 bits of the register specify the field number.

Special Cases: • if bb = 0, selection point is set to the start of the string buffer
• if bb > number of fields, selection point is set to the end of the string buffer

---

STRFIND **Find string in the string buffer** *(modified V3.1)*

Opcode: `E7 aa...00` where: `aa...00` is a zero-terminated string

Description: Search the string selection for the first occurrence of the specified string. If the string is found, the selection point is set to the matching substring. If the string is not found, the selection point is set to the end of the string selection.

STRINC **Increment string selection point** *(new V3.1)*

Opcode: `EE`

Description: The string selection point is incremented and the selection length is set to zero.

Special Cases: • the selection point will not increment past the end of the string

---

STRINS **Insert string**

Opcode: `E5 aa...00` where: `aa...00` is a zero-terminated string

Description: Insert the string in the string buffer at the current selection point. The selection point is updated to point immediately after the inserted string, so multiple insertions can be appended.

STRSEL **Set string selection point** *(modified V3.1)*

Opcode: `E4 nn mm` where: `nn` is the start of the selection
`mm` is the length of the selection

Description: Set the start of the string selection to character nn and the length of the selection to mm characters. Characters are numbered from 0 to n. If bit 7 of nn is set, then bits 6:0 of nn specify a register number, and the lower 8 bits of the register specify the start of the selection. If bit 7 of mm is set, then bits 6:0 of mm specify a register number, and the lower 8 bits of the register specify the length of the selection.

Special Cases:
- if nn > string length, start of selection is set to end of string
- if nn+mm > string length, selection is adjusted for the end of string

---

STRSET **Copy string to string buffer**

Opcode: `E3 aa...00` where: `aa...00` is a zero-terminated string

Description: Copy the string to the string buffer and set the selection point to the end of the string.

Special Cases:
- if nn > string length, start of selection is set to end of string

---

STRTOF **Convert string selection to floating point**

Opcode: `EA`

Description: Convert the string at the current selection point to a floating point value and store the result in register 0.

STRTOL **Convert string selection to long integer**

Opcode: `EB`

Description: Convert the string at the current selection point to a long integer value and store the result in register 0.

SWAP **Swap registers**

Opcode: `12 nn mm` where: `nn` and `mm` are register numbers

Description: tmp = reg[nn], reg[nn] = reg[mm], reg[mm] = tmp
The values of register nn and register mm are swapped.

SWAPA **Swap register A**

Opcode: `13 nn` where: `nn` is a register number

Description: tmp = reg[nn], reg[nn] = reg[A], reg[A] = tmp
The values of register nn and register A are swapped.

SYNC **Synchronization**

Opcode: `F0`
Returns: `5C`

Description: A sync character (`0x5C`) is sent in reply. This instruction is typically used after a reset to verify communications.

TABLE **Table lookup**

Opcode: `85 tc t1…tn` where: `tc` is the size of the table
`t1…tn` are 32-bit floating point or integer values

Description: reg[A] = value from table indexed by reg[0]
This opcode is only valid within a user function stored in the uM-FPU Flash memory or EEPROM memory. The value of the item in the table, indexed by register 0, is stored in register A. The first byte after the opcode specifies the size of the table, followed by groups of four bytes representing the 32-bit values for each item in the table. This instruction can be used to load either floating point values or long integer values. The long integer value in register 0 is used as an index into the table. The index number for the first table entry is zero.

Special Cases:
- if reg[0] <= 0, then the result is item 0
- if reg[0] > maximum size of table, then the result is the last item in the table

---

TAN **Tangent**
Opcode: `49`

Description: reg[A] = tan(reg[A])
Calculates the tangent of the angle (in radians) in register A and stored the result in register A.

Special Cases:
- if reg[A] is NaN or an infinity, then the result is NaN
- if reg[A] is 0.0, then the result is 0.0
- if reg[A] is –0.0, then the result is –0.0

TICKLONG **Load register 0 with millisecond ticks**
Opcode: `D9`

Description: reg[0] = ticks
Load register 0 with the ticks (in milliseconds).

TIMELONG **Load register 0 with time value in seconds**
Opcode: `D8`

Operation: reg[0] = time
Description: Load register 0 with the time (in seconds).

TIMESET **Set time value in seconds**
Opcode: `D7`

Description: time = reg[0], ticks = 0
The time (in seconds) is set from the value in register 0. The ticks (in milliseconds) is set to zero.

Special Cases:
- if reg[0] is -1, the timer is turned off.

TRACEOFF **Turn debug trace off**
Opcode: `F8`

Description: Used with the built-in debugger. If the debugger is not enabled, this instruction is ignored. Debug tracing is turned off, and a `{TRACE OFF}` message is sent to the serial output.

TRACEON **Turn debug trace on**

Opcode: `F9`

Description: Used with the built-in debugger. If the debugger is not enabled, this instruction is ignored. Debug tracing is turned on, and a `{TRACE ON}` message is sent to the serial output. The debug terminal will display a trace of all instructions executed until tracing is turned off.

TRACEREG **Display register value in debug trace**

Opcode: `FB nn` where: `nn` is a register number

Description: Used with the built-in debugger. If the debugger is not enabled, this instruction is ignored. If the debugger is enabled, the value of register `nn` will be displayed on the debug terminal.

TRACESTR **Display debug trace message**

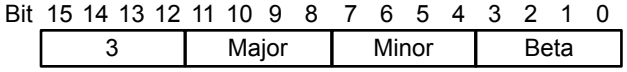
Opcode: `FA aa…00` where: `aa…00` is a zero-terminated string

Description: Used with the built-in debugger. If the debugger is not enabled, this instruction is ignored. If the debugger is enabled, a message will be displayed on the debug terminal. The zero terminated ASCII string to be displayed is sent immediately following the opcode.

VERSION **Copy the version string to the string buffer** *(modified V3.1)*

Opcode: `F3`

Description: The uM-FPU V3.1 version string is copied to the string buffer at the current selection point, and the version code is copied to register 0. The version code is represented as follows:



Bits 12-15 Chip Version (always set to 3)
Bits 8-11 Major Version
Bits 4-7 Minor Version
Bits 0-3 Beta Version

As an example, for the uM-FPU V3.1.3 general release:
version string: uM-FPU V3.1
version code: 0x3130

WRBLK **Write multiple 32-bit values** *(new V3.1)*

Opcode: `70 tc t1…tn` where: `tc` is the number of 32-bit values to write
`t1…tn` are 32-bit values

Description: reg[X] = t, X = X+1, for t = t0 to tn

This instruction is used to write multiple 32-bit values to the uM-FPU registers. The byte immediately following the opcode is the transfer count, and bits 6:0 specify the number of 32-bit values that follow (a value of zero specifies a transfer count of 128). If bit 7 of the transfer count is set, the bytes are reversed for each 32-bit value that follows. This allows for efficient data transfers when the native storage format of the microcontroller is the reverse of the uM-FPU format. The X register specifies the register to write to, and it is incremented after each 32-bit value is written.

Special Cases:
- the X register will not increment past the maximum register value of 127
- if PICMODE is enabled, the 32-bit values are assumed to be floating point values

---

XSAVE **Save register nn to register X**

Opcode: `0E nn` where: `nn` is a register number

Description: reg[X] = reg[nn], X = X + 1
Set register X to the value of register `nn`, and select the next register in sequence as register X.

Special Cases: • the X register will not increment past the maximum register value of 127

XSAVEA **Save register A to register X**

Opcode: `0F`

Description: reg[X] = reg[A], X = X + 1
Set register X to the value of register A, and select the next register in sequence as register X.

Special Cases: • the X register will not increment past the maximum register value of 127

Appendix A
uM-FPU V3.1 Instruction Summary

Instruction	Opcode	Arguments	Returns	Description
NOP	00			No Operation
SELECTA	01	nn		Select register A
SELECTX	02	nn		Select register X
CLR	03	nn		reg[nn] = 0
CLRA	04			reg[A] = 0
CLRX	05			reg[X] = 0, X = X + 1
CLR0	06			reg[0] = 0
COPY	07	mm,nn		reg[nn] = reg[mm]
COPYA	08	nn		reg[nn] = reg[A]
COPYX	09	nn		reg[nn] = reg[X], X = X + 1
LOAD	0A	nn		reg[0] = reg[nn]
LOADA	0B			reg[0] = reg[A]
LOADX	0C			reg[0] = reg[X], X = X + 1
ALOADX	0D			reg[A] = reg[X], X = X + 1
XSAVE	0E	nn		reg[X] = reg[nn], X = X + 1
XSAVEA	0F			reg[X] = reg[A], X = X + 1
COPY0	10	nn		reg[nn] = reg[0]
COPYI	11	bb,nn		reg[nn] = long(unsigned byte bb)
SWAP	12	nn,mm		Swap reg[nn] and reg[mm]
SWAPA	13	nn		Swap reg[nn] and reg[A]
LEFT	14			Left parenthesis
RIGHT	15			Right parenthesis
FWRITE	16	nn,b1,b2,b3,b4		Write 32-bit floating point to reg[nn]
FWRITEA	17	b1,b2,b3,b4		Write 32-bit floating point to reg[A]
FWRITEX	18	b1,b2,b3,b4		Write 32-bit floating point to reg[X]
FWRITE0	19	b1,b2,b3,b4		Write 32-bit floating point to reg[0]
FREAD	1A	nn	b1,b2,b3,b4	Read 32-bit floating point from reg[nn]
FREADA	1B		b1,b2,b3,b4	Read 32-bit floating point from reg[A]
FREADX	1C		b1,b2,b3,b4	Read 32-bit floating point from reg[X]
FREAD0	1D		b1,b2,b3,b4	Read 32-bit floating point from reg[0]
ATOF	1E	aa…00		Convert ASCII to floating point
FTOA	1F	bb		Convert floating point to ASCII
FSET	20	nn		reg[A] = reg[nn]
FADD	21	nn		reg[A] = reg[A] + reg[nn]
FSUB	22	nn		reg[A] = reg[A] - reg[nn]
FSUBR	23	nn		reg[A] = reg[nn] - reg[A]
FMUL	24	nn		reg[A] = reg[A] * reg[nn]
FDIV	25	nn		reg[A] = reg[A] / reg[nn]
FDIVR	26	nn		reg[A] = reg[nn] / reg[A]
FPOW	27	nn		reg[A] = reg[A] ** reg[nn]
FCMP	28	nn		Compare reg[A], reg[nn], Set floating point status
FSET0	29			reg[A] = reg[0]
FADD0	2A			reg[A] = reg[A] + reg[0]
FSUB0	2B			reg[A] = reg[A] - reg[0]

FSUBR0	2C			reg[A] = reg[0] - reg[A]
FMUL0	2D			reg[A] = reg[A] * reg[0]
FDIV0	2E			reg[A] = reg[A] / reg[0]
FDIVR0	2F			reg[A] = reg[0] / reg[A]
FPOW0	30			reg[A] = reg[A] ** reg[0]
FCMP0	31			Compare reg[A], reg[0], Set floating point status
FSETI	32	bb		reg[A] = float(bb)
FADDI	33	bb		reg[A] = reg[A] - float(bb)
FSUBI	34	bb		reg[A] = reg[A] - float(bb)
FSUBRI	35	bb		reg[A] = float(bb) - reg[A]
FMULI	36	bb		reg[A] = reg[A] * float(bb)
FDIVI	37	bb		reg[A] = reg[A] / float(bb)
FDIVRI	38	bb		reg[A] = float(bb) / reg[A]
FPOWI	39	bb		reg[A] = reg[A] ** bb
FCMPI	3A	bb		Compare reg[A], float(bb), Set floating point status
FSTATUS	3B	nn		Set floating point status for reg[nn]
FSTATUSA	3C			Set floating point status for reg[A]
FCMP2	3D	nn,mm		Compare reg[nn], reg[mm] Set floating point status
FNEG	3E			reg[A] = -reg[A]
FABS	3F			reg[A] = | reg[A] |
FINV	40			reg[A] = 1 / reg[A]
SQRT	41			reg[A] = sqrt(reg[A])
ROOT	42	nn		reg[A] = root(reg[A], reg[nn])
LOG	43			reg[A] = log(reg[A])
LOG10	44			reg[A] = log10(reg[A])
EXP	45			reg[A] = exp(reg[A])
EXP10	46			reg[A] = exp10(reg[A])
SIN	47			reg[A] = sin(reg[A])
COS	48			reg[A] = cos(reg[A])
TAN	49			reg[A] = tan(reg[A])
ASIN	4A			reg[A] = asin(reg[A])
ACOS	4B			reg[A] = acos(reg[A])
ATAN	4C			reg[A] = atan(reg[A])
ATAN2	4D	nn		reg[A] = atan2(reg[A], reg[nn])
DEGREES	4E			reg[A] = degrees(reg[A])
RADIANS	4F			reg[A] = radians(reg[A])
FMOD	50	nn		reg[A] = reg[A] MOD reg[nn]
FLOOR	51			reg[A] = floor(reg[A])
CEIL	52			reg[A] = ceil(reg[A])
ROUND	53			reg[A] = round(reg[A])
FMIN	54	nn		reg[A] = min(reg[A], reg[nn])
FMAX	55	nn		reg[A] = max(reg[A], reg[nn])
FCNV	56	bb		reg[A] = conversion(bb, reg[A])
FMAC	57	nn,mm		reg[A] = reg[A] + (reg[nn] * reg[mm])
FMSC	58	nn,mm		reg[A] = reg[A] - (reg[nn] * reg[mm])
LOADBYTE	59	bb		reg[0] = float(signed bb)
LOADUBYTE	5A	bb		reg[0] = float(unsigned byte)

LOADWORD	5B	b1,b2		reg[0] = float(signed b1*256 + b2)
LOADUWORD	5C	b1,b2		reg[0] = float(unsigned b1*256 + b2)
LOADE	5D			reg[0] = 2.7182818
LOADPI	5E			reg[0] = 3.1415927
LOADCON	5F	bb		reg[0] = float constant(bb)
FLOAT	60			reg[A] = float(reg[A])
FIX	61			reg[A] = fix(reg[A])
FIXR	62			reg[A] = fix(round(reg[A]))
FRAC	63			reg[A] = fraction(reg[A])
FSPLIT	64			reg[A] = integer(reg[A]), reg[0] = fraction(reg[A])
SELECTMA	65	nn,bb,bb		Select matrix A
SELECTMB	66	nn,bb,bb		Select matrix B
SELECTMC	67	nn,bb,bb		Select matrix C
LOADMA	68	bb,bb		reg[0] = Matrix A[bb, bb]
LOADMB	69	bb,bb		reg[0] = Matrix B[bb, bb]
LOADMC	6A	bb,bb		reg[0] = Matrix C[bb, bb]
SAVEMA	6B	bb,bb		Matrix A[bb, bb] = reg[A]
SAVEMB	6C	bb,bb		Matrix B[bb, bb] = reg[A]
SAVEMC	6D	bb,bb		Matrix C[bb, bb] = reg[A]
MOP	6E	bb		Matrix/Vector operation
FFT	6F	bb		Fast Fourier Transform
WRBLK	70	tc,t1…tn		Write multiple 32-bit values
RDBLK	71	tc	t1…tn	Read multiple 32-bit values
LOADIND	7A	nn		reg[0] = reg[reg[nn]]
SAVEIND	7B	nn		reg[reg[nn]] = reg[A]
INDA	7C	nn		Select register A using value in reg[nn]
INDX	7D	nn		Select register X using value in reg[nn]
FCALL	7E	fn		Call user-defined function in Flash
EECALL	7F	fn		Call user-defined function in EEPROM
RET	80			Return from user-defined function
BRA	81	bb		Unconditional branch
BRA,cc	82	cc,bb		Conditional branch
JMP	83	b1,b2		Unconditional jump
JMP,cc	84	cc,b1,b2		Conditional jump
TABLE	85	tc,t1…tn		Table lookup
FTABLE	86	cc,tc,t1…tn		Floating point reverse table lookup
LTABLE	87	cc,tc,t1…tn		Long integer reverse table lookup
POLY	88	tc,t1…tn		reg[A] = nth order polynomial
GOTO	89	nn		Computed GOTO
RET,cc	8A	cc		Conditional return from user-defined function
LWRITE	90	nn,b1,b2,b3,b4		Write 32-bit long integer to reg[nn]
LWRITEA	91	b1,b2,b3,b4		Write 32-bit long integer to reg[A]
LWRITEX	92	b1,b2,b3,b4		Write 32-bit long integer to reg[X], X = X + 1
LWRITE0	93	b1,b2,b3,b4		Write 32-bit long integer to reg[0]
LREAD	94	nn	b1,b2,b3,b4	Read 32-bit long integer from reg[nn]
LREADA	95		b1,b2,b3,b4	Read 32-bit long value from reg[A]

LREADX	96		b1,b2,b3,b4	Read 32-bit long integer from reg[X], X = X + 1
LREAD0	97		b1,b2,b3,b4	Read 32-bit long integer from reg[0]
LREADBYTE	98		bb	Read lower 8 bits of reg[A]
LREADWORD	99		b1,b2	Read lower 16 bits reg[A]
ATOL	9A	aa…00		Convert ASCII to long integer
LTOA	9B	bb		Convert long integer to ASCII
LSET	9C	nn		reg[A] = reg[nn]
LADD	9D	nn		reg[A] = reg[A] + reg[nn]
LSUB	9E	nn		reg[A] = reg[A] - reg[nn]
LMUL	9F	nn		reg[A] = reg[A] * reg[nn]
LDIV	A0	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LCMP	A1	nn		Signed compare reg[A] and reg[nn], Set long integer status
LUDIV	A2	nn		reg[A] = reg[A] / reg[nn] reg[0] = remainder
LUCMP	A3	nn		Unsigned compare reg[A] and reg[nn], Set long integer status
LTST	A4	nn		Test reg[A] AND reg[nn], Set long integer status
LSET0	A5			reg[A] = reg[0]
LADD0	A6			reg[A] = reg[A] + reg[0]
LSUB0	A7			reg[A] = reg[A] - reg[0]
LMUL0	A8			reg[A] = reg[A] * reg[0]
LDIV0	A9			reg[A] = reg[A] / reg[0] reg[0] = remainder
LCMP0	AA			Signed compare reg[A] and reg[0], set long integer status
LUDIV0	AB			reg[A] = reg[A] / reg[0] reg[0] = remainder
LUCMP0	AC			Unsigned compare reg[A] and reg[0], Set long integer status
LTST0	AD			Test reg[A] AND reg[0], Set long integer status
LSETI	AE	bb		reg[A] = long(bb)
LADDI	AF	bb		reg[A] = reg[A] + long(bb)
LSUBI	B0	bb		reg[A] = reg[A] - long(bb)
LMULI	B1	bb		reg[A] = reg[A] * long(bb)
LDIVI	B2	bb		reg[A] = reg[A] / long(bb) reg[0] = remainder
LCMPI	B3	bb		Signed compare reg[A] - long(bb), Set long integer status
LUDIVI	B4	bb		reg[A] = reg[A] / unsigned long(bb) reg[0] = remainder
LUCMPI	B5	bb		Unsigned compare reg[A] and long(bb), Set long integer status
LTSTI	B6	bb		Test reg[A] AND long(bb), Set long integer status
LSTATUS	B7	nn		Set long integer status for reg[nn]
LSTATUSA	B8			Set long integer status for reg[A]

LCMP2	B9	nn,mm		Signed long compare reg[nn], reg[mm] Set long integer status
LUCMP2	BA	nn,mm		Unsigned long compare reg[nn], reg[mm] Set long integer status
LNEG	BB			reg[A] = -reg[A]
LABS	BC			reg[A] = l reg[A] l
LINC	BD	nn		reg[nn] = reg[nn] + 1, set status
LDEC	BE	nn		reg[nn] = reg[nn] - 1, set status
LNOT	BF			reg[A] = NOT reg[A]
LAND	C0	nn		reg[A] = reg[A] AND reg[nn]
LOR	C1	nn		reg[A] = reg[A] OR reg[nn]
LXOR	C2	nn		reg[A] = reg[A] XOR reg[nn]
LSHIFT	C3	nn		reg[A] = reg[A] shift reg[nn]
LMIN	C4	nn		reg[A] = min(reg[A], reg[nn])
LMAX	C5	nn		reg[A] = max(reg[A], reg[nn])
LONGBYTE	C6	bb		reg[0] = long(signed byte bb)
LONGUBYTE	C7	bb		reg[0] = long(unsigned byte bb)
LONGWORD	C8	b1,b2		reg[0] = long(signed b1*256 + b2)
LONGUWORD	C9	b1,b2		reg[0] = long(unsigned b1*256 + b2)
SETSTATUS	CD	ss		Set status byte
SEROUT	CE	bb bb,bd bb,aa…00		Serial output
SERIN	CF	bb		Serial input
SETOUT	D0	bb		Set OUT1 and OUT2 output pins
ADCMODE	D1	bb		Set A/D trigger mode
ADCTRIG	D2			A/D manual trigger
ADCSCALE	D3	ch		ADCscale[ch] = reg[0]
ADCLONG	D4	ch		reg[0] = ADCvalue[ch]
ADCLOAD	D5	ch		reg[0] = float(ADCvalue[ch]) * ADCscale[ch]
ADCWAIT	D6			wait for next A/D sample
TIMESET	D7			time = reg[0]
TIMELONG	D8			reg[0] = time (long integer)
TICKLONG	D9			reg[0] = ticks (long integer)
EESAVE	DA	nn,ee		EEPROM[ee] = reg[nn]
EESAVEA	DB	ee		EEPROM[ee] = reg[A]
EELOAD	DC	nn,ee		reg[nn] = EEPROM[ee]
EELOADA	DD	ee		reg[A] = EEPROM[ee]
EEWRITE	DE	ee,bc,b1…bn		Store bytes starting at EEPROM[ee]
EXTSET	E0			external input count = reg[0]
EXTLONG	E1			reg[0] = external input counter
EXTWAIT	E2			wait for next external input
STRSET	E3	aa…00		Copy string to string buffer
STRSEL	E4	bb,bb		Set selection point
STRINS	E5	aa…00		Insert string at selection point
STRCMP	E6	aa…00		Compare string with string selection
STRFIND	E7	aa…00		Find string
STRFCHR	E8	aa…00		Set field separators
STRFIELD	E9	bb		Find field

STRTOF	EA			Convert string selection to floating point
STRTOL	EB			Convert string selection to long integer
READSEL	EC		aa…00	Read string selection
STRBYTE	ED	bb		Insert byte at selection point
STRINC	EE			Increment string selection point
STRDEC	EF			Decrement string selection point
SYNC	F0		5C	Get synchronization byte
READSTATUS	F1		ss	Read status byte
READSTR	F2		aa…00	Read string from string buffer
VERSION	F3			Copy version string to string buffer
IEEEMODE	F4			Set IEEE mode (default)
PICMODE	F5			Set PIC mode
CHECKSUM	F6			Calculate checksum for uM-FPU code
BREAK	F7			Debug breakpoint
TRACEOFF	F8			Turn debug trace off
TRACEON	F9			Turn debug trace on
TRACESTR	FA	aa…00		Send string to debug trace buffer
TRACEREG	FB	nn		Send register value to trace buffer
READVAR	FC	bb		Read internal register value
RESET	FF			Reset (9 consecutive FF bytes cause a reset, otherwise it is a NOP)

Notes:

Opcode	Opcode value in hexadecimal
Arguments	Additional data required by instruction
Returns	Data returned by instruction
nn	register number (0-127)
mm	register number (0-127)
fn	function number (0-63)
bb	8-bit value
b1,b2	16-bit value (b1 is MSB)
b1,b2,b3,b4	32-bit value (b1 is MSB)
b1…bn	string of 8-bit bytes
ss	Status byte
bd	baud rate and debug mode
cc	Condition code
ee	EEPROM address slot (0-255)
ch	A/D channel number
bc	Byte count
tc	32-bit value count
t1…tn	String of 32-bit values
aa…00	Zero terminated ASCII string