



**Micromega Corporation**

# **uM-FPU64**

## **64-bit Floating Point Coprocessor**

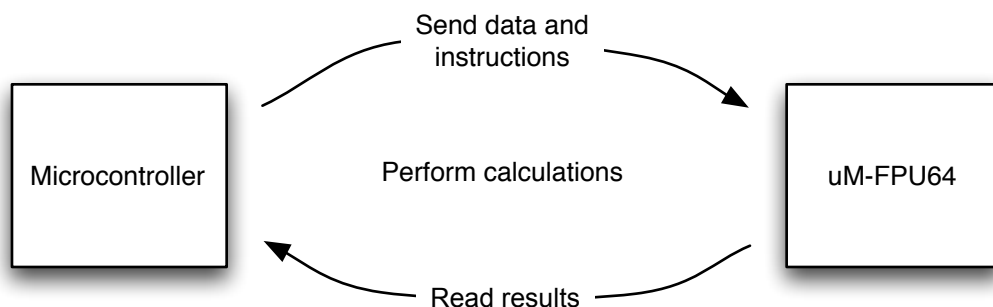
# **Instruction Set**

## **Release 411**

## **Introduction**

The uM-FPU64 floating point coprocessor provides extensive support for 32-bit IEEE 754 compatible floating point and integer operations, 64-bit IEEE 754 compatible floating point and integer operations, and local peripheral device support.

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.



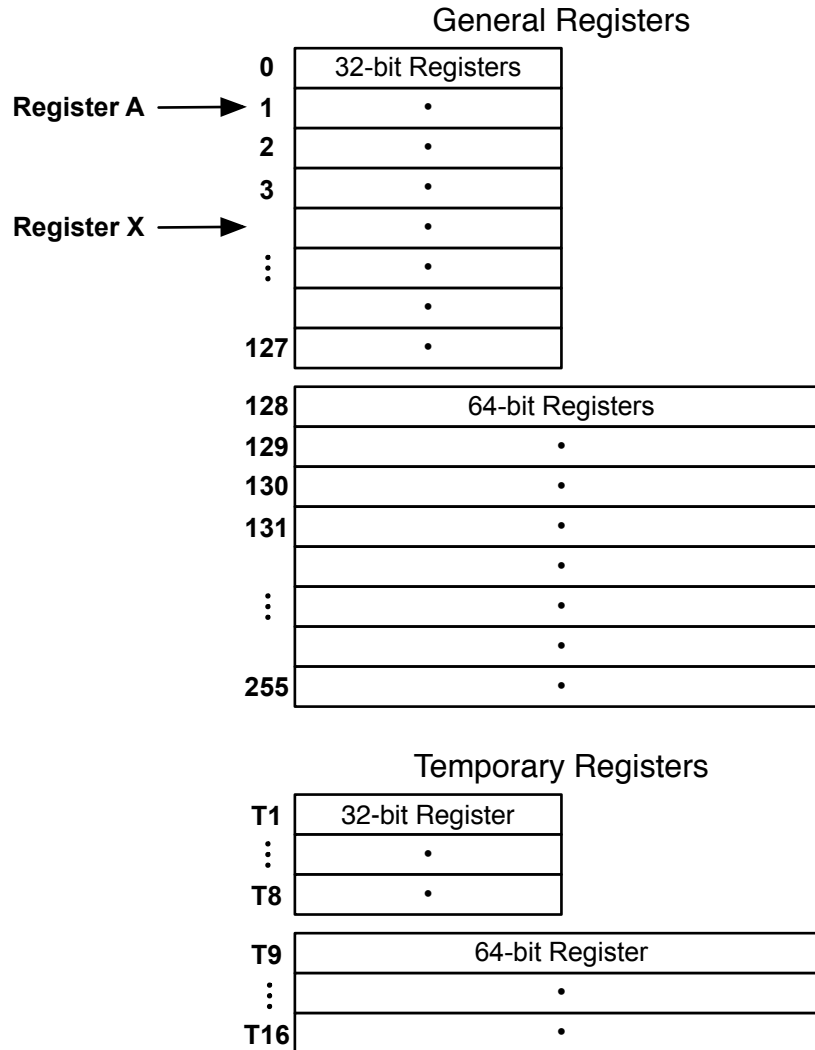
Instructions and data are sent to the uM-FPU using either a SPI or I<sup>2</sup>C interface. The uM-FPU64 chip has a 256 byte instruction buffer which allows for multiple instructions to sent. This improves the transfer times and allows the microcontroller to perform other tasks while the uM-FPU is performing 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<sup>2</sup>C 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-FPU64 Datasheet.

## uM-FPU Registers

The uM-FPU64 has 256 general purpose registers, and 16 temporary registers. They can be used for storing floating point or integer values. The general purpose registers are numbered 0 to 255, and can be directly accessed by the instruction set. Registers 0 to 127 are 32-bit registers, and registers 128 to 255 are 64-bit registers. The 16 temporary registers are used by the **LEFT** and **RIGHT** instructions to store temporary results. They can be accessed through register A, but can't be accessed directly by the instruction set.



## Register A

All mathematical operations on the uM-FPU64 use a working register called register A. The value in register A is used as an operand for the mathematical operation, and the results of the operation are stored back to register A. Any general purpose registers can be selected as register A, using the **SELECTA** instruction. For example,

**SELECTA, 5**                      *register 5 is selected as register A*

Register A also determines whether an operation is a 32-bit operation or a 64-bit operation. If register A is selected as register 0 to 127, the operation is 32-bit. If register A is selected as 128 to 255, the operation is 64-bit.

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*  
**CLR X**                      *clear register 16 (and increment register X)*  
**CLR X**                      *clear register 17 (and increment register X)*  
**CLR X**                      *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.

## Register 0 and Register 128

Register 0 and register 128 are implicitly used by many instructions. Register 0 is used for 32-bit operations, and register 128 is used for 64-bit operations. They are used by many instructions to pass values or to return values.

Register 0 and register 128 can be used as general purpose registers, but since many instructions use these registers, they are normally only used to store temporary values. For example,

**LOADPI**                      *load the value of pi to register 0 or 128*  
**FSET0**                      *store the value to register A*

## Register Abbreviations

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

reg[0]	register 0 (32-bit)
reg[128]	register 128 (64-bit)
reg[0   128]	register 0 (32-bit) or register 128 (64-bit)
reg[A]	register A
reg[X]	register X
reg[register]	any general purpose registers
reg[register1]	any general purpose registers
reg[register2]	any 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 is used as an example to describe the three different forms of these instructions. The **FADD, register** 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 **FADDI** 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, register** instruction is most general, but the **FADD0** and **FADDI, signedByte** instructions are more efficient for many common operations.

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

### Loading Floating Point Values

The following instructions are used to load data from the microprocessor and store it on the uM-FPU64 as 32-bit floating point values. Register A determines whether 32-bit or 64-bit values are stored.

<i>FWRITE, register, float32Value</i>	Write 32-bit floating point value to <i>register</i>
<i>FWRITE0, float32Value</i>	Write 32-bit floating point value to reg[0] or reg[128]
<i>FWRITEA, float32Value</i>	Write 32-bit floating point value to reg[A]
<i>FWRITEX, float32Value</i>	Write 32-bit floating point value to reg[X]
<i>DWRITE, register, float64Value</i>	Write 64-bit floating point value to <i>register</i>
<i>ATOF, string</i>	Convert ASCII string to floating point value and store in reg[0] or reg[128]
<i>LOADBYTE, signedByte</i>	Convert signed byte to floating point and store in reg[0] or reg[128]
<i>LOADUBYTE, unsignedByte</i>	Convert unsigned byte to floating point and store in reg[0] or reg[128]
<i>LOADWORD, signedWord</i>	Convert signed 16-bit value to floating point and store in reg[0] or reg[128]
<i>LOADUWORD, unsignedWord</i>	Convert unsigned 16-bit value to floating point and store in reg[0] or reg[128]
LOADE	Load the value of e (2.7182818) to reg[0] or reg[128]
LOADPI	Load the value of pi (3.1415927) to reg[0] or reg[128]
<i>FCOPYI, unsignedByte, register</i>	Convert signed 8-bit value to floating point and store in register

## Reading Floating Point Values

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

<code>FREAD, register [float32Value]</code>	Return 32-bit floating point value from <i>register</i>
<code>FREAD0 [float32Value]</code>	Return 32-bit floating point value from reg[0] or reg[128]
<code>FREADA [float32Value]</code>	Return 32-bit floating point value from reg[A]
<code>FREADX [float32Value]</code>	Return 32-bit floating point value from reg[X]
<code>DREAD, register [float64Value]</code>	Return 64-bit floating point value from <i>register</i>
<code>FTOA, format</code>	Convert floating point to ASCII string (READSTR used to read string)

## Additional Floating Point Instructions

<code>FNEG</code>	<code>SIN</code>	<code>FSTATUS, register</code>	<code>FLOOR</code>
<code>FABS</code>	<code>COS</code>	<code>FSTATUSA</code>	<code>CEIL</code>
<code>FINV</code>	<code>TAN</code>	<code>FCMP2, register1, register2</code>	<code>ROUND</code>
<code>SQRT</code>	<code>ASIN</code>	<code>FMAC, register1, register2</code>	<code>FRAC</code>
<code>ROOT, register</code>	<code>ACOS</code>	<code>FMSC, register1, register2</code>	
<code>LOG</code>	<code>ATAN</code>	<code>FCNV, conversion</code>	
<code>LOG10</code>	<code>ATAN2, register</code>	<code>FMIN, register</code>	
<code>EXP</code>	<code>DEGREES</code>	<code>FMAX, register</code>	
<code>EXP10</code>	<code>RADIANS</code>	<code>FMOD</code>	

## Matrix Instructions

<code>SELECTMA, register, rows, columns</code>	Select matrix A starting at <i>register</i> of size <i>rows</i> x <i>columns</i>
<code>SELECTMB, register, rows, columns</code>	Select matrix B starting at <i>register</i> of size <i>rows</i> x <i>columns</i>
<code>SELECTMC, register, rows, columns</code>	Select matrix C starting at <i>register</i> of size <i>rows</i> x <i>columns</i>
<code>LOADMA, row, column</code>	Load reg[0] with value from matrix A <i>row, column</i>
<code>LOADMB, row, column</code>	Load reg[0] with value from matrix B <i>row, column</i>
<code>LOADMC, row, column</code>	Load reg[0] with value from matrix C <i>row, column</i>
<code>SAVEMA, row, column</code>	Store reg[A] value to matrix A <i>row, column</i>
<code>SAVEMB, row, column</code>	Store reg[A] value to matrix A <i>row, column</i>
<code>SAVEMC, row, column</code>	Store reg[A] value to matrix A <i>row, column</i>
<code>MOP, action</code>	Perform matrix operation

## Fast Fourier Transform Instruction

<code>FFT, action</code>	Perform Fast Fourier Transform operation
--------------------------	--

## Conversion Instructions

<code>FLOAT</code>	Convert reg[A] from long integer to floating point
<code>FIX</code>	Convert reg[A] from floating point to long integer
<code>FIXR</code>	Convert reg[A] from floating point to long integer (with rounding)
<code>FSPLIT</code>	Set reg[A] = integer value, reg[0] or reg[128] = 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 describe the three different forms of the instructions. The **LADD, register** 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 **LADDI** 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, register** instruction is most general, but the **LADD0** and **LADDI, signedByte** instructions are more efficient for many common operations.

Register	Register 0	Immediate value	Description
<b>LSET, register</b>	LSET0	LSETI, signedByte	Set
<b>LADD, register</b>	LADD0	LADDI, signedByte	Add
<b>LSUB, register</b>	LSUB0	LSUBI, signedByte	Subtract
<b>LMUL, register</b>	LMUL0	LMULI, signedByte	Multiply
<b>LDIV, register</b>	LDIV0	LDIVI, signedByte	Divide
<b>LCMP, register</b>	LCMP0	LCMPI, signedByte	Compare
<b>LUDIV, register</b>	LUDIV0	LUDIVI, unsignedByte	Unsigned Divide
<b>LUCMP, register</b>	LUCMP0	LUCMPI, unsignedByte	Unsigned Compare
<b>LTST, register</b>	LTST0	LTSTI, unsignedByte	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.

<b>LWRITE, register, int32Value</b>	Write 32-bit long integer value to <i>register</i>
<b>LWRITE0, int32Value</b>	Write 32-bit long integer value to reg[0] or reg[128]
<b>LWRITEA, int32Value</b>	Write 32-bit long integer value to reg[A]
<b>LWRITEX, int32Value</b>	Write 32-bit long integer value to reg[X]
<b>DWRITE, register, intValue</b>	Write 64-bit floating point value to <i>register</i>
<b>ATOL, string</b>	Convert ASCII string to long integer value and store in reg[0] or reg[128]
<b>LONGBYTE, signedByte</b>	Convert signed byte to long integer and store in reg[0] or reg[128]
<b>LONGUBYTE, unsignedByte</b>	Convert unsigned byte to long integer and store in reg[0] or reg[128]
<b>LONGWORD, signedWord</b>	Convert signed 16-bit value to long integer and store in reg[0] or reg[128]
<b>LONGUWORD, unsignedByte</b>	Convert unsigned 16-bit value to long integer and store in reg[0] or reg[128]
<b>LCOPYI, unsignedByte, register</b>	Convert signed 8-bit value to long integer and store in register

## Reading Long Integer Values

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

<code>LREAD, register [int32Value]</code>	Returns 32-bit long integer value from <i>register</i>
<code>LREAD0 [int32Value]</code>	Returns 32-bit long integer value from reg[0] or reg[128]
<code>LREADA [int32Value]</code>	Returns 32-bit long integer value from reg[A]
<code>LREADX [int32Value]</code>	Returns 32-bit long integer value from reg[X]
<code>DREAD, register [int64Value]</code>	Return 64-bit floating point value from <i>register</i>
<code>LREADBYTE [byteValue]</code>	Returns 8-bit byte from reg[A]
<code>LREADWORD [wordValue]</code>	Returns 16-bit value from reg[A]
<code>LTOA, format</code>	Convert long integer to ASCII string (use READSTR to read string)

## Additional Long Integer Instructions

<code>LSTATUS, register</code>	<code>LCMP2, register1, register2</code>	<code>LAND, register</code>
<code>LSTATUSA</code>	<code>LUCMP2, register1, register2</code>	<code>LANDI, unsignedByte</code>
		<code>e</code>
<code>LNEG</code>	<code>LMIN, register</code>	<code>LOR, register</code>
<code>LABS</code>	<code>LMAX, register</code>	<code>LORI, unsignedByte</code>
<code>LNOT</code>	<code>LSHIFT, register</code>	<code>LXOR, register</code>
<code>LINC, register</code>	<code>LSHIFTI, signedByte</code>	
<code>LDEC, register</code>	<code>LBIT, unsignedByte, register</code>	

## General Purpose Instructions

<i>SELECTA, register</i>	<i>ALOADX</i>	<i>XSAVE, register</i>	<i>SETREAD</i>
<i>SELECTX, register</i>	<i>COPY, register1, register2</i>	<i>XSAVEA</i>	<i>SYNC</i>
<i>CLR, register</i>	<i>COPY0, register</i>	<i>INDA</i>	<i>VERSION</i>
<i>CLRA</i>	<i>COPYA, register</i>	<i>INDX</i>	<i>IEEEMODE</i>
<i>CLR<sub>X</sub></i>	<i>COPYX, register</i>	<i>LEFT</i>	<i>PICMODE</i>
<i>CLR<sub>0</sub></i>	<i>SWAP, register1, register2</i>	<i>RIGHT</i>	<i>SETARGS</i>
<i>LOAD, register</i>	<i>SWAPA, register</i>	<i>READVAR, item</i>	<i>CHECKSUM</i>
<i>LOADA</i>	<i>SETSTATUS, unsignedbyte</i>	<i>RESET</i>	<i>XOP</i>
<i>LOADX</i>	<i>READSTATUS</i>	<i>NOP</i>	

## Special Purpose Instructions

### Indirect Pointer Instructions

<i>SETIND, type, {register   address}</i>	Set indirect pointer
<i>ADDIND, register, unsignedByte</i>	Add to indirect pointer
<i>COPYIND, fromPtr, toPtr, countReg</i>	Copy using indirect pointers
<i>LOADIND, register</i>	Load reg[0   128] using indirect pointer
<i>SAVEIND, register</i>	Save reg[A] using indirect pointer
<i>RDIND, type, count [dataValue1...dataValueN]</i>	Read multiple data values from indirect pointer
<i>WRIND, type, count, dataValue1...dataValueN</i>	Write multiple data values to indirect pointer

### Stored Function Instructions

<i>FCALL, function</i>	Call user-defined function stored in Flash
<i>RET</i>	Return from user-defined function
<i>RET, conditionCode</i>	Conditional return from user-defined function
<i>BRA, relativeOffset</i>	Unconditional branch inside user-defined function
<i>BRA, conditionCode, relativeOffset</i>	Conditional branch inside user-defined function
<i>JMP, absoluteOffset</i>	Unconditional jump inside user-defined function
<i>JMP, conditionCode, absoluteOffset</i>	Conditional jump inside user-defined function
<i>GOTO, register</i>	Computed goto
<i>TABLE, tableSize, tableItem1...tableItemN</i>	Table lookup
<i>FTABLE, conditionCode, tableSize, tableItem1...tableItemN</i>	Floating point reverse table lookup
<i>LTABLE, conditionCode, tableSize, tableItem1...tableItemN</i>	Long integer reverse table lookup
<i>POLY, count, floatValue1...floatValueN</i>	N <sup>th</sup> order polynomial

### Background Event Processing

<i>EVENT, action{, function}</i>	Background event processing
----------------------------------	-----------------------------

### Analog to Digital Conversion Instructions

<i>ADCMODE, mode</i>	Select A/D trigger mode
<i>ADCTRIG</i>	Manual A/D trigger
<i>ADCSCALE, channel</i>	Set A/D floating point scale factor
<i>ADCLONG, channel</i>	Get raw long integer A/D reading
<i>ADCLOAD, channel</i>	Get scaled floating point A/D reading



ADCWAIT                      Wait for A/D conversion to complete

## Digital I/O Instructions

DIGIO, *action{,mode}*              Digital I/O  
DEVIO, *device,action{,...}*      Device I/O

## Timer Instructions

TIMESET                      Set timers  
TIMELONG                      Get time in seconds  
TICKLONG                      Get time in milliseconds  
RTC, *action*                      Real-time Clock  
DELAY, *period*                      Delay (in milliseconds)

## External Input Instructions

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

## String Manipulation Instructions

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

## Serial Input/Output

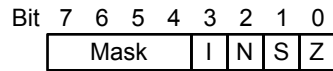
SEROUT, *action{,...}*              Serial Output  
SERIN, *action*                      Serial Input

## Debugging Instructions

BREAK                      Debug breakpoint  
TRACEOFF                      Turn debug trace off  
TRACEON                      Turn debug trace on  
TRACESTR, *string*                      Display string in debug trace  
TRACEREG, *register*                      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 7:4 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 3:0 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:

<i>Assembler Symbol</i>	<i>Test Condition</i>	<i>Description</i>
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

# uM-FPU64 Instruction Reference

## ACOS Arc Cosine

*Syntax:* **ACOS**

*Description:* 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 stored in register A.

$\text{reg}[A] = \text{acos}(\text{reg}[A])$

*Opcode:* **4B**

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

*See Also:* ASIN, ATAN, ATAN2, COS, SIN, TAN, DEGREES, RADIANS

## ADCLOAD Load scaled analog value

*Syntax:* **ADCLOAD, channel**

*Description:* Loads register 0 with the scaled floating point value of the analog reading from the specified channel.

if reg[A] is 32-bit,

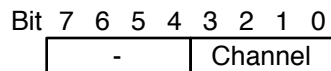
$\text{reg}[0] = (\text{float}(\text{ADCvalue}[\text{channel}] * \text{ADCscale}[\text{channel}])) + \text{ADCoffset}[\text{channel}]$

if reg[A] is 64-bit,

$\text{reg}[128] = (\text{float}(\text{ADCvalue}[\text{channel}] * \text{ADCscale}[\text{channel}])) + \text{ADCoffset}[\text{channel}]$

*Opcode:* **D5**

*Byte 2:* **channel**



Bits 3:0	Channel	Description
Value		
0 to 5		28-pin chip (AN0 to AN5)
0 to 8		44-pin chip (AN0 to AN8)

Waits until the analog-to-digital conversion is complete, then loads register 0 with the reading from the specified analog channel. The 12-bit value is converted to floating point, multiplied by the scale value for the selected channel, and added to the offset for the selected channel. The value is stored in register 0.

Note: 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.

*See Also:* ADCLONG, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

## ADCLONG Load raw analog value

*Syntax:* **ADCLONG, channel**

*Description:* Loads register 0 with the long integer value of the raw analog reading from the specified channel, or a pointer to the memory buffer containing the analog readings (if the PTR bit is set).

if reg[A] is 32-bit, reg[0] = ADCvalue[channel], status = longStatus(reg[0])

if reg[A] is 64-bit, reg[128] = ADCvalue[channel], status = longStatus(reg[128])

*Opcode:* **D4**

*Byte 2:* **channel**

Bit	7	6	5	4	3	2	1	0
	-	S	P	Channel				

Bit 5	<b>Size</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	0x00	Stores the analog reading in register 0.
	SIZE	0x20	Stores the size of the memory buffer in register 0. Used in block mode.
Bit 4	<b>Pointer</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	0x00	Stores the analog reading in register 0.
	PTR	0x10	Stores a pointer to the memory buffer register 0. Used in block mode.
Bits 3:0	<b>Channel</b>		
	<i>Value</i>	<i>Description</i>	
	0 to 5	28-pin chip (AN0 to AN5)	
	0 to 8	44-pin chip (AN0 to AN8)	

Waits until the analog-to-digital conversion is complete, then loads register 0 with the selected value. If bit 4 is zero, the 12-bit value reading from the specified analog channel is converted to a long integer and stored in register 0. If bit 4 is one, a pointer to the memory buffer containing the analog reading is stored in register 0. The memory buffer stores the 12-bit analog reading in sequential 16-bit words. The pointer option is normally used only when the ADC is configured for block mode sampling.

Note: 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 returned.

*See Also:* ADCLOAD, ADCMODE, ADCSCALE, ADCTRIG, ADCWAIT

## ADCMODE Set ADC trigger mode

*Syntax:*       **ADCMODE, mode**

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

*Opcode:*       **D1**

*Byte 2:*       **mode**

Bit	7	6	5	4	3	2	1	0
	Action				Options			

Bits 7:4	<b>Action</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	DISABLE	0x00	Disable analog conversions
	MANUAL	0x10	Manual trigger
	EXTIN	0x20	External trigger
	TIMER	0x30	Timer trigger
	EXTIN_BLK	0x40	External trigger, block mode
	TIMER_BLK	0x50	Timer trigger, block mode
	CHANNELS	0x60	Maximum number of ADC channels
	VREF	0x70	Select voltage reference

Bits 3:0       **Options**  
*See descriptions below.*

### DISABLE

ADCMODE, DISABLE  
 Disable analog conversions.

Bit	7	6	5	4	3	2	1	0
	0				-			

### MANUAL

ADCMODE, MANUAL+*repeat*  
 Manual trigger, single sample with repeat.

Bit	7	6	5	4	3	2	1	0
	1				Repeat			

Bits 3:0	<b>Repeat Count</b>	
	<i>Value</i>	<i>Description</i>
	0 to 15	For modes 1 to 3, 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. a value of 15 will result in 16 samples per trigger.

### EXTIN

ADCMODE, EXTIN+*repeat*  
 External trigger, single sample with repeat.

Bit	7	6	5	4	3	2	1	0
				2				Repeat

Bits 3:0

### Repeat Count

*Value*      *Description*

0 to 15      For modes 1 to 3, 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.  
a value of 15 will result in 16 samples per trigger.

## TIMER

ADCMODE, TIMER+*repeat*

Timer trigger, single sample with repeat.

Bit	7	6	5	4	3	2	1	0
				3				Repeat

Bits 3:0

### Repeat Count

*Value*      *Description*

0 to 15      For modes 1 to 3, 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.  
a value of 15 will result in 16 samples per trigger.

The value in register 0 specifies the time interval in microseconds. The minimum time interval is 100 microseconds and the maximum time interval is 4294.967 seconds. Short time intervals (from 100 microseconds to 2 milliseconds) are accurate to the microsecond, whereas longer time intervals (greater than 2 milliseconds) are accurate to the millisecond.

## EXTIN\_BLK

ADCMODE, EXTIN\_BLK

External trigger, block mode with continuous sampling.

Bit	7	6	5	4	3	2	1	0
				4				-

## TIMER\_BLK

ADCMODE, TIMER\_BLK

Timer trigger, block mode with continuous sampling.

Bit	7	6	5	4	3	2	1	0
				5				-

## CHANNELS

ADCMODE, CHANNELS, *max\_channel*

Sets the maximum number of ADC channels.

Bit	7	6	5	4	3	2	1	0
				6				Channels

Bits 3:0

***max\_channel***

<i>Value</i>	<i>Description</i>
0 to 5	28-pin chip (AN0 to AN5)
0 to 8	44-pin chip (AN0 to AN8)

Sets the total number of analog channels to convert. The value specified is the maximum channel number. e.g. A value of 2 will convert AN0, AN1, AN2.

**VREF**

ADCMODE, VREF, *vref\_bits*

Selects voltage reference.

Bit	7	6	5	4	3	2	1	0
	7						V-	V+

Bit 1

**VREF-**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
AVSS	0x00	AVSS is used as VREF-
AN1	0x02	AN1 is used as VREF-

Bit 0

**VREF+**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
AVDD	0x00	AVDD is used as VREF+
AN0	0x01	AN0 is used as VREF+

<i>Examples:</i>	ADCMODE, 0x10	Set manual trigger, single sample with one repeat per trigger.
	ADCMODE, 0x24	Set external trigger, single sample with five samples per trigger.
	LOADWORD, 1000	Set timer trigger every 1000 usec.
	ADCMODE, 0x30	Single sample with one repeat per trigger.
	ADCMODE, 0	Disable analog conversions.

*See Also:* ADCLOAD, ADCLONG, ADCSCALE, ADCTRIG, ADCWAIT

**ADCSCALE Set scale multiplier for ADC**

*Syntax:* **ADCSCALE, *channel***

*Description:* Set the scale value or offset value for the specified *channel* to the floating point value in register 0.

if reg[A] is 32-bit,

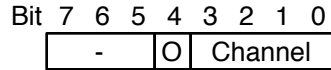
ADCscale[*channel*] = reg[0] or ADCOffset[*channel*] = reg[0]

if reg[A] is 64-bit,

ADCscale[*channel*] = reg[128] or ADCOffset[*channel*] = reg[128]

*Opcode:* **D3**

*Byte 2:* ***channel***



Bit 4	<b>Scale/Offset</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	0x00	Sets scale value.
	OFFSET	0x10	Sets offset value.
Bits 3:0	<b>Channel</b>		
	<i>Value</i>	<i>Description</i>	
	0 to 5	28-pin chip (AN0 to AN5)	
	0 to 8	44-pin chip (AN0 to AN8)	

Sets the scale value or offset value for *channel* to the floating point value in register 0. At device reset, the scale value for all channels is set to 1.0, and the offset value for all channels is set to zero.

*See Also:* ADCLOAD, ADCLONG, ADCMODE, ADCTRIG, ADCWAIT

---

## ADCTRIG Trigger an A/D conversion

*Syntax:* **ADCTRIG**

*Description:* Trigger an analog 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.

*Opcode:* **D2**

*See Also:* ADCLOAD, ADCLONG, ADCMODE, ADCSCALE, ADCWAIT

---

## ADCWAIT Wait for next A/D sample

*Syntax:* **ADCWAIT**

*Description:* Wait until the next analog conversion is complete and the analog values are ready.

*Opcode:* **D6**

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.

Note: 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.

*See Also:* ADCLOAD, ADCLONG, ADCMODE, ADCSCALE, ADCTRIG



**ADDIND      Add to Indirect Pointer**

*Syntax:*            **ADDIND, register, unsignedByte**

*Description:*      The long integer value in *register* is multiplied by the *unsignedByte* and by the data type size and the result is added to bits 23:0 of register 0 or 128 (pointer address). Bits 31:24 of register 0 or 128 are unchanged (pointer type). See the SETIND instruction for a description of pointers.

if reg[A] is 32-bit,  
     reg[0] (bits 31:24) = reg[0] (bits 31:24)  
     reg[0] (bits 23:0) = (reg[0] (bits23:0) + (reg[*register*] \* *unsignedByte* \* dataTypeSize)  
 if reg[A] is 64-bit,  
     reg[128] (bits 63:32) = 0  
     reg[128] (bits 31:24) = reg128 (bits 63:24)  
     reg[128] (bits 23:0) = (reg[128] (bits23:0) + (reg[*register*] \* *unsignedByte* \*  
     dataTypeSize

*Opcode:*            **78**

*Byte 2:*            **register**  
 Register number (0 to 255).

*Byte 3:*            **unsigned**  
 Unsigned byte (0 to 255).

*Special Cases:*    • if *register* = 0, the register value is not used in the pointer calculation  
                          • if *register* = 0 and *unsignedByte* = 0, the pointer is decremented by the data type size  
                          • if result is < 0, reg[0|128] (bits 23:0) is set to 0  
                          • if result is >= 0xFFFFFFFF, reg[0|128] (bits 23:0) is set to 0xFFFFFFFF

*See Also:*           SETIND, WRIND, RDIND, COPYIND, LOADIND, SAVEIND

**ALOADX      Load register A from register X**

*Syntax:*            **ALOADX**

*Description:*      Set register A to the value of register X, and increment X to select the next register in sequence.

reg[A] = reg[X] , X = X + 1

*Opcode:*            **0D**

*Special Cases:*    • if reg[A] is 32-bit and reg[X] is 64-bit, only the lower 32-bits of reg[X] are copied  
                          • if reg[A] is 64-bit and reg[X] is 32-bit, the upper 32-bits of reg[A] are set to zero

*See Also:*           LOAD, LOADA, LOADX, XSAVE, XSAVEA

---

## ASIN      Arc Sine

*Syntax:*            **ASIN**

*Description:*      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 is stored in register A.

$\text{reg[A]} = \text{asin}(\text{reg[A]})$

*Opcode:*            **4A**

*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

*See Also:*            ACOS, ATAN, ATAN2, COS, SIN, TAN, DEGREES, RADIANS

---

## ATAN      Arc Tangent

*Syntax:*            **ATAN**

*Description:*      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 is stored in register A.

$\text{reg[A]} = \text{atan}(\text{reg[A]})$

*Opcode:*            **4C**

*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

*See Also:*            ACOS, ASIN, ATAN2, COS, SIN, TAN, DEGREES, RADIANS

---

## ATAN2      Arc Tangent (with two registers)

*Syntax:*            **ATAN2, register**

*Description:*      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 of the specified register, and the result is stored in register A. This instruction is used to convert rectangular coordinates (register A, reg[register]) to polar coordinates (r, theta). The value of theta is stored in register A.

$\text{reg[A]} = \text{atan}(\text{reg[A]} / \text{reg[register]})$

*Opcode:*            **4D**

*Byte 2:*            **register**

Register number (0 to 255).

- Special Cases:*
- if reg[A] is 32-bit and *register* is 64-bit, the value from *register* is converted to a 32-bit value before being used, but the value stored in *register* remains unchanged
  - if reg[A] is 64-bit and *register* is 32-bit, the value in *register* is converted to a 64-bit value before being used, but value stored in *register* remains unchanged
  - if reg[A] or reg[*register*] is NaN, then the result is NaN
  - if reg[A] is 0.0 and reg[*register*] > 0, then the result is 0.0
  - if reg[A] > 0 and finite, and reg[*register*] is +inf, then the result is 0.0
  - if reg[A] is -0.0 and reg[*register*] > 0, then the result is -0.0
  - if reg[A] < 0 and finite, and reg[*register*] is +inf, then the result is -0.0
  - if reg[A] is 0.0 and reg[*register*] < 0, then the result is pi
  - if reg[A] > 0 and finite, and reg[*register*] is -inf, then the result is pi
  - if reg[A] is -0.0, and reg[*register*] < 0, then the result is -pi
  - if reg[A] < 0 and finite, and reg[*register*] is -inf, then the result is -pi
  - if reg[A] > 0, and reg[*register*] is 0.0 or -0.0, then the result is pi/2
  - if reg[A] is +inf, and reg[*register*] is finite, then the result is pi/2
  - if reg[A] < 0, and reg[*register*] is 0.0 or -0.0, then the result is -pi/2
  - if reg[A] is -inf, and reg[*register*] is finite, then the result is -pi/2
  - if reg[A] is +inf, and reg[*register*] is +inf, then the result is pi/4
  - if reg[A] is +inf, and reg[*register*] is -inf, then the result is 3\*pi/4
  - if reg[A] is -inf, and reg[*register*] is +inf, then the result is -pi/4
  - if reg[A] is -inf, and reg[*register*] is -inf, then the result is -3\*pi/4

*See Also:* ACOS, ASIN, ATAN, COS, SIN, TAN, DEGREES, RADIANS

---

## ATOF Convert ASCII string to floating point

*Syntax:* **ATOF, *string***

*Description:* Converts a zero terminated ASCII *string* to a floating point value. and stores the result in register 0 or register 128.

*Opcode:* **1E**

*Byte 2:* ***string***  
Zero-terminated ASCII string.

If register A is 32-bit, register 0 is loaded with the 32-bit floating point value. If register A is 64-bit, register 128 is loaded with the 64-bit floating point value. The *string* to convert is sent immediately following the opcode. The string can be in standard numeric format (e.g. 1.56, -0.5), or exponential format (e.g. 10E6). Conversion will stop at the first invalid character, but data bytes will continue to be read until a zero terminator is encountered. The *string* can contain the following characters:

- leading whitespace (space or tab)
- sign (+ or -)
- decimal digits (0 to 9)
- decimal point (.)
- decimal digits (0 to 9)
- exponential (E or e)

- sign (+ or -)
- decimal digits (0 to 9)

*Examples:*      **ATOF, "2.54"**      stores the value 2.54 in register 0 or 128  
                  **ATOF, "1E3"**      stores the value 1000.0 in register 0 or 128

*Special Cases:*    • if *string* length > 127, *string* will be truncated to 127 characters

*See Also:*          **ATOL, FTOA, LTOA, STRTOF, STRTOL**

## **ATOL      Convert ASCII string to long integer**

*Syntax:*            **ATOL, *string***

*Description:*      Converts a zero terminated ASCII *string* to a long integer value.

*Opcode:*            **9A**

*Byte 2:*            ***string***  
                  Zero-terminated ASCII string.

If register A is 32-bit, register 0 is loaded with the 32-bit long integer value. If register A is 64-bit, register 128 is loaded with the 64-bit long integer value. The *string* to convert is sent immediately following the opcode. Conversion will stop at the first invalid character, but data bytes will continue to be read until a zero terminator is encountered. The *string* can contain the following characters:

- leading whitespace (space or tab)
- sign (+ or -)
- decimal digits (0 to 9)

*Examples:*          **ATOL, "500000"**      stores the value 500000 in register 0 or 128  
                  **ATOL, "-5"**      stores the value -5 in register 0 or 128

*Special Cases:*    • if *string* length > 127, *string* will be truncated to 127 characters

*See Also:*          **ATOF, FTOA, LTOA, STRTOF, STRTOL**

## **BRA      Unconditional branch**

*Syntax:*            **BRA, *relativeAddress***

*Description:*      This instruction branches unconditionally to the instruction at the *relativeAddress*. If the *relativeAddress* is more than -128 to 127 bytes from the address of the next instruction, the JMP instruction must be used.

*Opcode:*            **81**

*Byte 2:*            ***relativeAddress***  
                  A signed byte value that is added to the address of the next instruction to determine the address to

branch to.

*Special Cases:* • only valid inside user-defined functions stored in Flash memory.

*See Also:* BRA,cc, JMP, JMP,cc, GOTO, RET, RET,cc

## BRA, cc Conditional branch

*Syntax:* **BRA,conditionCode, relativeAddress**

*Description:* If the condition is true, this instruction branches to the instruction at the *relativeAddress* address. If the condition is false, no branch occurs. If the *relativeAddress* is more than -128 to 127 bytes from the address of the next instruction, the JMP,cc instruction must be used.

*Opcode:* **82**

*Byte 2:* **conditionCode**  
The list of condition codes is as follows:

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
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

*Byte 3:* **relativeAddress**  
A signed byte value that is added to the address of the next instruction to determine the address to branch to.

*Special Cases:* • only valid inside user-defined functions stored in Flash memory.

*See Also:* BRA, JMP, JMP,cc, GOTO, RET, RET,cc

## BREAK Debug breakpoint

*Syntax:* **BREAK**

*Description:* If debug mode is enabled, a breakpoint occurs and the debug monitor is entered. If debug mode is disabled, the instruction is ignored.

*Opcode:* **F7**

Used in conjunction with the built-in debugger.

*See Also:* TRACEOFF, TRACEON, TRACEREG, TRACESTR

## **CEIL Ceiling**

*Syntax:* **CEIL**

*Description:* 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.

$\text{reg}[A] = \text{ceil}(\text{reg}[A])$

*Opcode:* **52**

*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

*See Also:* FLOOR, ROUND

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

*Syntax:* **CHECKSUM**

*Description:* A checksum is calculated for the uM-FPU64 code and user-defined functions stored in Flash. The checksum value is stored in register 0.

*Opcode:* **F6**

This can be used as a diagnostic test for confirming the state of a uM-FPU chip.

## **CLR Clear register**

*Syntax:* **CLR, register**

*Description:* Set the value of the specified register to zero.

$\text{reg}[\text{register}] = 0, \text{status} = \text{longStatus}(\text{reg}[\text{register}])$

*Opcode:* **03**

*Byte 2:* **register**

Register number (0 to 255).

- Special Cases:*
- if SETARGS is used, and *register* = 0
    - if reg[A] is 32-bit, the value is stored in registers 1 to 9
    - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* CLR0, CLRA, CLRX, SETARGS

## CLR0 Clear register 0

*Syntax:* **CLR0**

*Description:* Set the value of register 0 (32-bit) or register 128 (64-bit) to zero.

if reg[A] is 32-bit, reg[0] = 0, status = longStatus(reg[0])  
 if reg[A] is 64-bit, reg[128] = 0, status = longStatus(reg[128])

*Opcode:* **06**

- Special Cases:*
- if SETARGS is used,
    - if reg[A] is 32-bit, the value is stored in registers 1 to 9
    - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* CLR, CLRA, CLRX, SETARGS

## CLRA Clear register A

*Syntax:* **CLRA**

*Description:* Set the value of register A to zero.

reg[A] = 0, status = longStatus(reg[A])

*Opcode:* **04**

*See Also:* CLR, CLR0, CLRX

## CLRX Clear register X

*Syntax:* **CLRX**

*Description:* Set the value of register A to zero, and increment X to select the next register in sequence.

reg[X] = 0, status = longStatus(reg[X]), X = X + 1

*Opcode:* **05**

- Special Cases:*
- if reg[X] is 32-bit, X will not be incremented past register 127
  - if reg[X] is 64-bit, X will not be incremented past register 255

*See Also:* CLR, CLR0, CLRA

---

## **COPY**      **Copy registers**

*Syntax:*      **COPY, fromRegister, toRegister**

*Description:*      Copy the value from *fromRegister* to *toRegister*.

reg[*toRegister*] = reg[*fromRegister*], status = longStatus(reg[*toRegister*])

*Opcode:*      **07**

*Byte 2:*      ***fromRegister***  
Register number (0 to 255).

*Byte 3:*      ***toRegister***  
Register number (0 to 255).

*Special Cases:*      • if *toRegister* is 32-bit and *fromRegister* is 64-bit, the upper 32-bits of *fromRegister* are set to zero  
• if *toRegister* is 64-bit and *fromRegister* is 32-bit, only the lower 32-bits of *toRegister* are copied

*See Also:*      COPYA, COPYX, COPY0, FCOPYI, LCOPYI

---

## **COPYA**      **Copy register A**

*Syntax:*      **COPYA, register**

*Description:*      Copy the value of register A to *register*.

reg[*register*] = reg[A], status = longStatus(reg[A])

*Opcode:*      **08**

*Byte 2:*      ***register***  
Register number (0 to 255).

*Special Cases:*      • if reg[A] is 32-bit and *register* is 64-bit, the upper 32-bits of *register* are set to zero  
• if reg[A] is 64-bit and *register* is 32-bit, only the lower 32-bits of reg[A] are copied

*See Also:*      COPY, COPYX, COPY0, FCOPYI, LCOPYI

---

## **COPYIND**      **Copy using Indirect Pointers**

*Syntax:*      **COPYIND, fromRegister, toRegister, countRegister**

*Description:*      The number of data items specified by the *countRegister* are copied from the location pointed to by *fromRegister* to the location pointed to by *toRegister*. See the SETIND instruction for a description of pointers.



<i>Opcode:</i>	<b>79</b>
<i>Byte 2:</i>	<b><i>fromRegister</i></b> Register number (0 to 255). The register contains the from pointer.
<i>Byte 3:</i>	<b><i>toRegister</i></b> Register number (0 to 255). The register contains the to pointer.
<i>Byte 4:</i>	<b><i>countRegister</i></b> Register number (0 to 255). The register contains the number of items to copy.
<i>See Also:</i>	SETIND, ADDIND, WRIND, RDIND, LOADIND, SAVEIND

## **COPYX      Copy register X**

<i>Syntax:</i>	<b>COPYX, register</b>
<i>Description:</i>	Copy the value of register X to <i>register</i> , and increment X to select the next register in sequence.  reg[ <i>register</i> ] = reg[X], status = longStatus(reg[ <i>register</i> ]), X = X + 1
<i>Opcode:</i>	<b>09</b>
<i>Byte 2:</i>	<b><i>register</i></b> Register number (0 to 255).
<i>Special Cases:</i>	<ul style="list-style-type: none"> <li>• if reg[X] is 32-bit and <i>register</i> is 64-bit, the upper 32-bits of <i>register</i> are set to zero</li> <li>• if reg[X] is 64-bit and <i>register</i> is 32-bit, only the lower 32-bits of reg[X] are copied</li> </ul>
<i>See Also:</i>	COPY, COPYA, COPY0, FCOPYI, LCOPYI

## **COPY0      Copy register 0**

<i>Syntax:</i>	<b>COPY0, register</b>
<i>Description:</i>	<p>If register A is 32-bit, the value of register 0 is copied to <i>register</i>. If register A is 64-bit, the value of register 128 is copied to <i>register</i>.</p> <p>if reg[A] is 32-bit, then reg[<i>register</i>] = reg[0], status = longStatus(reg[0]) if reg[A] is 64-bit, then reg[<i>register</i>] = reg[128], status = longStatus(reg[128])</p>
<i>Opcode:</i>	<b>10</b>
<i>Byte 2:</i>	<b><i>register</i></b> Register number (0 to 255).
<i>Special Cases:</i>	<ul style="list-style-type: none"> <li>• if reg[A] is 32-bit and <i>register</i> is 64-bit, the upper 32-bits of <i>register</i> are set to zero</li> <li>• if reg[A] is 64-bit and <i>register</i> is 32-bit, only the lower 32-bits of reg[128] are copied</li> </ul>

*See Also:* COPY, COPYA, COPYX, FCOPYI, LCOPYI

## **COS      Cosine**

*Syntax:*      **COS**

*Description:*      Calculates the cosine of the angle (in radians) in register A and stores the result in register A.

reg[A] = cosine(reg[A])

*Opcode:*      **48**

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

*See Also:*      ACOS, ASIN, ATAN, ATAN, SIN, TAN, DEGREES, RADIANS

## **DEGREES      Convert radians to degrees**

*Syntax:*      **DEGREES**

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

*Opcode:*      **4E**

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

*See Also:*      ACOS, ASIN, ATAN, ATAN2, COS, SIN, TAN, RADIANS

## **DELAY      Delay (in milliseconds)**

*Syntax:*      **DELAY, *period***

*Description:*      The uM-FPU64 pauses for the number of milliseconds specified by *period*. If *period* is zero, then the number of milliseconds is loaded from register 0. If foreground/background processing has been enabled, the other process can continue execution during the delay period.

*Opcode:*      **DB**

*Bytes 2-3:*      ***period***  
A 16-bit unsigned value that specified the delay period in milliseconds. (0 to 65535)

*Special Cases:*      • the *period* is the minimum delay period, it can be up to one millisecond longer

*See Also:*      TIMESET, TIMELONG, TICKLONG, RTC

## **DEVIO      Device Input/Output**

*Syntax:*      **DEVIO, *device*, *action*{, ...}**

**Description:** This instruction provides support for devices interfaced to the uM-FPU64 chip using the digital pins. The DEVIO instruction is designed to interact with byte oriented I/O devices. It provides general I/O capabilities and higher level device specific support. All DEVIO instructions start with the opcode, followed by a byte that specifies the device, and a byte which specifies the action to perform with the device. Depending on the action, there may be additional bytes required by the instruction. The supported devices are: general access RAM, FIFO buffers, 1-wire bus, I<sup>2</sup>C bus, SPI bus, asynchronous serial port, counters, servo controllers, LCD, and VDrive2 USB Flash drives. Additional information is available in the *Using the uM-FPU64 DEVIO Instruction* document.

```
DEVIO, device, DISABLE
DEVIO, device, ENABLE, pin, config
DEVIO, device, {device specific actions}
DEVIO, device, WRITE_REG8{+MSB}{+LSB}, register
DEVIO, device, WRITE_REG16{+MSB}{+LSB}, register
DEVIO, device, WRITE_REG32{+MSB}{+LSB}, register
DEVIO, device, WRITE_REG64{+MSB}{+LSB}, register
DEVIO, device, WRITE_BYTE, byte
DEVIO, device, WRITE_WORD, byte, byte
DEVIO, device, WRITE_NBYTE, count, byte, ...
DEVIO, device, WRITE_REP, count, byte
DEVIO, device, WRITE_STR, string
DEVIO, device, WRITE_SBUF
DEVIO, device, WRITE_SSEL
DEVIO, device, WRITE_MEM, count
DEVIO, device, WRITE_MEMA, address, count
DEVIO, device, WRITE_MEMR, regAddr, regCount
DEVIO, device, READ_REG8{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, READ_REG16{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, READ_REG32{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, READ_REG64{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, READ_SKIP, count
DEVIO, device, READ_SBUF
DEVIO, device, READ_SSEL
DEVIO, device, READ_MEM, count
DEVIO, device, READ_MEMA, address, count
DEVIO, device, READ_MEMR, regAddr, regCount
DEVIO, device, READ_MEMR, regAddr, regCount
DEVIO, device, LOAD_DEVICE, xopdev
```

**Opcode:** **DA**

**Byte 2:** **device**

Bit	7	6	5	4	3	2	1	0
	Device				Number			

Bits 7:4	Device Type		
	IDE Symbol	IDE Value	Device Type
	MEM	0x00	Memory
	FIFO1	0x01	FIFO buffer 1
	FIFO2	0x02	FIFO buffer 2

FIFO3	0x03	FIFO buffer 3
FIFO4	0x04	FIFO buffer 4
OWIRE	0x10	1-Wire
I2C	0x20	I <sup>2</sup> C
SPI	0x30	SPI
ASYNC	0x40	Asynchronous Serial Port
COUNTER	0x50	Counter (Digital input)
SERVO	0x60	Servo Output
LCD	0x80	LCD
<i>loadable device</i>	0xA0	Reserved for loadable devices
<i>loadable device</i>	0xB0	Reserved for loadable devices
<i>loadable device</i>	0xC0	Reserved for loadable devices
<i>loadable device</i>	0xD0	Reserved for loadable devices
<i>loadable device</i>	0xE0	Reserved for loadable devices
<i>loadable device</i>	0xF0	Reserved for loadable devices

Bits 3:0	<b>Device Number</b>
<i>Value</i>	<i>Description</i>
0 - 15	Device number (for device types that support multiple devices)

**Byte 3:**      ***action***  
 An unsigned byte specifying the device action. A description of actions that are common to all devices is shown below. For device specific actions, see separate descriptions for each device type. (e.g. *DEVIO*, *ASYNC*)

**Disable (0x00)**

*DEVIO, device, DISABLE*  
 Disable the specified device and release the digital pins.

**Enable (0x01)**

*DEVIO, device, ENABLE, pin, config*  
 Enable the specified device and assign the digital pins. The enable instruction must be used to initialize a device before any other device instructions are used.

**Byte 4:**      ***pin***  
 Specifies the first pin used by the specified device (D0 to D23).

**Byte 5:**      ***config***  
 Configuration byte for initializing the device. See the device specific descriptions for details.

**Device Specific Actions (0x02 - 0x0F)**

For device specific actions, see the separate documentation for each device type. (e.g. *DEVIO*, *ASYNC*)

**Write 8-bit Value from Register (0x10, 0x14)**

*DEVIO, device, WRITE\_REG8, register*  
 Write the lower 8-bit value from the specified register to the device.

**Write 16-bit Value from Register (0x11, 0x15)**

*DEVIO, device, WRITE\_REG16{+MSB}{+LSB}, register*

Write the lower 16-bit value from the specified register to the device. The value can be written with the most significant byte or least significant byte first.

#### Write 32-bit Value from Register (0x12, 0x16)

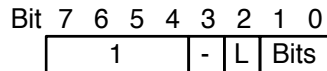
DEVIO, *device*, WRITE\_REG32{+MSB}{+LSB}, *register*

Write the lower 32-bit word from the specified register to the device. The value can be written with the most significant byte or least significant byte first.

#### Write 64-bit Value from Register (0x13, 0x17)

DEVIO, *device*, WRITE\_REG64{+MSB}{+LSB}, *register*

Write the 64-bit value from the specified register to the device. The value can be written with the most significant byte or least significant byte first.



Bit 3

#### Byte Order

*IDE Symbol*

*IDE Value*

*Description*

MSB

0x00

Most significant byte first

LSB

0x04

Least significant byte first

Bits 1:0

#### Number of Bits

*IDE Symbol*

*IDE Value*

*Description*

WRITE\_REG8

0x10

1 byte (8-bit)

WRITE\_REG16

0x11

2 bytes (16-bit)

WRITE\_REG32

0x12

3 bytes (32-bit)

WRITE\_REG64

0x13

4 bytes (64-bit)

Byte 4:

***register***

Register number (0 to 255).

#### Write Byte (0x20)

DEVIO, *device*, WRITE\_BYTE, *byte*

Write the 8-bit value specified to the device.

Byte 4:

***byte***

Unsigned 8-bit value to write to device.

#### Write Word (0x21)

DEVIO, *device*, WRITE\_WORD, *word*

Write the 16-bit value specified to the device.

Bytes 4-5:

***word***

Unsigned 16-bit value to write to device.

#### Write Multiple Bytes (0x22)

DEVIO, *device*, WRITE\_NBYTE, *count*, *byte*, ...

Write the number of bytes specified by *count* to the device.

Byte 4:

***count***

Unsigned 8-bit integer specifying the number of bytes to write.

Bytes 5-n:

***byte*, ...**

Unsigned 8-bit values to write to device.

### Write Repeat Byte (0x23)

DEVIO, *device*, WRITE\_REP, *count*, *byte*

Write the same byte repeatedly to the device the number of times specified by *count*.

Byte 4:

***count***

Unsigned 8-bit integer specifying the number of times to write the byte.

Byte 5:

***byte***

Unsigned 8-bit value to write to device.

### Write String (0x24)

DEVIO, *device*, WRITE\_STR, *string*

Write the zero-terminated *string* to the device. If *string* length > 127, *string* will be truncated to 127 characters. The zero terminator is not sent unless the device is MEM, FIFO1, FIFO2, or FIFO3.

Bytes 4-n:

***string***

Zero-terminated string to write to device.

### Write from String Buffer (0x25)

DEVIO, *device*, WRITE\_SBUF

Write the contents of the string buffer to the device. A zero terminator is also sent if the device is MEM, FIFO1, FIFO2, or FIFO3.

### Write from String Selection (0x26)

DEVIO, *device*, WRITE\_SSEL

Write the string selection to the device. A zero terminator is also sent if the device is MEM, FIFO1, FIFO2, or FIFO3.

### Write from Memory Address 0 (0x27)

DEVIO, *device*, WRITE\_MEM, *count*

Read *count* bytes from memory, starting at memory address 0, and write the bytes to the device.

Byte 4:

***count***

Unsigned 8-bit integer specifying the number of bytes to write.

### Write from Memory Address (0x28)

DEVIO, *device*, WRITE\_MEMA, *address*, *count*

Read *count* bytes from memory, starting at the memory address specified by *address*, and write the bytes to the device.

Byte 4-5:

***address***

Unsigned 16-bit integer specifying the memory address to write to.

Byte 6-7:

***count***

Unsigned 16-bit integer specifying the number of bytes to write.

### Write from Memory Address specified by Register (0x29)

DEVIO, *device*, WRITE\_MEMR, *regAddr*, *regCount*

Read the number of bytes specified in *regCount* from memory, starting at the memory address specified in *regAddr*, and write the bytes to the device.

Byte 4: **regAddr**  
The lower 16-bits of the register specify the memory address to write to.

Byte 5: **regCount**  
The lower 16-bits of the register specify the number of bytes to write.

#### Read 8-bit value to Register (0x30, 0x34, 0x38, 0x3C)

DEVIO, device, READ\_REG8{+ZE}{+SE}, register  
Read an 8-bit value from the device and store in the lower 8 bits of the specified register.  
The remaining bits in the register can be filled with either zero-extend or sign-extend.

#### Read 16-bit value to Register (0x31, 0x35, 0x39, 0x3D)

DEVIO, device, READ\_REG16{+MSB}{+LSB}{+ZE}{+SE}, register  
Read a 16-bit value from the device and store in the lower 16 bits of the specified register.  
The value can be stored with the most significant byte or least significant byte first. The remaining bits in the register can be filled with either zero-extend or sign-extend.

#### Read 32-bit value to Register (0x32, 0x36, 0x3A, 0x3E)

DEVIO, device, READ\_REG32{+MSB}{+LSB}{+ZE}{+SE}, register  
Read a 32-bit value from the device and store in the lower 32 bits of the specified register.  
The value can be stored with the most significant byte or least significant byte first. The remaining bits in the register can be filled with either zero-extend or sign-extend.

#### Read 64-bit value to Register (0x33, 0x37, 0x3B, 0x3F)

DEVIO, device, READ\_REG64{+MSB}{+LSB}{+ZE}{+SE}, register  
Read a 64-bit value from the device and store in specified register. The value can be stored with the most significant byte or least significant byte first. The remaining bits in the register can be filled with either zero-extend or sign-extend.

Bit 7 6 5 4 3 2 1 0

3	S	L	Bits
---	---	---	------

Bit 3 **Sign Extend**

IDE Symbol	IDE Value	Description
ZE	0x00	Zero extend.
SE	0x08	Sign extend.

Bit 2 **Byte Order**

IDE Symbol	IDE Value	Description
MSB	0x00	Most significant byte first
LSB	0x04	Least significant byte first

Bits 1:0 **Number of Bits**

IDE Symbol	IDE Value	Description
READ_REG8	0x30	1 byte (8-bit)
READ_REG16	0x31	2 bytes (16-bit)
READ_REG32	0x32	3 bytes (32-bit)
READ_REG64	0x33	4 bytes (64-bit)

Byte 4: **register**  
Register number (0 to 255).

#### Read and Skip Bytes (0x43)

**DEVIO, *device*, READ\_SKIP, *count***  
 Read and skip *count* bytes from device.

**Byte 4:**                    ***count***  
 Unsigned 8-bit integer specifying the number of bytes to skip.

#### **Read String to String Buffer (0x45)**

**DEVIO, *device*, READ\_SBUF**  
 Read zero-terminated string from device and store in string buffer.

#### **Read String to String Selection (0x46)**

**DEVIO, *device*, READ\_SSEL**  
 Read zero-terminated string from device and store at string selection point.

#### **Read to Memory Address 0 (0x47)**

**DEVIO, *device*, READ\_MEM, *count***  
 Read *count* bytes from the device and store in memory, starting at memory address 0.

**Byte 4:**                    ***count***  
 Unsigned byte specifying the number of bytes to read.

#### **Read to Memory Address (0x48)**

**DEVIO, *device*, READ\_MEMA, *address*, *count***  
 Read *count* bytes from the device and store in memory, starting at the memory address specified by *address*.

**Byte 4-5:**                ***address***  
 Unsigned 16-bit integer specifying the memory address to read from.

**Byte 6-7:**                ***count***  
 Unsigned 16-bit integer specifying the number of bytes to read.

#### **Read to Memory Address specified by Register (0x49)**

**DEVIO, *device*, READ\_MEMR, *regAddr*, *regCount***  
 Read the number of bytes specified in *regCount* from the device and store in memory starting at the memory address specified in *regAddr*.

**Byte 4:**                    ***regAddr***  
 The lower 16-bits of the register specify the memory address to read from.

**Byte 5:**                    ***regCount***  
 The lower 16-bits of the register specify the number of bytes to read.

#### **Load Device (0x5x)**

**DEVIO, *device*, LOAD\_DEVICE, *xopdev***  
 Attach the loadable *device* to the device code loaded in the XOP area of Flash memory as specified by *xopdev*. Ram is allocated from the dynamic allocation for use by the *device*. This call is required before using a loadable device.

---

## **DEVIO, ASYNC      Asynchronous Serial Port Interface**



*Syntax:*       **DEVIO,ASYNC,action{,...}**

*Description:*   This instruction provides support for sending and receiving data through an asynchronous serial connection. The serial connection can be configured as receive only (1 pin), transmit only (1 pin), receive and transmit (2 pins), or receive and transmit with hardware flow control (4 pins). The baud rate is selectable from 300 baud to 115,200 baud.

DEVIO, ASYNC, DISABLE  
DEVIO, ASYNC, ENABLE, *pin*, *config*

*Opcode:*       **DA**

*Byte 2:*       **ASYNC (0x40)**

*Byte 3:*       **action**  
An unsigned byte specifying the device action. Actions that are specific to the asynchronous serial device are shown below. For actions that are common to all devices, see the *DEVIO* description.

**Disable (0x00)**

DEVIO, ASYNC, DISABLE  
Disable the asynchronous serial connection and release the digital pins.

**Enable (0x01)**

DEVIO, ASYNC, ENABLE, *pin*, *config*  
Select the pins to use for the asynchronous serial connection, set the baud rate, and enable the asynchronous serial port.

*Byte 4:*       **pin**  
Specifies the pins used for the asynchronous serial connection.  
D0 to D8       28-pin uM-FPU64 chip  
D0 to D18      44-pin uM-FPU64 chip

**Pin Assignments**

Receive only

*pin*       Rx

Transmit only

*pin*       Tx

Receive and Transmit

*pin*       Rx

*pin*+1      Tx

Receive and Transmit with Flow Control

*pin*       Rx

*pin*+1      Tx

*pin*+2      /CTS

*pin*+3      /RTS

*Byte 5:*       **config**

Bit 7	6	5	4	3	2	1	0
-	-	Type	Baud Rate				

Bits 5:4       **Connection Type**

	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	RX	0x00	Receive Only
	TX	0x10	Transmit Only
	RX_TX	0x20	Receive and Transmit
	RX_TX_HW	0x30	Receive and Transmit with Hardware Flow Control
Bit 3:0	<b>Baud Rate</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	-	57,600 baud
	BAUD_300	0x01	300 baud
	BAUD_600	0x02	600 baud
	BAUD_1200	0x03	1200 baud
	BAUD_2400	0x04	2400 baud
	BAUD_4800	0x05	4800 baud
	BAUD_9600	0x06	9600 baud
	BAUD_19200	0x07	19200 baud
	BAUD_38400	0x08	38400 baud
	BAUD_57600	0x09	57600 baud
	BAUD_115200	0x0A	115200 baud

*See Also:* SERIN, SEROUT

---

## DEVIO, COUNTER     32-bit Counter Interface

*Syntax:*     **DEVIO, COUNTER+n, action{, ...}**

*Description:*     This instruction provides support for detecting and counting digital input changes. Optional support is provided for switch debouncing and automatic repeat when the input is held in the active state. If the active state is high, a rising edge on the digital input is counted. If the active state is low, a falling edge on the digital input is counted. If debouncing is enabled, changes to the digital input will be ignored for the *period* specified. The debounce period is set to 10 milliseconds by default. If a repeat value is specified, and the signal is held in the active state for the specified *delay*, the counter will increment at the specified *rate* while the signal remains in the active state.

```
DEVIO, COUNTER+n, DISABLE
DEVIO, COUNTER+n, ENABLE, pin, config
DEVIO, COUNTER+n, DEBOUNCE, period
DEVIO, COUNTER+n, REPEAT, delay, rate
DEVIO, COUNTER+n, READ_COUNT
DEVIO, COUNTER+n, EDGE1_MSEC
DEVIO, COUNTER+n, EDGE1_USEC
DEVIO, COUNTER+n, EDGE2_MSEC
DEVIO, COUNTER+n, EDGE2_USEC
```

*Opcode:*     **DA**

*Byte 2:*     **COUNTER+n (0x50-0x53)**

*Byte 3:*     **action**

An unsigned byte specifying the device action. Actions that are specific to counter devices are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

DEVIO, COUNTER+n, DISABLE

Disable the counter and release the digital pin.

### Enable (0x01)

DEVIO, COUNTER+n, ENABLE, *pin*, *config*

Selects the pin to use for the counter, the active level for counting, whether an event is associated with the counter, and enable the counter input.

Byte 4:

***pin***

Specifies the pin to use for the counter input.

D0 to D8 28-pin uM-FPU64 chip

D0 to D8 44-pin uM-FPU64 chip

Byte 5:

***config***

Bit 7 6 5 4 3 2 1 0

A	-	Event
---	---	-------

Bit 7

**Active State**

*IDE Symbol*

*IDE Value*

*Description*

LOW

0x00

Active low.

HIGH

0x80

Active high.

Bits 2:0

**Event Number**

*IDE Symbol*

*IDE Value*

*Description*

NO\_EVENT

0x00

No event.

EVENT1

0x01

Event 1.

EVENT2

0x02

Event 2.

EVENT4

0x03

Event 3.

EVENT5

0x04

Event 4.

EVENT6

0x05

Event 5.

EVENT7

0x06

Event 6.

EVENT1

0x07

Event 7.

### Set Debounce Period (0x02)

DEVIO, COUNTER+n, DEBOUNCE, *period*

Specifies the debounce period in milliseconds. The counters are initialized with a debounce period of 10 milliseconds.

Byte 4-5:

***period*** (*unsigned word*)

Specifies the debounce period in milliseconds (0 to 32677). Transitions on the counter input are ignored during the debounce period.

### Set Repeat Rate (0x03)

DEVIO, COUNTER+n, REPEAT, *delay*, *rate*

Specifies the automatic repeat parameters. The counters are initialized with no automatic repeat.

Byte 4-5:

***delay*** (*unsigned word*)

Specifies the delay in milliseconds before automatic repeat is enabled. (0 to 32677)

Byte 6-7:

***rate*** (*unsigned word*)

Specifies the rate in milliseconds that the counter will be incremented if the counter input remains active. (1 to 32677)

#### Read Count (0x04)

**DEVIO, COUNTER+n, READ\_COUNT**  
Returns the counter value in register 0.

#### Read Active Edge time in milliseconds (0x05)

**DEVIO, COUNTER+n, EDGE1\_MSEC**  
Returns the time in milliseconds (32-bit value) in register 0 or 128.

#### Read Active Edge time in microseconds (0x06)

**DEVIO, COUNTER+n, EDGE1\_USEC**  
Returns the time in microseconds (64-bit value) in register 0 or 128.

#### Read Not Active Edge time in milliseconds (0x07)

**DEVIO, COUNTER+n, EDGE2\_MSEC**  
Returns the time in milliseconds (32-bit value) in register 0 or 128.

#### Read Not Active Edge time in microseconds (0x08)

**DEVIO, COUNTER+n, EDGE2\_USEC**  
Returns the time in microseconds (64-bit value) in register 0 or 128.

---

## DEVIO, FIFO      FIFO Buffer Interface

*Syntax:*      **DEVIO,FIFO1,action{,...}**  
                 **DEVIO,FIFO2,action{,...}**  
                 **DEVIO,FIFO3,action{,...}**  
                 **DEVIO,FIFO4,action{,...}**

*Description:*      These instructions provide support for First In First Out (FIFO) buffers. They can be used to buffer data, or to transfer data from one process to another. Memory must be allocated to the FIFOs from the dynamic allocation area using one of the following instructions:

**DEVIO, MEM, ALLOCATE, memSize, fifoSize**  
**DEVIO, FIFOn, ALLOC\_MEM, size**  
**DEVIO, FIFOn, ALLOC\_MEMR, regSize**

**DEVIO, FIFOn, DISABLE**  
**DEVIO, FIFOn, ENABLE, pin, config**  
**DEVIO, FIFOn, CLEAR**  
**DEVIO, FIFOn, USED**  
**DEVIO, FIFOn, FREE**  
**DEVIO, FIFOn, STATUS**  
**DEVIO, FIFOn, CLEAR\_OVERFLOW**  
**DEVIO, FIFOn, ALLOC\_MEM, size**  
**DEVIO, FIFOn, ALLOC\_MEMR, regSize**

*Opcode:*      **DA**

*Byte 2:*      **FIFO1-FIFO3 (0x01-0x04)**

*Byte 3:*      **action**  
                 An unsigned byte specifying the device action. Actions that are specific to FIFO devices are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

DEVIO, FIFOn, DISABLE  
Disables the specified FIFO device.

### Enable (0x01)

DEVIO, FIFOn, ENABLE, *pin*, *config*  
Initialize the FIFO.

Byte 4:

***pin***  
Unused.

Byte 5:

***config***

Bit	7	6	5	4	3	2	1	0
				-			Type	

Bits 2:0

#### Event Type

IDE Symbol	IDE Value	Description
NO_EVENT	0x00	No event
EMPTY	0x01	Set event flag when buffer is empty.
NOT_EMPTY	0x02	Set event flag when data in buffer.
HALF_EMPTY	0x03	Set event flag when buffer is half empty.
HALF_FULL	0x04	Set event flag when buffer is half full.
FULL	0x05	Set event flag when buffer is full.
OVERFLOW	0x06	Set event flag when buffer overflows.

### Clear Buffer (03)

DEVIO, FIFOn, CLEAR  
Clear the buffer by resetting the input index and output index.

### Get Number of Bytes Used (0x04)

DEVIO, FIFOn, USED  
Gets the number of bytes currently used in the memory buffer and returns the value in register 0.

### Get Number of Bytes Free (0x05)

DEVIO, FIFOn, FREE  
Get the number of bytes currently available in the memory buffer and returns the value in register 0.

### Get Buffer Status (0x06)

DEVIO, FIFOn, STATUS  
Get the current status of the memory buffer. The buffer status is returned in register 0.  
The status byte is as follows:

Bit	7	6	5	4	3	2	1	0
	O	F	H	E	-		Type	
Bit 7	Buffer Overflow							
Bit 6	Buffer Full							
Bit 5	Buffer Half Full							
Bit 4	Buffer Empty							
Bits 2:0	Event Type							

### Clear Overflow Bit (0x07)

**DEVIO, FIFOn, CLEAR\_OVERFLOW**

Clear the overflow bit for the memory buffer. The overflow bit is set if an attempt is made to store data to the buffer when the buffer is already full. Once the overflow bit is set, no data will not be stored in the buffer until the overflow bit has been cleared.

### Dynamic Memory Allocation (0x08)

**DEVIO, FIFOn, ALLOC\_MEM, size**

The number of memory bytes specified by *size* are allocated from the dynamic allocation for use by FIFOn. If not enough bytes are available in the dynamic allocation, the size of FIFOn is set to zero.

Byte 4:

**size**

Unsigned 16-bit word specifying the number of consecutive memory bytes to allocate from the dynamic allocation to FIFOn.

### Dynamic Memory Allocation (0x09)

**DEVIO, FIFOn, ALLOC\_MEMR, regSize**

The number of memory bytes specified by the value of *regSize* are allocated from the dynamic allocation. The memory address of the first byte is returned in register 0. If there are not enough bytes available in the dynamic allocation, the size of FIFOn is set to zero.

Byte 4:

**regSize**

An 8-bit register value. The lower 16-bits of the register specify the number of consecutive memory bytes to allocate from the dynamic allocation to FIFOn.

---

## DEVIO, I2C

## I<sup>2</sup>C Bus Interface

Syntax:

**DEVIO, I2C, action{,...}**

Description:

This instruction provides support for communicating with I<sup>2</sup>C devices using a local I<sup>2</sup>C bus on the specified pair of digital pins.

DEVIO, I2C, DISABLE  
DEVIO, I2C, ENABLE, *pin*, *config*  
DEVIO, I2C, START\_WRITE, *device*  
DEVIO, I2C, STOP  
DEVIO, I2C, START\_READ, *byteCount*

Opcode:

**DA**

Byte 2:

**I2C (0x20)**

Byte 3:

**action**

An unsigned byte specifying the device action. Actions that are specific to I<sup>2</sup>C devices are shown below. For actions that are common to all devices, see the DEVIO description.

### Disable (0x00)

**DEVIO, I2C, DISABLE**

Disable the I<sup>2</sup>C bus and release the digital pins.

### Enable (0x01)

DEVIO, I2C, ENABLE, *pin*, *config*

Selects the pins to use for the I<sup>2</sup>C bus, the bus speed, and enables the I<sup>2</sup>C bus.

Byte 4:

***pin***

D0 to D8      28-pin uM-FPU64 chip  
D0 to D22     44-pin uM-FPU64 chip

#### Pin Assignments

*pin*            SDA  
*pin*+1        SCL

Byte 5:

***config***

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

Bit 0

#### Speed

IDE Symbol	IDE Value	Description
SLOW	0x00	100 kHz
FAST	0x01	400 kHz

### Start Write (0x02)

DEVIO, I2C, START\_WRITE, *device*

Sends the start write sequence to the I<sup>2</sup>C bus, and sets the status bits for the acknowledge.

Z=ACK, NZ=NAK

Byte 4:

***device***

Unsigned byte specifying the 8-bit I<sup>2</sup>C device address. (7-bit I<sup>2</sup>C device address left justified and a least significant bit of zero (e.g. 0x00 to 0xFF). If the *device* byte is zero, the lower 8 bits of register 0 are used for the device address. This allows for variable device addresses (e.g. for addressing multiple EEPROMs chips).

### Stop (0x03)

DEVIO, I2C, STOP

Sends the stop sequence to the I<sup>2</sup>C bus. This action is required to end a write transaction, but is not required for read transactions. Read transactions handle the stop sequence automatically.

### Start Read (0x04)

DEVIO, I2C, START\_READ, *byteCount*

Sends the start read sequence to the I<sup>2</sup>C bus, and specifies the number bytes in the read transfer. If the START\_READ action is optional, but can result in more efficient read transfers if multiple read actions are required. If no START\_READ action is used, then each DEVIO read operation is a separate read transaction.

Byte 4:

***byteCount***

Specifies the number bytes to read. A NAK will be sent after the last byte read followed by the stop sequence. Any of the DEVIO read operations can be used to read the bytes in the transfer

## DEVIO, LCD      LCD Interface

**Syntax:**      **DEVIO, LCD, action{,...}**

**Description:**      This instruction provides support for LCD displays that are compatible with the widely used HD44780 chipset. It uses a 4-bit parallel interface, and two control pins if configured as write-only, or three control pins if configured for read and write.

```
DEVIO, LCD, DISABLE
DEVIO, LCD, ENABLE, pin, config
DEVIO, LCD, CLEAR
DEVIO, LCD, HOME
DEVIO, LCD, MOVE, row, column
DEVIO, LCD, MOVE_REG, rowReg, colReg
DEVIO, LCD, CMD, command
DEVIO, LCD, INTERFACE, type
DEVIO, LCD, BACKLIGHT_ON
DEVIO, LCD, BACKLIGHT_OFF
```

**Opcode:**      **DA**

**Byte 2:**      **LCD (0x80)**

**Byte 3:**      **action**

An unsigned byte specifying the device action. Actions that are specific to the LCD device are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

```
DEVIO, LCD, DISABLE
    Disable the LCD device and release the digital pins.
```

### Enable (0x01)

```
DEVIO, LCD, ENABLE, pin, config
    Selects the pins to use for the LCD, configures and initializes the display.
```

**Byte 4:**      **pin**

Specifies the pins to use for the LCD interface.

D0 to D8      28-pin uM-FPU64 chip

D0 to D22      44-pin uM-FPU64 chip

### Pin Assignments

*pin* to *pin*+3      4-bit data

*pin*+4      E pin

*pin*+5      RS pin

*pin*+6      RW pin (if enabled)

**Byte 5:**      **config**

Bit	7	6	5	4	3	2	1	0
	-	R	F	Row			Col	



Bit 6	<b>RW pin setting</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	–	0x00	Read disabled, RW pin grounded
Bit 5	<b>Font</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	–	0x00	5x7 font
Bits 4:3	<b>Row</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	ROWS_1	0x00	1 row
Bits 2:0	<b>Column</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	COLS_8	0x00	8 columns
	COLS_12	0x01	12 columns
	COLS_16	0x02	16 columns
	COLS_20	0x03	20 columns
	COLS_40	0x04	40 columns

#### Clear Display (0x02)

DEVIO, LCD, CLEAR  
Clear the LCD display.

#### Home (0x03)

DEVIO, LCD, HOME  
Move cursor to the home position.

#### Move to row, column (0x04)

DEVIO, LCD, MOVE, *row*, *column*  
Move to the *row* and *column* specified.

Byte 4: ***row***  
Unsigned byte specifying the row number (0 to 3).

Byte 5: ***column***  
Unsigned byte specifying the column number (0 to 39).

#### Move to row, column using register (0x05)

DEVIO, LCD, MOVE\_REG, *rowReg*, *colReg*  
Move to the row and column specified by the values in *rowReg* and *colReg* registers.

Byte 4: ***rowReg***  
Register containing the row number.

Byte 5: ***column***  
Register containing column number.

#### Send Command (0x06)

DEVIO, LCD, CMD, *command*  
Send HD44780U compatible LCD command.

Byte 4:

**command**

Unsigned byte specifying the HD44780U compatible LCD command.

**Device Type (0x07)**

DEVIO, LCD, INTERFACE, *type*

Used to select an I<sup>2</sup>C controller for the LCD interface. If this command is not used, digital I/O pins are used for a direct interface. A `devio(I2C, ENABLE, ...)` function call must be done previously, and this function call must be done before the `devio(LCD, ENABLE, ...)` function call.

Byte 4:

**type**

Bit	7	6	5	4	3	2	1	0
	Device				Address			

Bits 6:4

**Device**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	digital pins (default)
ST7036	0x10	I <sup>2</sup> C controller
PCF8574	0x28	I <sup>2</sup> C controller
MCP23008	0x30	I <sup>2</sup> C controller

Bits 3:0

**Address**

Lower bits of I<sup>2</sup>C device address.

ST7036	2 bits	Device address: 0x78 to 0x7B
PCF8574	3 bits	Device address: 0x40 to 0x47
MCP23008	3 bits	Device address: 0x40 to 0x47

**Turn BackLight On (0x08)**

DEVIO, LCD, BACKLIGHT\_ON

When a PCF8574 or MCP23008 controller is used, this command turns on the backlight.

**Turn BackLight Off (0x09)**

DEVIO, LCD, BACKLIGHT\_OFF

When a PCF8574 or MCP23008 controller is used, this command turns off the backlight.

## DEVIO, MEM      Memory Interface

**Syntax:**      **DEVIO, MEM, action{, ...}**

**Description:**      This instruction stores data to the general memory area in RAM. The total amount of available RAM is 2304 bytes, which is split into a foreground memory, background memory, FIFO1, FIFO2, FIFO3, and FIFO4, dynamic allocation pool. The default allocation of RAM is as follows:

Foreground	2304 bytes
Background	0 bytes
FIFO1	0 bytes
FIFO2	0 bytes
FIFO3	0 bytes
FIFO4	0 bytes
Dynamic Allocation	0 bytes

The allocation can be changed with the *DEVIO, MEM, ALLOCATE* instruction. All memory not allocated to the Foreground, Background, FIFO1, FIFO2, FIFO3, FIFO4 is available for dynamic allocation to FIFOs or loadable devices using the *DEVIO, FIFO*n*, ALLOC\_MEM, size*, *DEVIO, FIFO*n*, ALLOC\_MEMR, regSize*, or *DEVIO, device, LOAD\_DEVICE, xopdev* instruction.

```
DEVIO, MEM, DISABLE
DEVIO, MEM, ENABLE, pin, config
DEVIO, MEM, ALLOCATE, memSize, fifoSize
DEVIO, MEM, ALLOCATE, memSize, fgSize
```

**Opcode:**      **DA**

**Byte 2:**      **MEM (0x00)**

**Byte 3:**      **action**  
                  An unsigned byte specifying the device action. Actions that are specific to the memory device are shown below. For actions that are common to all devices, see the *DEVIO* description.

### **Disable (0x00)**

```
DEVIO, MEM, DISABLE
This action is not required when using general memory in RAM.
```

### **Enable (0x01)**

```
DEVIO, MEM, ENABLE, pin, config
This action is not required when using general memory in RAM.
```

**Byte 4:**      **pin**  
                  Unused.

**Byte 5:**      **config**  
                  Unused.

### **Allocate Memory Buffers (0x02)**

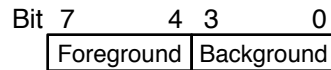
```
DEVIO, MEM, ALLOCATE, memSize, fifoSize
```

The 2304 bytes of available memory are allocated to the foreground, background, FIFO1, FIFO2, FIFO3, and FIFO4. The 4-bit value for each memory type specifies the amount of memory to allocate. At least 256 bytes are always allocated to the foreground. If the sum of all allocations is greater than the maximum 2304 bytes of available RAM, the foreground allocation is as specified, but no memory bytes are allocated to the background, FIFO1, FIFO2, FIFO3, or FIFO4. All remaining bytes are used for dynamic allocation.

Byte 4:

**memSize**

Unsigned byte specifying the number of bytes to allocate to the foreground and background memory buffers.

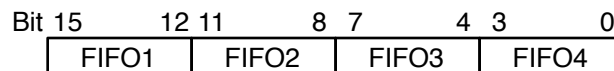


Bits 7:4      **Foreground Memory Allocation**  
 Bits 3:0      **Background Memory Allocation**

Byte 5-6:

**fifoSize** (*unsigned word*)

Unsigned 16-bit word specifying the number of bytes to allocate to the FIFO buffers.



Bits 15:12    **FIFO1 Memory Allocation**  
 Bits 11:8    **FIFO2 Memory Allocation**  
 Bits 7:4     **FIFO3 Memory Allocation**  
 Bits 3:0     **FIFO4 Memory Allocation**

**Memory Allocation Codes**

<i>Value</i>	<i>Description</i>
0x0	No memory
0x1	2 bytes
0x2	4 bytes
0x3	8 bytes
0x4	16 bytes
0x5	32 bytes
0x6	64 bytes
0x7	128 bytes
0x8	256 bytes
0x9	512 bytes
0xA	1024 bytes
0xB	2048 bytes
0xC	4096 bytes
0xD	8192 bytes
0xE	16384 bytes
0xF	default (FG: 1024 bytes, BG: 1024 bytes, FIFO1-4: 64 bytes)

**Allocate Memory Buffers (0x02)**

DEVIO, MEM, ALLOCATE, *memSize*, *fgSize*

If the foreground memory allocation bits (bits 7:4) of *memSize* are zero, then *fgSize* specifies the number of memory bytes allocated to the foreground, and no memory is allocated to FIFO1, FIFO2, FIFO3, and FIFO4. At least 256 bytes are always allocated to

the foreground. The background memory allocation bits (bits 3:0) of *memSize* specify the number of memory bytes allocated to the background. Any remaining bytes are available for dynamic allocation memory for FIFOs or loadable devices.

Byte 4: **0**

Byte 5-6: **fgSize** (*unsigned word*)  
 Unsigned 16-bit word specifying the number of memory bytes to allocate to the foreground.

---

## DEVIO, OWIRE     1-Wire Bus Interface

Syntax: **DEVIO,OWIRE,action{,...}**

Description: This instruction provides support for communicating with 1-Wire devices using a local 1-Wire bus on the specified digital pin.

DEVIO, OWIRE, DISABLE  
 DEVIO, OWIRE, ENABLE, *pin*, *config*  
 DEVIO, OWIRE, RESET  
 DEVIO, OWIRE, SELECT, *regAddr*  
 DEVIO, OWIRE, VERIFY, *regAddr*  
 DEVIO, OWIRE, SEARCH, *count*, *regAddr*  
 DEVIO, OWIRE, ALARM, *count*, *regAddr*  
 DEVIO, OWIRE, FAMILY\_SEARCH, *count*, *regAddr*  
 DEVIO, OWIRE, FAMILY\_ALARM, *count*, *regAddr*

Opcode: **DA**

Byte 2: **OWIRE (0x10-0x1F)**

Byte 3: **action**  
 An unsigned byte specifying the device action. Actions that are specific to 1-Wire devices are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

DEVIO, OWIRE, DISABLE  
 Disable the 1-wire bus and release the digital pin.

### Enable (0x01)

DEVIO, OWIRE, ENABLE, *pin*, *config*  
 The enable action is used to assign the digital pin for the 1-wire bus and initialize the bus.

Byte 4: **pin**  
 D0 to D8                      28-pin uM-FPU64 chip  
 D0 to D22                    44-pin uM-FPU64 chip

### Pin Assignments

pin                      1-Wire bus

Byte 5: **config**  
 Not used.

### Send Reset Pulse (0x02)

DEVIO, OWIRE, RESET

Sends a 1-wire reset pulse to the 1-wire bus.

### Select Device (0x03)

DEVIO, OWIRE, SELECT, *regAddr*

Selects the device on the 1-wire bus.

Byte 4:

***regAddr***

If *regAddr* is 32-bit, the SKIP\_ROM 1-wire command is used and no address is sent.

If *regAddr* is 64-bit, the MATCH\_ROM 1-wire command is used, and the 64-bit device address in *regAddr* is sent.

### Verify Device (0x04)

DEVIO, OWIRE, VERIFY, *regAddr*

Verifies that a device of the specified address is present on the 1-wire bus.

Byte 4:

***regAddr***

A 64-bit register specifying the device address.

### Search All (0x05)

DEVIO, OWIRE, SEARCH, *count*, *regAddr*

Searches for all devices on the 1-wire bus.

Byte 4:

***count***

Unsigned byte specifying the maximum number of addresses to store in consecutive registers starting at register *regAddr*.

Byte 5:

***regAddr***

The first of *count* 64-bit registers that the address of all devices found on the 1-wire bus.

### Alarm Search (0x06)

DEVIO, OWIRE, ALARM, *count*, *regAddr*

Searches for all devices on the 1-wire bus.

Byte 4:

***count***

Unsigned byte specifying the maximum number of addresses to store in consecutive registers starting at register *regAddr*.

Byte 5:

***regAddr***

The first of *count* 64-bit registers that the address of all devices found on the 1-wire bus that have an active alarm value.

### Family Search (0x07)

DEVIO, OWIRE, FAMILY\_SEARCH, *count*, *regAddr*

Searches for all devices on the 1-wire bus.

Byte 4:

***count***

Unsigned byte specifying the maximum number of addresses to store in consecutive registers starting at register *regAddr*.

*Byte 5:*                   **regAddr**  
The first of *count* 64-bit registers that the address of all devices found on the 1-wire bus that are part of the specified family of devices.

#### Family Alarm Search (0x08)

**DEVIO, OWIRE, FAMILY\_ALARM, count, regAddr**  
Searches for all devices on the 1-wire bus.

*Byte 4:*                   **count**  
Unsigned byte specifying the maximum number of addresses to store in consecutive registers starting at register *regAddr*.

*Byte 5:*                   **regAddr**  
The first of *count* 64-bit registers that the address of all devices found on the 1-wire bus that have an active alarm value and are part of the specified family of devices.

## DEVIO, SDFAT      SD card with FAT16 and FAT32 support

*Syntax:*               **DEVIO, SDFAT, action{,...}**

*Description:*       This instruction provides support for SD cards and supports the FAT16 and FAT32 file systems. The SD card can be used in raw mode for general non-volatile storage of up to 32GB of data, or in file mode with FAT16 and FAT32 support. Files stored on SD cards can be read by any device that supports the FAT16 and FAT32 file system..

DEVIO, SDFAT, DISABLE  
DEVIO, SDFAT, ENABLE, *pin, config*  
DEVIO, SDFAT, STATUS  
DEVIO, SDFAT, GET\_VALUE, *item*  
DEVIO, SDFAT, READ\_BLOCK, *regBlock, regPtr*  
DEVIO, SDFAT, WRITE\_BLOCK, *regBlock, regPtr*  
DEVIO, SDFAT, FIND, *filename*  
DEVIO, SDFAT, NEXT  
DEVIO, SDFAT, OPEN, *type, filename*  
DEVIO, SDFAT, CLOSE  
DEVIO, SDFAT, UPDATE  
DEVIO, SDFAT, GET\_POSITION  
DEVIO, SDFAT, SET\_POSITION, *regAddress*

*Opcode:*               **DA**

*Byte 2:*               **SDFAT (0xA0)**

*Byte 3:*               **action**  
An unsigned byte specifying the device action. Actions that are specific to the SDFAT device are shown below. For actions that are common to all devices, see the *DEVIO* description.

#### Disable (0x00)

**DEVIO, SDFAT, DISABLE**  
Disable the SDFAT device and release the digital pins.

#### Enable (0x01)

DEVIO, SDFAT, ENABLE, *pin*, *config*

Selects the pins to use for the SDFAT, configures and initializes the display.

Byte 4:

***pin***

Specifies the pins to use for the SD card select.

D0 to D8          28-pin uM-FPU64 chip

D0 to D22        44-pin uM-FPU64 chip

#### Pin Assignments

*pin*                  SD card select

Byte 5:

***config***

Bit	7	6	5	4	3	2	1	0
	E							Par

#### Bit 7          Write Block Enable

IDE Symbol	IDE Value	Description
-	0x00	WRITE_BLOCK action disabled
WRBLK_ENABLE	0x80	WRITE_BLOCK action disabled

#### Bits 1:0      Partition

IDE Symbol	IDE Value	Description
-	0x00	Use partition 1 on SD card (default)
-	0x01	Use partition 2 on SD card (default)
-	0x02	Use partition 3 on SD card (default)
-	0x03	Use partition 4 on SD card (default)

#### Status (0x02)

DEVIO, SDFAT, STATUS

Returns the current status of the SDFAT device in register 0.

0          OK

-1        End of File

< -2      Error value

#### Get Value (0x03)

DEVIO, SDFAT, GET\_VALUE, *item*

Returns the value specified by *item*.

0          Current status

1          Volume Size

#### Read Block (0x04)

DEVIO, SDFAT, READ\_BLOCK, *regBlock*, *regPtr*

#### Write Block (0x05)

DEVIO, SDFAT, WRITE\_BLOCK, *regBlock*, *regPtr*

#### Find File (0x06)

DEVIO, SDFAT, FIND, *filename*

#### Next File (0x07)

DEVIO, SDFAT, NEXT



### Open File (0x08)

DEVIO, SDFAT, OPEN, *type*, *filename*

### Close File (0x09)

DEVIO, SDFAT, CLOSE

### Update File (0x0A)

DEVIO, SDFAT, UPDATE

### Get Position (0x0B)

DEVIO, SDFAT, GET\_POSITION

### Set Position (0x0C)

DEVIO, SDFAT, SET\_POSITION, *regAddress*

## DEVIO, SERVO      Servo Control Interface

*Syntax:*            **DEVIO, SERVO+n, action{,...}**

*Description:*      This instruction is used to interface with servo controllers on the specified digital pins.

DEVIO, SERVO+n, DISABLE  
DEVIO, SERVO+n, ENABLE, *pin*, *config*  
DEVIO, SERVO+n, PULSE, *register*  
DEVIO, SERVO+n, SPEED, *register*  
DEVIO, SERVO+n, TIME, *register*  
DEVIO, SERVO+n, MOVE  
DEVIO, SERVO+n, HOME  
DEVIO, SERVO+n, READ\_PULSE  
DEVIO, SERVO+n, STATUS

*Opcode:*            **DA**

*Byte 2:*            **device (0x60-63)**

*Byte 3:*            **action**

An unsigned byte specifying the device action. Actions that are specific to servo controllers are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

DEVIO, SERVO+n, DISABLE  
Disable the servo controller and release the digital pin.

### Enable (0x01)

DEVIO, SERVO+n, ENABLE, *pin*, *config*  
Initialize the servo controller and optionally assign event.

*Byte 4:*            **pin**  
                    D0 to D8            28-pin uM-FPU64 chip  
                    D0 to D22        44-pin uM-FPU64 chip

Byte 5:

**config**

Bit 7	6	5	4	3	2	1	0
E		-					Event

Bit 7

**Extended Mode**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	Normal, Pulse widths (800 usec to 2200 usec).
EXTENDED	0x80	Extended Mode, Pulse widths (500 usec to 2500 usec).

Bits 2:0

**Event Number**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
NO_EVENT	0x00	No event.
EVENT1	0x01	Event 1.
EVENT2	0x02	Event 2.
EVENT4	0x03	Event 3.
EVENT5	0x04	Event 4.
EVENT6	0x05	Event 5.
EVENT7	0x06	Event 6.
EVENT1	0x07	Event 7.

**Set Pulse Width (0x02)**

DEVIO, SERVO+n, PULSE, *register*

Sets the pulse width of the servo to the value specified in the register..

Byte 4:

*register*

**Set Speed (0x03)**

DEVIO, SERVO+n, SPEED, *register*

Sets the speed of movement (microseconds/second) for the servo to the value specified in the register.

Byte 4:

*register*

**Set Time (0x04)**

DEVIO, SERVO+n, TIME, *register*

Sets the time of movement (milliseconds) for all servos in a group move to the value specified in the register. The movement will not occur until the MOVE action is received.

Byte 4:

*register*

**Move (0x05)**

DEVIO, SERVO+n, MOVE

Move all of the servos in a group move in the time specified by the TIME action.

**Move Home (0x06)**

DEVIO, SERVO+n, HOME

Move all active servos to the home position (1500 microseconds).

**Read Pulse Width (0x07)**

DEVIO, SERVO+n, READ

Return the current pulse width of the servo in register 0.

**Get Servo Status (0x08)**

**DEVIO, SERVO+*n*, STATUS**

Return the current status of all servos in register 0. The status bit is zero if the servo is in position, and one if the servo is still moving. The status value is as follows:

Bit	7	6	5	4	3	2	1	0
				-	3	2	1	0
Bit 3					Servo 3 Status			
Bit 2					Servo 2 Status			
Bit 1					Servo 1 Status			
Bit 0					Servo 0 Status			

---

## DEVIO, SPI      SPI Interface

**Syntax:**      **DEVIO, SPI+*n*, action{, ...}**

**Description:**      This instruction provides support for communicating with SPI devices using a local SPI bus on the specified digital pins.

DEVIO, SPI+*n*, DISABLE  
DEVIO, SPI+*n*, ENABLE, *pin*, *config*  
DEVIO, SPI+*n*, CS\_LOW  
DEVIO, SPI+*n*, CS\_HIGH

**Opcode:**      **DA**

**Byte 2:**      **device (0x30-0x3F)**

**Byte 3:**      **action**  
An unsigned byte specifying the device action. Actions that are specific to SPI devices are shown below. For actions that are common to all devices, see the *DEVIO* description.

### Disable (0x00)

DEVIO, SPI+*n*, DISABLE  
Disable the SPI device and release the digital pin.

### Enable (0x01)

DEVIO, SPI+*n*, ENABLE, *pin*, *config*  
Initialize the SPI device according to the configuration byte.

**Byte 4:**      ***pin***  
D0 to D8      28-pin uM-FPU64 chip  
D0 to D22      44-pin uM-FPU64 chip (Device 0)  
D0 to D22      44-pin uM-FPU64 chip (Device 1-15)

### Pin Assignments

Device 0

<i>pin</i>	SCLK	Serial clock (from FPU)
<i>pin</i> +1	MOSI	Master Output, Slave Input (from FPU)
<i>pin</i> +2	MISO	Master input, Slave Output (to FPU)

Device 1-15

<i>pin</i>	/CS	Slave chip select (from FPU)
------------	-----	------------------------------

Byte 5:

**config**

Bit	7	6	5	4	3	2	1	0
	R	Mode						Speed

Bit 7

**Read**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	Hold MOSI low during read.
READ_HIGH	0x80	Hold MOSI high during read.

Bits 6:5

**Clock Mode**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
MODE0	0x00	Idle state low, data captured on rising edge.
MODE1	0x20	Idle state low, data captured on falling edge.
MODE2	0x40	Idle state high, data captured on falling edge.
MODE3	0x60	Idle state high, data captured on rising edge.

Bits 4:0

**Speed**

<i>Value</i>	<i>Description</i>
0	78.125 kHz
1	89.285 kHz
2	104.166 kHz
3	125.000 kHz
4	156.25 kHz
5	208.333 kHz
6	312.500 kHz
7	625.000 kHz
8	1.250 MHz
9	2.500 MHz
10	1.667 MHz
11	2.000 MHz
12	2.500 MHz
13	3.333 MHz
14	5.000 MHz
15	10.000 MHz

**Set CS low (0x02)**

DEVIO, SPI+n, CS\_LOW

Sets the pin CS pin assigned to SPI device *n* low.

**Set CS high (0x03)**

DEVIO, SPI+n, CS\_HIGH

Sets the pin CS pin assigned to SPI device *n* high.

When multiple devices are used on the local SPI bus, device 0 is used to assign the hardware SPI pins, and device 1 to *n* are used to specify the pin that is connected to the /SS pin of each slave device. Each SPI device can have a unique SPI mode and clock speed.

## DIGIO Digital Input/Output

*Syntax:* **DIGIO, action{, mode}**

*Description:* This instruction is used to read and write the digital pins. The 28-pin uM-FPU64 chip has 9 digital pins (D0 to D8) and the 44-pin uM-FPU64 chip has 23 digital pins (D0 to D22). The byte immediately following the opcode specifies the action and pin number. The WRITE\_BITS, READ\_BITS, WRITE\_BITP, and READ\_BITP actions require an additional mode byte. The additional bytes used by the instruction, and the various actions that can be performed, are described below.

*Opcode:* **D0**

*Byte 2:* **action**

Bit	7	6	5	4	3	2	1	0
	Action			Pin				

Bits 7:5	<b>Action</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	LOW	0x00	Set Pin Low
	HIGH	0x20	Set Pin High
	TOGGLE	0x40	Toggle Pin
	INPUT	0x60	Read Input from Pin
	WRITE_BITS	0x80	Write Serial Bits to Pins
	READ_BITS	0xA0	Read Serial Bits from Pins
	WRITE_BITP	0xC0	Write Parallel Bits to Pins
	READ_BITP	0xE0	Read Parallel Bits from Pins

Bits 4:0	<b>Pin</b>	
	<i>Value</i>	<i>Description</i>
	D0 - D8	The 28-pin chip has digital pins D0 to D8.
	D0 - D22	The 44-pin chip has digital pins D0 to D22.

### Set Pin Low

DIGIO, LOW+*pin*

The *pin* is configured as an output and set low.

### Set Pin High

DIGIO, HIGH+*pin*

The *pin* is configured as an output and set high.

### Toggle Pin

DIGIO, TOGGLE+*pin*

The *pin* is configured as an output, and set to the opposite value as the current *pin* value.

### Read Input from Pin

DIGIO, INPUT+*pin*

The *pin* is configured as an input, and the value of the *pin* is read. The result is stored in the status register.

If *pin* is low, status = Z  
 If *pin* is high, status = NZ

### Write Serial Bits to Pins

DIGIO, WRITE\_BITS+*pin*, *mode*

The value in register 0 is written serially to *pin* and *pin*+1 according to the *mode* specified.

*pin* data pin  
*pin*+1 clock pin

Byte 3:

**mode** (for WRITE\_BITS and READ\_BITS actions)

Bit 7	6	5	4	3	2	1	0
P	M	F	# of Bits				

#### Bit 7 Sample Time (READ\_BITS)

IDE Symbol	IDE Value	Description
PRE	0x00	Sample pin level before clock pulse.
POST	0x80	Sample pin level after clock pulse.

#### Bit 6 Bit order

IDE Symbol	IDE Value	Description
MSB	0x00	Read most significant bit first.
LSB	0x40	Read least significant bit first.

#### Bit 5 Clock Speed

IDE Symbol	IDE Value	Description
FAST	0x00	Fast speed clock (743 kHz).
SLOW	0x20	Slow speed clock (534 kHz).

#### Bits 4:0 Number of Bits

Value	Description
0 - 31	The number of bits to transfer. A value of 0 specifies 32 bits.

### Read Serial Bits from Pins

DIGIO, READ\_BITS+*pin*, *mode*

The value is read serially from *pin* and *pin*+1, according to the *mode* specified, and the result is stored in register 0.

*pin* data pin  
*pin*+1 clock pin

Byte 3:

**mode**

(see description above)

### Write Parallel Bits to Pins

DIGIO, WRITE\_BITP+*pin*, *mode*

The *pins* are configured as outputs, and the value in register 0 is written in parallel to the specified *pins*.

Byte 3:

**mode** (for WRITE\_BITP and READ\_BITP actions)

Bit 7	6	5	4	3	2	1	0
I	-	-	# of Bits				

#### Bit 7 Invert

IDE Symbol	IDE Value	Description
------------	-----------	-------------

-	-	Bits are not inverted.
INVERT	0x40	Bits are inverted before write and after read.
<b>Bits 4:0 Number of Bits</b>		
<i>Value</i>	<i>Description</i>	
1 - 9	The 28-pin chip has 9 digital pins (D0 to D8).	
1 - 23	The 44-pin chip has 23 digital pins (D0 to D22).	

#### Read Parallel Bits from Pins

DIGIO, READ\_BITP+*pin*, *mode*

The *pins* are configured as inputs, and the value of the specified *pins* are read in parallel and stored in register 0.

Byte 3: **mode**  
(see description above)

*Examples:* DIGIO, LOW+1 Pin D1 is set to output low.  
DIGIO, WRITE\_BITS+3, 8 The lower 8 bits of register 0 is serially shifted out most significant bit first, using D3 as the data pin, and D4 as the clock pin.  
DIGIO, READ\_BITP+5, 4 The lower 4 bits of register 0 are set to the value of pins D5, D6, D7, and D8.

*See Also:* DEVIO

### DREAD Read 64-bit value

*Syntax:* **DREAD, register**

*Description:* The 64-bit value of *register* is returned.

*Opcode:* **73**

Byte 2: **register**  
Register number (0 to 255).

*Returns:* *byte1, byte2, byte3, byte4, byte5, byte6, byte7, byte8*  
The eight bytes representing the 64-bit value (MSB first) must be read immediately following this instruction.

*Special Cases:* • if *register* is 32-bit, the value is converted to 64-bit with sign extended before being sent.

*See Also:* SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, RDIND

### DWRITE Write 64-bit value

*Syntax:* **DWRITE, register, value**

*Description:* The 64-bit integer value is stored in *register*.

reg[*register*] = 64-bit value, status = longStatus(reg[*register*])

*Opcode:*       **72**

*Byte 2:*        **register**  
                  Register number (0 to 255).

*Bytes 3-10:*   **value**  
                  64-bit value represented by eight bytes (MSB first).

*Special Cases:*

- if *register* is 32-bit, only the lower 32-bits of the value are stored.
- if *register* = 0 or 128, and SETARGS is not active
  - if reg[A] is 32-bit, the value is stored in registers 0
  - if reg[A] is 64-bit, the value is stored in registers 128
- if *register* = 0 or 128, and SETARGS is active
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*       FWRITE, FWRITE0, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEA, LWRITEX, WRIND, SETARGS

## **EVENT       Background events**

*Syntax:*       **EVENT, action{, function}**

*Description:*   Used to manage background events.

*Opcode:*       **7F**

*Byte 2:*        **action**

Bit	7	6	5	4	3	2	1	0
	Action				Event			

Bits 7:4		<b>Action</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>	
	DISABLE	0x00	Disable event.	
	ENABLE	0x10	Enable event.	
	PERIOD	0x20	Set period for timer event (1 to 3).	
	SET	0x30	Set event flag.	
	CLEAR	0x40	Clear event flag.	
	WAIT	0x50	Wait for event flag.	
	TEST	0x60	Test event flag.	

Bits 3:0		<b>Event</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>	
	CALL	0x00	Call background function event.	
	EVENT1, TIMER1	0x01	Software event, Timer 1 event.	
	EVENT2, TIMER2	0x02	Software event, Timer 2 event.	
	EVENT3, TIMER3	0x03	Software event, Timer 3 event.	
	EVENT4, FIFO1	0x04	Software event, FIFO1 event.	



EVENT5, FIFO2	0x05	Software event, FIFO2 event.
EVENT6, FIFO3	0x06	Software event, FIFO3 event.
EVENT7, FIFO4	0x07	Software event, FIFO4 event.
SERIN	0x08	SERIN receive event.
ASYNC	0x09	DEVIO, ASYNC receive event.
EXTIN	0x0A	External input event.
ADC	0x0B	ADC ready event.
RTC	0x0C	Real-time clock event.
DELAY_FG	0x0E	Delay Foreground event. (SET only)
DELAY_BG	0x0F	Delay Background event. (SET only)

**EVENT, DISABLE+event**  
Disables a background event.

**EVENT, ENABLE+event, function**  
Enables a background event. When the event flag is set, the specified function is executed in the background. If the event is a timer event (TIMER1, TIMER2, TIMER3) the time period must also be set to a value other than zero to enable the timer.

Byte 3:

**function**  
Function number to execute when event occurs.

**EVENT, PERIOD+event**  
Sets the time period for timer events (TIMER1, TIMER2, TIMER3). If register A is 32-bit, the time period in milliseconds is read from the lower 16 bits of register 0. If register A is 64-bit, the time period in milliseconds is read from the lower 16 bits of register 128. If the period is set to zero, the timer will be disabled. If the period is set to a non-zero value, then timer is enabled.

**EVENT, SET+event**  
Set the event flag and causes the background function to execute.

**EVENT, CLEAR+event**  
Clears the event flag. The event flag is cleared automatically when an event occurs, so this action is not normally required.

**EVENT, WAIT+event**  
Waits until the event flag is set, then clears the event flag. If there are other instructions in the instruction buffer, or another instruction is sent before the **EVENT, WAIT+event** instruction has completed, it will terminate and clear the event flag.

**EVENT, TEST+event**  
Tests the event flag and sets the the internal status byte.  
Status = Z            event flag not set  
Status = NZ          event flag set

**Notes:** The ENABLE and DISABLE actions are used to implement background events. The SET, CLEAR, WAIT, and TEST actions are used to work with event flags in the foreground.

**Examples:**      **LOADWORD, 500**                      Load time period value..  
                 **EVENT, ENABLE+TIMER1, 1**   Enable timer 1 event with 500 msec period. Function 1 called

EVENT, ENABLE+RTC, 2      on event.  
    Enable RTC event. Function 2 called on event.

*Special Cases:*      • when setting the period for the TIMER1, TIMER2, TIMER3, TIMER4 events, only the lower 16 bits of reg[0 | 128] are used

---

## EXP      The value e raised to a power

*Syntax:*      **EXP**

*Description:*      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.

reg[A] = exp(reg[A])

*Opcode:*      **45**

*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

*See Also:*      FPOW, FPOWI, FPOW0, EXP10, LOG, LOG10, ROOT, SQRT

---

## EXP10      The value 10 raised to a power

*Syntax:*      **EXP10**

*Description:*      Calculates the value of 10 raised to the power of the floating point value in register A, and stores the result in register A.

reg[A] = exp10(reg[A])

*Opcode:*      **46**

*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 32-bit, and reg[A] is greater than 38, then the result is +infinity  
                                  • if reg[A] is 64-bit, and reg[A] is greater than 308, then the result is +infinity  
                                  • if reg[A] is -infinity, the result is 0.0  
                                  • if reg[A] is 32-bit, and reg[A] is less than -38, then the result is 0.0  
                                  • if reg[A] is 64-bit, and reg[A] is less than -308, then the result is 0.0

*See Also:*      FPOW, FPOWI, FPOW0, EXP, LOG, LOG10, ROOT, SQRT

---

## EXTLONG      Load value of external input counter

*Syntax:*      **EXTLONG**

*Description:*      Load register 0 with the external input count.

if reg[A] is 32-bit,  
     reg[0] = external input count, status = longStatus(reg[0])  
 if reg[A] is 64-bit,  
     reg[128] = external input count, status = longStatus(reg[128])

*Opcode:*       **E1**

*See Also:*     EXTSET, EXTWAIT

## **EXTSET     Set value of external input counter**

*Syntax:*       **EXTSET**

*Description:*   The external input count is set to the value in register 0. If the value is -1 (0xFFFFFFFF) the external input counter is disabled.

if reg[A] is 32-bit,  
     external input count = reg[0]  
 if reg[A] is 64-bit,  
     external input count = reg[128]

*Opcode:*       **E0**

*Special Cases:*   • if reg[A] is 64-bit, then only the lower 32 bits are used to set the external input count

*See Also:*     EXTLONG, EXTWAIT

## **EXTWAIT    Wait for next external input pulse**

*Syntax:*       **EXTWAIT**

*Description:*   Wait for the next external input to occur. If there are other instructions in the instruction buffer, or another instruction is sent before the EXTWAIT instruction has completed, it will terminate.

*Opcode:*       **E2**

*See Also:*     EXTLONG, EXTSET

## **FABS        Floating point absolute value**

*Syntax:*       **FABS**

*Description:*   Sets the floating value in register A to the absolute value.

reg[A] = | reg[A] |

*Opcode:*       **3F**

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

*See Also:* FNEG, LABS, LNEG

---

## **FADD** Floating point add

*Syntax:* **FADD, register**

*Description:* The floating point value in *register* is added to the value in register A.

$$\text{reg}[A] = \text{reg}[A] + \text{reg}[\text{register}]$$

*Opcode:* **21**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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

*See Also:* FADD0, LADD, LADDI, LADD0

---

## **FADDI** Floating point add immediate value

*Syntax:* **FADDI, signedByte**

*Description:* The signed byte value is converted to floating point and added to the value in register A.

$$\text{reg}[A] = \text{reg}[A] + \text{float}(\text{signedByte})$$

*Opcode:* **33**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

*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

*See Also:* FADD, FADD0, LADD, LADDI, LADD0

---

## **FADD0** Floating point add register 0

*Syntax:* **FADD0**

*Description:* If register A is 32-bit, the floating point value in register 0 is added to the value in register A. If register A is 64-bit, the floating point value in register 128 is added to the value in register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] + \text{reg}[0]$   
 if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] + \text{reg}[128]$

*Opcode:*       **2A**

*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

*See Also:*       FADD, FADDI, LADD, LADDI, LADD0

## **FCALL       Call Flash memory user defined function**

*Syntax:*       **FCALL, function**

*Description:*   The user-defined *function*, stored in Flash memory, is executed. Up to 16 levels of nesting is supported for function calls. The register A selection is stored by FCALL. If SETARGS was used prior to FCALL, the register A selection saved by the first SETARGS instruction is stored. If no SETARGS was used prior to FCALL, the current register A selection is stored. The register A selection is restored by the RET or RET, CC instruction that returns from the function being called. The uM-FPU IDE provides support for programming user defined functions in Flash memory using the serial debug monitor.

*Opcode:*       **7E**

*Byte 2:*       **function**  
 A function number (0 to 63).

*Special Cases:*

- only valid inside user-defined functions stored in Flash memory.
- if the user function is not defined, register 0 is set to NaN, and execution continues.

*See Also:*       BRA, BRA, CC, GOTO, JMP, JMP, CC, RET, RET, CC

## **FCMP       Floating point compare**

*Syntax:*       **FCMP, register**

*Description:*   Compares the floating point value in register A with the value in *register* and sets the internal status byte.

$\text{status} = \text{floatStatus}(\text{reg}[A] - \text{reg}[\text{register}])$

*Opcode:*       **28**

*Byte 2:*       **register**  
 Register number (0 to 255).

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 $\text{reg}[A] < \text{reg}[\text{register}]$
Bit 0	Zero	Set if $\text{reg}[A] = \text{reg}[\text{register}]$
		If neither Bit 0 or Bit 1 is set, $\text{reg}[A] > \text{reg}[\text{register}]$

*Special Cases:*

- if  $\text{reg}[A]$  is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if  $\text{reg}[A]$  is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:* FCMPI, FCMP0, FCMP2, LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2

## FCMPI Floating point compare immediate value

*Syntax:* **FCMPI, signedByte**

*Description:* 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	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 $\text{reg}[A] < \text{float}(\text{signedByte})$
Bit 0	Zero	Set if $\text{reg}[A] = \text{float}(\text{signedByte})$
		If neither Bit 0 or Bit 1 is set, $\text{reg}[A] > \text{float}(\text{signedByte})$

$\text{status} = \text{floatStatus}(\text{reg}[A] - \text{float}(\text{signedByte}))$

*Opcode:* **3A**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

The status byte can be read with the READSTATUS instruction.

*See Also:* FCMP, FCMP0, FCMP2, LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2

## FCMP0 Floating point compare register 0

*Syntax:* **FCMP0**

*Description:* If register A is 32-bit, the floating point value in register A is compared with the value in register 0, and the internal status byte is set. If register A is 64-bit, the signed long integer value in register A is compared with the value in register 128, and the internal status byte is set.

if reg[A] is 32-bit, status = floatStatus(reg[A] - reg[0])  
 if reg[A] is 64-bit, status = floatStatus(reg[A] - reg[128])

*Opcode:* **31**

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[0 | 128]  
 Bit 0    Zero            Set if reg[A] = reg[0 | 128]  
                               If neither Bit 0 or Bit 1 is set, reg[A] > reg[0 | 128]

*See Also:*    FCMP, FCMPI, FCMP2, LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2

## FCMP2      Floating point compare

*Syntax:*      **FCMP2, register1, register2**

*Description:*    Compares the floating point value in *register1* with the value in *register2* and sets the internal status byte.

status = floatStatus(reg[*register1*] - reg[*register2*])

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[*register2*] < reg[*register1*]  
 Bit 0    Zero            Set if reg[*register2*] = reg[*register1*]  
                               If neither Bit 0 or Bit 1 is set, reg[*register2*] > reg[*register1*]

*Opcode:*      **3D**

*Byte 2:*      **register1**  
 Register number (0 to 255).

*Byte 3:*      **register2**  
 Register number (0 to 255).

*Special Cases:*    • if *register1* is 32-bit and *register2* is 64-bit, the value is converted to 32-bit before being used  
                               • if *register1* is 64-bit and *register2* is 32-bit, the value is converted to 64-bit before being used

*See Also:*    FCMP, FCMPI, FCMP0, LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2

## FCNV      Floating point conversion

*Syntax:* **FCNV, conversion**

*Description:* Convert the value in register A using the *conversion* specified and store the result in register A. If register A is 32-bit, the conversion uses 32-bit constants. If register A is 64-bit, the conversion uses 64-bit constants.

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

*Opcode:* **56**

*Byte 2:* **conversion**

The conversion codes are as follows:

<i>Value</i>	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
0	F_C	00	Fahrenheit to Celsius
1	C_F	01	Celsius to Fahrenheit
2	IN_MM	02	inches to millimeters
3	MM_IN	03	millimeters to inches
4	IN_CM	04	inches to centimeters
5	CM_IN	05	centimeters to inches
6	IN_M	06	inches to meters
7	M_IN	07	meters to inches
8	FT_M	08	feet to meters
9	M_FT	09	meters to feet
10	YD_M	0A	yards to meters
11	M_YD	0B	meters to yards
12	MILES_KM	0C	miles to kilometers
13	KM_MILES	0D	kilometers to miles
14	NM_M	0E	nautical miles to meters
15	M_NM	0F	meters to nautical miles
16	ACRES_M2	10	acres to meters <sup>2</sup>
17	M2_ACRES	11	meters <sup>2</sup> to acres
18	OZ_G	12	ounces to grams
19	G_OZ	13	grams to ounces
20	LB_KG	14	pounds to kilograms
21	KG_LB	15	kilograms to pounds
22	USGAL_L	16	US gallons to liters
23	L_USGAL	17	liters to US gallons
24	UKGAL_L	18	UK gallons to liters
25	L_UKGAL	19	liters to UK gallons
26	USOZ_ML	1A	US fluid ounces to milliliters
27	ML_USOZ	1B	milliliters to US fluid ounces
28	UKOZ_ML	1C	UK fluid ounces to milliliters
29	ML_UKOZ	1D	milliliters to UK fluid ounces
30	CAL_J	1E	calories to Joules
31	J_CAL	1F	Joules to calories
32	HP_W	20	horsepower to watts
33	W_HP	21	watts to horsepower
34	ATM_KP	22	atmospheres to kilopascals



35	KP_ATM	23	kilopascals to atmospheres
36	MMHG_KP	24	mmHg to kilopascals
37	KP_MMHG	25	kilopascals to mmHg
38	DEG_RAD	26	degrees to radians
39	RAD_DEG	27	radians to degrees

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

*Examples:* FCNV, C\_F Converts the value in register A from celsius to fahrenheit.  
FCNV, IN\_CM Converts the value in register A from inches to centimeters.

---

## FCOPYI Copy Immediate value

*Syntax:* **FCOPYI, signedByte, register**

*Description:* The 8-bit signed value is converted to a long integer and copied to *register*.

$\text{reg}[\text{register}] = \text{float}(\text{signedByte}), \text{status} = \text{longStatus}(\text{reg}[\text{register}])$

*Opcode:* **5F**

*Byte 2:* **signedByte**  
An signed byte value (-128 to 127).

*Byte 3:* **register**  
Register number (0 to 255).

*See Also:* LCOPYI, COPY0, COPYA, COPYX

---

## FDIV Floating point divide

*Syntax:* **FDIV, register**

*Description:* The floating point value in register A is divided by the floating point value in *register*.

$\text{reg}[A] = \text{reg}[A] / \text{reg}[\text{register}]$

*Opcode:* **25**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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[*register*] is zero and reg[A] is not zero, then the result is infinity
- if reg[*register*] is infinity, then the result is zero

*See Also:* FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD, LDIV, LDIVI, LDIV0,

LUDIV, LUDIVI, LUDIV0

---

**FDIVI Floating point divide by immediate value***Syntax:* **FDIVI, *signedByte****Description:* The signed byte value is converted to floating point and the value in register A is divided by the converted value.
$$\text{reg}[A] = \text{reg}[A] / \text{float}(\text{signedByte})$$
*Opcode:* **37***Byte 2:* ***signedByte***  
A signed byte value (-128 to 127).*Special Cases:*

- if reg[A] is NaN, then the result is NaN
- if both values are zero, then the result is NaN
- if the *signedByte* is zero and reg[A] is not zero, then the result is infinity

*See Also:* FDIV, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

---

**FDIV0 Floating point divide by register 0***Syntax:* **FDIV0***Description:* If register A is 32-bit, the floating point value in register A is divided by the value in register 0. If register A is 64-bit, the floating point value in register A is divided by the value in register 128.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] / \text{reg}[0]$   
 if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] / \text{reg}[128]$

*Opcode:* **2E***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[0 | 128] is zero and reg[A] is not zero, then the result is infinity
- if reg[0 | 128] is infinity, then the result is zero

*See Also:* FDIV, FDIVI, FDIVR, FDIVRI, FDIVR0, FMOD, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

---

**FDIVR Floating point divide (reversed)***Syntax:* **FDIVR, *register****Description:* The floating point value in *register* is divided by the floating point value in register A and the result is stored in register A.
$$\text{reg}[A] = \text{reg}[\text{register}] / \text{reg}[A]$$

*Opcode:* **26**

*Byte 2:* ***register***  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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[*register*] is not zero, then the result is infinity
- if reg[A] is infinity, then the result is zero

*See Also:* FDIV, FDIVI, FDIV0, FDIVRI, FDIVR0, FMOD, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

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

*Syntax:* **FDIVRI, *signedByte***

*Description:* The signed byte value is converted to floating point and divided by the value in register A. The result is stored in register A.

$\text{reg[A]} = \text{float}(\text{signedByte}) / \text{reg[A]}$

*Opcode:* **38**

*Byte 2:* ***signedByte***  
A signed byte value (-128 to 127).

*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(*signedByte*) is not zero, then the result is infinity

*See Also:* FDIV, FDIVI, FDIV0, FDIVR, FDIVR0, FMOD, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

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

*Syntax:* **FDIVR0**

*Description:* If register A is 32-bit, 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. If register A is 64-bit, the floating point value in register 128 is divided by the floating point value in register A and the result is stored in register A.

if reg[A] is 32-bit,  $\text{reg[A]} = \text{reg[0]} / \text{reg[A]}$   
if reg[A] is 64-bit,  $\text{reg[A]} = \text{reg[128]} / \text{reg[A]}$

*Opcode:* **2F**

*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 | 128] is not zero, then the result is infinity
- if reg[A] is infinity, then the result is zero

*See Also:* FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FMOD, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

---

## FFT Fast Fourier Transform

*Syntax:* **FFT, action**

*Description:* This instruction performs Fast Fourier Transform (FFT) operations.

*Opcode:* **6F**

*Byte 2:* **action**

The type of action performed is specified by the action byte as follows:

<i>Value</i>	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
0	FIRST_STAGE	0x00	first stage of FFT
1	NEXT_STAGE	0x01	next stage of multistage FFT
2	NEXT_LEVEL	0x02	next level of multistage FFT
3	NEXT_BLOCK	0x03	next block of multistage FFT
+4	BIT_REVERSE	0x04	pre-processing bit reverse sort
+8	PRE_ADJUST	0x08	pre-processing for inverse FFT
+16	POST_ADJUST	0x10	post-processing for inverse FFT

The data for the FFT instruction must be 32-bit floating point values 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 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. If the matrix is stored in registers, the maximum matrix size is 64 points (if all 128 32-bit registers are used). If the matrix is stored in RAM, then maximum matrix size is 256 points.

*See Also:* COPYIND, LOADIND, LOADMA, SAVEIND, SAVEMA, SELECTMA

---

## FINV Floating point inverse

*Syntax:* **FINV**

*Description:* The inverse of the floating point value in register A is stored in register A.

$$\text{reg}[A] = 1 / \text{reg}[A]$$

*Opcode:* **40**

*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

*See Also:* FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0

## **FIX Convert floating point to long integer**

*Syntax:* **FIX**

*Description:* Converts the floating point value in register A to a long integer value.

reg[A] = fix(reg[A])

*Opcode:* **61**

*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)

*See Also:* FIXR, FLOAT, FRAC, FSPLIT

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

*Syntax:* **FIXR**

*Description:* Converts the floating point value in register A to a long integer value with rounding.

reg[A] = fix(round(reg[A]))

*Opcode:* **62**

*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)

*See Also:* FIX, FLOAT, FRAC, FSPLIT

## **FLOAT Convert long integer to floating point**

*Syntax:* **FLOAT**

*Description:* Converts the long integer value in register A to a floating point value.

reg[A] = float(reg[A])

*Opcode:* **60**

*See Also:* **FIX, FIXR, FRAC, FSPLIT**

---

## **FLOOR Floor**

*Syntax:* **FLOOR**

*Description:* 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.

$\text{reg}[A] = \text{floor}(\text{reg}[A])$

*Opcode:* **51**

*Special Cases:*

- if  $\text{reg}[A]$  is NaN, then the result is NaN
- if  $\text{reg}[A]$  is +infinity or -infinity, then the result is +infinity or -infinity
- if  $\text{reg}[A]$  is 0.0 or -0.0, then the result is 0.0 or -0.0

*See Also:* **CEIL, ROUND**

---

## **FMAC Multiply and add**

*Syntax:* **FMAC, register1, register2**

*Description:* The floating point value in *register1* is multiplied by the value in *register2* and the result is added to register A.

$\text{reg}[A] = \text{reg}[A] + (\text{reg}[\text{register1}] * \text{reg}[\text{register2}])$

*Opcode:* **57**

*Byte 2:* **register1**  
Register number (0 to 255).

*Byte 3:* **register2**  
Register number (0 to 255).

*Special Cases:*

- if  $\text{reg}[A]$  is 32-bit and *register1* or *register2* are 64-bit, the values are converted to 32-bit before being used
- if  $\text{reg}[A]$  is 64-bit and *register1* or *register2* are 32-bit, the values are converted to 64-bit before being used
- 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

*See Also:* **FMSC**

---

## **FMAX Floating point maximum**

*Syntax:*           **FMAX, register**

*Description:*     The maximum floating point value of register A and *register* is stored in register A.

$\text{reg}[A] = \max(\text{reg}[A], \text{reg}[\text{register}])$

*Opcode:*           **55**

*Byte 2:*           **register**

Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if either value is NaN, then the result is NaN

*See Also:*          FMIN, LMAX, LMIN

## **FMIN      Floating point minimum**

*Syntax:*           **FMIN, register**

*Description:*     The minimum floating point value of register A and *register* is stored in register A.

$\text{reg}[A] = \min(\text{reg}[A], \text{reg}[\text{register}])$

*Opcode:*           **54**

*Byte 2:*           **register**

Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if either value is NaN, then the result is NaN

*See Also:*          FMAX, LMAX, LMIN

## **FMOD      Floating point remainder**

*Syntax:*           **FMOD, register**

*Description:*     The floating point remainder of the floating point value in register A divided by *register* is stored in register A.

$\text{reg}[A] = \text{remainder of } \text{reg}[A] / (\text{reg}[\text{register}])$

*Opcode:*           **50**

*Byte 2:*           **register**

Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:* FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0

---

## FMSC Multiply and subtract from

*Syntax:* **FMSC, register1, register2**

*Description:* The floating point value in *register1* is multiplied by the value in *register2* and the result is subtracted from register A.

$$\text{reg}[A] = \text{reg}[A] - (\text{reg}[\text{register1}] * \text{reg}[\text{register2}])$$

*Opcode:* **58**

*Byte 2:* **register1**  
Register number (0 to 255).

*Byte 3:* **register2**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register1* or *register2* are 64-bit, the values are converted to 32-bit before being used
- if reg[A] is 64-bit and *register1* or *register2* are 32-bit, the values are converted to 64-bit before being used
- 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

*See Also:* FMAC

---

## FMUL Floating point multiply

*Syntax:* **FMUL, register**

*Description:* The floating point value in register A is multiplied by the value in *register*.

$$\text{reg}[A] = \text{reg}[A] * \text{reg}[\text{register}]$$

*Opcode:* **24**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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

*See Also:* FMULI, FMUL0, LMUL, LMULI, LMUL0

---

## FMULI Floating point multiply by immediate value

*Syntax:* **FMULI, signedByte**

*Description:* The signed byte value is converted to floating point and the value in register A is multiplied by the converted value.

$\text{reg}[A] = \text{reg}[A] * \text{float}[\text{signedByte}]$

*Opcode:* **36**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

*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

*See Also:* FMUL, FMUL0, LMUL, LMULI, LMUL0

---

## FMUL0 Floating point multiply by register 0

*Syntax:* **FMUL0**

*Description:* If register A is 32-bit, the floating point value in register A is multiplied by the value in register 0. If register A is 64-bit, the floating point value in register A is multiplied by the value in register 128. The result is stored in register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] * \text{reg}[0]$   
if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] * \text{reg}[128]$

*Opcode:* **2D**

*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

*See Also:* FMUL, FMULI, LMUL, LMULI, LMUL0

---

## FNEG Floating point negate

*Syntax:* **FNEG**

*Opcode:* **3E**

*Description:*  $\text{reg}[A] = -\text{reg}[A]$

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

*See Also:* FABS, LABS, LNEG

---

## **FPOW**      **Floating point power**

*Syntax:*      **FPOW, register**

*Description:*      The floating point value in register A is raised to the power of the floating point value in *register* and stored in register A.

$\text{reg}[A] = \text{reg}[A] ** \text{reg}[\text{register}]$

*Opcode:*      **27**

*Byte 2:*      **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if reg[*register*] is 0.0 or -0.0, then the result is 1.0
- if reg[*register*] is 1.0, then the result is the same as the A value
- if reg[*register*] is NaN, then the result is Nan
- if reg[A] is NaN and reg[*register*] is nonzero, then the result is NaN
- if  $|\text{reg}[A]| > 1$  and reg[*register*] is +infinite, then the result is +infinity
- if  $|\text{reg}[A]| < 1$  and reg[*register*] is -infinite, then the result is +infinity
- if  $|\text{reg}[A]| > 1$  and reg[*register*] is -infinite, then the result is 0.0
- if  $|\text{reg}[A]| < 1$  and reg[*register*] is +infinite, then the result is 0.0
- if  $|\text{reg}[A]| = 1$  and reg[*register*] is infinite, then the result is NaN
- if reg[A] is 0.0 and reg[*register*] > 0, then the result is 0.0
- if reg[A] is +infinity and reg[*register*] < 0, then the result is 0.0
- if reg[A] is 0.0 and reg[*register*] < 0, then the result is +infinity
- if reg[A] is +infinity and reg[*register*] > 0, then the result is +infinity
- if reg[A] is -0.0 and reg[*register*] > 0 but not a finite odd integer, then the result is 0.0
- if the reg[A] is -infinity and reg[*register*] < 0 but not a finite odd integer, then the result is 0.0
- if reg[A] is -0.0 and the reg[*register*] is a positive finite odd integer, then the result is -0.0
- if reg[A] is -infinity and reg[*register*] is a negative finite odd integer, then the result is -0.0
- if reg[A] is -0.0 and reg[*register*] < 0 but not a finite odd integer, then the result is +infinity
- if reg[A] is -infinity and reg[*register*] > 0 but not a finite odd integer, then the result is +infinity
- if reg[A] is -0.0 and reg[*register*] is a negative finite odd integer, then the result is -infinity
- if reg[A] is -infinity and reg[*register*] is a positive finite odd integer, then the result is -infinity
- if reg[A] < 0 and reg[*register*] is a finite even integer, then the result is equal to  $|\text{reg}[A]|$  to the power of reg[*register*]
- if reg[A] < 0 and reg[*register*] is a finite odd integer, then the result is equal to the negative of  $|\text{reg}[A]|$  to the power of reg[*register*]
- if reg[A] < 0 and finite and reg[*register*] is finite and not an integer, then the result is NaN

*See Also:* FPOWI, FPOW0, EXP, EXP10, LOG, LOG10, ROOT, SQRT

---

## FPOWI Floating point power by immediate value

*Syntax:* **FPOWI, signedByte**

*Description:* The signed byte value is converted to floating point and the floating point value in register A is raised to the power of the converted value and stored in register A.

$\text{reg}[A] = \text{reg}[A] ** \text{float}[\text{signedByte}]$

*Opcode:* **39**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

*Special Cases:*

- if *signedByte* is 0, then the result is 1.0
- if *signedByte* is 1, then the result is the same as the A value
- if *reg[A]* is NaN and *signedByte* is nonzero, then the result is NaN
- if *reg[A]* is 0.0 and *signedByte* > 0, then the result is 0.0
- if *reg[A]* is +infinity and *signedByte* < 0, then the result is 0.0
- if *reg[A]* is 0.0 and *signedByte* < 0, then the result is +infinity
- if *reg[A]* is +infinity and *signedByte* > 0, then the result is +infinity
- if *reg[A]* is -0.0 and *signedByte* > 0 but not an odd integer, then the result is 0.0
- if the *reg[A]* is -infinity and *signedByte* < 0 but not an odd integer, then the result is 0.0
- if *reg[A]* is -0.0 and *signedByte* is a positive odd integer, then the result is -0.0
- if *reg[A]* is -infinity and *signedByte* is a negative odd integer, then the result is -0.0
- if *reg[A]* is -0.0 and *signedByte* < 0 but not an odd integer, then the result is +infinity
- if *reg[A]* is -infinity and *signedByte* > 0 but not an odd integer, then the result is +infinity
- if *reg[A]* is -0.0 and *signedByte* is a negative odd integer, then the result is -infinity
- if *reg[A]* is -infinity and *signedByte* is a positive odd integer, then the result is -infinity
- if *reg[A]* < 0 and *signedByte* is an even integer,  
then the result is equal to |*reg[A]*| to the power of *signedByte*
- if *reg[A]* < 0 and *signedByte* is an odd integer,  
then the result is equal to the negative of |*reg[A]*| to the power of *signedByte*

*See Also:* FPOW, FPOW0, EXP, EXP10, LOG, LOG10, ROOT, SQRT

---

## FPOW0 Floating point power by register 0

*Syntax:* **FPOW0**

*Description:* If register A is 32-bit, 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. If register A is 64-bit, the floating point value in register A is raised to the power of the floating point value in register 128 and stored in register A.

if *reg[A]* is 32-bit,  $\text{reg}[A] = \text{reg}[A] ** \text{reg}[0]$   
if *reg[A]* is 64-bit,  $\text{reg}[A] = \text{reg}[A] ** \text{reg}[128]$

*Opcode:* **30**

- Special Cases:*
- if  $\text{reg}[0 \mid 128]$  is 0.0 or -0.0, then the result is 1.0
  - if  $\text{reg}[0 \mid 128]$  is 1.0, then the result is the same as the A value
  - if  $\text{reg}[0 \mid 128]$  is NaN, then the result is Nan
  - if  $\text{reg}[A]$  is NaN and  $\text{reg}[0 \mid 128]$  is nonzero, then the result is NaN
  - if  $|\text{reg}[A]| > 1$  and  $\text{reg}[0 \mid 128]$  is +infinite, then the result is +infinity
  - if  $|\text{reg}[A]| < 1$  and  $\text{reg}[0 \mid 128]$  is -infinite, then the result is +infinity
  - if  $|\text{reg}[A]| > 1$  and  $\text{reg}[0 \mid 128]$  is -infinite, then the result is 0.0
  - if  $|\text{reg}[A]| < 1$  and  $\text{reg}[0 \mid 128]$  is +infinite, then the result is 0.0
  - if  $|\text{reg}[A]| = 1$  and  $\text{reg}[0 \mid 128]$  is infinite, then the result is NaN
  - if  $\text{reg}[A]$  is 0.0 and  $\text{reg}[0 \mid 128] > 0$ , then the result is 0.0
  - if  $\text{reg}[A]$  is +infinity and  $\text{reg}[0 \mid 128] < 0$ , then the result is 0.0
  - if  $\text{reg}[A]$  is 0.0 and  $\text{reg}[0 \mid 128] < 0$ , then the result is +infinity
  - if  $\text{reg}[A]$  is +infinity and  $\text{reg}[0 \mid 128] > 0$ , then the result is +infinity
  - if  $\text{reg}[A]$  is -0.0 and  $\text{reg}[0 \mid 128] > 0$  but not a finite odd integer, then the result is 0.0
  - if the  $\text{reg}[A]$  is -infinity and  $\text{reg}[0 \mid 128] < 0$  but not a finite odd integer, then the result is 0.0
  - if  $\text{reg}[A]$  is -0.0 and the  $\text{reg}[0 \mid 128]$  is a positive finite odd integer, then the result is -0.0
  - if  $\text{reg}[A]$  is -infinity and  $\text{reg}[0 \mid 128]$  is a negative finite odd integer, then the result is -0.0
  - if  $\text{reg}[A]$  is -0.0 and  $\text{reg}[0 \mid 128] < 0$  but not a finite odd integer, then the result is +infinity
  - if  $\text{reg}[A]$  is -infinity and  $\text{reg}[0 \mid 128] > 0$  but not a finite odd integer, then the result is +infinity
  - if  $\text{reg}[A]$  is -0.0 and  $\text{reg}[0 \mid 128]$  is a negative finite odd integer, then the result is -infinity
  - if  $\text{reg}[A]$  is -infinity and  $\text{reg}[0 \mid 128]$  is a positive finite odd integer, then the result is -infinity
  - if  $\text{reg}[A] < 0$  and  $\text{reg}[0 \mid 128]$  is a finite even integer, then the result is equal to  $|\text{reg}[A]|$  to the power of  $\text{reg}[0 \mid 128]$
  - if  $\text{reg}[A] < 0$  and  $\text{reg}[0 \mid 128]$  is a finite odd integer, then the result is equal to the negative of  $|\text{reg}[A]|$  to the power of  $\text{reg}[0 \mid 128]$
  - if  $\text{reg}[A] < 0$  and finite and  $\text{reg}[0 \mid 128]$  is finite and not an integer, then the result is NaN

*See Also:* FPOW, FPOWI, EXP, EXP10, LOG, LOG10, ROOT, SQRT

---

## FRAC Get fractional part of floating point value

*Syntax:* **FRAC**

*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.

*Opcode:* **63**

*Special Cases:* • if  $\text{reg}[A]$  is NaN or infinity, then the result is NaN

*See Also:* FLOAT, FIX, FIXR, FSPLIT

---

## FREAD Read floating point value

*Syntax:* **FREAD, register**

*Return:* **float32Value**

*Description:* The floating point value of *register* is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

Return 32-bit floating point value from reg[*register*]

*Opcode:* **1A**

*Byte 2:* ***register***  
Register number (0 to 255).

*Return:* ***float32Value***  
Four bytes representing a 32-bit floating point value (MSB first).

*Special Cases:*

- if *register* is 64-bit, the value is converted to 32-bit before being sent.
- if PIC mode is selected, the value is converted to PIC format before being sent.

*See Also:* SETREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND

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

*Syntax:* **FREADA**

*Return:* ***float32Value***

*Description:* 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.

Return 32-bit floating point value from reg[A]

*Opcode:* **1B**

*Return:* ***float32Value***  
Four bytes representing the 32-bit floating point value (MSB first).

*Special Cases:*

- if reg[A] is 64-bit, the value is converted to 32-bit before being sent.
- if PIC mode is selected, the value is converted to PIC format before being sent.

*See Also:* SETREAD, FREAD, FREAD0, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND

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

*Syntax:* **FREADX**

*Return:* ***float32Value***

*Description:* 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.

Return 32-bit value floating point from reg[X], X = X + 1

*Opcode:*       **1C**

*Return:*        ***float32Value***  
Four bytes representing the 32-bit floating point value (MSB first).

*Special Cases:*   • if reg[X] is 64-bit, the value is converted to 32-bit before being sent.  
                      • if PIC mode is selected, the value is converted to PIC format before being sent.

*See Also:*        **SETREAD, FREAD, FREAD0, FREADA, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND**

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

*Syntax:*        **FREAD0**

*Return:*        ***float32Value***

*Description:*    If register A is 32-bit, the floating point value from register 0 is returned. If register A is 64-bit, the floating point value from register 128 is returned. The four bytes of the 32-bit floating point value must be read immediately following this instruction.

if reg[A] is 32-bit, return 32-bit floating point value from reg[0]  
if reg[A] is 64-bit, convert 64-bit value from reg[128] and return 32-bit floating point value

*Opcode:*        **1D**

*Return:*        ***float32Value***  
Four bytes representing the 32-bit floating point value (MSB first).

*Special Cases:*   • if reg[A] is 64-bit, the value is converted to 32-bit before being sent.  
                      • if PIC mode is selected, the value is converted to PIC format before being sent.

*See Also:*        **SETREAD, FREAD, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND**

## **FSET         Set register A**

*Syntax:*        **FSET, *register***

*Description:*    Set register A to the value of *register*.  
  
if reg[A] = reg[*register*]

*Opcode:*        **20**

*Byte 2:*        ***register***  
Register number (0 to 255).

*Special Cases:*   • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
                      • if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:* FSETI, FSET0, LSET, LSETI, LSET0

---

## **FSETI      Set register from immediate value**

*Syntax:*      **FSETI, signedByte**

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

reg[A] = float(signedByte)

*Opcode:*      **32**

*Byte 2:*      **signedByte**  
A signed byte value (-128 to 127).

*See Also:*      FSET, FSET0, LSET, LSETI, LSET0

---

## **FSET0      Set register A from register 0**

*Syntax:*      **FSET0**

*Description:*      If register A is 32-bit, it is set to the value of register 0. If register A is 64-bit, it is set to the value of register 128.

if reg[A] is 32-bit, reg[A] = reg[0]  
if reg[A] is 64-bit, reg[A] = reg[128]

*Opcode:*      **29**

*See Also:*      FSET, FSETI, LSET, LSETI, LSET0

---

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

*Syntax:*      **FSPLIT**

*Description:*      The integer portion of the original value in register A is stored in register A. If register A is 32-bit, the fractional portion of the original value is stored in register 0. If register A is 64-bit, the fractional portion of the original value is stored in register 128. Both values are stored as floating point values.

reg[A] = float(integer portion of reg[A])  
if reg[A] is 32-bit, reg[0] = fractional portion of reg[A]  
if reg[A] is 64-bit, reg[128] = fractional portion of reg[A]

*Opcode:*      **64**

*Special Cases:*      • if reg[A] is NaN or Infinity, reg[A] is set to zero  
                             • if reg[A] is NaN or Infinity, reg[0 | 128] is set to NaN

*See Also:*      **FLOAT, FIX, FIXR, FRAC**

---

## **FSTATUS    Get floating point status**

*Syntax:*      **FSTATUS, register**

*Description:*    Set the internal status byte to the floating point status of the value in *register*. The status byte can be used directly by instructions in user-defined functions, or read by the microcontroller with the READSTATUS instruction. It is set as follows:

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

status = floatstatus(reg[*register*])

*Opcode:*      **3B**

*Byte 2:*      **register**  
Register number (0 to 255).

*See Also:*      **FSTATUSA, LSTATUS, LSTATUSA, READSTATUS**

---

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

*Syntax:*      **FSTATUSA**

*Description:*    Set the internal status byte to the floating point status of the value in register A. The status byte can be used directly by instructions in user-defined functions, or read by the microcontroller with the READSTATUS instruction. It is set as follows:

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

status = floatstatus(reg[A])

*Opcode:*      **3C**

*See Also:*      **FSTATUS, LSTATUS, LSTATUSA, READSTATUS**

---

## **FSUB        Floating point subtract**

*Syntax:*      **FSUB, register**



*Description:* The floating point value in *register* is subtracted from the value in register A.

$\text{reg}[A] = \text{reg}[A] - \text{reg}[\text{register}]$

*Opcode:* **22**

*Byte 2:* ***register***  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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[*register*] is not +infinity, then the result is +infinity
- if reg[A] is -infinity and reg[*register*] is not -infinity, then the result is -infinity
- if reg[A] is not an infinity and reg[*register*] is an infinity, then the result is an infinity of the opposite sign as reg[*register*]

*See Also:* FSUBI, FSUB0, FSUBR, FSUBRI, FSUBR0, LSUB, LSUBI, LSUB0

## **FSUBI Floating point subtract immediate value**

*Syntax:* **FSUBI, *signedByte***

*Description:* The signed byte value is converted to floating point and subtracted from the value in register A.

$\text{reg}[A] = \text{reg}[A] - \text{float}[\text{signedByte}]$

*Opcode:* **34**

*Byte 2:* ***signedByte***  
A signed byte value (-128 to 127).

*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

*See Also:* FSUB, FSUB0, FSUBR, FSUBRI, FSUBR0, LSUB, LSUBI, LSUB0

## **FSUB0 Floating point subtract register 0**

*Syntax:* **FSUB0**

*Description:* If register A is 32-bit, the floating point value in register 0 is subtracted from the value in register A. If register A is 64-bit, the floating point value in register 128 is subtracted from the value in register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] - \text{reg}[0]$   
if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] - \text{reg}[128]$

*Opcode:*           **2B**

*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 | 128] is not +infinity, then the result is +infinity
- if reg[A] is -infinity and reg[0 | 128] is not -infinity, then the result is -infinity
- if reg[A] is not an infinity and reg[0 | 128] is an infinity, then the result is an infinity of the opposite sign as reg[0 | 128]

*See Also:*           FSUB, FSUBI, FSUBR, FSUBRI, FSUBR0, LSUB, LSUBI, LSUB0

## **FSUBR      Floating point subtract (reversed)**

*Syntax:*           **FSUBR, register**

*Description:*      The floating point value in register A is subtracted from the value in *register* and the result is stored in register A.

$\text{reg}[A] = \text{reg}[\text{register}] - \text{reg}[A]$

*Opcode:*           **23**

*Byte 2:*           **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- 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[*register*] is +infinity and reg[A] is not +infinity, then the result is +infinity
- if reg[*register*] is -infinity and reg[A] is not -infinity, then the result is -infinity
- if reg[*register*] is not an infinity and reg[A] is an infinity, then the result is an infinity of the opposite sign as reg[A]

*See Also:*           FSUB, FSUBI, FSUB0, FSUBRI, FSUBR0, LSUB, LSUBI, LSUB0

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

*Syntax:*           **FSUBRI, signedByte**

*Description:*      The signed byte value is converted to floating point and the value in register A is subtracted from the converted value. The result is stored in register A.

$\text{reg}[A] = \text{float}[\text{signedByte}] - \text{reg}[A]$

*Opcode:*           **35**

*Byte 2:*           **signedByte**  
A signed byte value (-128 to 127).

**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

**See Also:** FSUB, FSUBI, FSUB0, FSUBR, FSUBR0, LSUB, LSUBI, LSUB0

## FSUBR0 Floating point subtract register 0 (reversed)

**Syntax:** **FSUBR0**

**Description:** If register A is 32-bit, the floating point value in register A is subtracted from the value in register 0, and the result is stored in register A. If register A is 64-bit, the floating point value in register A is subtracted from the value in register 128, and the result is stored in register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[0] - \text{reg}[A]$   
 if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[128] - \text{reg}[A]$

**Opcode:** **2C**

**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[register] is +infinity and reg[0 | 128] is not +infinity, then the result is +infinity
- if reg[register] is -infinity and reg[A] is not -infinity, then the result is -infinity
- if reg[register] is not an infinity and reg[A] is an infinity, then the result is an infinity of the opposite sign as reg[A]

**See Also:** FSUB, FSUBI, FSUB0, FSUBR, FSUBRI, LSUB, LSUBI, LSUB0

## FTABLE Floating point reverse table lookup

**Syntax:** **FTABLE, conditionCode, tableSize, tableItem1..tableItemN**

**Description:** A reverse table lookup is performed on the floating point value in register A. The value is compared to the values in the 32-bit table using the *conditionCode*. The index number of the first table entry that satisfies the test condition is stored in register 0. If no entry is found, register 0 is unchanged. The index number for the first table entry is zero.

if reg[A] is 32-bit,  $\text{reg}[0] = \text{index of table entry that matches test conditions for reg}[A]$   
 if reg[A] is 64-bit,  $\text{reg}[128] = \text{index of table entry that matches test conditions for reg}[A]$

**Opcode:** **86**

**Byte 2:** **conditionCode**

The list of condition codes is as follows:

IDE Symbol	IDE Value	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

*Byte 3:*        **tableSize**

Specifies the number of 32-bit values in the table (0-255). If *tableSize* is 0, the number of 32-bit values in the table is 256.

*Bytes 4-n:*    **tableItem1..tableItemN**

The number of 32-bit values specified by *tableSize*. Each 32-bit value is represented by four bytes (MSB first).

*Special Cases:*

- only valid inside user-defined functions stored in Flash memory.
- if reg[A] is 64-bit, then the value is converted to 32-bit before being used

*See Also:*        TABLE, LTABLE, POLY

---

## FTOA        Convert floating point value to ASCII string

*Syntax:*        **FTOA, format**

*Description:*    The floating point value in register A is converted to an ASCII string.

*Opcode:*        **1F**

*Byte 2:*        **format**

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 *format* 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 *format* is non-zero, it is interpreted as a decimal number. The hundreds and tens digits specify the length of the converted string (to a maximum of 24), and the ones digit specifies the number of decimal points. 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)

<i>Value in register A</i>	<i>Format byte</i>	<i>Display format</i>
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.

*See Also:*                      STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, LTOA, READSTR, READSEL

---

## **FWRITE      Write floating point value**

*Syntax:*                      **FWRITE, register, float32Value**

*Description:*                      If the PIC data format has been selected (using the PICMODE instruction), the PIC format floating point value is converted to IEEE 754 format. If register is 32-bit, the floating point value is stored in register 0. If register is 64-bit, *float32Value* is converted to 64-bit before being stored in the register.

if *register* is 32-bit, reg[*register*] = 32-bit floating point value  
 if *register* is 64-bit, reg[*register*] = 32-bit value converted to 64-bit floating point

*Opcode:*                      **16**

*Byte 2:*                      **register**  
 Register number (0 to 255).

*Bytes 3 to 6:*                      **float32Value**  
 Four bytes representing a 32-bit floating point value (MSB first).

*Special Cases:*                      • if *register* is 64-bit, the *float32Value* is converted to 64-bit before being stored.  
    • if *register* = 0 or 128, and SETARGS is not active  
        • if reg[A] is 32-bit, the value is stored in registers 0  
        • if reg[A] is 64-bit, the value is stored in registers 128  
    • if *register* = 0 or 128, and SETARGS is active  
        • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
        • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*                      FWRITE0, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEA, LWRITEX, DWRITE, WRIND, SETARGS

---

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

*Syntax:* **FWRITEA, float32Value**

*Description:* If the PIC data format has been selected (using the PICMODE instruction), the PIC format floating point value is converted to IEEE 754 format. If register A is 32-bit, the floating point value is stored in register A. If register A is 64-bit, the 32-bit floating point value is converted to 64-bit before being stored in the register A.

if reg[A] is 32-bit, reg[A] = 32-bit floating point value  
 if reg[A] is 64-bit, reg[A] = 32-bit value converted to 64-bit floating point

*Opcode:* **17**

*Bytes 2 to 5:* **float32Value**  
 Four bytes representing a 32-bit floating point value (MSB first).

*See Also:* FWRITE, FWRITE0, FWRITEX, LWRITE, LWRITE0, LWRITEA, LWRITEX, DWRITE, WRIND

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

*Syntax:* **FWRITEX, float32Value**

*Description:* If the PIC data format has been selected (using the PICMODE instruction), the PIC format floating point value is converted to IEEE 754 format. If register X is 32-bit, the floating point value is stored in register X. If register X is 64-bit, the 32-bit floating point value is converted to 64-bit before being stored in the register X. Register X is incremented to the next register.

if reg[X] is 32-bit, reg[X] = 32-bit floating point value  
 if reg[X] is 64-bit, reg[X] = 32-bit value converted to 64-bit floating point  
 $X = X + 1$

*Opcode:* **18**

*Bytes 2 to 5:* **float32Value**  
 Four bytes representing a 32-bit floating point value (MSB first).

*Special Cases:*

- if reg[X] is 32-bit, it will not increment past register 127
- if reg[X] is 64-bit, it will not increment past register 255

*See Also:* FWRITE, FWRITE0, FWRITEA, LWRITE, LWRITE0, LWRITEA, LWRITEX, DWRITE, WRIND

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

*Syntax:* **FWRITE0, float32Value**

*Description:* If the PIC data format has been selected (using the PICMODE instruction), the PIC format floating point value is converted to IEEE 754 format. If register A is 32-bit, the floating point value is

stored in register 0. If register A is 64-bit, the 32-bit floating point value is converted to 64-bit before being stored in register 128.

if reg[A] is 32-bit, reg[0] = 32-bit floating point value  
 if reg[A] is 64-bit, reg[128] = 32-bit value converted to 64-bit floating point

*Opcode:* **19**

*Bytes 2 to 5:* **float32Value**  
 Four bytes representing a 32-bit floating point value (MSB first).

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* FWRITE, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEA, LWRITEX, DWRITE, WRIND, SETARGS

## GOTO Computed GOTO

*Syntax:* **GOTO, register**

*Description:* This instruction jumps to the address determined by adding the register value to the current function address.

*Opcode:* **89**

*Byte 2:* **register**  
 Register number (0 to 255).

This instruction is only valid in a user-defined function in Flash memory. If the register value is negative, or the new address is outside the address range of the function, a function return occurs.

*See Also:* BRA, BRA,CC, JMP, JMP,CC, RET, RET,CC

## IEEEMODE Select IEEE floating point format

*Syntax:* **IEEEMODE**

*Description:* Selects the IEEE 754 32-bit 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.

*Opcode:* **F4**

*See Also:* PICMODE

## INDA Select A using value in register

<i>Syntax:</i>	<b>INDA, register</b>
<i>Description:</i>	Select register A using the lower 8 bits of the value in <i>register</i> .  A = reg[ <i>register</i> ]
<i>Opcode:</i>	<b>7C</b>
<i>Byte 2:</i>	<b>register</b> Register number (0 to 255).
<i>See Also:</i>	SELECTA, SELECTX, INDX

## INDX      Select X using value in register

<i>Syntax:</i>	<b>INDX, register</b>
<i>Description:</i>	Select register X using the lower 8 bits of the value in <i>register</i> .  X = reg[ <i>register</i> ]
<i>Opcode:</i>	<b>7D</b>
<i>Byte 2:</i>	<b>register</b> Register number (0 to 255).
<i>See Also:</i>	SELECTA, SELECTX, INDA

## JMP      Unconditional jump

<i>Syntax:</i>	<b>JMP, address</b>
<i>Description:</i>	This instruction jumps unconditionally to the instruction at the <i>address</i> specified. If the jump is within -128 to 127 bytes of the address of the next instruction, the <b>BRA</b> instruction can be used.
<i>Opcode:</i>	<b>83</b>
<i>Bytes 2-3:</i>	<b>address</b> An unsigned word value that specifies the address of the next instruction.
<i>Special Cases:</i>	• only valid inside user-defined functions stored in Flash memory.
<i>See Also:</i>	BRA, BRA, cc, GOTO, JMP, cc, RET, RET, cc

## JMP, cc      Conditional jump

<i>Syntax:</i>	<b>JMP, conditionCode, address</b>
----------------	------------------------------------



*Description:* If the condition is true, this instruction jumps to the instruction at the *address* specified. If the condition is false, no jump occurs. If the jump is within -128 to 127 bytes of the address of the next instruction, the **BRA** instruction can be used.

*Opcode:* **84**

*Byte 2:* **conditionCode**

The list of condition codes is as follows:

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
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

*Bytes 3-4:* **address**

An unsigned word value that specifies the address of the next instruction.

*Special Cases:* • only valid inside user-defined functions stored in Flash memory.

*See Also:* BRA, BRA,cc, GOTO, JMP, RET, RET,cc

---

## **LABS Long Integer absolute value**

*Syntax:* **LABS**

*Description:* The absolute value of the long integer value in register A is stored in register A.

$\text{reg}[A] = |\text{reg}[A]|$ , status = longStatus(reg[A])

*Opcode:* **BC**

*See Also:* LNEG, FABS, FNEG

---

**LADD      Long integer add**

*Syntax:*      **LADD, register**

*Description:*      The long integer value in *register* is added to register A.

$\text{reg}[A] = \text{reg}[A] + \text{reg}[\text{register}], \text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:*      **9D**

*Byte 2:*      **register**  
Register number (0 to 255).

*Special Cases:*      • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
• if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:*      LADDI, LADD0, FADD, FADDI, FADD0

---

**LADDI      Long integer add immediate value**

*Syntax:*      **LADDI, signedByte**

*Description:*      The signed byte value is converted to a long integer and added to register A.

$\text{reg}[A] = \text{reg}[A] + \text{long}(\text{signedByte}), \text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:*      **AF**

*Byte 2:*      **signedByte**  
A signed byte value (-128 to 127).

*See Also:*      LADD, LADD0, FADD, FADDI, FADD0

---

**LADD0      Long integer add register 0**

*Syntax:*      **LADD0**

*Description:*      If register A is 32-bit, the long integer value in register 0 is added to register A. If register A is 64-bit, the long integer value in register 128 is added to register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] + \text{reg}[0]$   
if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] + \text{reg}[128]$   
 $\text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:*      **A6**

*See Also:*      LADD, LADDI, FADD, FADDI, FADD0

---

## LAND Long integer AND

**Syntax:** **LAND, register**

**Description:** The bitwise AND of the values in register A and *register* is stored in register A.

$\text{reg[A]} = \text{reg[A]} \text{ AND } \text{reg}[\text{register}], \text{ status} = \text{longStatus}(\text{reg[A]})$

**Opcode:** **C0**

**Byte 2:** **register**  
Register number (0 to 255).

**Special Cases:**

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

**See Also:** LANDI, LBIT, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR

---

## LANDI Long integer AND immediate value

**Syntax:** **LANDI, unsignedByte**

**Description:** The unsigned byte value is converted to a long integer and the bitwise AND of register A and the value is stored in register A.

$\text{reg[A]} = \text{reg[A]} \text{ AND } \text{long}(\text{signedByte}), \text{ status} = \text{longStatus}(\text{reg[A]})$

**Opcode:** **CB**

**Byte 2:** **unsignedByte**  
An unsigned byte value (0 to 255).

**See Also:** LAND, LBIT, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR

---

## LBIT Long integer Bit Clear, Set, Toggle, Test

**Syntax:** **LBIT, bitCode, register**

**Description:** The specified bit in *register* is cleared to zero, set to one, toggled, or tested. The action and bit number are specified by the *bitCode*. The status byte is set according to the state of selected bit after the action has been completed (Z if the bit is zero, NZ if the bit is non-zero).

**Opcode:** **74**

**Byte 2:** **bitCode**

Bit 7	6	5	4	3	2	1	0
Op		Bit Number					

Bits 7:6	Operation	IDE Symbol	IDE Value	Description
----------	-----------	------------	-----------	-------------

	CLEAR	0x00	Clear bit
	SET	0x40	Set bit
	TOGGLE	0x80	Toggle bit
	TEST	0xC0	Test bit
Bits 5:0	<b>Bit Number</b>		
	<i>Value</i>	<i>Description</i>	
	0-63	Bit Number	

*Byte 3:*        **register**  
Register number (0 to 255).

*See Also:*     LAND, LANDI, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR, LTST, LTSTI, LTSTI

---

## LCMP      Long integer compare

*Syntax:*        **LCMP, register**

*Description:*    Compares the signed long integer value in register A with the value in *register* and sets the internal status byte as follows:

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

Bit 1      Sign                    Set if reg[A] < reg[*register*]

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

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

status = longStatus(reg[A] - reg[*register*])

*Opcode:*        **A1**

*Byte 2:*        **register**  
Register number (0 to 255).

The status byte can be read with the **READSTATUS** instruction.

*Special Cases:*    • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
• if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:*        LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2, FCMP, FCMPI, FCMP0, FCMP2

---

## LCMPI      Long integer compare immediate value

*Syntax:*        **LCMPI, signedByte**

*Description:*    status = longStatus(reg[A] - long(*signedByte*))

The signed byte value is converted to long integer and compared to the signed long integer value in register A. The internal status byte is set as follows:

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

Bit 1      Sign                      Set if  $\text{reg}[A] < \text{long}(\text{signedByte})$   
 Bit 0      Zero                      Set if  $\text{reg}[A] = \text{long}(\text{signedByte})$   
    If neither Bit 0 or Bit 1 is set,  $\text{reg}[A] > \text{long}(\text{signedByte})$

*Opcode:*            **B3**

*Byte 2:*            ***signedByte***  
 A signed byte value (-128 to 127).

The status byte can be read with the **READSTATUS** instruction.

*See Also:*            LCMP, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2, FCMP, FCMPI, FCMP0, FCMP2

## LCMP0      Long integer compare register 0

*Syntax:*            **LCMP0**

*Description:*      if  $\text{reg}[A]$  is 32-bit,  $\text{status} = \text{longStatus}(\text{reg}[A] - \text{reg}[0])$   
                          if  $\text{reg}[A]$  is 64-bit,  $\text{status} = \text{longStatus}(\text{reg}[A] - \text{reg}[128])$

If register A is 32-bit, the signed long integer value in register A is compared with the value in register 0, and the internal status byte is set. If register A is 64-bit, the signed long integer value in register A is compared with the value in register 128, and the internal status byte is set as follows:

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

Bit 1      Sign                      Set if  $\text{reg}[A] < \text{reg}[0 \mid 128]$   
 Bit 0      Zero                      Set if  $\text{reg}[A] = \text{reg}[0 \mid 128]$   
    If neither Bit 0 or Bit 1 is set,  $\text{reg}[A] > \text{reg}[0 \mid 128]$

*Opcode:*            **AA**

The status byte can be read with the READSTATUS instruction.

*See Also:*            LCMP, LUCMPI, LCMP2, LUCMP, LUCMPI, LUCMP0, LUCMP2, FCMP, FCMPI, FCMP0, FCMP2

## LCMP2      Long integer compare

*Syntax:*            **LCMP2, register1, register2**

*Description:*      Compares the signed long integer value in *register1* with the value in *register2* and sets the internal status byte as follows:

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

Bit 1	Sign	Set if $\text{reg}[\text{register1}] < \text{reg}[\text{register2}]$
Bit 0	Zero	Set if $\text{reg}[\text{register1}] = \text{reg}[\text{register2}]$
		If neither Bit 0 or Bit 1 is set, $\text{reg}[\text{register1}] > \text{reg}[\text{register2}]$

$\text{status} = \text{longStatus}(\text{reg}[\text{register1}] - \text{reg}[\text{register2}])$

*Opcode:*       **B9**

*Byte 2:*        **register1**  
Register number (0 to 255).

*Byte 3:*        **register2**  
Register number (0 to 255).

The status byte can be read with the **READSTATUS** instruction.

*Special Cases:*   • if *register1* is 32-bit and *register2* is 64-bit, the value is converted to 32-bit before being used  
                      • if *register1* is 64-bit and *register2* is 32-bit, the value is converted to 64-bit before being used

*See Also:*        LCMP, LCMPI, LCMP0, LUCMP, LUCMPI, LUCMP0, LUCMP2, FCMP, FCMPI, FCMP0, FCMP2

## LCOPYI      Copy Immediate value

*Syntax:*        **LCOPYI, signedByte, register**

*Description:*   The 8-bit signed value is converted to a long integer and copied to *register*.

$\text{reg}[\text{register}] = \text{long}(\text{signedByte}), \text{status} = \text{longStatus}(\text{reg}[\text{register}])$

*Opcode:*        **11**

*Byte 2:*        **signedByte**  
An signed byte value (-128 to 127).

*Byte 3:*        **register**  
Register number (0 to 255).

*See Also:*        FCOPYI, COPY0, COPYA, COPYX

## LDEC        Long integer decrement

*Syntax:*        **LDEC, register**

*Description:*   The long integer value in *register* is decremented by one. The long integer status is stored in the status byte.

$\text{reg}[\text{register}] = \text{reg}[\text{register}] - 1, \text{status} = \text{longStatus}(\text{reg}[\text{register}])$

*Opcode:*        **BE**

*Byte 2:*        **register**  
 Register number (0 to 255).

*See Also:*     LINC

## **LDIV       Long integer divide**

*Syntax:*       **LDIV, register**

*Description:*   The long integer value in register A is divided by the signed value in *register*, and the result is stored in register A. If register A is 32-bit, the remainder is stored in register 0. If register A is 64-bit, the remainder is stored in register 128.

$\text{reg}[A] = \text{reg}[A] / \text{reg}[\text{register}]$   
 if reg[A] is 32-bit, reg[0] = remainder  
 if reg[A] is 64-bit, reg[128] = remainder  
 status = longStatus(reg[A])

*Opcode:*       **A0**

*Byte 2:*        **register**  
 Register number (0 to 255).

*Special Cases:*   • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
 • if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used  
 • if reg[*register*] is zero, the result is the largest positive integer  
 (32-bit: \$7FFFFFFF, 64-bit: \$FFFFFFFFFFFFFFFF)

*See Also:*       LDIVI, LDIV0, LUDIV, LUDIVI, LUDIV0, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

## **LDIVI      Long integer divide by immediate value**

*Syntax:*       **LDIVI, signedByte**

*Description:*   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. If register A is 32-bit, the remainder is stored in register 0. If register A is 64-bit, the remainder is stored in register 128.

$\text{reg}[A] = \text{reg}[A] / \text{long}(\text{signedByte})$   
 if reg[A] is 32-bit, reg[0] = remainder  
 if reg[A] is 64-bit, reg[128] = remainder  
 status = longStatus(reg[A])

*Opcode:*       **B2**

*Byte 2:*        **signedByte**  
 A signed byte value (-128 to 127).

*Special Cases:*   • if the signed byte value is zero, the result is the largest positive integer

(32-bit: \$7FFFFFFF, 64-bit:\$7FFFFFFFFFFFFFFF)

*See Also:* LDIV, LDIV0, LUDIV, LUDIVI, LUDIV0, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

---

## **LDIV0      Long integer divide by register 0**

*Syntax:*      **LDIV0**

*Description:*      If register A is 32-bit, the long integer value in register A is divided by the signed long integer value in register 0, and the result is stored in register A with the remainder stored in register 0. If register A is 64-bit, the long integer value in register A is divided by the signed long integer value in register 128, and the result is stored in register A with the remainder stored in register 128.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] / \text{reg}[0]$ ,  $\text{reg}[0] = \text{remainder}$   
 if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] / \text{reg}[128]$ ,  $\text{reg}[128] = \text{remainder}$   
 status = longStatus(reg[A])

*Opcode:*      **A9**

*Special Cases:*      • if reg[0 | 128] is zero, the result is the largest positive integer  
 (32-bit: \$7FFFFFFF, 64-bit:\$7FFFFFFFFFFFFFFF)

*See Also:* LDIV, LDIVI, LUDIV, LUDIVI, LUDIV0, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

---

## **LEFT      Left Parenthesis**

*Syntax:*      **LEFT**

*Description:*      Saves the current registerA and allocates a temporary register as register A.

*Opcode:*      **14**

The LEFT parenthesis 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 parenthesis 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. If register A is 32-bit, the 32-bit temporary registers are used. If register A is 64-bit, the 64-bit temporary registers are used.

*Special Cases:*      • If the maximum number of temporary register is exceeded, reg[A] is set to NaN, and the stack level is reset to zero.

*See Also:* RIGHT, SETARGS

---

## **LINC      Long integer increment**

*Syntax:*      **LINC, register**



*Description:* The long integer value in *register* is incremented by one. The long integer status is stored in the status byte.

$\text{reg}[\text{register}] = \text{reg}[\text{register}] + 1, \text{status} = \text{longStatus}(\text{reg}[\text{register}])$

*Opcode:* **BD**

*Byte 2:* ***register***  
Register number (0 to 255).

*See Also:* LDEC

## **LMAX**      **Long integer maximum**

*Syntax:* **LMAX, *register***

*Description:* The maximum signed long integer value of register A and *register* is stored in register A.

$\text{reg}[A] = \max(\text{reg}[A], \text{reg}[\text{register}]), \text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:* **C5**

*Byte 2:* ***register***  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if either value is NaN, then the result is NaN

*See Also:* LMIN, FMAX, FMIN

## **LMIN**      **Long integer minimum**

*Syntax:* **LMIN, *register***

*Description:* The minimum signed long integer value of register A and *register* is stored in register A.

$\text{reg}[A] = \min(\text{reg}[A], \text{reg}[\text{register}]), \text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:* **C4**

*Byte 2:* ***register***  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if either value is NaN, then the result is NaN

*See Also:* LMAX, FMAX, FMIN

## LMUL Long integer multiply

*Syntax:* **LMUL, register**

*Description:* The long integer value in register A is multiplied by the value in *register*.

$\text{reg}[A] = \text{reg}[A] * \text{reg}[\text{register}]$ , status = longStatus(reg[A])

*Opcode:* **9F**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:* LMULI, LMUL0, FMUL, FMULI, FMUL0

---

## LMULI Long integer multiply by immediate value

*Syntax:* **LMULI, signedByte**

*Description:* The signed byte value is converted to a long integer and register A is multiplied by the converted value.

$\text{reg}[A] = \text{reg}[A] * \text{long}(\text{signedByte})$ , status = longStatus(reg[A])

*Opcode:* **B1**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

*See Also:* LMUL, LMUL0, FMUL, FMULI, FMUL0

---

## LMUL0 Long integer multiply by register 0

*Syntax:* **LMUL0**

*Description:* If register A is 32-bit, the long integer value in register A is multiplied by the value in register 0. If register A is 64-bit, the long integer value in register A is multiplied by the value in register 128.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] * \text{reg}[0]$   
if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] * \text{reg}[128]$   
status = longStatus(reg[A])

*Opcode:* **A8**

*See Also:* LMUL, LMULI, FMUL, FMULI, FMUL0

---

**LNEG      Long integer negate***Syntax:*      **LNEG***Description:*      The negative of the long integer value in register A is stored in register A. $\text{reg}[A] = -\text{reg}[A], \text{status} = \text{longStatus}(\text{reg}[A])$ *Opcode:*      **BB***See Also:*      LABS, FABS, FNEG**LNOT      A = NOT A***Syntax:*      **LNOT***Description:*      The bitwise complement of the value in register A is stored in register A. $\text{reg}[A] = \text{NOT } \text{reg}[A], \text{status} = \text{longStatus}(\text{reg}[A])$ *Opcode:*      **BF***See Also:*      LAND, LANDI, LBIT, LOR, LORI, LSHIFT, LSHIFTI, LXOR**LOAD      Load register 0 with value of register***Syntax:*      **LOAD, register***Description:*      If register A is 32-bit, register 0 is loaded with the value in *register*. If register A is 64-bit, register 128 is loaded with the value in *register*.

if  $\text{reg}[A]$  is 32-bit,  $\text{reg}[0] = \text{reg}[\text{register}]$   
 if  $\text{reg}[A]$  is 64-bit,  $\text{reg}[128] = \text{reg}[\text{register}]$   
 $\text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:*      **0A**

*Byte 2:*      **register**  
 Register number (0 to 255).

*Special Cases:*      • if SETARGS is used  
                              • if  $\text{reg}[A]$  is 32-bit, the value is stored in registers 1 to 9  
                              • if  $\text{reg}[A]$  is 64-bit, the value is stored in registers 129 to 137

*See Also:*      LOADA, LOADX, ALOADX, XSAVE, XSAVEA, SETARGS**LOADA      Load register 0 with the value of register A***Syntax:*      **LOADA**

*Description:* If register A is 32-bit, register 0 is loaded with the value in register A. If register A is 64-bit, register 128 is loaded with the value in register A.

if reg[A] is 32-bit, reg[0] = reg[A], status = longStatus(reg[0])  
if reg[A] is 64-bit, reg[128] = reg[A], status = longStatus(reg[128])

*Opcode:* **0B**

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LOAD, LOADX, ALOADX, XSAVE, XSAVEA

---

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

*Syntax:*            **LOADBYTE, *signedByte***

*Description:*      If register A is 32-bit, register 0 is loaded with the 8-bit signed integer value converted to 32-bit floating point value. If register A is 64-bit, register 128 is loaded with the 8-bit signed integer value converted to 64-bit floating point value.

if reg[A] is 32-bit, reg[0] = float(*signedByte*)  
 if reg[A] is 64-bit, reg[128] = float(*signedByte*)

*Opcode:*            **59**

*Byte 2:*            ***signedByte***  
 A signed byte value (-128 to 127).

*Special Cases:*    • if SETARGS is used  
                          • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                          • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*            LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGBYTE,  
 LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

---

**LOADE          Load register 0 with floating point value of e**

*Syntax:*            **LOADE**

*Description:*      If register A is 32-bit, register 0 is loaded with the floating point value of e (2.7182818). If register A is 64-bit, register 128 is loaded with the floating point value of e (2.718281828459045).

if reg[A] is 32-bit, reg[0] = 2.7182818  
 if reg[A] is 64-bit, reg[128] = 2.718281828459045

*Opcode:*            **5D**

*Special Cases:*    • if SETARGS is used  
                          • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                          • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*            LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADPI, LONGBYTE,  
 LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

---

**LOADIND      Load Indirect**

*Syntax:*            **LOADIND, *register***

*Description:*      If register A is 32-bit, register 0 is loaded with the data value from the indirect pointer specified by *register*. If register A is 64-bit, register 128 is loaded with the data value from the indirect pointer specified by *register*. See the SETIND instruction for a description of pointers.

if reg[A] is 32-bit, reg[0] = data value pointed to by *register*  
 if reg[A] is 64-bit, reg[128] = data value pointed to by *register*

*Opcode:*       **7A**

*Byte 2:*        **register**  
 Register number (0 to 255).

*Special Cases:*   • if reg[A] is 32-bit and the data value pointed to by *register* is 64-bit, the value is converted to 32-bit before being used  
 • if reg[A] is 64-bit and the data value pointed to by *register* is 32-bit, the value is converted to 64-bit before being used

*Special Cases:*   • if SETARGS is active  
                       • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                       • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*        SETIND, ADDIND, WRIND, RDIND, COPYIND, SAVEIND, SETARGS

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

*Syntax:*        **LOADMA, row, column**

*Description:*   Load register 0 with a value from matrix A. Row and column numbers start from 0. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, reg[0] = matrix A [*row*, *column*]  
 if reg[A] is 64-bit, reg[128] = matrix A [*row*, *column*]

*Opcode:*        **68**

*Byte 2:*        **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:*        **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*   • if *row* or *column* is out of range, register 0 is set to NaN.  
 • if reg[A] is 64-bit, the value from the matrix is converted to 32-bit before being stored in register 128

*See Also:*        FFT, MOP, SELECTMA, SELECTMB, SELECTMC, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC

---

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

*Syntax:*            **LOADMA, row, column**

*Description:*    Load register 0 with a value from matrix B. Row and column numbers start from 0. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, reg[0] = matrix B [row, column]  
 if reg[A] is 64-bit, reg[128] = matrix B [row, column]

*Opcode:*            **69**

*Byte 2:*            **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:*            **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*    • if row or column is out of range, register 0 is set to NaN.  
                           • if reg[A] is 64-bit, the value from the matrix is converted to 32-bit before being stored in register 128

*See Also:*            MOP, SELECTMA, SELECTMB, SELECTMC, LOADMA, LOADMC, SAVEMA, SAVEMB, SAVEMC

---

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

*Syntax:*            **LOADMA, row, column**

*Description:*    Load register 0 with a value from matrix C. Row and column numbers start from 0. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, reg[0] = matrix C [row, column]  
 if reg[A] is 64-bit, reg[128] = matrix C [row, column]

*Opcode:*            **6A**

*Byte 2:*            **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:*            **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*

- if *row* or *column* is out of range, register 0 is set to NaN.
- if reg[A] is 64-bit, the value from the matrix is converted to 32-bit before being stored in register 128

*See Also:* MOP, SELECTMA, SELECTMB, SELECTMC, LOADMA, LOADMB, SAVEMA, SAVEMB, SAVEMC

## LOADPI Load register 0 with value of Pi

*Syntax:* **LOADPI**

*Description:* If register A is 32-bit, register 0 is loaded with the floating point value of pi (3.1415927). If register A is 64-bit, register 128 is loaded with the floating point value of pi (3.141592653589793).

if reg[A] is 32-bit, reg[0] = 3.1415927  
 if reg[A] is 64-bit, reg[128] = 3.141592653589793

*Opcode:* **5E**

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LONGBYTE, LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

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

*Syntax:* **LOADUBYTE, unsignedByte**

*Description:* If register A is 32-bit, register 0 is loaded with the 8-bit unsigned integer value converted to 32-bit floating point value. If register A is 64-bit, register 128 is loaded with the 8-bit unsigned integer value converted to 64-bit floating point value.

if reg[A] is 32-bit, reg[0] = float(unsignedByte)  
 if reg[A] is 64-bit, reg[128] = float(unsignedByte)

*Opcode:* **5A**

*Byte 2:* **unsignedByte**  
 An unsigned byte value (0 to 255).

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LOADBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGBYTE, LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS



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

**Syntax:** **LOADUWORD, unsignedWord**

**Description:** If register A is 32-bit, register 0 is loaded with the 16-bit unsigned integer value converted to 32-bit floating point value. If register A is 64-bit, register 128 is loaded with the 16-bit unsigned integer value converted to 64-bit floating point value.

if reg[A] a 32-bit, reg[0] = float(unsignedWord)  
if reg[A] a 64-bit, reg[128] = float(unsignedWord)

**Opcode:** **5C**

**Bytes 2-3:** **unsignedWord**  
An unsigned word value (0 to 65535).

**Special Cases:**

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

**See Also:** LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGBYTE, LONGUBYTE, LONGWORD, LOADIND, SETARGS

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

**Syntax:** **LOADWORD, signedWord**

**Description:** If register A is 32-bit, register 0 is loaded with the 16-bit signed integer value converted to 32-bit floating point value. If register A is 64-bit, register 128 is loaded with the 16-bit signed integer value converted to 64-bit floating point value.

if reg[A] is 32-bit, reg[0] = float(signedWord)  
if reg[A] is 64-bit, reg[128] = float(signedWord)

**Opcode:** **5B**

**Bytes 2-3:** **signedWord**  
A signed word value (-32768 to 32767).

**Special Cases:**

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

**See Also:** LOADBYTE, LOADUBYTE, LOADUWORD, LOADE, LOADPI, LONGBYTE, LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

---

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

*Syntax:*            **LOADX**

*Description:*      If register A is 32-bit, register 0 is loaded with the value in register X. If register A is 64-bit, register 128 is loaded with the value in register X.

if reg[A] is 32-bit,  $\text{reg}[0] = \text{reg}[X]$ ,  $\text{status} = \text{longStatus}(\text{reg}[0])$ ,  $X = X + 1$

if reg[A] is 64-bit,  $\text{reg}[128] = \text{reg}[X]$ ,  $\text{status} = \text{longStatus}(\text{reg}[128])$ ,  $X = X + 1$

*Opcode:*            **0C**

*Special Cases:*

- if reg[X] is 32-bit, it will not increment past register 127
- if reg[X] is 64-bit, it will not increment past register 255
- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*          LOAD, LOADA, ALOADX, XSAVE, XSAVEA, SETARGS

---

**LOG          Logarithm (base e)**

*Syntax:*            **LOG**

*Description:*      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.

$\text{reg}[A] = \log(\text{reg}[A])$

*Opcode:*            **43**

*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

*See Also:*          FPOW, FPOWI, FPOW0, EXP, EXP10, LOG10, ROOT, SQRT

---

**LOG10       Logarithm (base 10)**

*Syntax:*            **LOG10**

*Description:*      Calculates the base 10 logarithm of the floating point value in register A. The result is stored in register A.

$\text{reg}[A] = \log_{10}(\text{reg}[A])$

*Opcode:*            **44**

*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

*See Also:* FPOW, FPOWI, FPOW0, EXP, EXP10, LOG, ROOT, SQRT

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

*Syntax:* **LONGBYTE, signedByte**

*Description:* If register A is 32-bit, the 8-bit signed value is converted to a long integer and stored in register 0. If register A is 64-bit, the 8-bit signed value is converted to a long integer and stored in register 128.

if reg[A] is 32-bit, reg[0] = long(*signedByte*), status = longStatus(reg[0])  
 if reg[A] is 64-bit, reg[128] = long(*signedByte*), status = longStatus(reg[128])

*Opcode:* **C6**

*Byte 2:* **signedByte**  
 A signed byte value (-128 to 127).

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGUBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

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

*Syntax:* **LONGUBYTE, unsignedByte**

*Description:* If register A is 32-bit, the 8-bit unsigned value is converted to a long integer and stored in register 0. If register A is 64-bit, the 8-bit unsigned value is converted to a long integer and stored in register 128.

if reg[A] is 32-bit, reg[0] = long(*unsignedByte*), status = longStatus(reg[0])  
 if reg[A] is 64-bit, reg[128] = long(*unsignedByte*), status = longStatus(reg[128])

*Opcode:* **C7**

*Byte 2:* **unsignedByte**  
 An unsigned byte value (0 to 255).

*Special Cases:*

- if SETARGS is used
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGBYTE, LONGWORD, LONGUWORD, LOADIND, SETARGS

---

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

*Syntax:*           **LONGUWORD, unsignedByte**

*Description:*    If register A is 32-bit, the 16-bit unsigned value is converted to a long integer and stored in register 0. If register A is 64-bit, the 16-bit unsigned value is converted to a long integer and stored in register 128.

if reg[A] is 32-bit,  
                   reg[0] = long(unsigned (*unsignedWord*), status = longStatus(reg[0]))  
 if reg[A] is 64-bit,  
                   reg[128] = long(unsigned (*unsignedWord*), status = longStatus(reg[128]))

*Opcode:*           **C9**

*Bytes 2-3:*        **unsignedWord**  
 An unsigned word value (0 to 65535).

*Special Cases:*   • if SETARGS is used  
                       • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                       • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*         LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI,  
 LONGBYTE, LONGUBYTE, LONGWORD, LOADIND, SETARGS

---

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

*Syntax:*           **LONGWORD, signedByte**

*Description:*    If register A is 32-bit, the 16-bit signed value is converted to a long integer and stored in register 0. If register A is 64-bit, the 16-bit signed value is converted to a long integer and stored in register 128.

if reg[A] is 32-bit, reg[0] = long(signed (*signedWord*), status = longStatus(reg[0]))  
 if reg[A] is 64-bit, reg[128] = long(signed (*signedWord*), status = longStatus(reg[128]))

*Opcode:*           **C8**

*Bytes 2-3:*        **signedWord**  
 A signed word value (-32768 to 32767).

*Special Cases:*   • if SETARGS is used  
                       • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                       • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*         LOADBYTE, LOADUBYTE, LOADWORD, LOADUWORD, LOADE, LOADPI,  
 LONGBYTE, LONGUBYTE, LONGUWORD, LOADIND, SETARGS

---

**LOR          Long integer OR***Syntax:*        **LOR, *register****Description:*    The bitwise OR of the values in register A and *register* is stored in register A.
$$\text{reg}[A] = \text{reg}[A] \text{ OR } \text{reg}[\textit{register}], \text{ status} = \text{longStatus}(\text{reg}[A])$$
*Opcode:*        **C1***Byte 2:*        ***register***  
Register number (0 to 255).*Special Cases:*    • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
• if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used*See Also:*        LAND, LANDI, LBIT, LNOT, LOR, LSHIFT, LSHIFTI, LXOR**LORI        Long integer OR immediate value***Syntax:*        **LORI, *unsignedByte****Description:*    The unsigned byte value is converted to a long integer and the bitwise OR of register A and the value is stored in register A.
$$\text{reg}[A] = \text{reg}[A] \text{ OR } \text{long}(\textit{unsignedByte}), \text{ status} = \text{longStatus}(\text{reg}[A])$$
*Opcode:*        **CC***Byte 2:*        ***unsignedByte***  
An unsigned byte value (0 to 255).*See Also:*        LAND, LANDI, LBIT, LNOT, LOR, LSHIFT, LSHIFTI, LXOR**LREAD      Read long integer value***Syntax:*        **LREAD, *register****Description:*    The long integer value of *register* is returned. The four bytes of the 32-bit value must be read immediately following this instruction.
$$\text{return 32-bit integer value from reg}[\textit{register}]$$
*Opcode:*        **94***Byte 2:*        ***register***  
Register number (0 to 255).*Returns:*        ***int32Value***  
Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:* • if register is 64-bit, the value is converted to 32-bit before being sent.

*See Also:* SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

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

*Syntax:*            **LREADX**

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

return 32-bit integer value from reg[A]

*Opcode:*            **95**

*Returns:*            **int32Value**  
Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:* • if reg[A] is 64-bit, the value is converted to 32-bit before being sent.

*See Also:*            SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

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

*Syntax:*            **LREADBYTE**

*Description:*      The lower 8 bits of register A are returned. The byte containing the 8-bit value must be read immediately following the instruction.

Return 8-bit integer value from reg[A]

*Opcode:*            **98**

*Returns:*            **byteValue**  
One byte representing an 8-bit integer value.

*See Also:*            SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

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

*Syntax:*            **LREADWORD**

*Description:*      Returns the lower 16 bits of register A. The two bytes of the 16-bit value must be read immediately following this instruction.

Return 16-bit integer value from reg[A]

*Opcode:* **99**

*Returns:* **wordValue**  
Two bytes representing a 16-bit integer value (MSB first).

*See Also:* SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

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

*Syntax:* **LREADX**

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

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

*Opcode:* **96**

*Returns:* **int32Value**  
Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:* • if reg[X] is 64-bit, the value is converted to 32-bit before being sent.

*See Also:* SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADBYTE, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

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

*Syntax:* **LREAD0**

*Description:* If register A is 32-bit, the value of register 0 is returned. If register A is 64-bit, the value of register 128 is returned. The four bytes of the 32-bit value must be read immediately following this instruction.

if reg[A] is 32-bit, return 32-bit integer value from reg[0]  
if reg[A] is 64-bit, return 32-bit integer value from reg[128]

*Opcode:* **97**

*Returns:* **int32Value**  
Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:* • if reg[A] is 64-bit, the value from reg[128] is converted to 32-bit before being sent.

*See Also:* SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

## **LSET      Set register A**

*Syntax:*           **LSET, register**

*Description:*     Set register A to the value of *register*.

reg[A] = reg[*register*], status = longStatus(reg[A])

*Opcode:*           **9C**

*Byte 2:*           **register**

Register number (0 to 255).

*Special Cases:*   • if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used  
                          • if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:*         LSETI, LSET0, FSET, FSETI, FSET0

## **LSETI         Set register from immediate value**

*Syntax:*           **LSETI, signedByte**

*Description:*     The *signedByte* is converted to a long integer and stored in register A.

reg[A] = long(*signedByte*), status = longStatus(reg[A])

*Opcode:*           **AE**

*Byte 2:*           **signedByte**

A signed byte value (-128 to 127).

*See Also:*         LSET, LSET0, FSET, FSETI, FSET0

## **LSET0         Set register A from register 0**

*Syntax:*           **LSET0**

*Description:*     If register A is 32-bit, it is set to the value of register 0. If register A is 64-bit, it is set to the value of register 128.

if reg[A] is 32-bit, reg[A] = reg[0]  
 if reg[A] is 64-bit, reg[A] = reg[128]  
 status = longStatus(reg[A])

*Opcode:*           **A5**

*See Also:*         LSET, LSETI, FSET, FSETI, FSET0

## **LSHIFT        Long integer shift**



**Syntax:** **LSHIFT, register**

**Description:** The shift count is specified by the long integer value in *register*. Register A is shifted left or right depending on the shift count. If the shift count is positive, a left shift is performed with the number of bits equal to the shift count. If the shift count is -1 to -63, a logical right shift is performed with the number of bits equal to the absolute value of the shift count. If the shift count is -64 to -128, an arithmetic right shift is performed with the number of bits equal to the absolute value of the shift count - 64.

if  $\text{reg}[\text{register}] > 0$ , then  $\text{reg}[A] = \text{reg}[A]$  shifted left by  $\text{reg}[\text{register}]$  bits  
 $-63 < \text{reg}[\text{register}] < 0$  and , then  $\text{reg}[A] = \text{reg}[A]$  shifted right by  $-\text{reg}[\text{register}]$  bits  
 $-128 < \text{reg}[\text{register}] < -64$ , then  $\text{reg}[A] = \text{reg}[A]$  shifted right by  $-(\text{reg}[\text{register}]+64)$  bits  
 status = longStatus(reg[A])

**Opcode:** **C3**

**Byte 2:** **register**  
 Register number (0 to 255).

**Special Cases:**

- if  $\text{reg}[\text{register}] = 0$  or -64, no shift occurs
- if  $\text{reg}[A]$  is 32-bit and ( $\text{reg}[\text{register}] > 32$  or  $\text{reg}[\text{register}] < -32$ ), then  $\text{reg}[A] = 0$
- if  $\text{reg}[A]$  is 64-bit and ( $\text{reg}[\text{register}] > 64$  or  $\text{reg}[\text{register}] < -64$ ), then  $\text{reg}[A] = 0$

**See Also:** LAND, LANDI, LBIT, LNOT, LOR, LORI, LSHIFTI, LXOR

---

## **LSHIFTI Long integer shift using immediate value**

**Syntax:** **LSHIFTI, signedByte**

**Description:** The shift count is specified by the signed byte value. Register A is shifted left or right depending on the shift count. If the shift count is positive, a left shift is performed with the number of bits equal to the shift count. If the shift count is -1 to -63, a logical right shift is performed with the number of bits equal to the absolute value of the shift count. If the shift count is -64 to -128, an arithmetic right shift is performed with the number of bits equal to the absolute value of the shift count - 64.

*signedByte* > 0, then  $\text{reg}[A] = \text{reg}[A]$  shifted left by *signedByte* bits  
 $-63 < \text{signedByte} < 0$  and , then  $\text{reg}[A] = \text{reg}[A]$  shifted right by  $-\text{signedByte}$  bits  
 $-128 < \text{signedByte} < -64$ , then  $\text{reg}[A] = \text{reg}[A]$  shifted right by  $-(\text{signedByte}+64)$  bits  
 status = longStatus(reg[A])

**Opcode:** **CA**

**Byte 2:** **signedByte**  
 A signed byte value (-128 to 127).

**Special Cases:**

- if *signedByte* = 0 or -64, no shift occurs
- if  $\text{reg}[A]$  is 32-bit and (*signedByte* > 32 or *signedByte* < -32), then  $\text{reg}[A] = 0$
- if  $\text{reg}[A]$  is 64-bit and (*signedByte* > 64 or *signedByte* < -64), then  $\text{reg}[A] = 0$

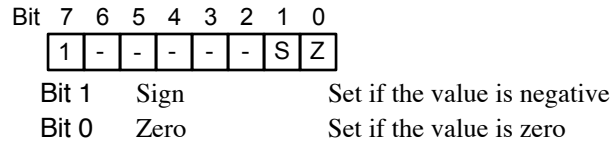
*See Also:* LAND, LANDI, LBIT, LNOT, LOR, LORI, LSHIFT, LXOR

---

## LSTATUS Get long integer status

*Syntax:* **LSTATUS, register**

*Description:* Set the internal status byte to the long integer status of the value in *register*. The status byte can be used directly by instructions in user-defined functions, or read by the microcontroller with the READSTATUS instruction. It is set as follows:



status = longStatus(reg[*register*])

*Opcode:* **B7**

*Byte 2:* **register**  
Register number (0 to 255).

*See Also:* FSTATUS, FSTATUSA, LSTATUSA, READSTATUS

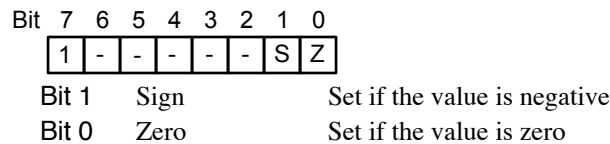
---

## LSTATUSA Get long integer status of register A

*Syntax:* **LSTATUSA**

*Description:* status = longStatus(reg[A])

Set the internal status byte to the long integer status of the value in register A. The status byte can be used directly by instructions in user-defined functions, or read by the microcontroller with the READSTATUS instruction. It is set as follows:



*Opcode:* **B8**

*See Also:* FSTATUS, FSTATUSA, LSTATUS, READSTATUS

---

## LSUB Long integer subtract

*Syntax:* **LSUB, register**

*Description:* The long integer value in *register* is subtracted from register A.

$\text{reg}[A] = \text{reg}[A] - \text{reg}[\text{register}]$ , status = longStatus(reg[A])

*Opcode:* **9E**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and register is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and register is 32-bit, the value is converted to 64-bit before being used

*See Also:* LSUBI, LSUB0, FSUB, FSUBI, FSUB0, FSUBR, FSUBRI, FSUBR0

## **LSUBI Long integer subtract immediate value**

*Syntax:* **LSUBI, signedByte**

*Description:* The signed byte value is converted to a long integer and subtracted from register A.

$\text{reg}[A] = \text{reg}[A] - \text{long}(\text{signedByte})$ , status = longStatus(reg[A])

*Opcode:* **B0**

*Byte 2:* **signedByte**  
A signed byte value (-128 to 127).

*See Also:* LSUB, LSUB0, FSUB, FSUBI, FSUB0, FSUBR, FSUBRI, FSUBR0

## **LSUB0 Long integer subtract register 0**

*Syntax:* **LSUB0**

*Description:* If register A is 32-bit, the long integer value in register 0 is subtracted from register A. If register A is 64-bit, the long integer value in register 128 is subtracted from register A.

if reg[A] is 32-bit,  $\text{reg}[A] = \text{reg}[A] - \text{reg}[0]$   
if reg[A] is 64-bit,  $\text{reg}[A] = \text{reg}[A] - \text{reg}[128]$   
status = longStatus(reg[A])

*Opcode:* **A7**

*See Also:* LSUB, LSUBI, FSUB, FSUBI, FSUB0, FSUBR, FSUBRI, FSUBR0

## **LTABLE Long integer reverse table lookup**

*Syntax:* **LTABLE, conditionCode, tableSize, TableItem1...TableItemN**

*Description:* It performs a reverse table lookup on a long integer value. The value in register A is compared to the values in the 32-bit table using the specified test condition. The index number of the first table entry that satisfied the test condition is stored in register 0. If no entry is found, register 0 is

unchanged. The index number for the first table entry is zero.

if reg[A] is 32-bit,  
 reg[0] = index of table entry that matches test conditions, status = longStatus(reg[0])  
 if reg[A] is 64-bit,  
 reg[128] = index of table entry that matches test conditions, status = longStatus(reg[128])

*Opcode:* **87**

*Byte 2:* ***conditionCode***

The list of condition codes is as follows:

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
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

*Byte 3:* ***tableSize***

Specifies the number of 32-bit values in the table (0-255). If *tableSize* is 0, the number of 32-bit values in the table is 256.

*Bytes 4-n:* ***TableItem1...TableItemN***

32-bit long integer values

*Special Cases:*

- only valid inside user-defined functions stored in Flash memory.
- if reg[A] is 64-bit, then the value is converted to 32-bit before being used.

*See Also:* TABLE, FTABLE, POLY

---

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

*Syntax:* **LTOA, *format***

*Description:* 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, depending on the size of the number. Examples of the converted string are as follows:

```
1
500000
-3598390
```

If the format byte is non-zero, a value between 1 and 24 specifies the length of the converted string. The converted string is right justified. If the format byte is positive, leading spaces are used. If the converted string is longer than the specified length, asterisks are stored. If the length is specified as zero, the string will be as long as necessary to represent the number.

#### ***Leading Zeroes***

If the format byte is negative, its absolute value specifies the length of the converted string, and leading zeros are used.

#### ***Unsigned***

If 100 is added to the format value the value is converted as an unsigned long integer, otherwise it is converted as a signed long integer.

#### ***Hexadecimal***

If the format byte is 40 to 56, the hexadecimal value of the register is stored. The length of the converted string is determined by subtracting 40 from the format byte. (e.g. 41 stores one hexadecimal digit, 42 stores two hexadecimal digits, ...). If the format byte is 40, then the maximum number of hexadecimal digits are stored. The maximum number of hexadecimal digits is 8 for a 32-bit register, and 16 for a 64-bit register.

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

<i>Value in register A</i>	<i>Format byte</i>	<i>Description</i>	<i>Display format</i>
-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
1000	-6	(signed 6, zero fill)	001000

The maximum length of the string is 24. This instruction is usually followed by a READSTR instruction to read the string.

stringbuffer = converted string

*Opcode:* **9B**

*Byte 2:* ***format***

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD,

STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, READSTR, READSEL

## LTST Long integer bit test

**Syntax:** **LTST, register**

**Description:** The internal status byte is set based on the result of a bitwise AND of the value in register A and *register*. The values of register A and *register* are not changed.

status = longStatus(reg[A] AND reg[*register*])

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				

**Opcode:** **A4**

**Byte 2:** **register**  
Register number (0 to 255).

**Special Cases:**

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

**See Also:** LTSTI, LTST0, LBIT, LCMP, LCMPI, LCMP0, LUCMP, LUCMPI, LUCMP0

## LTSTI Long integer bit test using immediate value

**Syntax:** **LTSTI, unsignedByte**

**Description:** The internal status byte is set based on the result of a bitwise AND of the value in register A and the unsigned byte value. The value of register A is not changed.

status = longStatus(reg[A] AND long(unsignedByte))

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				

**Opcode:** **B6**

**Byte 2:** **unsignedByte**  
An unsigned byte value (0 to 255).

**See Also:** LTST, LTST0, LBIT, LCMP, LCMPI, LCMP0, LUCMP, LUCMPI, LUCMP0

---

**LTST0 Long integer bit test register 0**

*Syntax:* **LTST0**

*Description:* If register A is 32-bit, the internal status byte is set based on the result of a bitwise AND of the value in register A and register 0. If register A is 64-bit, the internal status byte is set based on the result of a bitwise AND of the values in register A and register 128. The values of register A and register 0 are not changed.

if reg[A] is 32-bit, status = longStatus(reg[A] AND reg[0])

if reg[A] is 64-bit, status = longStatus(reg[A] AND reg[128])

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					

*Opcode:* **AD**

*See Also:* LTST, LTSTI, LBIT, LCMP, LCMPI, LCMP0, LUCMP, LUCMPI, LUCMP0

---

**LUCMP Unsigned long integer compare**

*Syntax:* **LUCMP, register**

*Description:* Compares the unsigned long integer value in register A with the value in *register* and sets the internal status byte.

status = longStatus(reg[A] - reg[register])

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 $\text{reg}[A] < \text{reg}[\text{register}]$					
Bit 0	Zero		Set if $\text{reg}[A] = \text{reg}[\text{register}]$					
If neither Bit 0 or Bit 1 is set, $\text{reg}[A] > \text{reg}[\text{register}]$								

*Opcode:* **A3**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

*See Also:* LCMP, LCMPI, LCMP0, LCMP2, LUCMPI, LUCMP0, LUCMP2, FCMP, FCMPI,

FCMP0, FCMP2

---

**LUCMPI     Unsigned long integer compare immediate value***Syntax:*     **LUCMPI, *unsignedByte****Description:*     The unsigned byte value is converted to long integer and compared to register A.status = longStatus(reg[A] - long(*unsignedByte*))

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(*unsignedByte*)Bit 0     Zero     Set if reg[A] = long(*unsignedByte*)If neither Bit 0 or Bit 1 is set, reg[A] > long(*unsignedByte*)*Opcode:*     **B5***Byte 2:*     ***unsignedByte***

An unsigned byte value (0 to 255).

*See Also:*     LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMP0, LUCMP2, FCMP, FCMPI, FCMP0, FCMP2

---

**LUCMP0     Unsigned long integer compare register 0***Syntax:*     **LUCMP0***Description:*     If register A is 32-bit register, the unsigned long integer value in register A is compared with the value in register 0, and the internal status byte is set. If register A is 64-bit, the signed long integer value in register A is compared with the value in register 128, and the internal status byte is set.

if reg[A] is 32-bit, status = longStatus(reg[A] - reg[0])

if reg[A] is 64-bit, status = longStatus(reg[A] - reg[128])

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] &lt; reg[0 | 128]

Bit 0     Zero     Set if reg[A] = reg[0 | 128]

If neither Bit 0 or Bit 1 is set, reg[A] &gt; reg[0 | 128]

*Opcode:*     **AC**

---

**LUCMP2     Unsigned long integer compare**



*Syntax:*           **LUCMP2, register1, register2**

*Description:*       Compares the unsigned long integer value in *register1* with the value in *register2* and sets the internal status byte.

$\text{status} = \text{longStatus}(\text{reg}[\text{register1}] - \text{reg}[\text{register2}])$

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  $\text{reg}[\text{register1}] < \text{reg}[\text{register2}]$

Bit 0     Zero                   Set if  $\text{reg}[\text{register1}] = \text{reg}[\text{register2}]$

If neither Bit 0 or Bit 1 is set,  $\text{reg}[\text{register1}] > \text{reg}[\text{register2}]$

*Opcode:*           **BA**

*Byte 2:*           **register1**  
Register number (0 to 255).

*Byte 3:*           **register2**  
Register number (0 to 255).

*Special Cases:*   • if  $\text{reg}[\text{register1}]$  is 32-bit and  $\text{reg}[\text{register2}]$  is 64-bit, the value is converted to 32-bit before being used  
• if  $\text{reg}[\text{register1}]$  is 64-bit and  $\text{reg}[\text{register2}]$  is 32-bit, the value is converted to 64-bit before being used

*See Also:*       LCMP, LCMPI, LCMP0, LCMP2, LUCMP, LUCMPI, LUCMP0, FCMP, FCMPI, FCMP0, FCMP2

## **LUDIV       Unsigned long integer divide**

*Syntax:*           **LUDIV, register**

*Description:*       The long integer value in register A is divided by the unsigned value in *register*, and the result is stored in register A. If register A is 32-bit, the remainder is stored in register 0. If register A is 64-bit, the remainder is stored in register 128.

$\text{reg}[A] = \text{reg}[A] / \text{reg}[\text{register}]$

if  $\text{reg}[A]$  is 32-bit,  $\text{reg}[0] = \text{remainder}$

if  $\text{reg}[A]$  is 64-bit,  $\text{reg}[128] = \text{remainder}$

$\text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:*           **A2**

*Byte 2:*           **register**  
Register number (0 to 255).

*Special Cases:*

- if reg[A] is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if reg[A] is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used
- if reg[*register*] is zero, the result is the largest unsigned integer (32-bit: \$FFFFFFFF, 64-bit: \$FFFFFFFFFFFFFFFF)

*See Also:* LDIV, LDIVI, LDIV0, LUDIVI, LUDIV0, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

## LUDIVI Unsigned long integer divide by immediate value

*Syntax:* **LUDIVI, unsignedByte**

*Description:* 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.

reg[A] = reg[A] / long(unsignedByte), status = longStatus(reg[A])  
 if reg[A] is 32-bit, reg[0] = remainder  
 if reg[A] is 64-bit, reg[128] = remainder

*Opcode:* **B4**

*Byte 2:* **unsignedByte**  
 An unsigned byte value (0 to 255).

*Special Cases:*

- if *unsignedByte* is zero, the result is the largest unsigned integer (32-bit: \$FFFFFFFF, 64-bit: \$FFFFFFFFFFFFFFFF)

*See Also:* LDIV, LDIVI, LDIV0, LUDIV, LUDIV0, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

## LUDIV0 Unsigned long integer divide by register 0

*Syntax:* **LUDIV0**

*Description:* If register A is 32-bit, the long integer value in register A is divided by the unsigned long integer value in register 0, and the result is stored in register A with the remainder stored in register 0. If register A is 64-bit, the long integer value in register A is divided by the unsigned long integer value in register 128, and the result is stored in register A with the remainder stored in register 128.

if reg[A] is 32-bit, reg[A] = reg[A] / reg[0], reg[0] = remainder  
 if reg[A] is 64-bit, reg[A] = reg[A] / reg[128], reg[128] = remainder  
 status = longStatus(reg[A])

*Opcode:* **AB**

*Special Cases:*

- if reg[0 | 128] is zero, the result is the largest unsigned integer (32-bit: \$FFFFFFFF, 64-bit: \$FFFFFFFFFFFFFFFF)

*See Also:* LDIV, LDIVI, LDIV0, LUDIV, LUDIVI, FDIV, FDIVI, FDIV0, FDIVR,

FDIVRI, FDIVR0, FMOD

---

**LWRITE      Write long integer value****Syntax:**      **LWRITE, register, int32Value****Description:**      The long integer value is stored in *register*. If *register* is 64-bit, *int32Value* is converted to 64-bit before being stored in the register.
$$\text{reg}[\text{register}] = 32\text{-bit long integer value, status} = \text{longStatus}(\text{reg}[\text{register}])$$
**Opcode:**      **90****Byte 2:**      **register**  
Register number (0 to 255).**Bytes 3 to 6:**      **int32Value**  
Four bytes representing a 32-bit integer value (MSB first).**Special Cases:**

- if *register* is 64-bit, the value is converted to 64-bit before being stored.
- if *register* = 0 or 128, and SETARGS is not active
  - if reg[A] is 32-bit, the value is stored in registers 0
  - if reg[A] is 64-bit, the value is stored in registers 128
- if *register* = 0 or 128, and SETARGS is active
  - if reg[A] is 32-bit, the value is stored in registers 1 to 9
  - if reg[A] is 64-bit, the value is stored in registers 129 to 137

**See Also:**      FWRITE, FWRITE0, FWRITEA, FWRITEX, LWRITE0, LWRITEA, LWRITEX, DWRITE, WRIND, SETARGS

---

**LWRITEA      Write long integer value to register A****Syntax:**      **LWRITEA, int32Value****Description:**      The long integer value is stored in register A.
$$\text{reg}[\text{A}] = 32\text{-bit long integer value, status} = \text{longStatus}(\text{reg}[\text{A}])$$
**Opcode:**      **91****Bytes 2 to 5:**      **int32Value**  
Four bytes representing a 32-bit integer value (MSB first).**Special Cases:**

- if reg[A] is 64-bit, the value is converted to 64-bit before being stored.

**See Also:**      FWRITE, FWRITE0, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEX, DWRITE, WRIND

---

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

*Syntax:*            **LWRTX, int32Value**

*Description:*     The long integer value is stored in register X.

reg[X] = 32-bit long integer value, status = longStatus(reg[X]), X = X + 1

*Opcode:*            **92**

*Bytes 2 to 5:*     **int32Value**

Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:*   • if reg[X] is 64-bit, the value is converted to 64-bit before being stored.

*See Also:*          FWRITE, FWRITE0, FWRITEA, FWRTEX, LWRITE, LWRITE0, LWRITEA, DWRITE, WRIND

---

**LWRITE0     Write long integer value to register0**

*Syntax:*            **LWRITE0, int32Value**

*Description:*     If register A is 32-bit, the long integer value is stored in register 0. If register A is 64-bit, the long integer value is stored in register 128.

if reg[A] is 32-bit, reg[0] = 32-bit long integer value,  
                                 status = longStatus(reg[0])

if reg[A] is 64-bit, reg[128] = 32-bit value converted to 64-bit floating point,  
                                 status = longStatus(reg[128])

*Opcode:*            **93**

*Bytes 2 to 5:*     **int32Value**

Four bytes representing a 32-bit integer value (MSB first).

*Special Cases:*   • if reg[A] is 64-bit, the value is converted to 64-bit before being stored.  
                         • if SETARGS is used  
                             • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
                             • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:*          FWRITE, FWRITE0, FWRITEA, FWRTEX, LWRITE, LWRITEA, LWRTEX, DWRITE, WRIND

---

**LXOR         Long integer XOR**

*Syntax:*            **LXOR, register**

*Description:*     The bitwise XOR of the values in register A and *register* is stored in register A.

$\text{reg}[A] = \text{reg}[A] \text{ XOR } \text{reg}[\text{register}], \text{status} = \text{longStatus}(\text{reg}[A])$

*Opcode:* **C2**

*Byte 2:* **register**  
Register number (0 to 255).

*Special Cases:*

- if  $\text{reg}[A]$  is 32-bit and *register* is 64-bit, the value is converted to 32-bit before being used
- if  $\text{reg}[A]$  is 64-bit and *register* is 32-bit, the value is converted to 64-bit before being used

---

## MOP      Matrix Operation

*Syntax:* **MOP, action {,byteCount, byte, ...}**

*Description:* Performs matrix operations on 32-bit floating point values. The matrices are stored in 32-bit registers or RAM. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

MOP, SCALAR\_SET  
 MOP, SCALAR\_ADD  
 MOP, SCALAR\_SUB  
 MOP, SCALAR\_SUBR  
 MOP, SCALAR\_MUL  
 MOP, SCALAR\_DIV  
 MOP, SCALAR\_DIVR  
 MOP, SCALAR\_POW  
 MOP, EWISE\_SET  
 MOP, EWISE\_ADD  
 MOP, EWISE\_SUB  
 MOP, EWISE\_SUBR  
 MOP, EWISE\_MUL  
 MOP, EWISE\_DIV  
 MOP, EWISE\_DIVR  
 MOP, EWISE\_POW  
 MOP, MULTIPLY  
 MOP, IDENTITY  
 MOP, DIAGONAL  
 MOP, TRANSPOSE  
 MOP, COUNT  
 MOP, SUM  
 MOP, AVE  
 MOP, MIN  
 MOP, MAX  
 MOP, COPY\_AB  
 MOP, COPY\_AC  
 MOP, COPY\_BA  
 MOP, COPY\_BC  
 MOP, COPY\_CA  
 MOP, COPY\_CB  
 MOP, DETERM  
 MOP, INVERSE  
 MOP, LOAD\_RA, byteCount, byte, ...  
 MOP, LOAD\_RB, byteCount, byte, ...

MOP, LOAD\_RC, *byteCount*, *byte*, ...  
 MOP, LOAD\_BA, *byteCount*, *byte*, ...  
 MOP, LOAD\_CA, *byteCount*, *byte*, ...  
 MOP, SAVE\_AR, *byteCount*, *byte*, ...  
 MOP, SAVE\_AB, *byteCount*, *byte*, ...  
 MOP, SAVE\_AC, *byteCount*, *byte*, ...  
 MOP, LU\_DECOMP  
 MOP, LU\_INVERSE  
 MOP, LU\_DETERM  
 MOP, LU\_SOLVE  
 MOP, CH\_DECOMP  
 MOP, CH\_INVERSE  
 MOP, CH\_DETERM  
 MOP, CH\_SOLVE

*Opcode:*       **6E**

*Byte 2:*       ***matrixOperation***

*Byte 3:*       ***byteCount***  
 The value specifies the number of bytes to follow.

*Bytes 4-n:*   ***byte*, ...**  
 A list of byte values.  
 These operations can be used to quickly load matrices, save results, or to extract and save matrix subsets.

*Details:*       The *action* selects one of the following operations:

<i>Value</i>	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
--------------	-------------------	------------------	--------------------

**Scalar Set (x00)**

MOP, SCALAR\_SET  
 For each element:  $MA[r,c] = reg[0]$

**Scalar Add (x01)**

MOP, SCALAR\_ADD  
 For each element:  $MA[r,c] = MA[r,c] + reg[0]$

**Scalar Subtract (x02)**

MOP, SCALAR\_SUB  
 For each element:  $MA[r,c] = MA[r,c] - reg[0]$

**Scalar Subtract (reverse) (x03)**

MOP, SCALAR\_SUBR  
 For each element:  $MA[r,c] = reg[0] - MA[r,c]$

**Scalar Multiply (x04)**

MOP, SCALAR\_MUL  
 For each element:  $MA[r,c] = MA[r,c] * reg[0]$

**Scalar Divide (x05)**

**MOP, SCALAR\_DIV**  
 For each element:  $MA[r,c] = MA[r,c] / reg[0]$

#### Scalar Divide (reverse) (x06)

**MOP, SCALAR\_DIVR**  
 For each element:  $MA[r,c] = reg[0] / MA[r,c]$

#### Scalar Power (x07)

**MOP, SCALAR\_POW**  
 For each element:  $MA[r,c] = MA[r,c] ** reg[0]$

#### Element-wise Set (x08)

**MOP, EWISE\_SET**  
 For each element:  $MA[r,c] = MB[r,c]$

#### Element-wise Add (x09)

**MOP, EWISE\_ADD**  
 For each element:  $MA[r,c] = MA[r,c] + MB[r,c]$

#### Element-wise Subtract (x0A)

**MOP, EWISE\_SUB**  
 For each element:  $MA[r,c] = MA[r,c] - MB[r,c]$

#### Element-wise Subtract (reverse) (x0B)

**MOP, EWISE\_SUBR**  
 For each element:  $MA[r,c] = MB[r,c] - MA[r,c]$

#### Element-wise Multiply (x0C)

**MOP, EWISE\_MUL**  
 For each element:  $MA[r,c] = MA[r,c] * MB[r,c]$

#### Element-wise Divide (x0D)

**MOP, EWISE\_DIV**  
 For each element:  $MA[r,c] = MA[r,c] / MB[r,c]$

#### Element-wise Divide (reverse) (x0E)

**MOP, EWISE\_DIVR**  
 For each element:  $MA[r,c] = MB[r,c] / MA[r,c]$

#### Element-wise Power (x0F)

**MOP, EWISE\_POW**  
 For each element:  $MA[r,c] = MA[r,c] ** MB[r,c]$

#### Matrix Multiply (x10)

**MOP, MULTIPLY**  
 Calculate:  $MA = MB * MC$

#### Identity Matrix (x11)

**MOP, IDENTITY**  
 MA = identity matrix

#### Diagonal Matrix (x12)

MOP, DIAGONAL  
MA = diagonal matrix

**Transpose (x13)**

MOP, TRANSPOSE  
MA = transpose MB

**Count (x14)**

MOP, COUNT  
reg[0] = count of all elements in MA

**Sum (x15)**

MOP, SUM  
reg[0] = sum of all elements in MA

**Average (x16)**

MOP, AVE  
reg[0] = average of all elements in MA

**Minimum (x17)**

MOP, MIN  
reg[0] = minimum of all elements in MA

**Maximum (x18)**

MOP, MAX  
reg[0] = maximum of all elements in MA

**Copy Matrix A to Matrix B (x19)**

MOP, COPY\_AB  
Matrix B is set to a copy of matrix A.

**Copy Matrix A to Matrix C (x1A)**

MOP, COPY\_AC  
Matrix C is set to a copy of matrix A.

**Copy Matrix B to Matrix A (x1B)**

MOP, COPY\_BA  
Matrix A is set to a copy of matrix B.

**Copy Matrix B to Matrix C (x1C)**

MOP, COPY\_BC  
Matrix C is set to a copy of matrix B.

**Copy Matrix C to Matrix A (x1D)**

MOP, COPY\_CA  
Matrix A is set to a copy of matrix C.

**Copy Matrix C to Matrix B (x1E)**

MOP, COPY\_CB  
Matrix B is set to a copy of matrix C.

**Matrix Determinant (x1F)**



**MOP, DETERM**

reg[0] = determinant of MA

This operation is only implemented for 2x2 and 3x3 matrices. To calculate the determinant of larger matrices use the LU decomposition or Cholesky decomposition matrix operations.

### **Matrix Inverse (x20)**

**MOP, INVERSE**

MA = inverse of MA

This operation is only implemented for 2x2 and 3x3 matrices. To calculate the inverse of larger matrices use the LU decomposition or Cholesky decomposition matrix operations.

### **Load Matrix from Registers (x21, 0x22, 0x23)**

**MOP, LOAD\_RA, byteCount, byte, ...**

Load matrix A from registers.

**MOP, LOAD\_RB, byteCount, byte, ...**

Load matrix B from registers.

**MOP, LOAD\_RC, byteCount, byte, ...**

Load matrix C from registers.

The load register operations take a list of register numbers and sequentially copy the indexed register values to the matrix specified. Register 0 is cleared to zero before the indexed values are copied, to provide an easy way to load zero values to a matrix. If an index is negative, the absolute value is used as an index, and the negative value of the indexed register is copied.

### **Load Matrix to Matrix (x24, 0x25)**

**MOP, LOAD\_BA, byteCount, byte, ...**

Load matrix A from matrix B.

**MOP, LOAD\_CA, byteCount, byte, ...**

Load matrix A from matrix C.

The load matrix operations take a list of matrix indices and sequentially copy the indexed matrix values to Matrix A. If an index value is negative, the absolute value is used as an index, and the negative value of the indexed value is copied. An index of 0x80 is used to copy the negative of the value at index 0.

### **Save Matrix A to Registers (x26)**

**MOP, SAVE\_AR, byteCount, byte, ...**

This matrix operation takes a list of register numbers and sequentially copies the values from matrix A to the specified registers. If an index value is negative, the matrix A value for that index position is not stored.

### **Save Matrix A to Matrix B (x27)**

**MOP, SAVE\_AB, byteCount, byte, ...**

This matrix operation take a list of matrix indices and sequentially copies the values from matrix A to matrix B. If an index value is negative, the matrix A value for that index position is not stored.

### **Save Matrix A to Matrix C (0x28)**

`MOP, SAVE_AC, byteCount, byte, ...`

This matrix operation take a list of matrix indices and sequentially copies the values from matrix A to matrix C. If an index value is negative, the matrix A value for that index position is not stored.

### **LU Decomposition (0x29)**

`MOP, LU_DECOMP`

MC  $n \times n$  matrix is augmented to  $n+2 \times n$  and the LU decomposition of the original  $n \times n$  matrix is stored in MC.

### **LU Matrix Inverse (0x2A)**

`MOP, LU_INVERSE`

MA = inverse of original MC  $n \times n$  matrix

The `MOP, LU_DECOMP` operation must be done before the `MOP, LU_INVERSE` operation.

### **LU Matrix Determinant (0x2B)**

`MOP, LU_DETERM`

reg[0] = determinant of original  $n \times n$  MC matrix

The `MOP, LU_DECOMP` operation must be done before the `MOP, LU_DETERM` operation.

### **LU Matrix Solve (0x2C)**

`MOP, LU_SOLVE`

The LU decomposition matrix stored in MC is used to solve the set of  $n$  linear equations. The input vector is stored in row  $n$  of the augmented MC matrix, and the solution vector is returned in row  $n$  of the augmented MC matrix. The `MOP, LU_DECOMP` operation must be done before the first `MOP, LU_SOLVE` operation. Multiple `MOP, LU_SOLVE` operations can be done without repeating the first `MOP, LU_DECOMP` operation.

### **Cholesky Decomposition (0x2D)**

`MOP, CH_DECOMP`

MC  $n \times n$  matrix is augmented to  $n+2 \times n$  and the Cholesky decomposition of the original  $n \times n$  matrix is stored in MC.

### **Cholesky Matrix Inverse (0x2E)**

`MOP, CH_INVERSE`

The inverse of the original MC  $n \times n$  matrix is stored in the first  $n \times n$  elements of MC. The `MOP, CH_DECOMP` operation must be done before the `MOP, CH_INVERSE` operation. The inverse matrix overwrites the Cholesky decomposition matrix.

### **Cholesky Matrix Determinant (0x2F)**

`MOP, CH_DETERM`

reg[0] = determinant of original  $n \times n$  MC matrix

The `MOP, CH_DECOMP` operation must be done before the `MOP, CH_DETERM` operation.

### **Cholesky Matrix Solve (0x30)**

`MOP, CH_SOLVE`

The LU decomposition matrix stored in MC is used to solve the set of  $n$  linear equations. The input vector is stored in row  $n$  of the augmented MC matrix, and the solution vector is returned in row  $n$  of the augmented MC matrix. The `MOP, CH_DECOMP` operation must be done before the first `MOP, CH_SOLVE` operation. Multiple `MOP, CH_SOLVE` operations can be done without repeating the first `MOP, CH_DECOMP` operation.

- Special Cases:*
- matrix operations are restricted to 32-bit floating point.
  - indirect pointers must be used to select matrices in RAM.
  - in a background process, a matrix that starts at register 0 to 15 must not extend beyond register 15.
  - in a background process, larger matrices should be stored using registers 16 to 127, or RAM.

*See Also:* SELECTMA, SELECTMB, SELECTMC, LOADMA, LOADMB, LOADMC, SAVEMA, SAVEMB, SAVEMC

---

## **NOP      No operation**

*Syntax:*      **NOP**

*Description:*      No operation.

*Opcode:*      **00**

---

## **PICMODE      Select PIC floating point format**

*Syntax:*      **PICMODE**

*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 32-bit floating point data in the alternate format. Normally this instruction would be issued immediately after the reset as part of the initialization code. The IEEEEMODE instruction can be used to revert to standard IEEE 754 32-bit floating point mode.

*Opcode:*      **F5**

*See Also:*      IEEEEMODE

---

## **POLY      A = nth order polynomial**

*Syntax:*      **POLY, count, float32Value1...float32ValueN**

*Description:*      This instruction is only valid in a user-defined function in Flash memory. The value of the specified polynomial is calculated and stored in register A. The general form of the polynomial is:

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

The value of x is the initial value of register A. An  $n^{\text{th}}$  order polynomial will have n+1 coefficients stored in the table. The coefficient values  $A_0, A_1, A_2, \dots$  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.

reg[A] = result of nth order polynomial calculation

*Opcode:*      **88**

**Byte 2:** **count**  
The number of 32-bit floating point values that follow.

**Bytes 3-n:** **float32Value1...float32ValueN**  
Each 32-bit floating point value is represented by four bytes (MSB first).

*Assembler Example:*

POLY, 2 ; polynomial  $3x + 5$   
 #float 3.0  
 #float 5.0

*Compiler Example:*

```
value = POLY(x, 3.0, 5.0)      ; value = 3x + 5
value = POLY(x, 1, 0, 0, 1)    ; value = x3 + 1
```

*Special Cases:*

- only valid inside user-defined functions stored in Flash memory.
- if `reg[A]` is 64-bit, then the value is converted to 32-bit before being used, and the result is converted to 64-bit before being stored.

*See Also:*      TABLE, FTABLE, LTABLE

**RADIANS**      Convert degrees to radians

*Syntax:* **RADIANS**

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

```
reg[A] = radians(reg[A])
```

*Opcode:*           **4F**

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

*See Also:* ACOS, ASIN, ATAN, ATAN2, COS, SIN, TAN, DEGREES

<b>RDIND</b>	<b>Read data using indirect pointer</b>
--------------	---

**Syntax:** **RDIND, dataType, pointer, count**

**Description:** Read *count* data values of the specified *dataType* from the *pointer* location. If *count* = 0, then the count is loaded from the lower 16 bits of register 0. The pointer can be a register pointer or a

*Opcode:* **EC**

*Returns:* **byte1...byteN**

A zero-terminated string.

*See Also:* SETREAD, STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR

## READSTATUS Return the last status byte

*Syntax:* **READSTATUS**

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

*Opcode:* **F1**

*Returns:* **status**

The status byte.

*See Also:* SETREAD, FSTATUS, FSTATUSA, LSTATUS, LSTATUSA, SETSTATUS

## READSTR Read string

*Syntax:* **READSTR**

*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.

*Opcode:* **F2**

*Returns:* **byte1...byteN**

A zero-terminated string.

*See Also:* SETREAD, STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSEL

## READVAR Read internal variable

*Syntax:* **READVAR, item**

*Description:* Sets register 0 to the internal value selected by *item*.

- |   |                  |
|---|------------------|
| 0 | Register A       |
| 1 | Register X       |
| 2 | Matrix A pointer |
| 3 | Matrix A rows    |
| 4 | Matrix A columns |
| 5 | Matrix B pointer |

6	Matrix B rows
7	Matrix B columns
8	Matrix C pointer
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
19	Silicon revision number
20	Device type (0x11 for 28-pin device, 0x12 for 44-pin device)
21	Number of Functions loaded in Flash
22	Number of XOP Instructions loaded in Flash
23	Error Status

if reg[A] is 32-bit, reg[0] = internal register value, status = longStatus(reg[0])  
if reg[A] is 64-bit, reg[128] = internal register value, status = longStatus(reg[128])

*Opcode:* **FC**

*Byte 2:* **item**  
Selects the internal value to load into register 0.

---

## RESET Reset

*Syntax:* **RESET**

*Description:* Nine consecutive 0xFF bytes will cause the uM-FPU to reset. If less than nine consecutive 0xFF bytes are received, they are treated as NOPs.

*Opcode:* **FF**

---

## RET Return from user-defined function

*Syntax:* **RET**

*Description:* This instruction unconditionally returns from the current function. It restores the register A selection to the value stored by FCALL. This instruction is only valid in user-defined function stored in Flash memory.

*Opcode:* **80**

*Special Cases:* • only valid inside user-defined functions stored in Flash memory.

*See Also:* FCALL, BRA, BRA,cc, GOTO, JMP, JMP,cc, RET, RET,cc

---

## RET, cc      Conditional return from user-defined function

*Syntax:*            **RET, conditionCode**

*Description:*      If the condition is true, this instruction returns from the current function. If the condition is false, no return occurs. It restores the register A selection to the value stored by FCALL. This instruction is only valid in user-defined function stored in Flash memory.

*Opcode:*            **8A**

*Byte 2:*            **conditionCode**  
The list of condition codes is as follows:

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
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

This instruction is only valid in a user-defined function in Flash memory.

*Special Cases:*    • only valid inside user-defined functions stored in Flash memory.

*See Also:*            FCALL, BRA, BRA,cc, GOTO, JMP, JMP,cc, RET, RET,cc

---

## RIGHT      Right Parenthesis

*Syntax:*            **RIGHT**

*Description:*      If register A is 32-bit, the value of register A is loaded to register 0. If register A is 64-bit, the value of register A is loaded to register 128. 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. This is 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.



*Opcode:* **15**

*Special Cases:* • if no left parenthesis is currently outstanding, then register 0 (32-bit) or register 128 (64-bit) is set to NaN.

*Special Cases:* • if SETARGS is used  
 • if reg[A] is 32-bit, the value is stored in registers 1 to 9  
 • if reg[A] is 64-bit, the value is stored in registers 129 to 137

*See Also:* LEFT, SETARGS

## ROOT Calculate n<sup>th</sup> root

*Syntax:* **ROOT, register**

*Description:* Calculates the n<sup>th</sup> root of the floating point value in register A and stores the result in register A. It is equivalent to raising register A to the power of (1 / n), where n is the floating point value in register.

$\text{reg}[A] = \text{reg}[A] ** (1 / \text{reg}[\text{register}])$

*Opcode:* **42**

*Byte 2:* **register**  
 Register number (0 to 255).

*Special Cases:* • if reg[A] is 32-bit and register is 64-bit, the value is converted to 32-bit before being used  
 • if reg[A] is 64-bit and register is 32-bit, the value is converted to 64-bit before being used  
 • see the description of the POWER instruction for the special cases of (1/reg[register])  
 • if reg[register] is infinity, then (1 / reg[register]) is zero  
 • if reg[register] is zero, then (1 / reg[register]) is infinity

*See Also:* FPOW, FPOWI, FPOW0, EXP, EXP10, LOG, LOG10, SQRT

## ROUND Floating point Rounding

*Syntax:* **ROUND**

*Description:* The floating point value equal to the nearest integer to the floating point value in register A is stored in register A.

$\text{reg}[A] = \text{round}(\text{reg}[A])$

*Opcode:* **53**

*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

*See Also:* CEIL, FLOOR

## RTC Real-time Clock

*Syntax:* **RTC,action {,register}**

*Description:* Used to manage the real-time clock.

*Opcode:* **DC**

*Byte 2:* **action**

Bit	7	6	5	4	3	2	1	0
	Action				Options			

### Bits 7:4 Action

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
INIT	0x00	Initialize the real-time clock mode.
START	0x10	Start real-time clock.
STOP	0x20	Stop real-time clock.
ALARM_MASK	0x30	Set alarm mask.
WRITE_TIME	0x40	Write real-time clock date/time value.
WRITE_ALARM	0x50	Write alarm data/time value.
READ_TIME	0x60	Read real-time clock date/time value.
READ_ALARM	0x70	Read alarm date/time value.
NUM_TO_STR	0x80	Convert date/time number to string.
STR_TO_NUM	0x90	Convert date/time string to number.
NUM_TO_DATE	0xA0	Convert date/time number to values stored in consecutive registers.
DATE_TO_NUM	0xB0	Convert values stored in consecutive registers to date/time number.

### Bits 3:0 Options

*See descriptions below.*

### Initialize

RTC, INIT+options  
Initialize the Real-time clock.

Bit	7	6	5	4	3	2	1	0
	0				E	S	C	A

### Bits 3 Enable RTCC output pin

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	The RTC output pin is disabled.
RTCC	0x08	The RTC output pin is enabled.

### Bits 2 Type of output

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
ALARM_OUT	0x00	Toggle RTC on each alarm event.
HZ_OUT	0x04	One Hz output.

### Bits 1 Calibration

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	No Calibration.

	CAL	0x02	Set calibration from lower 8 bits of register 0.
Bits 0	<b>Alarm Event</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	0x00	Alarm event disabled.
	ALARM_ON	0x00	Alarm event enabled.

### Start and Stop the Real-time Clock

RTC, START+*options*  
 RTC, STOP+*options*  
 Start and stop the real-time clock.

Bit	7	6	5	4	3	2	1	0
	Action			-				

### Set Alarm Mask

RTC, ALARM\_MASK+*options*  
 Specify the alarm mask. The mask is used in combination with the alarm time to determine when an alarm occurs. An alarm sets the RTC event flag and can optionally be output on the RTCC pin. The RTC event can be used to trigger an action at a specific time, or schedule.

Bit	7	6	5	4	3	2	1	0
	Action			Mask				

Bits 3:0	<b>Alarm Mask</b>		
	<i>Value</i>	<i>Description</i>	
	0	Alarm event every half second.	
	1	Alarm event every second.	
	2	Alarm event every 10 seconds.	
	3	Alarm event every minute.	
	4	Alarm event every 10 minutes.	
	5	Alarm event every hour.	
	6	Alarm event every day.	
	7	Alarm event every week.	
	8	Alarm event every month.	
	9	Alarm event every year.	

### Write Time, Write Alarm Time, Read Time, Read Alarm Time

RTC, WRITE\_TIME+*options*  
 RTC, WRITE\_ALARM+*options*  
 RTC, READ\_TIME+*options*  
 RTC, READ\_ALARM+*options*  
 Used to write and read the real time and alarm time values. The date and time value is read into or written out to register 0 or the string buffer. The entire date and time or specific date and time values can be selected.

Bit	7	6	5	4	3	2	1	0
	Action			S	Item			

Bits 3	<b>Numeric/String Select</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	-	0x00	The numeric value in register 0 is used.

	STR	0x08	A string value is used. If the string selection is not empty, the string selection is used, otherwise the string buffer is used.
Bits 2:0	<b>Item</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	DATE_TIME	0x00	Year, Month, Day, Hour, Minute, Second. Number: The numbers of seconds since 2000-01-01 00:00:00 String: YYYY-MM-DD HH:MM:SS
	SECOND	0x01	Seconds. Number: 0 to 59 String: 00...59
	MINUTE	0x02	Minutes. Number: 0 to 59 String: 00...59
	HOURL	0x03	Hours. Number: 0 to 23 String: 00...23
	DAY	0x04	Day. Number: 1 to 31 String: 01...31
	MONTH	0x05	Month. Number: 1 to 12 String: 01...12
	YEAR	0x06	Year. Number: 0 to 99 String: 2000...2099
	WEEKDAY	0x07	Weekday. Number: 0 to 6 String: 0...6

#### Convert Date/Time Number to String

RTC, NUM\_TO\_STR+options

Converts the date and time value in register 0 to a string, and stores it in the string buffer. If the string selection is not empty, the string selection is used, otherwise the string buffer is used.

Bit	7	6	5	4	3	2	1	0
	Action			Options				

Bits 3:0	<b>Item</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	DATE_TIME	0x00	YYYY-MM-DD HH:MM:SS
	DATE	0x01	YYYY-MM-DD
	TIME	0x02	HH:MM:SS

#### Convert String to Date/Time Number

RTC, STR\_TO\_NUM+options

Converts the date and time string in the string buffer to a numeric value, and stores it in register 0. If the string selection is not empty, the string selection is used, otherwise the string buffer is used.

Bit	7	6	5	4	3	2	1	0
	Action			Options				

Bits 3:0	<b>Item</b>		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	DATE_TIME	0x00	YYYY-MM-DD HH:MM:SS

DATE	0x01	YYYY-MM-DD
TIME	0x02	HH:MM:SS

### Convert Date/Time Number to Values stored in Consecutive Registers

**RTC, NUM\_TO\_DATE, *register***

Converts the date and time number in register 0 to date and time values stored in seven consecutive 32-bit registers, starting at the register specified. The values are stored as follows:

<i>register</i>	second (0-59)
<i>register+1</i>	minute (0-59)
<i>register+2</i>	hour (0-23)
<i>register+3</i>	day (1-31)
<i>register+4</i>	month (1-12)
<i>register+5</i>	year (0-99)
<i>register+6</i>	weekday (0-6) 0 = Sunday

### Convert Values stored in Consecutive Registers to Date/Time Number

**RTC, DATE\_TO\_NUM, *register***

Converts the date and time values stored in seven consecutive 32-bit registers, starting at the register specified, to a date and time number stored in register 0. The values are stored as shown above.

The 32-bit integer date/time format is the numbers of seconds since 2000-01-01 00:00:00.

The string date/time format is:  
YYYY-MM-DD HH:MM:SS

*Examples:*      **RTC, INIT+ALARM\_ON**      Disable RTCC pin clock, no calibration, enable alarm events.  
                 **RTC, START**                      Start the real-time clock.

*See Also:*      **TIMESET, TIMELONG, TICKLONG, DELAY**

---

## SAVEIND      Save using Indirect Pointer

*Syntax:*      **SAVEIND, *register***

*Description:*      The value of register A is stored at the indirect pointer specified by *register*. See the SETIND instruction for a description of pointers.

data value pointed to by *register* = reg[A]

*Opcode:*      **7B**

*Byte 2:*      ***register***  
Register number (0 to 255).

*Special Cases:*      • if data value pointed to by *register* is 32-bit and reg[A] is 64-bit, the value is converted to 32-bit before being saved  
                             • if data value pointed to by *register* is 64-bit and reg[A] is 32-bit, the value is converted to 64-bit before being used

*See Also:* SETIND, ADDIND, WRIND, RDIND, COPYIND, LOADIND

---

## SAVEMA Save register 0 value to matrix A

*Syntax:* **SAVEMA, row, column**

*Description:* Store the register 0 value to matrix A at the row, column specified. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, matrix A [row, column] = reg[0]  
 if reg[A] is 64-bit, matrix A [row, column] = reg[128]

*Opcode:* **6B**

*Byte 2:* **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:* **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*

- if row or column is out of range, no value is stored in the matrix
- if reg[A] is 64-bit, the value from register 128 is converted to 32-bit before being stored in the matrix

*See Also:* FFT, LOADMA, LOADMB, LOADMC, MOP, SAVEMB, SAVEMC, SELECTMA, SELECTMB, SELECTMC

---

## SAVEMB Save register 0 value to matrix B

*Syntax:* **SAVEMB, row, column**

*Description:* Store the register 0 value to matrix B at the row, column specified. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, matrix B [row, column] = reg[0]  
 if reg[A] is 64-bit, matrix B [row, column] = reg[128]

*Opcode:* **6C**

*Byte 2:* **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:* **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*

- if *row* or *column* is out of range, no value is stored in the matrix
- if register A is 64-bit, the value from register 128 is converted to 32-bit before being stored in the matrix

*See Also:*      **LOADMA, LOADMB, LOADMC, MOP, SAVEMA, SAVEMC, SELECTMA, SELECTMB, SELECTMC**

## **SAVEMC      Save register 0 value to matrix C**

*Syntax:*            **SAVEMC, row, column**

*Description:*      Store the register 0 value to matrix C at the row, column specified. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

if reg[A] is 32-bit, matrix C [*row, column*] = reg[0]  
 if reg[A] is 64-bit, matrix C [*row, column*] = reg[128]

*Opcode:*            **6D**

*Byte 2:*            **rows**  
 If bit 7 = 0, bits 6:0 specify the row of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the row of the matrix.

*Byte 3:*            **columns**  
 If bit 7 = 0, bits 6:0 specify the column of the matrix.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the column of the matrix.

*Special Cases:*

- if *row* or *column* is out of range, no value is stored in the matrix
- if reg[A] is 64-bit, the value from register 128 is converted to 32-bit before being stored in the matrix

*See Also:*            **LOADMA, LOADMB, LOADMC, MOP, SAVEMA, SAVEMB, SELECTMA, SELECTMB, SELECTMC**

## **SELECTA      Select A**

*Syntax:*            **SELECTA, register**

*Description:*      The *register* specified is selected as register A.

A = register

*Opcode:*            **01**

*Byte 2:*            **register**  
 Register number (0 to 255).

*See Also:*            **INDA, INDX, SELECTX**

---

## SELECTMA Select matrix A

**Syntax:**        **SELECTMA, register, rows, columns**

**Description:**    The *register* specifies the start of matrix A, and size of the matrix in *rows* and *columns*. The matrix is stored in sequential registers or RAM. If the matrix is stored in registers, register X is set to the first element of the matrix so that the FREADX, FWRITE, LREADX, LWRITE, SAVEX, SETX, LOADX instructions can be immediately used to store values to or retrieve values from the matrix. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

Select matrix A  
if register matrix, X = *register*

**Opcode:**        **65**

**Byte 2:**        **register**  
If bit 7 = 0, bits 6:0 specify a register number for the start of the matrix (0 to 127).  
If bit 7 = 1, bits 6:0 specify a register number, and the register contains an indirect pointer to the start of the matrix.

**Byte 3:**        **rows**  
If bit 7 = 0, bits 6:0 specify the number of rows in matrix.  
If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of rows in matrix.

**Byte 4:**        **columns**  
If bit 7 = 0, bits 6:0 specify the number of columns in matrix.  
If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of columns in matrix.

**Special Cases:**

- matrix operations are restricted to 32-bit floating point.
- indirect pointers must be used to select matrices in RAM.
- if matrix is too large for the available registers or RAM, a 1x1 matrix is defined and the 0,0 element is set to NaN.
- in a background process, a matrix that starts at register 0 to 15 must not extend beyond register 15.
- in a background process, larger matrices should be defined using registers 16 to 127, or RAM.

**See Also:**        FFT, LOADMA, LOADMB, LOADMC, MOP, SAVEMA, SAVEMB, SAVEMC, SELECTMB, SELECTMC

---

## SELECTMB Select matrix B

**Syntax:**        **SELECTMB, register, rows, columns**

**Description:**    The *register* specifies the start of matrix B, and size of the matrix in *rows* and *columns*. The matrix is stored in sequential registers or RAM. If the matrix is stored in registers, register X is also set to the first element of the matrix so that the FREADX, FWRITE, LREADX, LWRITE, SAVEX, SETX, LOADX instructions can be immediately used to store values to or retrieve



values from the matrix. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

Select matrix B  
if register matrix,  $X = \text{register}$

*Opcode:*       **66**

*Byte 2:*       **register**  
If bit 7 = 0, bits 6:0 specify a register number for the start of the matrix (0 to 127).  
If bit 7 = 1, bits 6:0 specify a register number, and the register contains an indirect pointer to the start of the matrix.

*Byte 3:*       **rows**  
If bit 7 = 0, bits 6:0 specify the number of rows in matrix.  
If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of rows in matrix.

*Byte 4:*       **columns**  
If bit 7 = 0, bits 6:0 specify the number of columns in matrix.  
If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of columns in matrix.

*Special Cases:*

- matrix operations are restricted to 32-bit floating point.
- indirect pointers must be used to select matrices in RAM.
- if an indirect pointer is a Flash pointer or the datatype is not Float32, the instruction is ignored.
- if matrix is too large for the available registers or RAM, a 1x1 matrix is defined and the 0,0 element is set to NaN.
- in a background process, a matrix that starts at register 0 to 15 must not extend beyond register 15.
- in a background process, larger matrices should be defined using registers 16 to 127, or RAM.

*See Also:*       LOADMA, LOADMB, LOADMC, MOP, SAVEMA, SAVEMC, SAVEMC, SELECTMB, SELECTMC

## SELECTMC Select matrix C

*Syntax:*       **SELECTMC, register, rows, columns**

*Description:*   The *register* specifies the start of matrix C, and size of the matrix in *rows* and *columns*. The matrix is stored in sequential registers or RAM. If the matrix is stored in registers, register X is also set to the first element of the matrix so that the FREADX, FWRITE, LREADX, LWRITE, SAVEX, SETX, LOADX instructions can be immediately used to store values to or retrieve values from the matrix. Additional information is available in the *Using the uM-FPU64 Matrix Instructions* document.

Select matrix C  
if register matrix,  $X = \text{register}$

*Opcode:*       **67**

*Byte 2:*       **register**  
If bit 7 = 0, bits 6:0 specify a register number for the start of the matrix (0 to 127).

If bit 7 = 1, bits 6:0 specify a register number, and the register contains an indirect pointer to the start of the matrix.

*Byte 3:*

**rows**

If bit 7 = 0, bits 6:0 specify the number of rows in matrix.

If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of rows in matrix.

*Byte 4:*

**columns**

If bit 7 = 0, bits 6:0 specify the number of columns in matrix.

If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the number of columns in matrix.

*Special Cases:*

- matrix operations are restricted to 32-bit floating point.
- indirect pointers must be used to select matrices in RAM.
- if an indirect pointer is a Flash pointer or the datatype is not Float32, the instruction is ignored.
- if matrix is too large for the available registers or RAM, a 1x1 matrix is defined and the 0,0 element is set to NaN.
- in a background process, a matrix that starts at register 0 to 15 must not extend beyond register 15.
- in a background process, larger matrices should be defined using registers 16 to 127, or RAM.

*See Also:*

LOADMA, LOADMB, LOADMC, MOP, SAVEMA, SAVEMB, SAVEMC, SELECTMB, SELECTMC

## SELECTX      Select register X

*Syntax:*

**SELECTX, register**

*Description:*

The *register* specified is selected as the register X.

*X = register*

*Opcode:*

**02**

*Byte 2:*

**register**

Register number (0 to 255).

*See Also:*

INDA, INDX, SELECTA

## SERIN      Serial input

*Syntax:*

**SERIN, action**

*Description:*

This instruction is used to read serial data from the SERIN pin or one of the digital I/O pins.

If the debug monitor is enabled, and the SERIN pin is selected, the serial input is handled by the debugger. The uM-FPU64 IDE can provide a terminal emulator, or simulate both character mode and NMEA mode input.

SERIN, DISABLE  
SERIN, ENABLE\_CHAR

SERIN, STATUS\_CHAR  
 SERIN, READ\_CHAR  
 SERIN, ENABLE\_NMEA  
 SERIN, STATUS\_NMEA  
 SERIN, READ\_NMEA

*Opcode:* **CF**

*Byte 2:* **action**

Bit	7	6	5	4	3	2	1	0
	Device				Action			

Bit 7:4	<b>Device</b>			
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>	
	-	0x00	SERIN pin	
	ASYNC	0x40	Digital I/O pin assigned by DEVIO, ASYNC	
Bits 7:5	<b>Action</b>			
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>	
	DISABLE	0x00	Disable serial input.	
	ENABLE_CHAR	0x01	Enable character mode input.	
	STATUS_CHAR	0x02	Get character mode input status.	
	READ_CHAR	0x03	Read character.	
	ENABLE_NMEA	0x04	Enable NMEA input.	
	STATUS_NMEA	0x05	Get NMEA input status.	
	READ_NMEA	0x06	Read NMEA sentence.	

If the input pin is SERIN:

- 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.

If the input pin is a digital I/O pin:

- the pin must first be initialized for serial input using the DEVIO, ASYNC instruction.
- the baud rate is specified using the DEVIO, ASYNC instruction

#### Disable Input (0x00, 0x40)

SERIN, DISABLE

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

#### Enable Character Mode Input (0x01, 0x41)

SERIN, ENABLE\_CHAR

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, STATUS\_CHAR instruction and input characters can be read using the SERIN, READ\_CHAR instruction.

#### Get Character Mode Input Status (0x02, 0x42)

SERIN, STATUS\_CHAR

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.

### Read Character (0x03, 0x43)

**SERIN, READ\_CHAR**

Read next serial input character. The serial input character is stored in register 0. If this instruction is the last instruction in the instruction buffer, it will wait for the next available input character. If there are other instructions in the instruction buffer, or another instruction is sent before the **SERIN, READ\_CHAR** instruction has completed, it will terminate and store a zero value in register 0.

### Enable NMEA Input Mode (0x04, 0x44)

**SERIN, ENABLE\_NMEA**

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, READ\_NMEA** instruction, and the NMEA input status can be checked with the **SERIN, STATUS\_NMEA** instruction.

### Read NMEA Input Status (0x05, 0x45)

**SERIN, STATUS\_NMEA**

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.

### Read NMEA Sentence (0x06, 0x46)

**SERIN, READ\_NMEA**

Transfer next 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. If there are other instructions in the instruction buffer, or another instruction is sent before the **SERIN, READ\_NMEA** instruction has completed, it will terminate and store an empty sentence.

#### Examples:

<b>SERIN, ENABLE_CHAR</b>	Enable character input on SERIN pin.
<b>SERIN, ASYNC+ENABLE_CHAR</b>	Enable character input on DEVIO, ASYNC pin.
<b>SERIN, READ_CHAR</b>	Read the next character received on the SERIN pin and store the value in register 0.

*See Also:* **DEVIO, ASYNC, SEROUT**

---

## SEROUT      Serial Output

*Syntax:*      **SEROUT, action{, mode} / {, string}**

*Description:* This instruction is used to enable and disable Debug Mode, set the baud rate for the SERIN and SEROUT pins, and to write serial data to the SEROUT pin or one of the digital I/O pins.

A secondary use for this instruction is for data logging. Data channels 1-3 are provided for this purpose. If the debug monitor is enabled, and the SEROUT pin or data channels 1-3 are selected, the serial output is sent to the debugger. This information is handled by the uM-FPU64 IDE and can be displayed in a terminal emulator, as a text log, or as a table and graph. If the debugger is not enabled, output to data channels 1-3 is suppressed.

SEROUT, SET\_BAUD, *mode*  
 SEROUT, WRITE\_STR, *string*  
 SEROUT, WRITE\_SBUF  
 SEROUT, WRITE\_SSEL  
 SEROUT, WRITE\_CHAR  
 SEROUT, WRITE\_STRZ, *string*  
 SEROUT, WRITE\_FLOAT, *register*, *format*  
 SEROUT, WRITE\_LONG, *register*, *format*  
 SEROUT, WRITE\_COMMA  
 SEROUT, WRITE\_CRLF

*Opcode:* **CE**

*Byte 2:* **action**

Bit 7	6	5	4	3	2	1	0
Device				Action			

Bit 7:4

**Device**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
-	0x00	SEROUT pin
IDE1	0x10	Debug mode, data channel 1
IDE2	0x20	Debug mode, data channel 2
IDE3	0x30	Debug mode, data channel 3
ASYNC	0x40	Digital I/O pin assigned by DEVIO, ASYNC

Bits 7:5

**Action**

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
SET_BAUD	0x00	Set baud rate and debug mode.
WRITE_STR	0x01	Write string.
WRITE_SBUF	0x02	Write string buffer.
WRITE_SSEL	0x03	Write string selection.
WRITE_CHAR	0x04	Write character.
WRITE_STRZ	0x05	Write string and zero terminator.
WRITE_FLOAT	0x06	Write floating point value.
WRITE_LONG	0x07	Write long integer value.
WRITE_COMMA	0x08	Write comma.
WRITE_CRLF	0x09	Write carriage return, linefeed.

If the output pin is SEROUT:

- the instruction is ignored if Debug Mode is enabled
- the baud rate for serial output is set with the SEROUT, 0 instruction.

If the output pin is a digital I/O pin:

- the pin must first be initialized for serial output using the DEVIO instruction.
- the baud rate is specified using the DEVIO instruction

### Set Baud Rate and Debug Mode (0x00, 0x10, 0x20, 0x30, 0x40)

SEROUT, SET\_BAUD, *mode*

This instruction sets the baud rate for the SERIN and SEROUT pins, and enables or disables Debug Mode. The mode is specified by the byte immediately following the opcode and action byte.

Byte 3:

***mode***

<i>Value</i>	<i>Description</i>
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

If the serial output pin is SEROUT (ASYNC0), the baud rate and debug mode is set according to the value. If the mode value is 0, a {DEBUG ON} message is sent to the serial output and the baud rate is changed. If the mode value is 1 through 10, if the debug mode is enabled, and a {DEBUG OFF} message is sent to the SEROUT before the baud rate is changed.

If the serial output pin is a digital I/O pin (ASYNC1), the SET\_BAUD instruction is ignored. The serial port must be initialized with the DEVIO, INIT instruction.

### Write String (0x01, 0x11, 0x21 0x31, 0x41)

SEROUT, WRITE\_STR, *string*

The zero-terminated 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. If *string* length > 127, *string* will be truncated to 127 characters.

### Write String Buffer (0x02, 0x12, 0x22, 0x32, 0x42)

SEROUT, WRITE\_SBUF

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

### Write String Selection (0x03, 0x13, 0x23, 0x33, 0x43)

SEROUT, WRITE\_SSEL

The current string selection is sent to the serial output. The instruction is ignored if Debug Mode is enabled.

### Write Character (0x04, 0x14, 0x24, 0x34, 0x44)

SEROUT, WRITE\_CHAR

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

#### Write String and Zero-Terminator (0x05, 0x15, 0x25, 0x35, 0x45)

SEROUT, WRITE\_STRZ, *string*

The zero-terminated text string specified by the instruction is sent to the serial output, including the zero-terminator. The instruction is ignored if Debug Mode is enabled.

#### Write Float (0x06, 0x16, 0x26, 0x36, 0x46)

SEROUT, WRITE\_FLOAT, *register*, *format*

The value in *register* is converted to a floating point string using the *format* specified and sent to the serial output.

Byte 3:           ***register***  
                  Register number (0-255).

Byte 4:           ***format***  
                  Conversion format (see FTOA instruction).

#### Write Long (0x07, 0x17, 0x27, 0x37, 0x47)

SEROUT, WRITE\_LONG, *register*, *format*

The value in *register* is converted to a long integer string using the *format* specified and sent to the serial output.

Byte 3:           ***register***  
                  Register number (0-255).

Byte 4:           ***format***  
                  Conversion format (see LTOA instruction).

#### Write Comma (0x08, 0x18, 0x28, 0x38, 0x48)

SEROUT, WRITE\_COMMA

A comma is sent to the serial output.

#### Write Carriage Return, Linefeed (0x09, 0x19, 0x29, 0x39, 0x49)

SEROUT, WRITE\_CRLF

A carriage return and linefeed is sent to the serial output.

#### Examples:

SEROUT, ENABLE_CHAR	Enable character input on SERIN pin.
SEROUT, ASYNC+ENABLE_CHAR	Enable character input on DEVIO, ASYNC pin.
SEROUT, READ_CHAR	Read the next character received on the SERIN pin and store the value in register 0.

See Also:       DEVIO, ASYNC, SERIN

---

## SETARGS   Enable FCALL argument loading

Syntax:       **SETARGS**

Description:   The SETARGS instruction is used to facilitate the passing of arguments to a user-defined function. After a SETARGS instruction, instructions that normally load values to register 0, will instead load

values sequentially into registers 1 to 9, if register A is a 32-bit register, or registers 129-137, if register A is a 64-bit register. The register A selection is saved by the first SETARGS instruction. Each additional SETARGS before the next FCALL will toggle between 32-bit and 64-bit argument loading by toggled the register A selection between register 0 and register 128. Argument loading is disabled by the next FCALL instruction.

*Opcode:* **DD**

*See Also:* CLR, CLR0, DWRITE, FWRITE, FWRITE0, LOAD, LOADA, LOADX, LOADBYTE, LOADE, LOADIND, LOADPI, LOADUBYTE, LOADUWORD, LOADWORD, LONGBYTE, LONGUBYTE, LONGUWORD, LONGWORD, LWRITE, LWRITE0, RIGHT

---

## SETIND Set indirect pointer

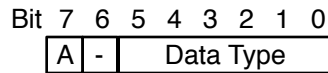
*Syntax:* **SETIND, type, register**  
**SETIND, type, address**  
**SETIND, type, function, offset**

*Description:* Register 0 or 128 is set to the value of an indirect pointer. Indirect pointers can point to registers or memory. If *type* specifies a register pointer, the pointer will point to the *register* specified. If *type* specifies a memory or DMA pointer, the pointer will point to the specified *memory address*. If *type* specifies a Flash pointer, the pointer will point to Flash data in the specified *function* and *offset*.

if reg[A] is 32-bit,  
     reg[0] = indirect pointer  
 if reg[A] is 64-bit,  
     reg[0] = indirect pointer

*Opcode:* **77**

*Byte 2:* **type**  
 This type is stored in the type field of the indirect pointer.



### Bits 7 Auto Increment

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
INC	0x80	Auto-increment the pointer

### Bits 5:0 Data Type

<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
REG_LONG	0x00	Register, Long integer data
REG_FLOAT	0x01	Register, Floating point data
MEM_INT8	0x08	Memory, 8-bit signed integer data
MEM_UINT8	0x09	Memory, 8-bit unsigned integer data
MEM_INT16	0x0A	Memory, 16-bit signed integer data
MEM_UINT16	0x0B	Memory, 16-bit unsigned integer data
MEM_LONG32	0x0C	Memory, 32-bit long integer data
MEM_FLOAT32	0x0D	Memory, 32-bit floating point data
MEM_LONG64	0x0E	Memory, 64-bit long integer data
MEM_FLOAT64	0x0F	Memory, 64-bit floating point data



DMA_INT8	0x18	DMA, 8-bit signed integer data
DMA_UINT8	0x19	DMA, 8-bit unsigned integer data
DMA_INT16	0x1A	DMA, 16-bit signed integer data
DMA_UINT16	0x1B	DMA, 16-bit unsigned integer data
DMA_LONG32	0x1C	DMA, 32-bit long integer data
DMA_FLOAT32	0x1D	DMA, 32-bit floating point data
DMA_LONG64	0x1E	DMA, 64-bit long integer data
DMA_FLOAT64	0x1F	DMA, 64-bit floating point data
FLASH_INT8	0x28	Flash, 8-bit signed integer data
FLASH_UINT8	0x29	Flash, 8-bit unsigned integer data
FLASH_INT16	0x2A	Flash, 16-bit signed integer data
FLASH_UINT16	0x2B	Flash, 16-bit unsigned integer data
FLASH_LONG32	0x2C	Flash, 32-bit long integer data
FLASH_FLOAT32	0x2D	Flash, 32-bit floating point data
FLASH_LONG64	0x2E	Flash, 64-bit long integer data
FLASH_FLOAT64	0x2F	Flash, 64-bit floating point data

### Register Pointer

*Byte 3:*        **register**  
                     Register number (0 to 255). This value is stored in the address field of the indirect pointer.

### Memory or DMA Pointer

*Byte 3-4:*    **address**  
                     A 16-bit value the specifies the memory address (0 to 65535). This value is stored in the address field of the indirect pointer.

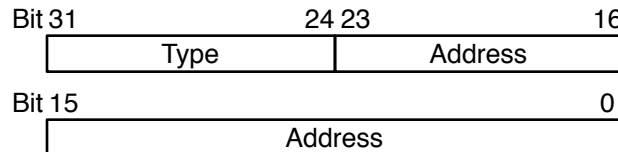
### Flash Pointer

*Byte 3:*        **function**  
                     Function number (0 to 63). This value is stored in the upper 8 bits of the indirect pointer address field.

*Byte 4-5:*    **offset**  
                     A 16-bit value the specifies the memory address (0 to 65535). This value is stored in the lower 16 bits of the indirect pointer address field.

### Notes:

The indirect pointer stored in Register A has the following format:



*Special Cases:*    • if reg[A] is 64-bit, the upper 32 bits of reg[128] are set to zero.

*See Also:*        ADDIND, COPYIND, LOADIND, RDIND, SAVEIND, WRIND

---

**SETREAD    Set Read Mode**

*Syntax:*            **SETREAD**

*Description:*      This instruction should be used by the foreground process prior to any read instruction. It ensures that the FPU stays in foreground mode, and is ready to send data when the read instruction is received from the microcontroller. Background processes will not run until the next read instruction has finished executing.

*Opcode:*            **FD**

*See Also:*            FREAD, FREADA, FREADX, FREAD0, LREAD, LREADA, LREADX, LREAD0, LREADBYTE, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

---

**SETSTATUS    Set status byte**

*Syntax:*            **SETSTATUS, status**

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

status = *status*

*Opcode:*            **CD**

*Byte 2:*            **status**

The 8-bit value to store as the internal status value.

*See Also:*            SETREAD, FSTATUS, FSTATUSA, LSTATUS, LSTATUSA, SETSTATUS

---

**SIN            Sine**

*Syntax:*            **SIN**

*Description:*      Calculates the sine of the angle (in radians) in register A and stores the result in register A.

reg[A] = sin(reg[A])

*Opcode:*            **47**

*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

*See Also:*            ACOS, ASIN, ATAN, ATAN2, COS, TAN, DEGREES, RADIANS

---

**SQRT          Square root**

*Syntax:*            **SQRT**

*Description:* Calculates the square root of the floating point value in register A and stores the result in register A.

$\text{reg}[A] = \text{sqrt}(\text{reg}[A])$

*Opcode:* **41**

*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

*See Also:* FPOW, FPOWI, FPOW0, EXP, EXP10, LOG, LOG10, ROOT

## STRBYTE Insert byte at string selection

*Syntax:* **STRBYTE**

*Description:* If register A is 32-bit, the lower 8 bits of register 0 are stored as an 8-bit character in the string buffer at the current selection point. If register A is 64-bit, the lower 8 bits of register 128 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.

*Opcode:* **ED**

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

## STRCMP Compare string with string selection

*Syntax:* **STRCMP, *string***

*Description:* The *string* is compared with the string at the current selection point of the string buffer and the internal status byte is set.

$\text{status} = \text{longStatus of string compare}$

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

*Opcode:* **E6**

*Bytes 2-n:* ***string***  
A zero-terminated string.

*Special Cases:*

- if *string* length > 127, *string* will be truncated to 127 characters

*See Also:* STRSET, STRSEL, STRINS, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

## STRDEC Decrement string selection point

*Syntax:* **STRDEC**

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

*Opcode:* **EF**

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

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

## STRFCHR Set field separator characters

*Syntax:* **STRFCHR, string**

*Description:* The *string* specifies a list of characters (maximum of six) to be used as field separators by the STRFIELD instruction. The default field separator is a comma.

*Opcode:* **E8**

*Bytes 2-n:* **string**  
A zero-terminated string.

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

## STRFIELD Find field in string

*Syntax:* **STRFIELD, field**

*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.

*Opcode:* **E9**

*Byte 2:* **field**  
If bit 7 = 0, bits 6:0 specify the field.  
If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the field.

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

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRFIND Find string in the string buffer

*Syntax:* **STRFIND, *string***

*Description:* Search the current string selection in the string buffer 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 current string selection.

*Opcode:* **E7**

*Bytes 2-n:* ***string***  
A zero-terminated string.

*Special Cases:* • if *string* length > 127, *string* will be truncated to 127 characters

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRINC Increment string selection point

*Syntax:* **STRINC**

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

*Opcode:* **EE**

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

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRINS Insert string

*Syntax:* **STRINS, *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.

*Opcode:* **E5**

*Bytes 2-n:* ***string***  
A zero-terminated string.

*Special Cases:* • if *string* length > 127, *string* will be truncated to 127 characters

*See Also:* STRSET, STRSEL, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRSEL Set string selection point

*Syntax:* **STRSEL, start, length**

*Description:* Set the start of the string selection to character *start* and the length of the selection to *length* characters. Characters are numbered from 0 to n.

*Opcode:* **E4**

*Byte 2:* **start**  
 If bit 7 = 0, bits 6:0 specify the start of the string selection.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the start of the string selection.

*Byte 3:* **length**  
 If bit 7 = 0, bits 6:0 specify the length of the string selection.  
 If bit 7 = 1, bits 6:0 specify a register number, and the lower 8 bits of the register specify the length of the string selection.

*Special Cases:*

- if *start* > string length, start of selection is set to end of string
- if *start+length* > string length, selection is adjusted for the end of string

*See Also:* STRSET, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRSET Copy string to string buffer

*Syntax:* **STRSET, string**

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

*Opcode:* **E3**

*Bytes 2-n:* **string**  
 A zero-terminated string.

*Special Cases:*

- if *string* length > 127, *string* will be truncated to 127 characters

*See Also:* STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, STRTOL, FTOA, LTOA, READSTR, READSEL

---

## STRTOF Convert string selection to floating point

*Syntax:* **STRTOF**

*Description:* Convert the string at the current selection point to a floating point value. If register A is 32-bit, the

result is stored in register 0. If register A is 64-bit, the result is stored in register 128.

*Opcode:* **EA**

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOL, FTOA, LTOA, READSTR, READSEL

## STRTOL Convert string selection to long integer

*Syntax:* **STRTOL**

*Description:* Convert the string at the current selection point to a long integer value. If register A is 32-bit, the result is stored in register 0. If register A is 64-bit, the result is stored in register 128.

*Opcode:* **EB**

*See Also:* STRSET, STRSEL, STRINS, STRCMP, STRFIND, STRFCHR, STRFIELD, STRINC, STRDEC, STRBYTE, STRTOF, FTOA, LTOA, READSTR, READSEL

## SWAP Swap registers

*Syntax:* **SWAP, register1, register2**

*Description:* The values of *register1* and *register2* are swapped.

tmp = reg[*register1*], reg[*register1*] = reg[*register2*], reg[*register2*] = tmp

*Opcode:* **12**

*Byte 2:* **register1**  
Register number (0 to 255).

*Byte 3:* **register2**  
Register number (0 to 255).

*Special Cases:*

- if *register2* is 32-bit and *register1* is 64-bit, only the lower 32-bits of *register1* are copied and the upper 32-bits of *register1* are set to zero
- if *register2* is 64-bit and *register1* is 32-bit, only the lower 32-bits of *register2* are copied and the upper 32-bits of *register2* are set to zero

*See Also:* SWAPA

## SWAPA Swap register A

*Syntax:* **SWAPA, register**

*Description:* The values of *register* and register A are swapped.

tmp = reg[*register*], reg[*register*] = reg[A], reg[A] = tmp

<i>Opcode:</i>	<b>13</b>
<i>Byte 2:</i>	<b><i>register</i></b> Register number (0 to 255).
<i>Special Cases:</i>	<ul style="list-style-type: none"> <li>• if reg[A] is 32-bit and <i>register</i> is 64-bit, only the lower 32-bits of <i>register</i> are copied and the upper 32-bits of <i>register</i> are set to zero</li> <li>• if <i>register</i> is 64-bit and reg[A] is 32-bit, only the lower 32-bits of reg[A] are copied and the upper 32-bits of reg[A] are set to zero</li> </ul>
<i>See Also:</i>	SWAP

## **SYNC      Synchronization**

<i>Syntax:</i>	<b>SYNC</b>
<i>Description:</i>	A sync character (0x5C) is sent in reply. This instruction is typically used after a reset to verify communications.
<i>Opcode:</i>	<b>F0</b>
<i>Returns:</i>	<b>5C</b>

## **TABLE      Table lookup**

<i>Syntax:</i>	<b>TABLE, <i>tableSize</i>, <i>tableItem1</i>...<i>tableItemN</i></b>
<i>Description:</i>	<p>This opcode is only valid within a user function stored in the uM-FPU64 Flash memory. The value of the item in the 32-bit 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.</p> <p>reg[0] = value from table indexed by reg[0]  if reg[A] is 32-bit, reg[0] = value from table indexed by reg[0]  if reg[A] is 64-bit, reg[128] = value from table indexed by lower 32-bits of reg[128]</p>
<i>Opcode:</i>	<b>85</b>
<i>Byte 2:</i>	<b><i>tableSize</i></b> Specifies the number of 32-bit values in the table (0-255). If <i>tableSize</i> is 0, the number of 32-bit values in the table is 256.
<i>Bytes 3-n:</i>	<b><i>TableItem1</i>...<i>TableItemN</i></b> 32-bit integer values or 32-bit floating point values
<i>Special Cases:</i>	<ul style="list-style-type: none"> <li>• only valid inside user-defined functions stored in Flash memory.</li> <li>• if the number of table entries doesn't match the table size, the instruction is ignored.</li> </ul>



- if  $\text{reg}[0 \mid 128] \leq 0$ , then zero is returned.
- if  $\text{reg}[0 \mid 128] > \text{maximum size of table}$ , then zero is returned.
- if  $\text{reg}[A]$  is 64-bit, the lower 32-bit are set to the value in the table and the upper 32-bits are zero.

*See Also:* FTABLE, LTABLE, POLY

---

## TAN Tangent

*Syntax:* **TAN**

*Description:* Calculates the tangent of the angle (in radians) in register A and stores the result in register A.

$\text{reg}[A] = \tan(\text{reg}[A])$

*Opcode:* **49**

*Special Cases:*

- if  $\text{reg}[A]$  is NaN or an infinity, then the result is NaN
- if  $\text{reg}[A]$  is 0.0, then the result is 0.0
- if  $\text{reg}[A]$  is -0.0, then the result is -0.0

*See Also:* ACOS, ASIN, ATAN, ATAN2, COS, SIN, DEGREES, RADIANS

---

## TICKLONG Load register 0 with millisecond or microsecond ticks

*Syntax:* **TICKLONG**

*Description:* Load register 0 (32-bit) with the time in milliseconds, or register 128 (64-bit) with the time in microseconds.

if  $\text{reg}[A]$  is 32-bit,  $\text{reg}[0] = \text{ticks in milliseconds}$ ,  $\text{status} = \text{longStatus}(\text{reg}[0])$   
 if  $\text{reg}[A]$  is 64-bit,  $\text{reg}[128] = \text{ticks in microseconds}$ ,  $\text{status} = \text{longStatus}(\text{reg}[128])$

*Opcode:* **D9**

*Special Cases:*

- if  $\text{reg}[A]$  is 64-bit, then upper 32-bits are set to zero

*See Also:* TIMESET, TIMELONG, RTC, DELAY

---

## TIMELONG Load register 0 with time value in seconds

*Syntax:* **TIMELONG**

*Description:* Load register 0 (32-bit) or register 128 (64-bit) with the time in seconds.

if  $\text{reg}[A]$  is 32-bit,  $\text{reg}[0] = \text{time in seconds}$ ,  $\text{status} = \text{longStatus}(\text{reg}[0])$   
 if  $\text{reg}[A]$  is 64-bit,  $\text{reg}[128] = \text{time in seconds}$ ,  $\text{status} = \text{longStatus}(\text{reg}[128])$

*Opcode:* **D8**

*Special Cases:* • if reg[A] is 64-bit, then upper 32-bits are set to zero

*See Also:* `TIMESET`, `TICKLONG`, `RTC`, `DELAY`

## **TIMESET Set time value in seconds**

*Syntax:* **TIMESET**

*Description:* The time in seconds is set from the value in register 0 (32-bit) or register 128 (64-bit). If the value in register 0 or 128 is -1, the timer is disabled.

if reg[A] is 32-bit, time in seconds = reg[0], ticks = time in milliseconds

if reg[A] is 64-bit, time in seconds = reg[128], ticks = time in microseconds

*Opcode:* **D7**

*Special Cases:* • if reg[0 | 128] is -1, the timer is turned off.

*See Also:* `TIMELONG`, `TICKLONG`, `RTC`, `DELAY`

## **TRACEOFF Turn debug trace off**

*Syntax:* **TRACEOFF**

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

*Opcode:* **F8**

*See Also:* `TRACEON`, `TRACEREG`, `TRACESTR`, `BREAK`

## **TRACEON Turn debug trace on**

*Syntax:* **TRACEON**

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

*Opcode:* **F9**

*See Also:* `TRACEOFF`, `TRACEREG`, `TRACESTR`, `BREAK`

## **TRACEREG Display register value in debug trace**

*Syntax:* **TRACEREG, register**

*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* will be displayed on the debug terminal as follows:  
32-bit register: {R1:3F800000}

64-bit register: {R129:3FF0000000000000}

*Opcode:* **FB**

*Byte 2:* **register**  
Register number (0 to 255).

*See Also:* TRACEOFF, TRACEON, TRACESTR, BREAK

## TRACESTR Display debug trace message

*Syntax:* **TRACESTR, string**

*Description:* Used with the built-in debugger. If the debugger is not enabled, this instruction is ignored. If the debugger is enabled, the *string* will be displayed on the debug terminal. If the string is of the form \$Rxx, a READVAR value is output as a hexadecimal string, where xx is the decimal value of the READVAR value.

*Opcode:* **FA**

*Bytes 2-n:* **string**  
A zero-terminated string.

*Examples:* TRACESTR, "test" sends {"test"} to debug terminal  
TRACESTR, "\$R14" sends {"\$R14":0000} to debug terminal (READVAR, 14 value)

*See Also:* TRACEOFF, TRACEON, TRACEREG, BREAK

## VERSION Copy the version string to the string buffer

*Syntax:* **VERSION**

*Description:* The uM-FPU64 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:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	4				Major				Minor				Beta			
Bits 15:12	Chip Version (always set to 4)															
Bits 11:8	Major Version															
Bits 7:4	Minor Version															
Bits 3:0	Beta Version															

As an example:

version string: uM-FPU64 r402  
version code: 0x4100

*Opcode:* **F3**

**WRIND**      **Write data using indirect pointer**

**Syntax:** `WRIND, dataType, pointer, count, value1...valueN`

*Description:* Write *count* data values of the specified *dataType* to the *pointer* location. If *count* = 0, then the count is loaded from the lower 16 bits of register 0. The pointer can be a register pointer or a memory pointer. If *dataType* is different then the data type of the *pointer* data conversion is automatically performed. See the SETIND instruction for a description of pointers. The WRIND instruction has been optimized for 32-bit transfers of the same data type (e.g. long-to-long or float-to-float). These transfers can be done at the maximum transfer rate without filling the instruction buffer. Transfers that require data conversions may require an additional delay between data transfers to avoid exceeding the 256 byte FPU instruction buffer.

*Opcode:*           **70**

*Byte 2:*            **dataType**

Bit	7	6	5	4	3	2	1	0
	-				Data Type			

Bits 3:0	Data Type		
	<i>IDE Symbol</i>	<i>IDE Value</i>	<i>Description</i>
	INT8	0x08	8-bit signed integer data
	UINT8	0x09	8-bit unsigned integer data
	INT16	0x0A	16-bit signed integer data
	UINT16	0x0B	16-bit unsigned integer data
	LONG32	0x0C	32-bit long integer data
	FLOAT32	0x0D	32-bit floating point data
	LONG64	0x0E	64-bit long integer data
FLOAT64	0x0F	64-bit float point data	

**Byte 3:** ***pointer***  
The register number of a register that contains a pointer (0 to 255).

**Byte 4:** **count**  
An 8-bit value that specifies the number of data items to read from the pointer location (0 to 255). If *count* = 0, the lower 16 bits of register 0 specify the number of data items to read from the pointer location.

Bytes 5-n: **value1...valueN**  
Data values of the type specified by *dataType*.

*Special Cases:*

- if *dataType* is 32-bit floating point, and PICMODE is enabled, the values are converted to IEEE-754 format before being stored

*See Also:* SETIND, ADDIND, RDIND, COPYIND, LOADIND, SAVEIND  
FWRITE, FWRITE0, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEA,  
LWRITEX, DWRITE

---

**XOP      Execute extended opcode instruction stored in Flash memory**

*Syntax:*      **XOP, xop\_number, arg1, arg2, arg2**

*Description:*      Executes the extended opcode instruction specified by *xop\_number*, using arguments *arg1*, *arg2*, *arg3*. The value of the arguments depends on the XOP instruction. Separate documentation for XOP library files specify the arguments required for each XOP. All XOP instructions have three argument bytes. If an XOP requires fewer than three arguments, zero bytes are used as fillers.

The uM-FPU64 IDE software is used to load the code for XOP instructions into Flash memory.

*Opcode:*      **FE**

*Byte 2:*      **xop\_number**  
An 8-bit value specifying the XOP number which is used to determine the location of the XOP ode programmed in Flash memory.

*Byte 3:*      **arg1**  
An 8-bit value used as the first argument for the XOP instruction.

*Byte 4:*      **arg2**  
An 8-bit value used as the second argument for the XOP instruction.

*Byte 5:*      **arg3**  
An 8-bit value used as the third argument for the XOP instruction.

*Examples:*

*ASM:*      XOP, 01, 10, 20, 30 ; call XOP 1 with arguments 10, 20, and 30

*Compiler:*      qa[4] equ    F%  
                 qb[4] equ    F%  
                 qc[4] equ    F%  
                 q\_add(qa, qb, qc)    ; call XOP q\_add with quaternions qa, qb, qc

---

**XSAVE      Save register value to register X**

*Syntax:*      **XSAVE, register**

*Description:*      Set register X to the value of *register*, and select the next register in sequence as register X.

reg[X] = reg[register], status = longStatus(reg[X]), X = X + 1

*Opcode:*      **0E**

*Byte 2:*      **register**  
Register number (0 to 255).

*Special Cases:*      • if *register* is 32-bit and reg[X] is 64-bit, the upper 32-bits of reg[X] are set to zero

- if *register* is 64-bit and reg[X] is 32-bit, only the lower 32-bits of *register* are copied
- if reg[X] is 32-bit, it will not increment past register 127
- if reg[X] is 64-bit, it will not increment past register 255

*See Also:*      **LOAD, LOADA, LOADX, ALOADX, XSAVEA**

---

## **XSAVEA      Save register A to register X**

*Syntax:*            **XSAVEA**

*Description:*      Set register X to the value of register A, and select the next register in sequence as register X.

$\text{reg[X]} = \text{reg[A]}$ ,  $\text{status} = \text{longStatus}(\text{reg[X]})$ ,  $X = X + 1$

*Opcode:*            **0F**

*Special Cases:*

- if reg[A] is 32-bit and reg[X] is 64-bit, the upper 32-bits of reg[X] are set to zero
- if reg[A] is 64-bit and reg[X] is 32-bit, only the lower 32-bits of reg[A] are copied
- if reg[X] is 32-bit, it will not increment past register 127
- if reg[X] is 64-bit, it will not increment past register 255

*See Also:*            **LOAD, LOADA, LOADX, ALOADX, XSAVE**

---

## Appendix A

### uM-FPU64 Instruction Summary

Instruction	Opcode	Arguments	Returns	Description
NOP	00			No Operation
SELECTA	01	<i>register</i>		Select register A, A = <i>register</i>
SELECTX	02	<i>register</i>		Select register X, X = <i>register</i>
CLR	03	<i>register</i>		reg[ <i>register</i> ] = 0
CLRA	04			reg[A] = 0
CLR <sub>X</sub>	05			reg[X] = 0, X = X + 1
CLR <sub>0</sub>	06			reg[0   128] = 0
COPY	07	<i>register1</i> , <i>register2</i>		reg[ <i>register2</i> ] = reg[ <i>register1</i> ]
COPYA	08	<i>register</i>		reg[ <i>register</i> ] = reg[A]
COPYX	09	<i>register</i>		reg[ <i>register</i> ] = reg[X], X = X + 1
LOAD	0A	<i>register</i>		reg[0   128] = reg[ <i>register</i> ]
LOADA	0B			reg[0   128] = reg[A]
LOADX	0C			reg[0   128] = reg[X], X = X + 1
ALOADX	0D			reg[A] = reg[X], X = X + 1
XSAVE	0E	<i>register</i>		reg[X] = reg[ <i>register</i> ], X = X + 1
XSAVEA	0F			reg[X] = reg[A], X = X + 1
COPY <sub>0</sub>	10	<i>register</i>		reg[ <i>register</i> ] = reg[0   128]
LCOPYI	11	<i>signedByte</i> , <i>register</i>		reg[ <i>register</i> ] = long( <i>signedByte</i> )
SWAP	12	<i>register1</i> , <i>register2</i>		Swap reg[ <i>register1</i> ] and reg[ <i>register2</i> ]
SWAPA	13	<i>register</i>		Swap reg[ <i>register</i> ] and reg[A]
LEFT	14			Left parenthesis
RIGHT	15			Right parenthesis
FWRITE	16	<i>register</i> , <i>float32Value</i>		Write 32-bit floating point to reg[ <i>register</i> ]
FWRITEA	17	<i>float32Value</i>		Write 32-bit floating point to reg[A]
FWRITEX	18	<i>float32Value</i>		Write 32-bit floating point to reg[X]
FWRITE <sub>0</sub>	19	<i>float32Value</i>		Write 32-bit floating point to reg[0   128]
FREAD	1A	<i>register</i>	<i>float32Value</i>	Read 32-bit floating point from reg[ <i>register</i> ]
FREADA	1B		<i>float32Value</i>	Read 32-bit floating point from reg[A]
FREADX	1C		<i>float32Value</i>	Read 32-bit floating point from reg[X]
FREAD <sub>0</sub>	1D		<i>float32Value</i>	Read 32-bit floating point from reg[0   128]
ATOF	1E	<i>string</i>		Convert ASCII to floating point
FTOA	1F	<i>format</i>		Convert floating point to ASCII
FSET	20	<i>register</i>		reg[A] = reg[ <i>register</i> ]
FADD	21	<i>register</i>		reg[A] = reg[A] + reg[ <i>register</i> ]
FSUB	22	<i>register</i>		reg[A] = reg[A] - reg[ <i>register</i> ]
FSUBR	23	<i>register</i>		reg[A] = reg[ <i>register</i> ] - reg[A]
FMUL	24	<i>register</i>		reg[A] = reg[A] * reg[ <i>register</i> ]
FDIV	25	<i>register</i>		reg[A] = reg[A] / reg[ <i>register</i> ]
FDIVR	26	<i>register</i>		reg[A] = reg[ <i>register</i> ] / reg[A]
FPOW	27	<i>register</i>		reg[A] = reg[A] ** reg[ <i>register</i> ]

FCMP	28	<i>register</i>		Compare reg[A] and reg[ <i>register</i> ], and set status
FSET0	29			reg[A] = reg[0   128]
FADD0	2A			reg[A] = reg[A] + reg[0   128]
FSUB0	2B			reg[A] = reg[A] - reg[0   128]
FSUBR0	2C			reg[A] = reg[0] - reg[A]
FMUL0	2D			reg[A] = reg[A] * reg[0   128]
FDIV0	2E			reg[A] = reg[A] / reg[0   128]
FDIVR0	2F			reg[A] = reg[0   128] / reg[A]
FPOW0	30			reg[A] = reg[A] ** reg[0   128]
FCMP0	31			Compare reg[A] and reg[0   128]
FSETI	32	<i>signedByte</i>		reg[A] = float( <i>signedByte</i> )
FADDI	33	<i>signedByte</i>		reg[A] = reg[A] - float( <i>signedByte</i> )
FSUBI	34	<i>signedByte</i>		reg[A] = reg[A] - float( <i>signedByte</i> )
FSUBRI	35	<i>signedByte</i>		reg[A] = float( <i>signedByte</i> ) - reg[A]
FMULI	36	<i>signedByte</i>		reg[A] = reg[A] * float( <i>signedByte</i> )
FDIVI	37	<i>signedByte</i>		reg[A] = reg[A] / float( <i>signedByte</i> )
FDIVRI	38	<i>signedByte</i>		reg[A] = float( <i>signedByte</i> ) / reg[A]
FPOWI	39	<i>signedByte</i>		reg[A] = reg[A] ** <i>signedByte</i>
FCMPI	3A	<i>signedByte</i>		Compare reg[A] and float( <i>signedByte</i> ), and set floating point status
FSTATUS	3B	<i>register</i>		Set floating point status for <i>register</i>
FSTATUSA	3C			Set floating point status for reg[A]
FCMP2	3D	<i>register1</i> , <i>register2</i>		Compare reg[ <i>register1</i> ] and reg[ <i>register2</i> ], and 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	<i>register</i>		reg[A] = root(reg[A], reg[ <i>register</i> ])
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	<i>register</i>		reg[A] = atan2(reg[A], reg[ <i>register</i> ])
DEGREES	4E			reg[A] = degrees(reg[A])
RADIANS	4F			reg[A] = radians(reg[A])
FMOD	50	<i>register</i>		reg[A] = reg[A] MOD reg[ <i>register</i> ]
FLOOR	51			reg[A] = floor(reg[A])
CEIL	52			reg[A] = ceil(reg[A])
ROUND	53			reg[A] = round(reg[A])
FMIN	54	<i>register</i>		reg[A] = min(reg[A], reg[ <i>register</i> ])
FMAX	55	<i>register</i>		reg[A] = max(reg[A], reg[ <i>register</i> ])
FCNV	56	<i>conversion</i>		reg[A] = conversion(reg[A])



FMAC	57	<i>register1, register2</i>		$\text{reg}[A] = \text{reg}[A] + (\text{reg}[\text{register1}] * \text{reg}[\text{register2}])$
FMSC	58	<i>register1, register2</i>		$\text{reg}[A] = \text{reg}[A] - (\text{reg}[\text{register1}] * \text{reg}[\text{register2}])$
LOADBYTE	59	<i>signedByte</i>		$\text{reg}[0 \mid 128] = \text{float}(\text{signedByte})$
LOADUBYTE	5A	<i>unsignedByte</i>		$\text{reg}[0 \mid 128] = \text{float}(\text{unsignedByte})$
LOADWORD	5B	<i>signedWord</i>		$\text{reg}[0 \mid 128] = \text{float}(\text{signedWord})$
LOADUWORD	5C	<i>unsignedWord</i>		$\text{reg}[0 \mid 128] = \text{float}(\text{unsignedWord})$
LOADE	5D			$\text{reg}[0 \mid 128] = 2.7182818$
LOADPI	5E			$\text{reg}[0 \mid 128] = 3.1415927$
FCOPYI	5F	<i>signedByte, register</i>		$\text{reg}[\text{register}] = \text{float}(\text{signedByte})$
FLOAT	60			$\text{reg}[A] = \text{float}(\text{reg}[A])$
FIX	61			$\text{reg}[A] = \text{fix}(\text{reg}[A])$
FIXR	62			$\text{reg}[A] = \text{fix}(\text{round}(\text{reg}[A]))$
FRAC	63			$\text{reg}[A] = \text{fraction}(\text{reg}[A])$
FSPLIT	64			$\text{reg}[A] = \text{integer}(\text{reg}[A]),$ $\text{reg}[0 \mid 128] = \text{fraction}(\text{reg}[A])$
SELECTMA	65	<i>register, rows, columns</i>		Select matrix A, starting at <i>register</i> . size = <i>rows</i> x <i>columns</i>
SELECTMB	66	<i>register, rows, columns</i>		Select matrix B, starting at <i>register</i> . size = <i>rows</i> x <i>columns</i>
SELECTMC	67	<i>register, rows, columns</i>		Select matrix C, starting at <i>register</i> . size = <i>rows</i> x <i>columns</i>
LOADMA	68	<i>row, column</i>		$\text{reg}[0] = \text{Matrix A}[\text{row}, \text{column}]$
LOADMB	69	<i>row, column</i>		$\text{reg}[0] = \text{Matrix B}[\text{row}, \text{column}]$
LOADMC	6A	<i>row, column</i>		$\text{reg}[0] = \text{Matrix C}[\text{row}, \text{column}]$
SAVEMA	6B	<i>row, column</i>		$\text{Matrix A}[\text{row}, \text{column}] = \text{reg}[0]$
SAVEMB	6C	<i>row, column</i>		$\text{Matrix B}[\text{row}, \text{column}] = \text{reg}[0]$
SAVEMC	6D	<i>row, column</i>		$\text{Matrix C}[\text{row}, \text{column}] = \text{reg}[0]$
MOP	6E	<i>action</i>		Matrix/Vector operation
FFT	6F	<i>action</i>		Fast Fourier Transform
WRIND	70	<i>dataType, pointer, count, value1... valueN</i>		Write multiple data values to indirect pointer
RDIND	71	<i>dataType, pointer, count</i>	<i>value1...valueN</i>	Read multiple data values from indirect pointer
DWRITE	72	<i>register, value64</i>		Write 64-bit value
DREAD	73	<i>register</i>	<i>value64</i>	Read 64-bit value
LBIT	74	<i>action, register</i>		Bit Clear, Set, Toggle, Test
SETIND	77	<i>type, {register   address   function, offset}</i>		Set indirect pointer
ADDIND	78	<i>register, unsignedByte</i>		Add to indirect pointer
COPYIND	79	<i>register1, register2, register3</i>		Copy using indirect pointers
LOADIND	7A	<i>register</i>		Load $\text{reg}[0 \mid 128]$ using indirect pointer
SAVEIND	7B	<i>register</i>		Save $\text{reg}[A]$ using indirect pointer

INDA	7C	<i>register</i>		Select register A using reg[ <i>register</i> ] value
INDX	7D	<i>register</i>		Select register X using reg[ <i>register</i> ] value
FCALL	7E	<i>function</i>		Call user-defined function stored in Flash
EVENT	7F	<i>action</i> {, <i>function</i> }		Background Events
RET	80			Return from user-defined function
BRA	81	<i>relativeOffset</i>		Unconditional branch
BRA, cc	82	<i>conditionCode</i> , <i>relativeOffset</i>		Conditional branch
JMP	83	<i>absoluteOffset</i>		Unconditional jump
JMP, cc	84	<i>conditionCode</i> , <i>absoluteOffset</i>		Conditional jump
TABLE	85	<i>tableSize</i> , <i>tableItem1...</i> <i>tableItemN</i>		Table lookup
FTABLE	86	<i>conditionCode</i> , <i>tableSize</i> , <i>tableItem1...</i> <i>tableItemN</i>		Floating point reverse table lookup
LTABLE	87	<i>conditionCode</i> , <i>tableSize</i> , <i>tableItem1...</i> <i>tableItemN</i>		Long integer reverse table lookup
POLY	88	<i>count</i> , <i>float32Value1...</i> <i>float32ValueN</i>		reg[A] = nth order polynomial
GOTO	89	<i>register</i>		Computed GOTO
RET, cc	8A	<i>conditionCode</i>		Conditional return from user-defined function
LWRITE	90	<i>register</i> , <i>int32Value</i>		Write 32-bit long integer to reg[ <i>register</i> ]
LWRITEA	91	<i>int32Value</i>		Write 32-bit long integer to reg[A]
LWRITEX	92	<i>int32Value</i>		Write 32-bit long integer to reg[X], X = X + 1
LWRITE0	93	<i>int32Value</i>		Write 32-bit long integer to reg[0   128]
LREAD	94	<i>register</i>	<i>int32Value</i>	Read 32-bit long integer from reg[ <i>register</i> ]
LREADA	95		<i>int32Value</i>	Read 32-bit long value from reg[A]
LREADX	96		<i>int32Value</i>	Read 32-bit long integer from reg[X], X = X + 1
LREAD0	97		<i>int32Value</i>	Read 32-bit long integer from reg[0   128]
LREADBYTE	98		<i>byteValue</i>	Read lower 8 bits of reg[A]
LREADWORD	99		<i>wordValue</i>	Read lower 16 bits reg[A]
ATOL	9A	<i>string</i>		Convert ASCII to long integer
LTOA	9B	<i>format</i>		Convert long integer to ASCII
LSET	9C	<i>register</i>		reg[A] = reg[ <i>register</i> ]
LADD	9D	<i>register</i>		reg[A] = reg[A] + reg[ <i>register</i> ]
LSUB	9E	<i>register</i>		reg[A] = reg[A] - reg[ <i>register</i> ]
LMUL	9F	<i>register</i>		reg[A] = reg[A] * reg[ <i>register</i> ]
LDIV	A0	<i>register</i>		reg[A] = reg[A] / reg[ <i>register</i> ] reg[0   128] = remainder

LCMP	A1	<i>register</i>		Signed compare reg[A] and reg[ <i>register</i> ], and set status
LUDIV	A2	<i>register</i>		reg[A] = reg[A] / reg[ <i>register</i> ] reg[0   128] = remainder
LUCMP	A3	<i>register</i>		Unsigned compare reg[A] and reg[ <i>register</i> ], and set long integer status
LTST	A4	<i>register</i>		Test reg[A] AND reg[ <i>register</i> ], and set long integer status
LSET0	A5			reg[A] = reg[0]
LADD0	A6			reg[A] = reg[A] + reg[0   128]
LSUB0	A7			reg[A] = reg[A] - reg[0   128]
LMUL0	A8			reg[A] = reg[A] * reg[0   128]
LDIV0	A9			reg[A] = reg[A] / reg[0   128] reg[0] = remainder
LCMP0	AA			Signed compare reg[A] and reg[0   128], and set long integer status
LUDIV0	AB			reg[A] = reg[A] / reg[0   128] reg[0] = remainder
LUCMP0	AC			Unsigned compare reg[A] and reg[0   128], and set long integer status
LTST0	AD			Test reg[A] AND reg[0   128], and set long integer status
LSETI	AE	<i>signedByte</i>		reg[A] = long( <i>signedByte</i> )
LADDI	AF	<i>signedByte</i>		reg[A] = reg[A] + long( <i>signedByte</i> )
LSUBI	B0	<i>signedByte</i>		reg[A] = reg[A] - long( <i>signedByte</i> )
LMULI	B1	<i>signedByte</i>		reg[A] = reg[A] * long( <i>signedByte</i> )
LDIVI	B2	<i>signedByte</i>		reg[A] = reg[A] / long( <i>signedByte</i> ) reg[0   128] = remainder
LCMPI	B3	<i>signedByte</i>		Signed compare reg[A] - long( <i>signedByte</i> ), and set long integer status
LUDIVI	B4	<i>unsignedByte</i>		reg[A] = reg[A] / long( <i>unsignedByte</i> ) reg[0   128] = remainder
LUCMPI	B5	<i>unsignedByte</i>		Unsigned integer compare reg[A] and long( <i>unsignedByte</i> ), and set status
LTSTI	B6	<i>unsignedByte</i>		Test reg[A] AND long( <i>unsignedByte</i> ), and set long integer status
LSTATUS	B7	<i>register</i>		Set long integer status for reg[ <i>register</i> ]
LSTATUSA	B8			Set long integer status for reg[A]
LCMP2	B9	<i>register1</i> , <i>register2</i>		Signed integer compare reg[ <i>register1</i> ], reg[ <i>register2</i> ], and set status
LUCMP2	BA	<i>register1</i> , <i>register2</i>		Unsigned integer compare reg[ <i>register1</i> ], reg[ <i>register2</i> ], and set status
LNEG	BB			reg[A] = -reg[A]
LABS	BC			reg[A] = absolute value (reg[A] )
LINC	BD	<i>register</i>		reg[ <i>register</i> ] = reg[ <i>register</i> ] + 1
LDEC	BE	<i>register</i>		reg[ <i>register</i> ] = reg[ <i>register</i> ] - 1
LNOT	BF			reg[A] = NOT reg[A]
LAND	C0	<i>register</i>		reg[A] = reg[A] AND reg[ <i>register</i> ]
LOR	C1	<i>register</i>		reg[A] = reg[A] OR reg[ <i>register</i> ]
LXOR	C2	<i>register</i>		reg[A] = reg[A] XOR reg[ <i>register</i> ]
LSHIFT	C3	<i>register</i>		reg[A] = reg[A] shift reg[ <i>register</i> ]

LMIN	C4	<i>register</i>		reg[A] = min(reg[A], reg[ <i>register</i> ])
LMAX	C5	<i>register</i>		reg[A] = max(reg[A], reg[ <i>register</i> ])
LONGBYTE	C6	<i>signedByte</i>		reg[0   128] = long( <i>signedByte</i> )
LONGUBYTE	C7	<i>unsignedByte</i>		reg[0   128] = long( <i>unsignedByte</i> )
LONGWORD	C8	<i>signedWord</i>		reg[0   128] = long( <i>signedWord</i> )
LONGUWORD	C9	<i>unsignedWord</i>		reg[0   128] = long( <i>unsignedWord</i> )
LSHIFTI	CA	<i>unsignedByte</i>		reg[A] = reg[A] shift <i>unsignedByte</i>
LANDI	CB	<i>unsignedByte</i>		reg[A] = reg[A] AND <i>unsignedByte</i>
LORI	CC	<i>unsignedByte</i>		reg[A] = reg[A] OR <i>unsignedByte</i>
SETSTATUS	CD	<i>status</i>		Set status byte
SEROUT	CE	<i>action</i> , <i>{baud}</i> / <i>{string}</i>		Serial output
SERIN	CF	<i>action</i>		Serial input
DIGIO	D0	<i>action</i> , <i>{mode}</i>		Digital I/O
ADCMODE	D1	<i>mode</i>		Set A/D trigger mode
ADCTRIG	D2			A/D manual trigger
ADCSCALE	D3	<i>channel</i>		ADCscale[ch] = reg[0]
ADCLONG	D4	<i>channel</i>		reg[0] = ADCvalue[ <i>channel</i> ]
ADCLOAD	D5	<i>channel</i>		reg[0] = float(ADCvalue[ <i>channel</i> ]) * ADCscale[ <i>channel</i> ]
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)
DEVIO	DA	<i>device</i> , <i>action</i> <i>{, ...}</i>		Device I/O
DELAY	DB	<i>period</i>		Delay (in milliseconds)
RTC	DC	<i>action</i>		Real-time Clock
SETARGS	DD			Enable FCALL argument loading
EXTSET	E0			external input count = reg[0]
EXTLONG	E1			reg[0] = external input counter
EXTWAIT	E2			wait for next external input
STRSET	E3	<i>string</i>		Copy string to string buffer
STRSEL	E4	<i>start</i> , <i>length</i>		Set selection point
STRINS	E5	<i>string</i>		Insert string at selection point
STRCMP	E6	<i>string</i>		Compare string with string selection
STRFIND	E7	<i>string</i>		Find string
STRFCHR	E8	<i>string</i>		Set field separators
STRFIELD	E9	<i>field</i>		Find field
STRTOF	EA			Convert string selection to floating point
STRTOL	EB			Convert string selection to long integer
READSEL	EC		<i>string</i>	Read string selection
STRBYTE	ED			Insert byte at selection point
STRINC	EE			Increment string selection point
STRDEC	EF			Decrement string selection point
SYNC	F0		<i>5C</i>	Get synchronization byte
READSTATUS	F1		<i>status</i>	Read status byte
READSTR	F2		<i>string</i>	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	<i>string</i>		Send string to debug trace buffer
TRACEREG	FB	<i>register</i>		Send register value to trace buffer
READVAR	FC	<i>item</i>		Read internal register value
SETREAD	FD			Set read mode
XOP	FE	<i>xop_number, byte, byte, byte</i>		Executes extended opcode instruction stored in Flash memory
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
register	register number (0-255).
register1	register number (0-255).
register2	register number (0-255).
function	function number (0-63).
byteValue	8-bit integer value.
signedByte	8-bit signed integer value.
unsignedByte	8-bit unsigned integer value.
wordValue	16-bit integer value (MSB first).
signedWord	16-bit signed integer value.
unsignedWord	16-bit unsigned integer value.
int32Value	32-bit integer value (MSB first).
float32Value	32-bit floating point value (MSB first).
status	Status byte.
string	Zero-terminated string.
baud	Baud rate and debug mode.
conditionCode	Condition code.
absoluteOffset	User-defined function offset (absolute offset).
relativeOffset	User-defined function offset (-128 to +127 from current offset).
channel	A/D channel number.
count	Byte count.
tableSize	Number of table items
tableItem1...tableItemN	Table values.
start	String of string selection.
length	Length of string selection.
item	Internal value to read.
conversion	Selects the conversion to perform.
xop_number	XOP instruction number
arg1, arg2, arg3	XOP argument bytes

## Appendix B

### Revision History

#### Release 409, 411

##### Modified Instructions

DEVIO	DEVIO, SDFAT	RDIND	WRIND
DEVIO, FIFO <sub>n</sub>	FTABLE	TABLE	
DEVIO, MEM	LTABLE	TRACESTR	

##### New Instructions

DEVIO, loadable  
devices

##### Deleted Instructions

DEVIO, VDRIVE2

#### Release 408

##### Modified Instructions

FTOA	SELECTMA	SELECTMC
READVAR	SELECTMB	XOP

#### Release 407, 406

##### Modified Instructions

DEVIO,LCD	TABLE
DEVIO,SPI	

#### Release 405

##### Modified Instructions

DEVIO,FIFO <sub>n</sub>	DEVIO,LCD	RTC	TIMESET
DEVIO,I2C	DIGIO	TICKLONG	

#### Release 404

##### Modified Instructions

MOP

#### Release 402

##### Modified Instructions

ADDIND	COPYIND	DELAY	DEVIO,COUNTER
--------	---------	-------	---------------

EVENT  
FCALL  
FFT  
FTOA  
LEFT  
LOADIND  
LOADMA

LOADMB  
LOADMC  
MOP  
READVAR  
RET  
RET,CC  
RTC

SAVEIND  
SAVEMA  
SAVEMB  
SAVEMC  
SELECTA  
SELECTMA  
SELECTMB

SELECTMC  
SERIN  
SEROUT  
SETARGS  
SETIND

## Release 401

### New Instructions

ADDIND  
COPYIND  
DELAY  
DEVIO  
DIGIO

DREAD  
DWRITE  
EVENT  
FCOPYI  
LANDI

LBIT  
LCOPYI  
LORI  
LSHIFTI  
RDIND

RTC  
SETARGS  
SETIND  
SETREAD  
WRIND

### Modified Instructions

ADCLOAD  
ADCLONG  
ADCMODE

ADCSCALE  
LOADIND  
READVAR

SAVEIND  
SERIN  
SEROUT