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 I²C interface. The uM-FPU64 chip has a 256 byte instruction buffer which allows for multiple instructions to be 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 I²C interfaces.

Instructions consist of a single opcode byte, optionally followed by additional 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.

### General Registers

<table>
<thead>
<tr>
<th>Register A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>127</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>129</td>
</tr>
<tr>
<td>130</td>
</tr>
<tr>
<td>131</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

### Temporary Registers

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T8</td>
</tr>
<tr>
<td>T9</td>
</tr>
<tr>
<td>T16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

### 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,

\[ \text{FADD, 4} \quad \text{add the value of register 4 to register A} \]

**Register X**

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

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

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

**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, register** 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, signedByte** 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.

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

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.

- **FWRITE, register, float32Value**: Write 32-bit floating point value to register
- **FWRITE0, float32Value**: Write 32-bit floating point value to reg[0] or reg[128]
- **FWRITEA, float32Value**: Write 32-bit floating point value to reg[A]
- **FWRITEX, float32Value**: Write 32-bit floating point value to reg[X]
- **DWRITE, register, float64Value**: Write 64-bit floating point value to register
- **ATOF, string**: Convert ASCII string to floating point value and store in reg[0] or reg[128]
- **LOADBYTE, signedByte**: Convert signed byte to floating point and store in reg[0] or reg[128]
- **LOADUBYTE, unsignedByte**: Convert unsigned byte to floating point and store in reg[0] or reg[128]
- **LOADWORD, signedWord**: Convert signed 16-bit value to floating point and store in reg[0] or reg[128]
- **LOADUWORD, unsignedWord**: 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]
- **FCOPYI, unsignedByte, register**: Convert signed 8-bit value to floating point and store in register
Overview

Micromega Corporation
5
uM-FPU64 Instruction Set - Release 411

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

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

Additional Floating Point Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEG</td>
<td>SIN</td>
</tr>
<tr>
<td>FABS</td>
<td>COS</td>
</tr>
<tr>
<td>FINV</td>
<td>TAN</td>
</tr>
<tr>
<td>SQRT</td>
<td>ASIN</td>
</tr>
<tr>
<td>ROOT,register</td>
<td>ACOS</td>
</tr>
<tr>
<td>LOG</td>
<td>ATAN</td>
</tr>
<tr>
<td>LOG10</td>
<td>ATAN2,register</td>
</tr>
<tr>
<td>EXP</td>
<td>DEGREES</td>
</tr>
<tr>
<td>EXP10</td>
<td>RADIANS</td>
</tr>
</tbody>
</table>

Matrix Instructions

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

Fast Fourier Transform Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT, action</td>
<td>Perform Fast Fourier Transform operation</td>
</tr>
</tbody>
</table>

Conversion Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>Convert reg[A] from long integer to floating point</td>
</tr>
<tr>
<td>FIX</td>
<td>Convert reg[A] from floating point to long integer</td>
</tr>
<tr>
<td>FIXR</td>
<td>Convert reg[A] from floating point to long integer (with rounding)</td>
</tr>
<tr>
<td>FSMIT</td>
<td>Set reg[A] = integer value, reg[0] or reg[128] = fractional value</td>
</tr>
</tbody>
</table>
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, 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.

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

Loading Long Integer Values

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

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

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

Additional Long Integer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTATUS, register</td>
<td>LCMP2, register1, register2</td>
</tr>
<tr>
<td>LSTATUSA</td>
<td>LUCMP2, register1, register2</td>
</tr>
<tr>
<td>LNEG</td>
<td>LMIN, register</td>
</tr>
<tr>
<td>LABS</td>
<td>LMAX, register</td>
</tr>
<tr>
<td>LNOT</td>
<td>LSHIFT, register</td>
</tr>
<tr>
<td>LINC, register</td>
<td>LSHIFTI, signedByte</td>
</tr>
<tr>
<td>LDEC, register</td>
<td>LBIT, unsignedByte, register</td>
</tr>
</tbody>
</table>
Overview

Micromega Corporation
uM-FPU64 Instruction Set - Release 411

General Purpose Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTA, register</td>
<td>ALLODX</td>
</tr>
<tr>
<td>SELECTX, register</td>
<td>COPY, register1, register2</td>
</tr>
<tr>
<td>CLR, register</td>
<td>COPY0, register</td>
</tr>
<tr>
<td>CLRA</td>
<td>COPYA, register</td>
</tr>
<tr>
<td>CLRX</td>
<td>COPYX, register</td>
</tr>
<tr>
<td>CLR0</td>
<td>SWAP, register1, register2</td>
</tr>
<tr>
<td>LOAD, register</td>
<td>SWAPA, register</td>
</tr>
<tr>
<td>LOADA</td>
<td>SETSTATUS, unsignedbyte</td>
</tr>
<tr>
<td>LOADX</td>
<td>READSTATUS</td>
</tr>
<tr>
<td>XSAVE, register</td>
<td>SETREAD</td>
</tr>
<tr>
<td>XSAVEA</td>
<td>SYNC</td>
</tr>
<tr>
<td>INDA</td>
<td>VERSION</td>
</tr>
<tr>
<td>INDX</td>
<td>IEEEMODE</td>
</tr>
<tr>
<td>LEFT</td>
<td>SETARGS</td>
</tr>
<tr>
<td>RIGHT</td>
<td></td>
</tr>
<tr>
<td>READVAR, item</td>
<td>CHECKSUM</td>
</tr>
<tr>
<td>RESET</td>
<td>XOP</td>
</tr>
</tbody>
</table>

Special Purpose Instructions

Indirect Pointer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETIND, type, {register</td>
<td>Set indirect pointer</td>
</tr>
<tr>
<td>ADDIND, register,</td>
<td>Add to indirect pointer</td>
</tr>
<tr>
<td>COPYIND, fromPtr,</td>
<td>Copy using indirect pointers</td>
</tr>
<tr>
<td>LOADIND, register</td>
<td>Load reg[0</td>
</tr>
<tr>
<td>SAVEIND, register</td>
<td>Save reg[A] using indirect pointer</td>
</tr>
<tr>
<td>RDIND, type, count</td>
<td>Read multiple data values from indirect pointer</td>
</tr>
<tr>
<td>WRIND, type, count,</td>
<td>Write multiple data values to indirect pointer</td>
</tr>
</tbody>
</table>

Stored Function Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCALL, function</td>
<td>Call user-defined function stored in Flash</td>
</tr>
<tr>
<td>RET</td>
<td>Return from user-defined function</td>
</tr>
<tr>
<td>RET, conditionCode</td>
<td>Conditional return from user-defined function</td>
</tr>
<tr>
<td>BRA, relativeOffset</td>
<td>Unconditional branch inside user-defined function</td>
</tr>
<tr>
<td>BRA, conditionCode,</td>
<td>Conditional branch inside user-defined function</td>
</tr>
<tr>
<td>JMP, absoluteOffset</td>
<td>Unconditional jump inside user-defined function</td>
</tr>
<tr>
<td>JMP, conditionCode,</td>
<td>Conditional jump inside user-defined function</td>
</tr>
<tr>
<td>GOTO, register</td>
<td>Computed goto</td>
</tr>
<tr>
<td>TABLE, tableSize,</td>
<td>Table lookup</td>
</tr>
<tr>
<td>FTABLE, conditionCode</td>
<td>Floating point reverse table lookup</td>
</tr>
<tr>
<td>LTABLE, conditionCode</td>
<td>Long integer reverse table lookup</td>
</tr>
<tr>
<td>POLY, count, floatValue</td>
<td>Nth order polynomial</td>
</tr>
</tbody>
</table>

Background Event Processing

EVENT, action{,function} | Background event processing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCMODE, mode</td>
<td>Select A/D trigger mode</td>
</tr>
<tr>
<td>ADCTRIG</td>
<td>Manual A/D trigger</td>
</tr>
<tr>
<td>ADCSCALE, channel</td>
<td>Set A/D floating point scale factor</td>
</tr>
<tr>
<td>ADCLONG, channel</td>
<td>Get raw long integer A/D reading</td>
</tr>
<tr>
<td>ADCLOAD, channel</td>
<td>Get scaled floating point A/D reading</td>
</tr>
</tbody>
</table>
Overview

**ADCW A I T**  
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

ST R FCHR, 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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td>I</td>
<td>N</td>
<td>S</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Assembler Symbol</th>
<th>Test Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>NZ</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
<tr>
<td>LT</td>
<td>0x72</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>0x62</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>0x70</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>0x60</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>PZ</td>
<td>0x71</td>
<td>Positive Zero</td>
</tr>
<tr>
<td>MZ</td>
<td>0x73</td>
<td>Negative Zero</td>
</tr>
<tr>
<td>INF</td>
<td>0xC8</td>
<td>Infinity</td>
</tr>
<tr>
<td>FIN</td>
<td>0xC0</td>
<td>Finite</td>
</tr>
<tr>
<td>PINF</td>
<td>0xE8</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>MINF</td>
<td>0xEA</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>0x44</td>
<td>Not-a-Number (NaN)</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x00</td>
<td>True</td>
</tr>
<tr>
<td>FALSE</td>
<td>0xFF</td>
<td>False</td>
</tr>
</tbody>
</table>
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,
  \[ \text{reg}[0] = (\text{float}(\text{ADCvalue[channel]}) \times \text{ADCscale[channel]}) + \text{ADCoffset[channel]} \]

if reg[A] is 64-bit,
  \[ \text{reg}[128] = (\text{float}(\text{ADCvalue[channel]}) \times \text{ADCscale[channel]}) + \text{ADCoffset[channel]} \]

Opcodes:     D5

Byte 2:     channel

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td></td>
</tr>
</tbody>
</table>

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
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

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 5</th>
<th>Size</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0x00</td>
<td>Stores the analog reading in register 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td>0x20</td>
<td>Stores the size of the memory buffer in register 0. Used in block mode.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 4</th>
<th>Pointer</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>0x00</td>
<td>Stores the analog reading in register 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTR</td>
<td>0x10</td>
<td>Stores a pointer to the memory buffer register 0. Used in block mode.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 3:0</th>
<th>Channel</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 5</td>
<td>28-pin chip (AN0 to AN5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 to 8</td>
<td>44-pin chip (AN0 to AN8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:  \texttt{ADCMODE, mode}

Description:  Set the trigger mode of the A/D converter. The \textit{mode} is interpreted as follows:

\textbf{Opcode: D1}

\textbf{Byte 2: mode}

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
Action & & & & & & & & \\
Options & & & & & & & & \\
\hline
\end{tabular}

Bits 7:4  \textbf{Action}

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE</td>
<td>0x00</td>
<td>Disable analog conversions</td>
</tr>
<tr>
<td>MANUAL</td>
<td>0x10</td>
<td>Manual trigger</td>
</tr>
<tr>
<td>EXTIN</td>
<td>0x20</td>
<td>External trigger</td>
</tr>
<tr>
<td>TIMER</td>
<td>0x30</td>
<td>Timer trigger</td>
</tr>
<tr>
<td>EXTIN_BLK</td>
<td>0x40</td>
<td>External trigger, block mode</td>
</tr>
<tr>
<td>TIMER_BLK</td>
<td>0x50</td>
<td>Timer trigger, block mode</td>
</tr>
<tr>
<td>CHANNELS</td>
<td>0x60</td>
<td>Maximum number of ADC channels</td>
</tr>
<tr>
<td>VREF</td>
<td>0x70</td>
<td>Select voltage reference</td>
</tr>
</tbody>
</table>

Bits 3:0  \textbf{Options}

\textit{See descriptions below.}

\textbf{DISABLE}

\texttt{ADCMODE, DISABLE}

Disable analog conversions.

\begin{tabular}{|c|c|c|}
\hline
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & & & & & & & & \\
\hline
\end{tabular}

\textbf{MANUAL}

\texttt{ADCMODE, MANUAL+repeat}

Manual trigger, single sample with repeat.

\begin{tabular}{|c|c|c|}
\hline
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
1 & & & & & & & & \\
\hline
Repeat & & & & & & & & \\
\hline
\end{tabular}

Bits 3:0  \textbf{Repeat Count}

\begin{tabular}{|c|c|c|}
\hline
Value & 0 to 15 & Description \\
\hline
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. \\
\hline
\end{tabular}

\textbf{EXTIN}

\texttt{ADCMODE, EXTIN+repeat}

External trigger, single sample with repeat.
<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bits 3:0</strong> Repeat Count</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 15</td>
<td>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.</td>
</tr>
</tbody>
</table>

**TIMER**

ADCMODE, TIMER+repeat

Timer trigger, single sample with repeat.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bits 3:0</strong> Repeat Count</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 15</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>-</th>
</tr>
</thead>
</table>

**TIMER_BLK**

ADCMODE, TIMER_BLK

Timer trigger, block mode with continuous sampling.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>-</th>
</tr>
</thead>
</table>

**CHANNELS**

ADCMODE, CHANNELS, max_channel

Sets the maximum number of ADC channels.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bits 3:0</strong> max_channel</td>
<td></td>
</tr>
</tbody>
</table>
### Value Description

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5</td>
<td>28-pin chip (AN0 to AN5)</td>
</tr>
<tr>
<td>0 to 8</td>
<td>44-pin chip (AN0 to AN8)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>AVSS</td>
<td>0x00</td>
<td>AVSS is used as VREF-</td>
</tr>
<tr>
<td>6</td>
<td>AN1</td>
<td>0x02</td>
<td>AN1 is used as VREF-</td>
</tr>
</tbody>
</table>

#### Bit 1 VREF-

<table>
<thead>
<tr>
<th>Bit</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AVDD</td>
<td>0x00</td>
<td>AVDD is used as VREF+</td>
</tr>
<tr>
<td>0</td>
<td>AN0</td>
<td>0x01</td>
<td>AN0 is used as VREF+</td>
</tr>
</tbody>
</table>

**Examples:**

- **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,

\[
\text{ADCScale}[\text{channel}] = \text{reg}[0] \quad \text{or} \quad \text{ADCoffset}[\text{channel}] = \text{reg}[0]
\]

if reg[A] is 64-bit,

\[
\text{ADCScale}[\text{channel}] = \text{reg}[128] \quad \text{or} \quad \text{ADCoffset}[\text{channel}] = \text{reg}[128]
\]

**Opcode:** D3

**Byte 2:** channel
Bit 7 6 5 4 3 2 1 0
- O Channel

Bit 4                  Scale/Offset
IDE Symbol      IDE Value     Description
-             0x00          Sets scale value.
OFFSET         0x10          Sets offset value.

Bits 3:0      Channel
Value     Description
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 >= 0xFFFFFF, reg[0|128] (bits 23:0) is set to 0xFFFFFF

**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.

reg[A] = asin(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.

reg[A] = atan(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.

reg[A] = atan(reg[A] / reg[register])

*Opcode:*  
4D

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

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

\[ \text{ATOF}, \text{string} \]

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

**Opcode:**

\[ \text{1E} \]

**Byte 2:**

\[ \text{string} \]

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)
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, "2.54"  stores the value 2.54 in register 0 or 128
ATOL, "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:   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:

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>NZ</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
<tr>
<td>LT</td>
<td>0x72</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>0x62</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>0x70</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>0x60</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>PZ</td>
<td>0x71</td>
<td>Positive Zero</td>
</tr>
<tr>
<td>MZ</td>
<td>0x73</td>
<td>Negative Zero</td>
</tr>
<tr>
<td>INF</td>
<td>0xC8</td>
<td>Infinity</td>
</tr>
<tr>
<td>FIN</td>
<td>0xC0</td>
<td>Finite</td>
</tr>
<tr>
<td>PINF</td>
<td>0xE8</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>MINF</td>
<td>0xEA</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>0x44</td>
<td>Not-a-Number (NaN)</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x00</td>
<td>True</td>
</tr>
<tr>
<td>FALSE</td>
<td>0xFF</td>
<td>False</td>
</tr>
</tbody>
</table>

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**
**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: \textit{COPY,fromRegister,toRegister}

Description: Copy the value from \textit{fromRegister} to \textit{toRegister}.

\[ \text{reg[toRegister]} = \text{reg[fromRegister]}, \text{status} = \text{longStatus(reg[toRegister])} \]

Opcode: 07

Byte 2: \textit{fromRegister}
Register number (0 to 255).

Byte 3: \textit{toRegister}
Register number (0 to 255).

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

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

COPYA

Copy register A

Syntax: \textit{COPYA,register}

Description: Copy the value of register A to \textit{register}.

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

Opcode: 08

Byte 2: \textit{register}
Register number (0 to 255).

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

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

COPYIND

Copy using Indirect Pointers

Syntax: \textit{COPYIND,fromRegister,toRegister,countRegister}

Description: The number of data items specified by the \textit{countRegister} are copied from the location pointed to by \textit{fromRegister} to the location pointed to by \textit{toRegister}. See the SETIND instruction for a description of pointers.
 Opcode: 79

Byte 2: fromRegister
Register number (0 to 255). The register contains the from pointer.

Byte 3: toRegister
Register number (0 to 255). The register contains the to pointer.

Byte 4: countRegister
Register number (0 to 255). The register contains the number of items to copy.

See Also: SETIND, ADDIND, WRIND, RDIND, LOADIND, SAVEIND

COPYX Copy register X

Syntax: COPYX, register

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

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

Opcode: 09

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

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

See Also: COPY, COPYA, COPY0, FCOPYI, LCOPYI

COPY0 Copy register 0

Syntax: COPY0, register

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

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

Opcode: 10

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[128] are copied
**COS**

**Cosine**

**Syntax:**

```cos```

**Description:**

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

\[
\text{reg}[A] = \cos(\text{reg}[A])
\]

**Opcode:**

48

**Special Cases:**

- If \(\text{reg}[A]\) is NaN or an infinity, then the result is NaN

**See Also:**

ACOS, ASIN, ATAN, ATAN2, COS, 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 \(\text{reg}[A]\) 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, FC 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_REG8{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, WRITE_REG16{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, WRITE_REG32{+MSB}{+LSB}{+ZE}{+SE}, register
DEVIO, device, WRITE_REG64{+MSB}{+LSB}{+ZE}{+SE}, register
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, LOAD_DEVICE, xopdev
```

**Opcode:** DA

**Byte 2:** device

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Device Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 7:4</td>
<td>Device Type</td>
</tr>
<tr>
<td>IDE Symbol</td>
<td>IDE Value</td>
</tr>
<tr>
<td>MEM</td>
<td>0x00</td>
</tr>
<tr>
<td>FIFO1</td>
<td>0x01</td>
</tr>
<tr>
<td>FIFO2</td>
<td>0x02</td>
</tr>
</tbody>
</table>
### Device Number

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 15</td>
<td>Device number (for device types that support multiple devices)</td>
</tr>
</tbody>
</table>

#### 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)**

DEVI0, \texttt{device}, \texttt{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)**

DEVI0, \texttt{device}, \texttt{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.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{1} \texttt{-} \texttt{L} Bits</td>
</tr>
</tbody>
</table>

**Bit 3 \textit{Byte Order}**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>0x00</td>
<td>Most significant byte first</td>
</tr>
<tr>
<td>LSB</td>
<td>0x04</td>
<td>Least significant byte first</td>
</tr>
</tbody>
</table>

**Bits 1:0 \textit{Number of Bits}**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_REG8</td>
<td>0x10</td>
<td>1 byte (8-bit)</td>
</tr>
<tr>
<td>WRITE_REG16</td>
<td>0x11</td>
<td>2 bytes (16-bit)</td>
</tr>
<tr>
<td>WRITE_REG32</td>
<td>0x12</td>
<td>3 bytes (32-bit)</td>
</tr>
<tr>
<td>WRITE_REG64</td>
<td>0x13</td>
<td>4 bytes (64-bit)</td>
</tr>
</tbody>
</table>

**Byte 4:** \texttt{register}

Register number (0 to 255).

**Write Byte (0x20)**

DEVI0, \texttt{device}, \texttt{WRITE\_BYTE, byte}

Write the 8-bit value specified to the device.

**Byte 4:** \texttt{byte}

Unsigned 8-bit value to write to device.

**Write Word (0x21)**

DEVI0, \texttt{device}, \texttt{WRITE\_WORD, word}

Write the 16-bit value specified to the device.

**Bytes 4-5:** \texttt{word}

Unsigned 16-bit value to write to device.

**Write Multiple Bytes (0x22)**

DEVI0, \texttt{device}, \texttt{WRITE\_NBYTE, count, byte, ...}

Write the number of bytes specified by \texttt{count} to the device.

**Byte 4:** \texttt{count}

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

**Bytes 5-n:** \texttt{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: \textit{regAddr}

The lower 16-bits of the register specify the memory address to write to.

Byte 5: \textit{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)

\texttt{DEVIO, device, READ\_REG\{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)

\texttt{DEVIO, device, READ\_REG\{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)

\texttt{DEVIO, device, READ\_REG\{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)

\texttt{DEVIO, device, READ\_REG\{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 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: \textit{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: \[ \text{DEVIO}, \text{ASYNC}, \text{action}\{,\ldots\} \]

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.

\[
\begin{align*}
\text{DEVIO, ASYNC, DISABLE} \\
\text{DEVIO, ASYNC, ENABLE, pin, config}
\end{align*}
\]

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)
   \[ \text{DEVIO, ASYNC, DISABLE} \]
   Disable the asynchronous serial connection and release the digital pins.

Enable (0x01)
   \[ \text{DEVIO, ASYNC, ENABLE, pin, config} \]
   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
   \[ \text{pin} \text{ Rx} \]
   Transmit only
   \[ \text{pin} \text{ Tx} \]
   Receive and Transmit
   \[ \text{pin} \text{ Rx} \]
   \[ \text{pin+1} \text{ Tx} \]
   Receive and Transmit with Flow Control
   \[ \text{pin} \text{ Rx} \]
   \[ \text{pin+1} \text{ Tx} \]
   \[ \text{pin+2} \text{/CTS} \]
   \[ \text{pin+3} \text{/RTS} \]

Byte 5: config
   Bit 7 6 5 4 3 2 1 0
   \[ \text{Type Baud Rate} \]
   
   Bits 5:4 Connection Type
### IDE Symbol | IDE Value | Description
--- | --- | ---
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 Baud Rate

| IDE Symbol | IDE Value | Description |
--- | --- | ---
- | - | 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 | 0xA | 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

- `DISABLE`
- `ENABLE, pin, config`
- `DEBOUNCE, period`
- `REPEAT, delay, rate`
- `READ_COUNT`
- `EDGE1_MSEC`
- `EDGE1_USEC`
- `EDGE2_MSEC`
- `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

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Event Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IDE Symbol</td>
</tr>
<tr>
<td>LOW</td>
<td>0x00</td>
</tr>
<tr>
<td>HIGH</td>
<td>0x80</td>
</tr>
</tbody>
</table>

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)
DEVI0, COUNTER+n, READ_COUNT
Returns the counter value in register 0.

Read Active Edge time in milliseconds (0x05)
DEVI0, COUNTER+n, EDGE1_MSEC
Returns the time in milliseconds (32-bit value) in register 0 or 128.

Read Active Edge time in microseconds (0x06)
DEVI0, 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)
DEVI0, 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)
DEVI0, COUNTER+n, EDGE2_USEC
Returns the time in microseconds (64-bit value) in register 0 or 128.

**DEVIO, FIFO**

**FIFO Buffer Interface**

**Syntax:**

DEVI0,FIFO1,action{,...}
DEVI0,FIFO2,action{,...}
DEVI0,FIFO3,action{,...}
DEVI0,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:

DEVI0, MEM, ALLOCATE, memSize, fifoSize
DEVI0, FIFO1, ALLOC_MEM, size
DEVI0, FIFO1, ALLOC_MEMR, regSize

DEVI0, FIFO1, DISABLE
DEVI0, FIFO1, ENABLE, pin, config
DEVI0, FIFO1, CLEAR
DEVI0, FIFO1, USED
DEVI0, FIFO1, FREE
DEVI0, FIFO1, STATUS
DEVI0, FIFO1, CLEAR_OVERFLOW
DEVI0, FIFO1, ALLOC_MEM, size
DEVI0, FIFO1, 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 DEVI0 description.
Disable (0x00)
DEVIO, FIFO, DISABLE
Disables the specified FIFO device.

Enable (0x01)
DEVIO, FIFO, ENABLE, pin, config
Initialize the FIFO.

Byte 4:
pin
Unused.

Byte 5:
config

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Type</td>
</tr>
</tbody>
</table>

Bits 2:0 Event Type

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_EVENT</td>
<td>0x00</td>
<td>No event</td>
</tr>
<tr>
<td>EMPTY</td>
<td>0x01</td>
<td>Set event flag when buffer is empty.</td>
</tr>
<tr>
<td>NOT_EMPTY</td>
<td>0x02</td>
<td>Set event flag when data in buffer.</td>
</tr>
<tr>
<td>HALF_EMPTY</td>
<td>0x03</td>
<td>Set event flag when buffer is half empty.</td>
</tr>
<tr>
<td>HALF_FULL</td>
<td>0x04</td>
<td>Set event flag when buffer is half full.</td>
</tr>
<tr>
<td>FULL</td>
<td>0x05</td>
<td>Set event flag when buffer is full.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>0x06</td>
<td>Set event flag when buffer overflows.</td>
</tr>
</tbody>
</table>

Clear Buffer (03)
DEVIO, FIFO, CLEAR
Clear the buffer by resetting the input index and output index.

Get Number of Bytes Used (0x04)
DEVIO, FIFO, 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, FIFO, FREE
Get the number of bytes currently available in the memory buffer and returns the value in register 0.

Get Buffer Status (0x06)
DEVIO, FIFO, STATUS
Get the current status of the memory buffer. The buffer status is returned in register 0.
The status byte is as follows:

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFH[HE] - Type</td>
</tr>
</tbody>
</table>

| 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, FIFO, 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, FIFO, ALLOC_MEM, size
The number of memory bytes specified by size are allocated from the dynamic allocation for use by FIFO. If not enough bytes are available in the dynamic allocation, the size of FIFO 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 FIFO.

Dynamic Memory Allocation (0x09)
DEVIO, FIFO, 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 FIFO 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 FIFO.

DEVIO, I2C I2C Bus Interface

Syntax: DEVIO,I2C,action{,...}

Description: This instruction provides support for communicating with IC devices using a local IC 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 IC devices are shown below. For actions that are common to all devices, see the DEVIO description.

Disable (0x00)
DEVIO, I2C, DISABLE
Disable the IC bus and release the digital pins.
Enable (0x01)

DEVIO, I2C, ENABLE, pin, config

Selects the pins to use for the I2C bus, the bus speed, and enables the I2C 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 0: Speed
    - IDE Symbol: ID E Value
    - SLOW: 0x00
    - FAST: 0x01

Start Write (0x02)

DEVIO, I2C, START_WRITE, device

Sends the start write sequence to the I2C bus, and sets the status bits for the acknowledge.

**Byte 4:**

- **device**
  - Unsigned byte specifying the 8-bit I2C device address. (7-bit I2C 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 I2C 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 I2C 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:  \texttt{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.

\begin{verbatim}
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
\end{verbatim}

Opcode: \texttt{DA}

Byte 2: \texttt{LCD (0x80)}

Byte 3: \texttt{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.

Enable (0x01)
\texttt{DEVIO, LCD, ENABLE, pin, config}
Selects the pins to use for the LCD, configures and initializes the display.

Byte 4: \texttt{pin}
Specifies the pins to use for the LCD interface.
\begin{itemize}
\item D0 to D8  28-pin uM-FPU64 chip
\item D0 to D22 44-pin uM-FPU64 chip
\end{itemize}

Pin Assignments
\begin{itemize}
\item \texttt{pin to pin+3}  4-bit data
\item \texttt{pin+4}  E pin
\item \texttt{pin+5}  RS pin
\item \texttt{pin+6}  RW pin (if enabled)
\end{itemize}

Byte 5: \texttt{config}

\begin{tabular}{ccccc}
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\hline
\texttt{R} & \texttt{F} & Row & Col
\end{tabular}
Bit 6  **RW pin setting**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>Read disabled, RW pin grounded</td>
</tr>
<tr>
<td>READ_ENABLED</td>
<td>0x40</td>
<td>Read enabled, RW pin required</td>
</tr>
</tbody>
</table>

Bit 5  **Font**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>5x7 font</td>
</tr>
<tr>
<td>FONT_5x10</td>
<td>0x20</td>
<td>5x10 font</td>
</tr>
</tbody>
</table>

Bits 4:3  **Row**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWS_1</td>
<td>0x00</td>
<td>1 row</td>
</tr>
<tr>
<td>ROWS_2</td>
<td>0x08</td>
<td>2 rows</td>
</tr>
<tr>
<td>ROWS_4</td>
<td>0x10</td>
<td>4 rows</td>
</tr>
</tbody>
</table>

Bits 2:0  **Column**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLS_8</td>
<td>0x00</td>
<td>8 columns</td>
</tr>
<tr>
<td>COLS_12</td>
<td>0x01</td>
<td>12 columns</td>
</tr>
<tr>
<td>COLS_16</td>
<td>0x02</td>
<td>16 columns</td>
</tr>
<tr>
<td>COLS_20</td>
<td>0x03</td>
<td>20 columns</td>
</tr>
<tr>
<td>COLS_40</td>
<td>0x04</td>
<td>40 columns</td>
</tr>
</tbody>
</table>

**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: \textit{command}
Unsigned byte specifying the HD44780U compatible LCD command.

Device Type (0x07)
\texttt{DEVIO, LCD, INTERFACE, type}
Used to select an I2C controller for the LCD interface. If this command is not used, digital I/O pins are used for a direct interface. A \texttt{devio(I2C, ENABLE, ...)} function call must be done previously, and this function call must be done before the \texttt{devio(LCD, ENABLE, ...)} function call.

Byte 4:
\textit{type}

Bit 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>Device</th>
<th>Address</th>
</tr>
</thead>
</table>

Bits 6:4  \textbf{Device}
\textbf{IDE Symbol} | \textbf{IDE Value} | \textbf{Description}
- | 0x00 | digital pins (default)
ST7036 | 0x10 | I2C controller
PCF8574 | 0x28 | I2C controller
MCP23008 | 0x30 | I2C controller

Bits 3:0  \textbf{Address}
Lower bits of I2C 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)
\texttt{DEVIO, LCD, BACKLIGHT_ON}
When a PCF8574 or MCP23008 controller is used, this command turns on the backlight.

Turn BackLight Off (0x09)
\texttt{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,FIFOn,ALLOC_MEM,regSize, DEVIO,FIFOn,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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreground Memory Allocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Background Memory Allocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Byte 5-6:**

`fifoSize` *(unsigned word)*

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Memory Allocation Codes**

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

**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 \texttt{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.

\textbf{Byte 4:} \hfill \texttt{0}

\textbf{Byte 5-6:} \hfill \texttt{fgSize (unsigned word)}

Unsigned 16-bit word specifying the number of memory bytes to allocate to the
foreground.

---

**DEVIO, OWIRE**  
1-Wire Bus Interface

\textbf{Syntax:} \hfill \texttt{DEVIO,OWIRE,action\{,…\}}

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

\begin{verbatim}
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
\end{verbatim}

\textbf{Opcode:} \hfill DA

\textbf{Byte 2:} \hfill OWIRE (0x10–0x1F)

\textbf{Byte 3:} \hfill \textit{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 \texttt{DEVIO} description.

\textbf{Disable (0x00)}  
\texttt{DEVIO, OWIRE, DISABLE}

Disable the 1-wire bus and release the digital pin.

\textbf{Enable (0x01)}  
\texttt{DEVIO, OWIRE, ENABLE, pin, config}

The enable action is used to assign the digital pin for the 1-wire bus and initialize the bus.

\textbf{Byte 4:} \hfill \textit{pin}

\begin{itemize}
  \item D0 to D8 \quad \text{28-pin uM-FPU64 chip}
  \item D0 to D22 \quad \text{44-pin uM-FPU64 chip}
\end{itemize}

\textbf{Pin Assignments}  
\begin{itemize}
  \item \texttt{pin} \quad \text{1-Wire bus}
\end{itemize}

\textbf{Byte 5:} \hfill \textit{config}

Not used.

---

Micromega Corporation
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: \textit{regAddr}

The first of \textit{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.

\textbf{DEVIO, SDFAT} \quad SD card with FAT16 and FAT32 support

\textbf{Syntax:}

```
DEVIO, SDFAT, action{,…}
```

\textbf{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
```

\textbf{Opcode:} \quad DA

\textbf{Byte 2:} \quad SDFAT (0xA0)

\textbf{Byte 3:} \quad \textit{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 \textit{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*

<table>
<thead>
<tr>
<th>Bit</th>
<th>Write Block Enable</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>-</td>
<td>0x00</td>
<td>WRITE_BLOCK action disabled</td>
</tr>
<tr>
<td>6</td>
<td>WRBLK_ENABLE</td>
<td>WRBLKN</td>
<td>0x80</td>
<td>WRITE_BLOCK action disabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 1:0</th>
<th>Partition</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Use partition 1 on SD card (default)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0x00</td>
<td>0x00</td>
<td>Use partition 2 on SD card (default)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0x01</td>
<td>0x01</td>
<td>Use partition 3 on SD card (default)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>0x02</td>
<td>0x02</td>
<td>Use partition 4 on SD card (default)</td>
</tr>
</tbody>
</table>

**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` 

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

Bit 7

**Extended Mode**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>Normal, Pulse widths (800 usec to 2200 usec).</td>
</tr>
<tr>
<td>EXTENDED</td>
<td>0x80</td>
<td>Extended Mode, Pulse widths (500 usec to 2500 usec).</td>
</tr>
</tbody>
</table>

Bits 2:0

**Event Number**

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_EVENT</td>
<td>0x00</td>
<td>No event.</td>
</tr>
<tr>
<td>EVENT1</td>
<td>0x01</td>
<td>Event 1.</td>
</tr>
<tr>
<td>EVENT2</td>
<td>0x02</td>
<td>Event 2.</td>
</tr>
<tr>
<td>EVENT3</td>
<td>0x03</td>
<td>Event 3.</td>
</tr>
<tr>
<td>EVENT4</td>
<td>0x04</td>
<td>Event 4.</td>
</tr>
<tr>
<td>EVENT5</td>
<td>0x05</td>
<td>Event 5.</td>
</tr>
<tr>
<td>EVENT6</td>
<td>0x06</td>
<td>Event 6.</td>
</tr>
<tr>
<td>EVENT7</td>
<td>0x07</td>
<td>Event 7.</td>
</tr>
</tbody>
</table>

**Set Pulse Width (0x02)**

`DEVIO, SERVO+n, PULSE, register`

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

**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.

**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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Servo 3 Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 2</td>
<td>Servo 2 Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 1</td>
<td>Servo 1 Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 0</td>
<td>Servo 0 Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Device 1-15

```
pin  /CS  Slave chip select (from FPU)
```
Byte 5:  

config

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Mode</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit 7

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>Hold MOSI low during read.</td>
</tr>
<tr>
<td>READ_HIGH</td>
<td>0x80</td>
<td>Hold MOSI high during read.</td>
</tr>
</tbody>
</table>

Bits 6:5

Clock Mode

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

Bits 4:0

Speed

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

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:**

\[ \text{DIGIO, action\{,mode\}} \]

**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:**

\[ \text{D0} \]

**Byte 2:**

\[ \text{action} \]

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

<table>
<thead>
<tr>
<th>Bits 4:0</th>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>D0 - D8</td>
<td>The 28-pin chip has digital pins D0 to D8.</td>
</tr>
<tr>
<td></td>
<td>D0 - D22</td>
<td>The 44-pin chip has digital pins D0 to D22.</td>
</tr>
</tbody>
</table>

**Set Pin Low**

\[ \text{DIGIO, LOW+pin} \]

The pin is configured as an output and set low.

**Set Pin High**

\[ \text{DIGIO, HIGH+pin} \]

The pin is configured as an output and set high.

**Toggle Pin**

\[ \text{DIGIO, TOGGLE+pin} \]

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

**Read Input from Pin**

\[ \text{DIGIO, INPUT+pin} \]

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)

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>M</td>
<td>F</td>
<td># of Bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bit 7**  
Sample Time (READ_BITS)

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRE</td>
<td>0x00</td>
<td>Sample pin level before clock pulse.</td>
</tr>
<tr>
<td>POST</td>
<td>0x80</td>
<td>Sample pin level after clock pulse.</td>
</tr>
</tbody>
</table>

**Bit 6**  
Bit order

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>0x00</td>
<td>Read most significant bit first.</td>
</tr>
<tr>
<td>LSB</td>
<td>0x40</td>
<td>Read least significant bit first.</td>
</tr>
</tbody>
</table>

**Bit 5**  
Clock Speed

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>0x00</td>
<td>Fast speed clock (743 kHZ).</td>
</tr>
<tr>
<td>SLOW</td>
<td>0x20</td>
<td>Slow speed clock (534 kHZ).</td>
</tr>
</tbody>
</table>

**Bits 4:0**  
Number of Bits

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 31</td>
<td>The number of bits to transfer. A value of 0 specifies 32 bits.</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>-</td>
<td>-</td>
<td># of Bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bit 7**  
Invert

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
</table>
- - Bits are not inverted.

**INVERT** 0x40 Bits are inverted before write and after read.

<table>
<thead>
<tr>
<th>Bits 4:0</th>
<th>Number of Bits</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 9</td>
<td>The 28-pin chip has 9 digital pins (D0 to D8).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 - 23</td>
<td>The 44-pin chip has 23 digital pins (D0 to D22).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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**.

\[
\text{reg}[	ext{register}] = 64\text{-bit value, status} = \text{longStatus}(	ext{reg}[	ext{register}])
\]
Opcode: \texttt{72}

Byte 2: \texttt{register}

Register number (0 to 255).

Bytes 3-10: \texttt{value}

64-bit value represented by eight bytes (MSB first).

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

See Also: \texttt{FWRITE, FWRITE0, FWRITEA, FWRITEX, LWRITE, LWRITE0, LWRITEA, LWRI

EVENT Background events

Syntax: \texttt{EVENT,action{,function}}

Description: Used to manage background events.

Opcode: \texttt{7F}

Byte 2: \texttt{action}

<table>
<thead>
<tr>
<th>Bit 7:6:5:4:3:2:1:0</th>
<th>Action</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{DISABLE}</td>
<td>0x00</td>
<td>Disable event.</td>
</tr>
<tr>
<td>\texttt{ENABLE}</td>
<td>0x10</td>
<td>Enable event.</td>
</tr>
<tr>
<td>\texttt{PERIOD}</td>
<td>0x20</td>
<td>Set period for timer event (1 to 3).</td>
</tr>
<tr>
<td>\texttt{SET}</td>
<td>0x30</td>
<td>Set event flag.</td>
</tr>
<tr>
<td>\texttt{CLEAR}</td>
<td>0x40</td>
<td>Clear event flag.</td>
</tr>
<tr>
<td>\texttt{WAIT}</td>
<td>0x50</td>
<td>Wait for event flag.</td>
</tr>
<tr>
<td>\texttt{TEST}</td>
<td>0x60</td>
<td>Test event flag.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CALL}</td>
<td>0x00</td>
<td>Call background function event.</td>
</tr>
<tr>
<td>\texttt{EVENT1,TIMER1}</td>
<td>0x01</td>
<td>Software event, Timer 1 event.</td>
</tr>
<tr>
<td>\texttt{EVENT2,TIMER2}</td>
<td>0x02</td>
<td>Software event, Timer 2 event.</td>
</tr>
<tr>
<td>\texttt{EVENT3,TIMER3}</td>
<td>0x03</td>
<td>Software event, Timer 3 event.</td>
</tr>
<tr>
<td>\texttt{EVENT4,FIFO1}</td>
<td>0x04</td>
<td>Software event, FIFO 1 event.</td>
</tr>
</tbody>
</table>
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.
SYNC         0x09  DEVO, ASYNQ 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
Enable RTC event. Function 2 called on event.

Special Cases:
• when setting the period for the TIMER1, TIMER2, TIMR3, 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.

\[
\text{reg}[A] = \exp(\text{reg}[A])
\]

Opcode: 45

Special Cases:
• if \text{reg}[A] is NaN, then the result is NaN
• if \text{reg}[A] is +infinity or greater than 88, then the result is +infinity
• if \text{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.

\[
\text{reg}[A] = \exp10(\text{reg}[A])
\]

Opcode: 46

Special Cases:
• if \text{reg}[A] is NaN, then the result is NaN
• if \text{reg}[A] is +infinity, then the result is +infinity
• if \text{reg}[A] is 32-bit, and \text{reg}[A] is greater than 38, then the result is +infinity
• if \text{reg}[A] is 64-bit, and \text{reg}[A] is greater than 308, then the result is +infinity
• if \text{reg}[A] is –infinity, the result is 0.0
• if \text{reg}[A] is 32-bit, and \text{reg}[A] is less than -38, then the result is 0.0
• if \text{reg}[A] is 64-bit, and \text{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, reg[A] = reg[A] + reg[0]

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.

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

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:
Micromega Corporation

---

**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:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- **Bit 2** Not-a-Number Set if either value is not a valid number
- **Bit 1** Sign Set if reg[A] < float(signedByte)
- **Bit 0** Zero Set if reg[A] = float(signedByte)

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

```
status = floatStatus(reg[A] - float(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.

---

**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:**

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

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]</td>
</tr>
<tr>
<td>Bit 2 Not-a-Number Set if either value is not a valid number</td>
</tr>
<tr>
<td>Bit 1 Sign Set if reg[A] &lt; reg[0</td>
</tr>
<tr>
<td>Bit 0 Zero Set if reg[A] = reg[0</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]</td>
</tr>
<tr>
<td>Bit 2 Not-a-Number Set if either value is not a valid number</td>
</tr>
<tr>
<td>Bit 1 Sign Set if reg[register2] &lt; reg[register1]</td>
</tr>
<tr>
<td>Bit 0 Zero Set if reg[register2] = reg[register1] If neither Bit 0 or Bit 1 is set, reg[register2] &gt; reg[register1]</td>
</tr>
</tbody>
</table>

**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: \texttt{FCNV,conversion}

Description: Convert the value in register A using the \textit{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.

\[\text{reg}[A] = \text{the converted value of reg}[A]\]

Opcode: \texttt{56}

Byte 2: \textit{conversion}

The conversion codes are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F_C</td>
<td>00</td>
<td>Fahrenheit to Celsius</td>
</tr>
<tr>
<td>1</td>
<td>C_F</td>
<td>01</td>
<td>Celsius to Fahrenheit</td>
</tr>
<tr>
<td>2</td>
<td>IN_MM</td>
<td>02</td>
<td>inches to millimeters</td>
</tr>
<tr>
<td>3</td>
<td>MM_IN</td>
<td>03</td>
<td>millimeters to inches</td>
</tr>
<tr>
<td>4</td>
<td>IN_CM</td>
<td>04</td>
<td>inches to centimeters</td>
</tr>
<tr>
<td>5</td>
<td>CM_IN</td>
<td>05</td>
<td>centimeters to inches</td>
</tr>
<tr>
<td>6</td>
<td>IN_M</td>
<td>06</td>
<td>inches to meters</td>
</tr>
<tr>
<td>7</td>
<td>M_IN</td>
<td>07</td>
<td>meters to inches</td>
</tr>
<tr>
<td>8</td>
<td>FT_M</td>
<td>08</td>
<td>feet to meters</td>
</tr>
<tr>
<td>9</td>
<td>M_FT</td>
<td>09</td>
<td>meters to feet</td>
</tr>
<tr>
<td>10</td>
<td>YD_M</td>
<td>0A</td>
<td>yards to meters</td>
</tr>
<tr>
<td>11</td>
<td>M_YD</td>
<td>0B</td>
<td>meters to yards</td>
</tr>
<tr>
<td>12</td>
<td>MILES_KM</td>
<td>0C</td>
<td>miles to kilometers</td>
</tr>
<tr>
<td>13</td>
<td>KM_MILES</td>
<td>0D</td>
<td>kilometers to miles</td>
</tr>
<tr>
<td>14</td>
<td>NM_M</td>
<td>0E</td>
<td>nautical miles to meters</td>
</tr>
<tr>
<td>15</td>
<td>M_NM</td>
<td>0F</td>
<td>meters to nautical miles</td>
</tr>
<tr>
<td>16</td>
<td>ACRES_M2</td>
<td>10</td>
<td>acres to meters2</td>
</tr>
<tr>
<td>17</td>
<td>M2_ACRES</td>
<td>11</td>
<td>meters 2 to acres</td>
</tr>
<tr>
<td>18</td>
<td>OZ_G</td>
<td>12</td>
<td>ounces to grams</td>
</tr>
<tr>
<td>19</td>
<td>G_OZ</td>
<td>13</td>
<td>grams to ounces</td>
</tr>
<tr>
<td>20</td>
<td>LB_KG</td>
<td>14</td>
<td>pounds to kilograms</td>
</tr>
<tr>
<td>21</td>
<td>KG_LB</td>
<td>15</td>
<td>kilograms to pounds</td>
</tr>
<tr>
<td>22</td>
<td>USGAL_L</td>
<td>16</td>
<td>US gallons to liters</td>
</tr>
<tr>
<td>23</td>
<td>L_USGAL</td>
<td>17</td>
<td>liters to US gallons</td>
</tr>
<tr>
<td>24</td>
<td>UKGAL_L</td>
<td>18</td>
<td>UK gallons to liters</td>
</tr>
<tr>
<td>25</td>
<td>L_UKGAL</td>
<td>19</td>
<td>liters to UK gallons</td>
</tr>
<tr>
<td>26</td>
<td>USOZ_ML</td>
<td>1A</td>
<td>US fluid ounces to milliliters</td>
</tr>
<tr>
<td>27</td>
<td>ML_USOZ</td>
<td>1B</td>
<td>milliliters to US fluid ounces</td>
</tr>
<tr>
<td>28</td>
<td>UKOZ_ML</td>
<td>1C</td>
<td>UK fluid ounces to milliliters</td>
</tr>
<tr>
<td>29</td>
<td>ML_UKOZ</td>
<td>1D</td>
<td>milliliters to UK fluid ounces</td>
</tr>
<tr>
<td>30</td>
<td>CAL_J</td>
<td>1E</td>
<td>calories to Joules</td>
</tr>
<tr>
<td>31</td>
<td>J_CAL</td>
<td>1F</td>
<td>Joules to calories</td>
</tr>
<tr>
<td>32</td>
<td>HP_W</td>
<td>20</td>
<td>horsepower to watts</td>
</tr>
<tr>
<td>33</td>
<td>W_HP</td>
<td>21</td>
<td>watts to horsepower</td>
</tr>
<tr>
<td>34</td>
<td>ATM_KP</td>
<td>22</td>
<td>atmospheres to kilopascals</td>
</tr>
</tbody>
</table>
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, FDIVO, FDIVR, FDIVRI, FDIVR0, FMOD, LDIV, LDIVI, LDIVO
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.

        reg[A] = reg[A] / float(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, LDIVO, LUDIV, LUDIVI, LUDIVO

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, reg[A] = reg[A] / reg[0]

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, LDIVO, LUDIV, LUDIVI, LUDIVO

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.

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, LUDIVO

---

**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:

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

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.

\[
\text{reg}[A] = \frac{1}{\text{reg}[A]}
\]

**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])
```
**FLOOR**  
**Floor**

**Syntax:**

\[ \text{FLOOR} \]

**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:**

\[ \text{FMAC, register}_1, \text{register}_2 \]

**Description:**
The floating point value in \( \text{register}_1 \) is multiplied by the value in \( \text{register}_2 \) and the result is added to register A.

\[ \text{reg}[A] = \text{reg}[A] + (\text{reg}[\text{register}_1] \times \text{reg}[\text{register}_2]) \]

**Opcode:** 57

**Byte 2:**

\( \text{register}_1 \)
Register number (0 to 255).

**Byte 3:**

\( \text{register}_2 \)
Register number (0 to 255).

**Special Cases:**
- if \( \text{reg}[A] \) is 32-bit and \( \text{register}_1 \) or \( \text{register}_2 \) are 64-bit, the values are converted to 32-bit before being used
- if \( \text{reg}[A] \) is 64-bit and \( \text{register}_1 \) or \( \text{register}_2 \) 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**
**FMAX, register**

**Syntax:**

\[ FMAX, \text{register} \]

**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 \( \text{reg}[A] \) 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
- if either value is NaN, then the result is NaN

**See Also:** FMIN, LMAX, LMIN

---

**FMIN, register**

**Floating point minimum**

**Syntax:**

\[ \text{FMIN}, \text{register} \]

**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 \( \text{reg}[A] \) 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
- if either value is NaN, then the result is NaN

**See Also:** FMAX, LMAX, LMIN

---

**FMOD, register**

**Floating point remainder**

**Syntax:**

\[ \text{FMOD}, \text{register} \]

**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, LUDIVO

---

**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.

\[
	ext{reg}[A] = \text{reg}[A] - (\text{reg}[\text{register1}] \times \text{reg}[\text{register2}])
\]

**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.

\[
	ext{reg}[A] = \text{reg}[A] \times \text{reg}[\text{register}]
\]

**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.

\[ \text{reg}[A] = \text{reg}[A] \times \text{float[signedByte]} \]

Opcode: 36

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

Special Cases:
• if \( \text{reg}[A] \) is NaN, then the result is NaN
• if the signed byte is zero and \( \text{reg}[A] \) 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.

\[ \begin{align*}
\text{if } \text{reg}[A] \text{ is 32-bit}, & \quad \text{reg}[A] = \text{reg}[A] \times \text{reg}[0] \\
\text{if } \text{reg}[A] \text{ is 64-bit}, & \quad \text{reg}[A] = \text{reg}[A] \times \text{reg}[128]
\end{align*} \]

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.

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

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 | reg[A] | > 1 and reg[register] is +infinite, then the result is +infinity
• if | reg[A] | < 1 and reg[register] is -infinite, then the result is +infinity
• if | reg[A] | > 1 and reg[register] is -infinite, then the result is 0.0
• if | reg[A] | < 1 and reg[register] is +infinite, then the result is 0.0
• if | 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 | 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 | 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
**FPOWI**

**Floating point power by immediate value**

**Syntax:**

\[ \text{FPOWI, signedByte} \]

**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.

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

**Opcode:**

39

**Byte 2:**

\[ \text{signedByte} \]

A signed byte value (-128 to 127).

**Special Cases:**

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

**See Also:**

FPOW, FPOW0, EXP, EXP10, LOG, LOG10, ROOT, SQRT

---

**FPOW0**

**Floating point power by register 0**

**Syntax:**

\[ \text{FPOW0} \]

**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.

\[
\begin{align*}
\text{if reg[A] is 32-bit, } \text{reg[A]} &= \text{reg[A]} ^* \text{reg[0]} \\
\text{if reg[A] is 64-bit, } \text{reg[A]} &= \text{reg[A]} ^* \text{reg[128]}
\end{align*}
\]

**Opcode:**

30
### 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 reg[A] is NaN or infinity, then the result is NaN

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

---

### 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, FREAD, 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, FREADA, 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.

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

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.

\[ \begin{align*}
\text{if } \text{reg}[A] \text{ is 32-bit, } & \text{reg}[A] = \text{reg}[0] \\
\text{if } \text{reg}[A] \text{ is 64-bit, } & \text{reg}[A] = \text{reg}[128]
\end{align*} \]

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.

\[ \begin{align*}
\text{reg}[A] &= \text{float(integer portion of reg}[A]) \\
\text{if} \text{reg}[A] \text{is 32-bit, } & \text{reg}[0] = \text{fractional portion of reg}[A] \\
\text{if} \text{reg}[A] \text{is 64-bit, } & \text{reg}[128] = \text{fractional portion of reg}[A]
\end{align*} \]

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
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>N</th>
<th>Z</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>N</th>
<th>Z</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
**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 \text{reg}[A] is NaN, then the result is NaN
• if \text{reg}[A] is +infinity, then the result is +infinity
• if \text{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.

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

**Opcode:** 86

**Byte 2:** conditionCode

The list of condition codes is as follows:

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>NZ</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
</tbody>
</table>
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: \textit{tableSize}
Specifies the number of 32-bit values in the table (0-255). If \textit{tableSize} is 0, the number of 32-bit values in the table is 256.

Bytes 4-n: \textit{tableItem1..tableItemN}
The number of 32-bit values specified by \textit{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: \texttt{FTOA, format}

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

Opcode: 1F

Byte 2: \texttt{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 \texttt{FTOA} opcode is the format byte and determines the format of the converted value.

If \texttt{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:

<table>
<thead>
<tr>
<th>Value</th>
<th>ASCII String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>NaN</td>
</tr>
<tr>
<td>10e20</td>
<td>Infinity</td>
</tr>
<tr>
<td>3.1415927</td>
<td>-Infinity</td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Value in register A</th>
<th>Format byte</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.567</td>
<td>61 (6.1)</td>
<td>123.6</td>
</tr>
<tr>
<td>123.567</td>
<td>62 (6.2)</td>
<td>123.57</td>
</tr>
<tr>
<td>123.567</td>
<td>42 (4.2)</td>
<td><em>.</em>*</td>
</tr>
<tr>
<td>0.9999</td>
<td>20 (2.0)</td>
<td>1</td>
</tr>
<tr>
<td>0.9999</td>
<td>31 (3.1)</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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` is 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` is 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, FWRITEA, 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 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).
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
Syntax: **INDA, register**

**Description:** Select register A using the lower 8 bits of the value in register.

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

**Opcode:** 7C

**Byte 2:** \( \text{register} \)

Register number (0 to 255).

**See Also:** SELECTA, SELECTX, INDX

---

**INDX**  
Select X using value in register

**Syntax:** **INDX, register**

**Description:** Select register X using the lower 8 bits of the value in register.

\[ X = \text{reg}[\text{register}] \]

**Opcode:** 7D

**Byte 2:** \( \text{register} \)

Register number (0 to 255).

**See Also:** SELECTA, SELECTX, INDA

---

**JMP**  
Unconditional jump

**Syntax:** **JMP, address**

**Description:** This instruction jumps unconditionally to the instruction at the address specified. If the jump is within -128 to 127 bytes of the address of the next instruction, the **BRA** instruction can be used.

**Opcode:** 83

**Bytes 2-3:** \( \text{address} \)

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,cc, RET, RET,cc

---

**JMP, cc**  
Conditional jump

**Syntax:** **JMP, conditionCode, address**
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: \textit{conditionCode}

The list of condition codes is as follows:

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>NZ</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
<tr>
<td>LT</td>
<td>0x72</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>0x62</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>0x70</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>0x60</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>PZ</td>
<td>0x71</td>
<td>Positive Zero</td>
</tr>
<tr>
<td>MZ</td>
<td>0x73</td>
<td>Negative Zero</td>
</tr>
<tr>
<td>INF</td>
<td>0xC8</td>
<td>Infinity</td>
</tr>
<tr>
<td>FIN</td>
<td>0xC0</td>
<td>Finite</td>
</tr>
<tr>
<td>PINF</td>
<td>0xE8</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>MINF</td>
<td>0xEA</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>0x44</td>
<td>Not-a-Number (NaN)</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x00</td>
<td>True</td>
</tr>
<tr>
<td>FALSE</td>
<td>0xFF</td>
<td>False</td>
</tr>
</tbody>
</table>

Bytes 3-4: \textit{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

\textbf{LABS} \hspace{1cm} \textbf{Long Integer absolute value}

Syntax: LABS

Description: The absolute value of the long integer value in register A is stored in register A.

\[ \text{reg[A]} = |\text{reg[A]}|, \text{status} = \text{longStatus}(\text{reg[A]}) \]

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.

\[
\text{if } \text{reg}[A] \text{ is 32-bit, } \text{reg}[A] = \text{reg}[A] + \text{reg}[0] \\
\text{if } \text{reg}[A] \text{ is 64-bit, } \text{reg}[A] = \text{reg}[A] + \text{reg}[128] \\
\text{status} = \text{longStatus}(\text{reg}[A])
\]

**Opcode:**

```
A6
```

**See Also:**

LADD, LADDI, FADD, FADDI, FADD0
**LAND**  Long integer AND

**Syntax:**  \texttt{LAND,register}

**Description:**  The bitwise AND of the values in register A and \texttt{register} is stored in register A.

\[
\text{reg}[A] = \text{reg}[A] \text{ AND } \text{reg}[\text{register}], \text{ status } = \text{longStatus(reg}[A])
\]

**Opcode:**  \texttt{C0}

**Byte 2:**  \texttt{register}

Register number (0 to 255).

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

**See Also:**  \texttt{LANDI, LBIT, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR}

---

**LANDI**  Long integer AND immediate value

**Syntax:**  \texttt{LANDI,\texttt{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.

\[
\text{reg}[A] = \text{reg}[A] \text{ AND } \text{long}(\text{signedByte}), \text{ status } = \text{longStatus(reg}[A])
\]

**Opcode:**  \texttt{CB}

**Byte 2:**  \texttt{\texttt{unsignedByte}}

An unsigned byte value (0 to 255).

**See Also:**  \texttt{LAND, LBIT, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR}

---

**LBIT**  Long integer Bit Clear, Set, Toggle, Test

**Syntax:**  \texttt{LBIT,bitCode,register}

**Description:**  The specified bit in \texttt{register} is cleared to zero, set to one, toggled, or tested. The action and bit number are specified by the \texttt{bitCode}. The status byte is set according to the state of selected bit after the action has been completed (\texttt{Z} if the bit is zero, \texttt{NZ} if the bit is non-zero).

**Opcode:**  \texttt{74}

**Byte 2:**  \texttt{bitCode}

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Op</th>
<th>Bit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 7:6</td>
<td><strong>Operation</strong></td>
<td><strong>IDE Symbol</strong></td>
</tr>
</tbody>
</table>

---
CLEAR 0x00 Clear bit
SET 0x40 Set bit
TOGGLE 0x80 Toggle bit
TEST 0xC0 Test bit

Bits 5:0 Bit Number
Value Description
0-63 Bit Number

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

See Also: LAND, LANDI, LNOT, LOR, LORI, LSHIFT, LSHIFTI, LXOR, LTST, 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>- - - - - - S</td>
</tr>
<tr>
<td>Z</td>
<td>- - - - - - Z</td>
</tr>
</tbody>
</table>

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:
Micromega Corporation

93

uM-FPU64 Instruction Set - Release 411
Bit 1  Sign  Set if reg[register1] < reg[register2]
Bit 0  Zero  Set if reg[register1] = reg[register2]
        If neither Bit 0 or Bit 1 is set, reg[register1] > reg[register2]

status = longStatus(reg[register1] - reg[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.

reg[register] = long(signedByte), status = longStatus(reg[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.

reg[register] = reg[register] - 1, status = longStatus(reg[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: $7FFFFFFFFFFFFFF$)

**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, LDIVO, LUDIV, LUDIVI, LUDIV0, FDIV, FDIVI, FDIVO, FDIVR, 
FDIVRI, FDIVRO, FMOD

---

**LDIVO**

**Long integer divide by register 0**

**Syntax:** 
LDIVO

**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, reg[A] = reg[A] / reg[0], reg[0] = 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, FDIVO, FDIVR, 
FDIVRI, FDIVRO, FMOD

---

**LEFT**

**Left Parenthesis**

**Syntax:**
LEFT

**Description:**
Saves the current register A 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.

\[ \text{reg[register]} = \text{reg[register]} + 1, \text{status} = \text{longStatus(reg[register])} \]

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.

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

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.

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

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.

\[
\text{reg}[A] = \text{reg}[A] \times \text{reg}[\text{register}], \quad \text{status} = \text{longStatus}(\text{reg}[A])
\]

**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.

\[
\text{reg}[A] = \text{reg}[A] \times \text{long}(\text{signedByte}), \quad \text{status} = \text{longStatus}(\text{reg}[A])
\]

**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.

\[
\begin{align*}
\text{if} \ \text{reg}[A] \text{ is 32-bit,} \quad &\text{reg}[A] = \text{reg}[A] \times \text{reg}[0] \\
\text{if} \ \text{reg}[A] \text{ is 64-bit,} \quad &\text{reg}[A] = \text{reg}[A] \times \text{reg}[128] \\
\text{status} &= \text{longStatus}(\text{reg}[A])
\end{align*}
\]

**Opcode:**  
A8

**See Also:**  
LMUL, LMULI, FMUL, FMULI, FMUL0
**LNEG**  Long integer negate

*Syntax:*  \textbf{LNEG}

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

\[
\text{reg}[A] = -\text{reg}[A], \text{status} = \text{longStatus(reg}[A]\text{)}
\]

*Opcode:*  \textbf{BB}

*See Also:*  LABS, FABS, FNEG

---

**LNOT**  \text{A} = \text{NOT} \text{A}

*Syntax:*  \textbf{LNOT}

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

\[
\text{reg}[A] = \text{NOT} \text{reg}[A], \text{status} = \text{longStatus(reg}[A]\text{)}
\]

*Opcode:*  \textbf{BF}

*See Also:*  LAND, LANDI, LBIT, LOR, LORI, LSHIFT, LSHIFTI, LXOR

---

**LOAD**  Load register 0 with value of register

*Syntax:*  \textbf{LOAD}, \texttt{register}

*Description:*  If register A is 32-bit, register 0 is loaded with the value in \textit{register}. If register A is 64-bit, register 128 is loaded with the value in \textit{register}.

\[
\begin{align*}
\text{if reg[A] is 32-bit, reg[0] = reg[register]} \\
\text{if reg[A] is 64-bit, reg[128] = reg[register]} \\
\text{status = longStatus(reg[A])}
\end{align*}
\]

*Opcode:*  \textbf{0A}

*Byte 2:*  \texttt{register}

Register number (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:*  LOADA, LOADX, ALOGDX, XSAVE, XSAVEA, SETARGS

---

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

*Syntax:*  \textbf{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, LOADE, 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, LONGBIND, 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, LONGBIND, 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.

\[
\text{if reg}[A]\text{ a 32-bit, } \text{reg}[0] = \text{float}(\text{unsignedWord}) \\
\text{if reg}[A]\text{ a 64-bit, } \text{reg}[128] = \text{float}(\text{unsignedWord})
\]

**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.

\[
\text{if reg}[A]\text{ is 32-bit, } \text{reg}[0] = \text{float}(\text{signedWord}) \\
\text{if reg}[A]\text{ is 64-bit, } \text{reg}[128] = \text{float}(\text{signedWord})
\]

**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, LOADWORD, LOADUWORD, LOADE, LOADPI, LONGBYTE, LONGUBYTE, LONGWORD, 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, reg[0] = reg[X], status = longStatus(reg[0]), X = X + 1
if reg[A] is 64-bit, reg[128] = reg[X], status = longStatus(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.

reg[A] = log(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.

reg[A] = log10(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, LONGBYTE, 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, LONGWORD, LOADIND, SETARGS
LOR  Long integer OR

Syntax: \texttt{LOR, register}

Description: The bitwise OR of the values in register A and register is stored in register A.

\[ \text{reg}[A] = \text{reg}[A] \text{ OR reg[register]}, \text{ status } = \text{longStatus(reg[A])} \]

Opcode: \texttt{C1}

Byte 2: \texttt{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, LORI, LSHIFT, LSHIFTI, LXOR

LORI  Long integer OR immediate value

Syntax: \texttt{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.

\[ \text{reg}[A] = \text{reg}[A] \text{ OR long(unsignedByte)}, \text{ status } = \text{longStatus(reg[A])} \]

Opcode: \texttt{CC}

Byte 2: \texttt{unsignedByte}
An unsigned byte value (0 to 255).

See Also: LAND, LANDI, LBIT, LNOT, LOR, LSHIFT, LSHIFTI, LXOR

LREAD  Read long integer value

Syntax: \texttt{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.

\[ \text{return 32-bit integer value from reg[register]} \]

Opcode: \texttt{94}

Byte 2: \texttt{register}
Register number (0 to 255).

Returns: \texttt{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, LREADX, LREADBYTE, 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, LREADX, LREADBYTE, 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, LREAD0, LREADA, LREADBYTE, LREADWORD, DREAD, RDIND, READSTR, READSEL, READSTATUS

LSET  Set register A
**LSET**, register

**Syntax:** \( \text{LSET, register} \)

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

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

**Opcode:** 9C

**Byte 2:** \text{register}

Register number (0 to 255).

**Special Cases:**
- if \(\text{reg}[A]\) is 32-bit and \text{register} is 64-bit, the value is converted to 32-bit before being used
- if \(\text{reg}[A]\) is 64-bit and \text{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:** \( \text{LSETI, signedByte} \)

**Description:** The \text{signedByte} is converted to a long integer and stored in register A.

\[
\text{reg}[A] = \text{long}(\text{signedByte}), \text{status} = \text{longStatus}(\text{reg}[A])
\]

**Opcode:** AE

**Byte 2:** \text{signedByte}

A signed byte value (-128 to 127).

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

---

**LSET0**  
Set register A from register 0

**Syntax:** \text{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.

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

**Opcode:** A5

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

---

**LSHIFT**  
Long integer shift
Syntax: \texttt{LSHIFT,register}

Description: The shift count is specified by the long integer value in \texttt{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.

\[
\begin{align*}
\text{if } \text{reg[register]} & > 0, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted left by } \text{reg[register]} \text{ bits} \\
-63 & < \text{reg[register]} < 0 \text{ and }, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted right by } -\text{reg[register]} \text{ bits} \\
-128 & < \text{reg[register]} < -64, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted right by } -(\text{reg[register]}+64) \text{ bits} \\
\text{status} & = \text{longStatus(reg[A])}
\end{align*}
\]

Opcode: \texttt{C3}

Byte 2: \texttt{register}
Register number (0 to 255).

Special Cases: • if \text{reg[register]} = 0 or -64, no shift occurs
• if \text{reg[A]} is 32-bit and (\text{reg[register]} > 32 or \text{reg[register]} < -32), then \text{reg[A]} = 0
• if \text{reg[A]} is 64-bit and (\text{reg[register]} > 64 or \text{reg[register]} < -64), then \text{reg[A]} = 0

See Also: LAND, LANDI, LBIT, LNOT, LOR, LORI, LSHIFTI, LXOR

\textbf{LSHIFTI} Long integer shift using immediate value

Syntax: \texttt{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.

\[
\begin{align*}
\text{signedByte} & > 0, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted left by } \text{signedByte} \text{ bits} \\
-63 & < \text{signedByte} < 0 \text{ and }, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted right by } -\text{signedByte} \text{ bits} \\
-128 & < \text{signedByte} < -64, \text{ then } \text{reg[A]} = \text{reg[A]} \text{ shifted right by } -(\text{signedByte}+64) \text{ bits} \\
\text{status} & = \text{longStatus(reg[A])}
\end{align*}
\]

Opcode: \texttt{CA}

Byte 2: \texttt{signedByte}
A signed byte value (-128 to 127).

Special Cases: • if \text{signedByte} = 0 or -64, no shift occurs
• if \text{reg[A]} is 32-bit and (\text{signedByte} > 32 or \text{signedByte} < -32), then \text{reg[A]} = 0
• if \text{reg[A]} is 64-bit and (\text{signedByte} > 64 or \text{signedByte} < -64), then \text{reg[A]} = 0
LSTATUS  Get long integer status

Syntax:  

\[ \text{LSTATUS, register} \]

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>S</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Opcode:  \( B7 \)

Byte 2:  \( \text{register} \)
Register number (0 to 255).

See Also:  FSTATUS, FSTATUSA, LSTATUSA, READSTATUS

LSTATUSA  Get long integer status of register A

Syntax:  

\[ \text{LSTATUSA} \]

Description:  \[ \text{status} = \text{longStatus(reg[A])} \]

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>S</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Opcode:  \( B8 \)

See Also:  FSTATUS, FSTATUSA, LSTATUS, READSTATUS

LSUB  Long integer subtract

Syntax:  

\[ \text{LSUB, register} \]

Description:  The long integer value in register is subtracted from register A.
reg[A] = reg[A] - reg[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.

reg[A] = reg[A] - long(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, reg[A] = reg[A] - reg[0]
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:

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>N0</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
<tr>
<td>LT</td>
<td>0x72</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>0x62</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>0x70</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>0x60</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>PZ</td>
<td>0x71</td>
<td>Positive Zero</td>
</tr>
<tr>
<td>MZ</td>
<td>0x73</td>
<td>Negative Zero</td>
</tr>
<tr>
<td>INF</td>
<td>0xC8</td>
<td>Infinity</td>
</tr>
<tr>
<td>FIN</td>
<td>0xC0</td>
<td>Finite</td>
</tr>
<tr>
<td>PINF</td>
<td>0xE8</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>MINF</td>
<td>0xEA</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>0x44</td>
<td>Not-a-Number (NaN)</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x00</td>
<td>True</td>
</tr>
<tr>
<td>FALSE</td>
<td>0xFF</td>
<td>False</td>
</tr>
</tbody>
</table>

**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 Zeros**

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 an 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)

<table>
<thead>
<tr>
<th>Value in register A</th>
<th>Format byte</th>
<th>Description</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>10</td>
<td>(signed 10)</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>110</td>
<td>(unsigned 10)</td>
<td>4294967295</td>
</tr>
<tr>
<td>-1</td>
<td>4</td>
<td>(signed 4)</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>104</td>
<td>(unsigned 4)</td>
<td>****</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>(signed 4)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>(unformatted)</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>6</td>
<td>(signed 6)</td>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
<td>-6</td>
<td>(signed 6, zero fill)</td>
<td>001000</td>
</tr>
</tbody>
</table>

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, STRFCRN, STRFIELD,
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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

- 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:**

\[ \text{LTST0} \]

**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 \( \text{reg}[A] \) is 32-bit, \( \text{status} = \text{longStatus}(\text{reg}[A] \text{ AND } \text{reg}[0]) \)
- if \( \text{reg}[A] \) is 64-bit, \( \text{status} = \text{longStatus}(\text{reg}[A] \text{ AND } \text{reg}[128]) \)

The status byte can be read with the \text{READSTATUS} instruction. It is set as follows:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>S</td>
</tr>
</tbody>
</table>

- Bit 1: **Sign** — Set if the MSB of the result is set
- Bit 0: **Zero** — Set the result is zero

**Opcode:**

\[ \text{AD} \]

**See Also:** LTST, LTSTI, LBIT, LCMP, LCMPI, LCMP0, LUCMP, LUCMPI, LUCMP0

**LUCMP**  
**Unsigned long integer compare**

**Syntax:**

\[ \text{LUCMP, register} \]

**Description:**
Compares the unsigned long integer value in register A with the value in \text{register} and sets the internal status byte.

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

The status byte can be read with the \text{READSTATUS} instruction. It is set as follows:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>S</td>
</tr>
</tbody>
</table>

- 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:**

\[ \text{A3} \]

**Byte 2:** \text{register}  
Register number (0 to 255).

**Special Cases:**
- if \( \text{reg}[A] \) is 32-bit and \text{register} is 64-bit, the value is converted to 32-bit before being used
- if \( \text{reg}[A] \) is 64-bit and \text{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.

\[
\text{status} = \text{longStatus(reg}[A]\text{ - long(unsignedByte))}
\]

The status byte can be read with the READSTATUS instruction. It is set as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>Bit 1</td>
<td>Sign</td>
<td>Set if reg[A] &lt; long(unsignedByte)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 0</td>
<td>Zero</td>
<td>Set if reg[A] = long(unsignedByte)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If neither Bit 0 or Bit 1 is set, reg[A] &gt; long(unsignedByte)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

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

The status byte can be read with the READSTATUS instruction. It is set as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>Bit 1</td>
<td>Sign</td>
<td>Set if reg[A] &lt; reg[0</td>
<td>128]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit 0</td>
<td>Zero</td>
<td>Set if reg[A] = reg[0</td>
<td>128]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If neither Bit 0 or Bit 1 is set, reg[A] &gt; reg[0</td>
<td>128]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Opcode:**  AC

LUCMP2  Unsigned long integer compare
### LUCMP2, register1, register2

**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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Z</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **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

**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 \(\text{reg}[\text{A}]\) is 32-bit, \(\text{reg}[0] = \text{remainder}\)

if \(\text{reg}[\text{A}]\) is 64-bit, \(\text{reg}[128] = \text{remainder}\)

** 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, LUDIVO, 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.

\[
\text{reg}[A] = \text{reg}[A] \div \text{long(unsignedByte)}, \text{status} = \text{longStatus(reg}[A])
\]

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, LUDIVO, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD

---

**LUDIVO**

*Unsigned long integer divide by register 0*

**Syntax:**

`LUDIVO`

**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.

\[
\text{if reg}[A] \text{ is 32-bit, reg}[A] = \text{reg}[A] \div \text{reg}[0], \text{reg}[0] = \text{remainder}
\]

\[
\text{if reg}[A] \text{ is 64-bit, reg}[A] = \text{reg}[A] \div \text{reg}[128], \text{reg}[128] = \text{remainder}
\]

\[
\text{status} = \text{longStatus(reg}[A])
\]

**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, LUDIVO, FDIV, FDIVI, FDIV0, FDIVR, FDIVRI, FDIVR0, FMOD
LWRITE  Write long integer value

Syntax:  \textit{LWRITE, register, int32Value}

Description:  The long integer value is stored in \textit{register}. If \textit{register} is 64-bit, \textit{int32Value} is converted to 64-bit before being stored in the register.

\[ \text{reg[register]} = 32\text{-bit long integer value, status = longStatus(reg[register])} \]

Opcode:  90

Byte 2:  \textit{register}
Register number (0 to 255).

Bytes 3 to 6:  \textit{int32Value}
Four bytes representing a 32-bit integer value (MSB first).

Special Cases:  
  • if \textit{register} is 64-bit, the value is converted to 64-bit before being stored.
  • if \textit{register} = 0 or 128, and \textit{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 \textit{register} = 0 or 128, and \textit{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:  \textit{LWRITEA, int32Value}

Description:  The long integer value is stored in register A.

\[ \text{reg[A]} = 32\text{-bit long integer value, status = longStatus(reg[A])} \]

Opcode:  91

Bytes 2 to 5:  \textit{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
LWRITEX  Write long integer value to register X

Syntax:  

LWRITEX, int32Value

Description:  The long integer value is stored in register X.

\[
\text{reg}[X] = 32\text{-bit long integer value, status = longStatus(reg}[X]), \quad X = X + 1
\]

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, FWRITEX, 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.

\[
\begin{align*}
&\text{if reg[A] is 32-bit, reg}[0] = 32\text{-bit long integer value,} \\
&\quad \text{status = longStatus(reg}[0]) \\
&\text{if reg[A] is 64-bit, reg}[128] = 32\text{-bit value converted to 64-bit floating point,} \\
&\quad \text{status = longStatus(reg}[128])
\end{align*}
\]

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, FWRITEX, LWRITE, LWRITE0, LWRITEA, 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.
reg[A] = reg[A] XOR reg[register], status = longStatus(reg[A])

Opcode: C2

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

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>IDE</th>
<th>Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x00</td>
<td>MOP</td>
<td>SCALAR_SET</td>
<td>x00</td>
<td>Scalar Set MA[r,c] = reg[0]</td>
</tr>
<tr>
<td>x01</td>
<td>MOP</td>
<td>SCALAR_ADD</td>
<td>x01</td>
<td>Scalar Add MA[r,c] = MA[r,c] + reg[0]</td>
</tr>
<tr>
<td>x02</td>
<td>MOP</td>
<td>SCALAR_SUB</td>
<td>x02</td>
<td>Scalar Subtract MA[r,c] = MA[r,c] - reg[0]</td>
</tr>
<tr>
<td>x03</td>
<td>MOP</td>
<td>SCALAR_SUBR</td>
<td>x03</td>
<td>Scalar Subtract (reverse) MA[r,c] = reg[0] - MA[r,c]</td>
</tr>
<tr>
<td>x04</td>
<td>MOP</td>
<td>SCALAR_MUL</td>
<td>x04</td>
<td>Scalar Multiply MA[r,c] = MA[r,c] * reg[0]</td>
</tr>
<tr>
<td>x05</td>
<td>MOP</td>
<td>SCALAR_DIV</td>
<td>x05</td>
<td>Scalar Divide MA[r,c] = MA[r,c] / reg[0]</td>
</tr>
</tbody>
</table>
MOP, SCALAR_DIV
For each element: \( MA[r,c] = MA[r,c] / \text{reg}[0] \)

**Scalar Divide (reverse) (x06)**
MOP, SCALAR_DIVR
For each element: \( MA[r,c] = \text{reg}[0] / MA[r,c] \)

**Scalar Power (x07)**
MOP, SCALAR_POW
For each element: \( MA[r,c] = MA[r,c] ** \text{reg}[0] \)

**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 x n matrix is augmented to n+2 x n and the LU decomposition of the original n x n matrix is stored in MC.

**LU Matrix Inverse (0x2A)**
MOP, LU_INVERSE
MA = inverse of original MC n x 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 x 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 x n matrix is augmented to n+2 x n and the Cholesky decomposition of the original n x n matrix is stored in MC.

**Cholesky Matrix Inverse (0x2E)**
MOP, CH_INVERSE
The inverse of the original MC n x n matrix is stored in the first n x 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 x 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 IEEEMODE instruction can be used to revert to standard IEEE 754 32-bit floating point mode.

**Opcode:**

F5

**See Also:** IEEEMODE

---

**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:

\[ y = A_0 + A_1x + A_2x^2 + \ldots A_nx^n \]

The value of x is the initial value of register A. An n\textsuperscript{th} order polynomial will have n+1 coefficients stored in the table. The coefficient values A\textsubscript{0}, A\textsubscript{1}, A\textsubscript{2}, … are stored as a series of 32-bit floating point values (4 bytes) stored in order from A\textsubscript{n} to A\textsubscript{0}. If a given term in the polynomial is not needed, a zero must be is stored for that value.

\[ \text{reg[A]} = \text{result of nth order polynomial calculation} \]

**Opcode:**

88
Byte 2: \( \textit{count} \)
The number of 32-bit floating point values that follow.

Bytes 3-n: \( \textit{float32Value1...float32ValueN} \)
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 = x^3 + 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.

\[ \text{reg}[A] = \text{radians}(\text{reg}[A]) \]

Opcode:  
4F

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

See Also:  
ACOS, ASIN, ATAN, ATAN2, COS, SIN, TAN, DEGREES

RDIND  
Read data using indirect pointer

Syntax:  
RDIND, dataType, pointer, count

Description:  
Read \( count \) data values of the specified \( \text{dataType} \) from the \( \text{pointer} \) 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. The data items must be read immediately following this instruction. See the SETIND instruction for a description of pointers. The RDIND 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:* 71

*Byte 2:* *dataType*

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Data Type</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Bits 3:0</th>
<th>Data Type</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT8</td>
<td>0x08</td>
<td>8-bit signed integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UINT8</td>
<td>0x09</td>
<td>8-bit unsigned integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT16</td>
<td>0X0A</td>
<td>16-bit signed integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UINT16</td>
<td>0x0B</td>
<td>16-bit unsigned integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG32</td>
<td>0x0C</td>
<td>32-bit long integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT32</td>
<td>0x0D</td>
<td>32-bit floating point data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG64</td>
<td>0x0E</td>
<td>64-bit long integer data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT64</td>
<td>0x0F</td>
<td>64-bit float point data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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.

*Special Cases:* • if *dataType* is 32-bit floating point, and PICMODE is enabled, the values are converted from IEEE-754 format before being sent

*See Also:* SETIND, ADDIND, WRIND, COPYIND, LOADIND, SAVEIND, SETREAD, FREAD, FREAD0, FREADA, FREADX, LREAD, LREAD0, LREADA, LREADX, LREADBYTE, LREADWORD, DREAD

---

**READSEL**  Read string selection

**Syntax:** READSEL

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

**Opcode:** EC
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
if \( \text{reg}[A] \) is 32-bit, \( \text{reg}[0] = \text{internal register value} \), \( \text{status} = \text{longStatus}(\text{reg}[0]) \)

if \( \text{reg}[A] \) is 64-bit, \( \text{reg}[128] = \text{internal register value} \), \( \text{status} = \text{longStatus}(\text{reg}[128]) \)

**Opcode:** FC

**Byte 2:** \( \text{item} \)

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 then 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 \(\text{FCALL}\). 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:  \texttt{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:  \texttt{conditionCode}

The list of condition codes is as follows:

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0x51</td>
<td>Zero</td>
</tr>
<tr>
<td>EQ</td>
<td>0x51</td>
<td>Equal</td>
</tr>
<tr>
<td>NZ</td>
<td>0x50</td>
<td>Not Zero</td>
</tr>
<tr>
<td>NE</td>
<td>0x50</td>
<td>Not Equal</td>
</tr>
<tr>
<td>LT</td>
<td>0x72</td>
<td>Less Than</td>
</tr>
<tr>
<td>LE</td>
<td>0x62</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>GT</td>
<td>0x70</td>
<td>Greater Than</td>
</tr>
<tr>
<td>GE</td>
<td>0x60</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>PZ</td>
<td>0x71</td>
<td>Positive Zero</td>
</tr>
<tr>
<td>MZ</td>
<td>0x73</td>
<td>Negative Zero</td>
</tr>
<tr>
<td>INF</td>
<td>0xC8</td>
<td>Infinity</td>
</tr>
<tr>
<td>FIN</td>
<td>0xC0</td>
<td>Finite</td>
</tr>
<tr>
<td>PINF</td>
<td>0xE8</td>
<td>Positive Infinity</td>
</tr>
<tr>
<td>MINF</td>
<td>0xEA</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>0x44</td>
<td>Not-a-Number (NaN)</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x00</td>
<td>True</td>
</tr>
<tr>
<td>FALSE</td>
<td>0xFF</td>
<td>False</td>
</tr>
</tbody>
</table>

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:  \texttt{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 \texttt{RIGHT} parenthesis is the outermost parenthesis, the register A selection from before the first \texttt{LEFT} parenthesis is restored, otherwise the previous temporary register is selected as register. This is used together with the \texttt{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\textsuperscript{th} root

Syntax: ROOT, register

Description: Calculates the n\textsuperscript{th} 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.

\[ \text{reg}[A] = \text{reg}[A] ^ { \left( \frac{1}{\text{reg}[\text{register}]} \right)} \]

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/\text{reg}[\text{register}])
  • if \text{reg}[\text{register}] is infinity, then (1 / \text{reg}[\text{register}]) is zero
  • if \text{reg}[\text{register}] is zero, then (1 / \text{reg}[\text{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:  \texttt{RTC,action \{,register\}}

Description:  Used to manage the real-time clock.

Opcode:  \texttt{DC}

Byte 2:  action

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
Action & & & & & & & & \\
Options & & & & & & & & \\
\hline
\end{tabular}
\end{center}

Bits 7:4  \textbf{Action}

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0x00</td>
<td>Initialize the real-time clock mode.</td>
</tr>
<tr>
<td>START</td>
<td>0x10</td>
<td>Start real-time clock.</td>
</tr>
<tr>
<td>STOP</td>
<td>0x20</td>
<td>Stop real-time clock.</td>
</tr>
<tr>
<td>ALARM_MASK</td>
<td>0x30</td>
<td>Set alarm mask.</td>
</tr>
<tr>
<td>WRITE_TIME</td>
<td>0x40</td>
<td>Write real-time clock date/time value.</td>
</tr>
<tr>
<td>WRITE_ALARM</td>
<td>0x50</td>
<td>Write alarm data/time value.</td>
</tr>
<tr>
<td>READ_TIME</td>
<td>0x60</td>
<td>Read real-time clock date/time value.</td>
</tr>
<tr>
<td>READ_ALARM</td>
<td>0x70</td>
<td>Read alarm date/time value.</td>
</tr>
<tr>
<td>NUM_TO_STR</td>
<td>0x80</td>
<td>Convert date/time number to string.</td>
</tr>
<tr>
<td>STR_TO_NUM</td>
<td>0x90</td>
<td>Convert date/time string to number.</td>
</tr>
<tr>
<td>NUM_TO_DATE</td>
<td>0xA0</td>
<td>Convert date/time number to values stored in consecutive registers.</td>
</tr>
<tr>
<td>DATE_TO_NUM</td>
<td>0xB0</td>
<td>Convert values stored in consecutive registers to date/time number.</td>
</tr>
</tbody>
</table>

Bits 3:0  \textbf{Options}

\textit{See descriptions below.}

\textbf{Initialize}

\texttt{RTC, INIT+options}

Initialize the Real-time clock.

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Bit & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0 & E & S & C & A \\
\hline
\end{tabular}
\end{center}

Bits 3  \textbf{Enable RTCC output pin}

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>The RTC output pin is disabled.</td>
</tr>
<tr>
<td>RTCC</td>
<td>0x08</td>
<td>The RTC output pin is enabled.</td>
</tr>
</tbody>
</table>

Bits 2  \textbf{Type of output}

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALARM_OUT</td>
<td>0x00</td>
<td>Toggle RTC on each alarm event.</td>
</tr>
<tr>
<td>HZ_OUT</td>
<td>0x04</td>
<td>One Hz output.</td>
</tr>
</tbody>
</table>

Bits 1  \textbf{Calibration}

<table>
<thead>
<tr>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0x00</td>
<td>No Calibration.</td>
</tr>
</tbody>
</table>
CAL 0x02

Set calibration from lower 8 bits of register 0.

Bits 0

<table>
<thead>
<tr>
<th>Alarm Event</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td>0x00</td>
<td>Alarm event disabled.</td>
</tr>
<tr>
<td>ALARM_ON</td>
<td></td>
<td>0x00</td>
<td>Alarm event enabled.</td>
</tr>
</tbody>
</table>

Start and Stop the Real-time Clock

```
RTC, START+options
RTC, STOP+options
```

Start and stop the real-time clock.

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     
```

### Bits 3:0 Alarm Mask

**Value**

- **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 Numeric/String Select

**IDE Symbol**

- **-**: 0x00

**Description**

- The numeric value in register 0 is used.
A string value is used. If the string selection is not empty, the string selection is used, otherwise the string buffer is used.

<table>
<thead>
<tr>
<th>Bits 2:0</th>
<th>Item</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE_TIME</td>
<td>0x00</td>
<td>DATE_TIME</td>
<td>YYYY-MM-DD HH:MM:SS</td>
<td></td>
</tr>
</tbody>
</table>

- **SECOND** 0x01: Seconds. Number: 0 to 59
  - String: 00...59
- **MINUTE** 0x02: Minutes. Number: 0 to 59
  - String: 00...59
- **HOUR** 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.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Action</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 3:0</td>
<td>Item</td>
<td>IDE Symbol</td>
</tr>
<tr>
<td>DATE_TIME</td>
<td>0x00</td>
<td>DATE_TIME</td>
</tr>
<tr>
<td>DATE</td>
<td>0x01</td>
<td>DATE</td>
</tr>
<tr>
<td>TIME</td>
<td>0x02</td>
<td>TIME</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Action</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 3:0</td>
<td>Item</td>
<td>IDE Symbol</td>
</tr>
<tr>
<td>DATE_TIME</td>
<td>0x00</td>
<td>DATE_TIME</td>
</tr>
</tbody>
</table>
Convert Date/Time Number to Values stored in Consecutive Registers

RTCA, 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:

- register: second (0-59)
- register+1: minute (0-59)
- register+2: hour (0-23)
- register+3: day (1-31)
- register+4: month (1-12)
- register+5: year (0-99)
- register+6: weekday (0-6) 0 = Sunday

Convert Values stored in Consecutive Registers to Date/Time Number

RTCA, 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: TIMESSET, 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
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.

\[
\begin{align*}
\text{if reg}[A] \text{ is 32-bit, matrix A } [\text{row}, \text{column}] & = \text{reg}[0] \\
\text{if reg}[A] \text{ is 64-bit, matrix A } [\text{row}, \text{column}] & = \text{reg}[128]
\end{align*}
\]

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.

\[
\begin{align*}
\text{if reg}[A] \text{ is 32-bit, matrix B } [\text{row}, \text{column}] & = \text{reg}[0] \\
\text{if reg}[A] \text{ is 64-bit, matrix B } [\text{row}, \text{column}] & = \text{reg}[128]
\end{align*}
\]

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.
**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, SAVEMC, 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: \[ \text{SELECTMA, register, rows, columns} \]

Description: The \textit{register} specifies the start of matrix A, and size of the matrix in \textit{rows} and \textit{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 \texttt{FREADX}, \texttt{FWRITEX}, \texttt{LREADX}, \texttt{LWRITEX}, \texttt{SAVEX}, \texttt{SETX}, \texttt{LOADX} instructions can be immediately used to store values to or retrieve values from the matrix. Additional information is available in the \textit{Using the uM-FPU64 Matrix Instructions} document.

Select matrix A if register matrix, \( X = \text{register} \)

Opcode: \( 65 \)

Byte 2: \textit{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: \textit{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: \textit{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: \texttt{FFT, LOADMA, LOADMB, LOADM C, MOP, SAVEMA, SAVEMB, SAVEM C, SELECTMB, SELECTMC}

SELECTMB Select matrix B

Syntax: \[ \text{SELECTMB, register, rows, columns} \]

Description: The \textit{register} specifies the start of matrix B, and size of the matrix in \textit{rows} and \textit{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 \texttt{FREADX}, \texttt{FWRITEX}, \texttt{LREADX}, \texttt{LWRITEX}, \texttt{SAVEX}, \texttt{SETX}, \texttt{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:** \( \text{register} \)  
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:** \( \text{rows} \)  
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:** \( \text{columns} \)  
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:**  \( \text{SELECTMC}, \text{register}, \text{rows}, \text{columns} \)

**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`, `FWRITEX`, `LREADX`, `LWRITEX`, `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:** \( \text{register} \)  
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.

\[ X = \text{register} \]

**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.

\[ \text{SERIN, DISABLE} \]
\[ \text{SERIN, ENABLE_CHAR} \]
SERIN, STATUS_CHAR
SERIN, READ_CHAR
SERIN, ENABLE_NMEA
SERIN, STATUS_NMEA
SERIN, READ_NMEA

Opcode: CF

Byte 2: action

<table>
<thead>
<tr>
<th>Bit 7:4</th>
<th>Device</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>Bit 7:4</td>
<td>Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
</tbody>
</table>

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:**

- SERIN, ENABLE_CHAR  
  Enable character input on SERIN pin.
- SERIN, ASYNC+ENABLE_CHAR  
  Enable character input on DEVIO,ASYNC pin.
- SERIN, READ_CHAR  
  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

<table>
<thead>
<tr>
<th>Bit 7:4 Device</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 7:4 Device</td>
<td>IDE Symbol</td>
<td>IDE Value</td>
<td>Description</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0x00</td>
<td>SEROUT pin</td>
</tr>
<tr>
<td>IDE1</td>
<td></td>
<td>0x10</td>
<td>Debug mode, data channel 1</td>
</tr>
<tr>
<td>IDE2</td>
<td></td>
<td>0x20</td>
<td>Debug mode, data channel 2</td>
</tr>
<tr>
<td>IDE3</td>
<td></td>
<td>0x30</td>
<td>Debug mode, data channel 3</td>
</tr>
<tr>
<td>ASYNC</td>
<td></td>
<td>0x40</td>
<td>Digital I/O pin assigned by DEVIO,ASYNC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 7:5 Action</th>
<th>IDE Symbol</th>
<th>IDE Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_BAUD</td>
<td>0x00</td>
<td></td>
<td>Set baud rate and debug mode.</td>
</tr>
<tr>
<td>WRITE_STR</td>
<td>0x01</td>
<td></td>
<td>Write string.</td>
</tr>
<tr>
<td>WRITE_SBUF</td>
<td>0x02</td>
<td></td>
<td>Write string buffer.</td>
</tr>
<tr>
<td>WRITE_SSEL</td>
<td>0x03</td>
<td></td>
<td>Write string selection.</td>
</tr>
<tr>
<td>WRITE_CHAR</td>
<td>0x04</td>
<td></td>
<td>Write character.</td>
</tr>
<tr>
<td>WRITE_STRZ</td>
<td>0x05</td>
<td></td>
<td>Write string and zero terminator.</td>
</tr>
<tr>
<td>WRITE_FLOAT</td>
<td>0x06</td>
<td></td>
<td>Write floating point value.</td>
</tr>
<tr>
<td>WRITE_LONG</td>
<td>0x07</td>
<td></td>
<td>Write long integer value.</td>
</tr>
<tr>
<td>WRITE_COMMA</td>
<td>0x08</td>
<td></td>
<td>Write comma.</td>
</tr>
<tr>
<td>WRITE_CRLF</td>
<td>0x09</td>
<td></td>
<td>Write carriage return, linefeed.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Byte 3: mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>57600 baud, Debug Mode enabled</td>
</tr>
<tr>
<td>1</td>
<td>300 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>2</td>
<td>600 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>3</td>
<td>1200 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>4</td>
<td>2400 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>5</td>
<td>4800 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>6</td>
<td>9600 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>7</td>
<td>19200 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>8</td>
<td>38400 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>9</td>
<td>57600 baud, Debug Mode disabled</td>
</tr>
<tr>
<td>10</td>
<td>115200 baud, Debug Mode disabled</td>
</tr>
</tbody>
</table>

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)**

```assembly
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)**

```assembly
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)**

```assembly
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)**

```assembly
SEROUT, WRITE_COMMA
```

A comma is sent to the serial output.

**Write Carriage Return, Linefeed (0x09, 0x19, 0x29, 0x39, 0x49)**

```assembly
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.

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

**Bits 7**

<table>
<thead>
<tr>
<th>Auto Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE Symbol</td>
</tr>
<tr>
<td>INC</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>

**Bits 5:0**

<table>
<thead>
<tr>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE Symbol</td>
</tr>
<tr>
<td>REG_LONG</td>
</tr>
<tr>
<td>REG_FLOAT</td>
</tr>
<tr>
<td>MEM_INT8</td>
</tr>
<tr>
<td>MEM_UINT8</td>
</tr>
<tr>
<td>MEM_INT16</td>
</tr>
<tr>
<td>MEM_UINT16</td>
</tr>
<tr>
<td>MEM_LONG32</td>
</tr>
<tr>
<td>MEM_FLOAT32</td>
</tr>
<tr>
<td>MEM_LONG64</td>
</tr>
<tr>
<td>MEM_FLOAT64</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Register, Long integer data</td>
</tr>
<tr>
<td>Register, Floating point data</td>
</tr>
<tr>
<td>Memory, 8-bit signed integer data</td>
</tr>
<tr>
<td>Memory, 8-bit unsigned integer data</td>
</tr>
<tr>
<td>Memory, 16-bit signed integer data</td>
</tr>
<tr>
<td>Memory, 16-bit unsigned integer data</td>
</tr>
<tr>
<td>Memory, 32-bit long integer data</td>
</tr>
<tr>
<td>Memory, 32-bit floating point data</td>
</tr>
<tr>
<td>Memory, 64-bit long integer data</td>
</tr>
<tr>
<td>Memory, 64-bit floating point data</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>24 23</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type</td>
<td>Address</td>
</tr>
<tr>
<td>Bit 15</td>
<td>0</td>
<td>Address</td>
</tr>
</tbody>
</table>

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
```

Micromega Corporation 154 uM-FPU64 Instruction Set - Release 411
**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, STRTOP, STRTOL, PTOA, 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.

status = longStatus of string compare

The status byte can be read with the READSTATUS instruction. It is set as follows:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>S</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

- 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, STRFCHR, 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
STRFIND  Find string in the string buffer

Syntax:  \texttt{STRFIND, string}

Description:  Search the current string selection in the string buffer for the first occurrence of the specified \textit{string}. If the \textit{string} is found, the selection point is set to the matching substring. If the \textit{string} is not found, the selection point is set to the end of the current string selection.

Opcode:  \texttt{E7}

Bytes 2-n:  \textit{string}
A zero-terminated string.

Special Cases:  • if \textit{string} length > 127, \textit{string} will be truncated to 127 characters

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

STRINC  Increment string selection point

Syntax:  \texttt{STRINC}

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

Opcode:  \texttt{EE}

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

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

STRINS  Insert string

Syntax:  \texttt{STRINS, string}

Description:  Insert the \textit{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:  \texttt{E5}

Bytes 2-n:  \textit{string}
A zero-terminated string.

Special Cases:  • if \textit{string} length > 127, \textit{string} will be truncated to 127 characters
See Also: STRSET, STRSEL, STRCMP, STRFIND, STRFCHR, STRFIELD, STRING, 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, STRING, 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, STRING, 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.

\[
tmp = \text{reg}[\text{register1}], \text{reg}[\text{register1}] = \text{reg}[\text{register2}], \text{reg}[\text{register2}] = \text{tmp}
\]

**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.

\[
tmp = \text{reg}[\text{register}], \text{reg}[\text{register}] = \text{reg}[A], \text{reg}[A] = \text{tmp}
\]
**Opcode:** 13

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

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

**See Also:** SWAP

---

**SYNC**  
**Synchronization**

**Syntax:**  
`SYNC`

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

**Opcode:** F0

**Returns:** 5C

---

**TABLE**  
**Table lookup**

**Syntax:**  
`TABLE, tableSize, tableItem1...tableItemN`

**Description:**  
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.

\[
\text{reg}[0] = \text{value from table indexed by reg}[0] \\
\text{if reg}[A] \text{ is 32-bit, reg}[0] = \text{value from table indexed by reg}[0] \\
\text{if reg}[A] \text{ is 64-bit, reg}[128] = \text{value from table indexed by lower 32-bits of reg}[128]
\]

**Opcode:** 85

**Byte 2:**  
*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 3-n:**  
`TableItem1...TableItemN`  
32-bit integer values or 32-bit floating point values

**Special Cases:**  
- only valid inside user-defined functions stored in Flash memory.  
- if the number of table entries doesn’t match the table size, the instruction is ignored.
• if reg[0 | 128] <= 0, then zero is returned.
• if reg[0 | 128] > maximum size of table, then zero is returned.
• if 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 reg[A] is NaN or an infinity, then the result is NaN
• if reg[A] is 0.0, then the result is 0.0
• if reg[A] is –0.0, then the result is –0.0

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

Opcode: D9

Special Cases:
• if 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 reg[A] is 32-bit, reg[0] = time in seconds, status = longStatus(reg[0])
if reg[A] is 64-bit, reg[128] = time in seconds, status = longStatus(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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Major</td>
<td>Minor</td>
<td>Beta</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit 7 6 5 4 3 2 1 0</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 3:0 IDE Symbol</td>
<td>IDE Value</td>
</tr>
<tr>
<td>INT8 0x08</td>
<td></td>
</tr>
<tr>
<td>UINT8 0x09</td>
<td></td>
</tr>
<tr>
<td>INT16 0x0A</td>
<td></td>
</tr>
<tr>
<td>UINT16 0x0B</td>
<td></td>
</tr>
<tr>
<td>LONG32 0x0C</td>
<td></td>
</tr>
<tr>
<td>FLOAT32 0x0D</td>
<td></td>
</tr>
<tr>
<td>LONG64 0x0E</td>
<td></td>
</tr>
<tr>
<td>FLOAT64 0x0F</td>
<td></td>
</tr>
</tbody>
</table>

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.

reg[X] = reg[A], status = longStatus(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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>00</td>
<td></td>
<td>No Operation</td>
<td></td>
</tr>
<tr>
<td>SELECTA</td>
<td>01</td>
<td>register</td>
<td>Select register A, A = register</td>
<td></td>
</tr>
<tr>
<td>SELECTX</td>
<td>02</td>
<td>register</td>
<td>Select register X, X = register</td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>03</td>
<td>register</td>
<td>reg[register] = 0</td>
<td></td>
</tr>
<tr>
<td>CLRA</td>
<td>04</td>
<td></td>
<td>reg[A] = 0</td>
<td></td>
</tr>
<tr>
<td>CLRX</td>
<td>05</td>
<td></td>
<td>reg[X] = 0, X = X + 1</td>
<td></td>
</tr>
<tr>
<td>CLR0</td>
<td>06</td>
<td></td>
<td>reg[0</td>
<td>128] = 0</td>
</tr>
<tr>
<td>COPY</td>
<td>07</td>
<td>register1, register2</td>
<td>reg[register2] = reg[register1]</td>
<td></td>
</tr>
<tr>
<td>COPYA</td>
<td>08</td>
<td>register</td>
<td>reg[register] = reg[A]</td>
<td></td>
</tr>
<tr>
<td>COPYX</td>
<td>09</td>
<td>register</td>
<td>reg[register] = reg[X], X = X + 1</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>0A</td>
<td>register</td>
<td>reg[0</td>
<td>128] = reg[register]</td>
</tr>
<tr>
<td>LOADA</td>
<td>0B</td>
<td></td>
<td>reg[0</td>
<td>128] = reg[A]</td>
</tr>
<tr>
<td>LOADX</td>
<td>0C</td>
<td></td>
<td>reg[0</td>
<td>128] = reg[X], X = X + 1</td>
</tr>
<tr>
<td>ALOADX</td>
<td>0D</td>
<td></td>
<td>reg[A] = reg[X], X = X + 1</td>
<td></td>
</tr>
<tr>
<td>XSAVE</td>
<td>0E</td>
<td>register</td>
<td>reg[X] = reg[register], X = X + 1</td>
<td></td>
</tr>
<tr>
<td>XSAVEA</td>
<td>0F</td>
<td></td>
<td>reg[X] = reg[A], X = X + 1</td>
<td></td>
</tr>
<tr>
<td>COPY0</td>
<td>10</td>
<td>register</td>
<td>reg[register] = reg[0</td>
<td>128]</td>
</tr>
<tr>
<td>LCOPYI</td>
<td>11</td>
<td>signedByte, register</td>
<td>reg[register] = long(signedByte)</td>
<td></td>
</tr>
<tr>
<td>SWAP</td>
<td>12</td>
<td>register1, register2</td>
<td>Swap reg[register1] and reg[register2]</td>
<td></td>
</tr>
<tr>
<td>SWAPA</td>
<td>13</td>
<td>register</td>
<td>Swap reg[register] and reg[A]</td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>14</td>
<td></td>
<td>Left parenthesis</td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td>15</td>
<td></td>
<td>Right parenthesis</td>
<td></td>
</tr>
<tr>
<td>FWRITE</td>
<td>16</td>
<td>register, float32Value</td>
<td>Write 32-bit floating point to reg[register]</td>
<td></td>
</tr>
<tr>
<td>FWRITEA</td>
<td>17</td>
<td>float32Value</td>
<td>Write 32-bit floating point to reg[A]</td>
<td></td>
</tr>
<tr>
<td>FWRITEX</td>
<td>18</td>
<td>float32Value</td>
<td>Write 32-bit floating point to reg[X]</td>
<td></td>
</tr>
<tr>
<td>FWRITE0</td>
<td>19</td>
<td>float32Value</td>
<td>Write 32-bit floating point to reg[0</td>
<td>128]</td>
</tr>
<tr>
<td>FREAD</td>
<td>1A</td>
<td>register</td>
<td>float32Value</td>
<td>Read 32-bit floating point from reg[register]</td>
</tr>
<tr>
<td>FREADA</td>
<td>1B</td>
<td>float32Value</td>
<td>Read 32-bit floating point from reg[A]</td>
<td></td>
</tr>
<tr>
<td>FREADX</td>
<td>1C</td>
<td>float32Value</td>
<td>Read 32-bit floating point from reg[X]</td>
<td></td>
</tr>
<tr>
<td>FREADD0</td>
<td>1D</td>
<td>float32Value</td>
<td>Read 32-bit floating point from reg[0</td>
<td>128]</td>
</tr>
<tr>
<td>ATOF</td>
<td>1E</td>
<td>string</td>
<td>Convert ASCII to floating point</td>
<td></td>
</tr>
<tr>
<td>FTOA</td>
<td>1F</td>
<td>format</td>
<td>Convert floating point to ASCII</td>
<td></td>
</tr>
<tr>
<td>FSET</td>
<td>20</td>
<td>register</td>
<td>reg[A] = reg[register]</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Opcode</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMP</td>
<td>28</td>
<td>Compare reg[A] and reg[register], and set status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSET0</td>
<td>29</td>
<td>reg[A] = reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FADD0</td>
<td>2A</td>
<td>reg[A] = reg[A] + reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FSUB0</td>
<td>2B</td>
<td>reg[A] = reg[A] - reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FSUBR0</td>
<td>2C</td>
<td>reg[A] = reg[0] - reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL0</td>
<td>2D</td>
<td>reg[A] = reg[A] * reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FDIV0</td>
<td>2E</td>
<td>reg[A] = reg[A] / reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FDIVR0</td>
<td>2F</td>
<td>reg[A] = reg[0</td>
<td>128] / reg[A]</td>
<td></td>
</tr>
<tr>
<td>FPOW0</td>
<td>30</td>
<td>reg[A] = reg[A] ** reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FCMP0</td>
<td>31</td>
<td>Compare reg[A] and reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>FSETI</td>
<td>32</td>
<td>reg[A] = float(signedByte)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADDI</td>
<td>33</td>
<td>reg[A] = reg[A] - float(signedByte)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUBI</td>
<td>34</td>
<td>reg[A] = reg[A] - float(signedByte)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUBRI</td>
<td>35</td>
<td>reg[A] = float(signedByte) - reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMULI</td>
<td>36</td>
<td>reg[A] = reg[A] * float(signedByte)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIVI</td>
<td>37</td>
<td>reg[A] = reg[A] / float(signedByte)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIVRI</td>
<td>38</td>
<td>reg[A] = float(signedByte) / reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPOWI</td>
<td>39</td>
<td>reg[A] = reg[A] ** signedByte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMPI</td>
<td>3A</td>
<td>Compare reg[A] and float(signedByte), and set floating point status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTATUS</td>
<td>3B</td>
<td>Set floating point status for register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTATUSA</td>
<td>3C</td>
<td>Set floating point status for reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMP2</td>
<td>3D</td>
<td>Compare reg[register1] and reg[register2], and set floating point status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNEG</td>
<td>3E</td>
<td>reg[A] = -reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FABS</td>
<td>3F</td>
<td>reg[A] =</td>
<td>reg[A]</td>
<td></td>
</tr>
<tr>
<td>FINV</td>
<td>40</td>
<td>reg[A] = 1 / reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT</td>
<td>41</td>
<td>reg[A] = sqrt(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROOT</td>
<td>42</td>
<td>reg[A] = root(reg[A], reg[register])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG</td>
<td>43</td>
<td>reg[A] = log(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG10</td>
<td>44</td>
<td>reg[A] = log10(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>45</td>
<td>reg[A] = exp(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP10</td>
<td>46</td>
<td>reg[A] = exp10(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN</td>
<td>47</td>
<td>reg[A] = sin(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COS</td>
<td>48</td>
<td>reg[A] = cos(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAN</td>
<td>49</td>
<td>reg[A] = tan(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIN</td>
<td>4A</td>
<td>reg[A] = asin(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACOS</td>
<td>4B</td>
<td>reg[A] = acos(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN</td>
<td>4C</td>
<td>reg[A] = atan(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN2</td>
<td>4D</td>
<td>reg[A] = atan2(reg[A], reg[register])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEGREES</td>
<td>4E</td>
<td>reg[A] = degrees(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RADIANS</td>
<td>4F</td>
<td>reg[A] = radians(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOOR</td>
<td>51</td>
<td>reg[A] = floor(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEIL</td>
<td>52</td>
<td>reg[A] = ceil(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUND</td>
<td>53</td>
<td>reg[A] = round(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMIN</td>
<td>54</td>
<td>reg[A] = min(reg[A], reg[register])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMAX</td>
<td>55</td>
<td>reg[A] = max(reg[A], reg[register])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCNV</td>
<td>56</td>
<td>reg[A] = conversion(reg[A])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Opcode</td>
<td>Description</td>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>FMAC</td>
<td>57</td>
<td><code>register1, register2</code></td>
<td>reg[A] = reg[A] + (reg[register1] * reg[register2])</td>
<td></td>
</tr>
<tr>
<td>LOADBYTE</td>
<td>59</td>
<td><code>signedByte</code></td>
<td>reg[0</td>
<td>128] = float(signedByte)</td>
</tr>
<tr>
<td>LOADUBYTE</td>
<td>5A</td>
<td><code>unsignedByte</code></td>
<td>reg[0</td>
<td>128] = float(unsignedByte)</td>
</tr>
<tr>
<td>LOADWORD</td>
<td>5B</td>
<td><code>signedWord</code></td>
<td>reg[0</td>
<td>128] = float(signedWord)</td>
</tr>
<tr>
<td>LOAUDWORD</td>
<td>5C</td>
<td><code>unsignedWord</code></td>
<td>reg[0</td>
<td>128] = float(unsignedWord)</td>
</tr>
<tr>
<td>LOADE</td>
<td>5D</td>
<td></td>
<td>reg[0</td>
<td>128] = 2.7182818</td>
</tr>
<tr>
<td>LOADPI</td>
<td>5E</td>
<td></td>
<td>reg[0</td>
<td>128] = 3.1415927</td>
</tr>
<tr>
<td>FCOPYI</td>
<td>5F</td>
<td><code>signedByte, register</code></td>
<td>reg[register] = float(signedByte)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>60</td>
<td></td>
<td>reg[A] = float(reg[A])</td>
<td></td>
</tr>
<tr>
<td>FIX</td>
<td>61</td>
<td></td>
<td>reg[A] = fix(reg[A])</td>
<td></td>
</tr>
<tr>
<td>FIXR</td>
<td>62</td>
<td></td>
<td>reg[A] = fix(round(reg[A]))</td>
<td></td>
</tr>
<tr>
<td>FRAC</td>
<td>63</td>
<td></td>
<td>reg[A] = fraction(reg[A])</td>
<td></td>
</tr>
<tr>
<td>FSPLIT</td>
<td>64</td>
<td></td>
<td>reg[A] = integer(reg[A]), reg[0</td>
<td>128] = fraction(reg[A])</td>
</tr>
<tr>
<td>SELECTMA</td>
<td>65</td>
<td><code>register, rows, columns</code></td>
<td>Select matrix A, starting at register. size = rows x columns</td>
<td></td>
</tr>
<tr>
<td>SELECTMB</td>
<td>66</td>
<td><code>register, rows, columns</code></td>
<td>Select matrix B, starting at register. size = rows x columns</td>
<td></td>
</tr>
<tr>
<td>SELECTMC</td>
<td>67</td>
<td><code>register, rows, columns</code></td>
<td>Select matrix C, starting at register. size = rows x columns</td>
<td></td>
</tr>
<tr>
<td>LOADMA</td>
<td>68</td>
<td><code>row, column</code></td>
<td>reg[0] = Matrix A[row, column]</td>
<td></td>
</tr>
<tr>
<td>LOADMB</td>
<td>69</td>
<td><code>row, column</code></td>
<td>reg[0] = Matrix B[row, column]</td>
<td></td>
</tr>
<tr>
<td>LOADMC</td>
<td>6A</td>
<td><code>row, column</code></td>
<td>reg[0] = Matrix C[row, column]</td>
<td></td>
</tr>
<tr>
<td>SAVEMA</td>
<td>6B</td>
<td><code>row, column</code></td>
<td>Matrix A[row, column] = reg[0]</td>
<td></td>
</tr>
<tr>
<td>SAEMB</td>
<td>6C</td>
<td><code>row, column</code></td>
<td>Matrix B[row, column] = reg[0]</td>
<td></td>
</tr>
<tr>
<td>SAVEMC</td>
<td>6D</td>
<td><code>row, column</code></td>
<td>Matrix C[row, column] = reg[0]</td>
<td></td>
</tr>
<tr>
<td>MOP</td>
<td>6E</td>
<td><code>action</code></td>
<td>Matrix/Vector operation</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>6F</td>
<td><code>action</code></td>
<td>Fast Fourier Transform</td>
<td></td>
</tr>
<tr>
<td>WRIND</td>
<td>70</td>
<td><code>dataType, pointer, count, value1...valueN</code></td>
<td>Write multiple data values to indirect pointer</td>
<td></td>
</tr>
<tr>
<td>RDIND</td>
<td>71</td>
<td><code>dataType, pointer, count, value1...valueN</code></td>
<td>Read multiple data values from indirect pointer</td>
<td></td>
</tr>
<tr>
<td>DWRITE</td>
<td>72</td>
<td><code>register, value64</code></td>
<td>Write 64-bit value</td>
<td></td>
</tr>
<tr>
<td>DREAD</td>
<td>73</td>
<td><code>register, value64</code></td>
<td>Read 64-bit value</td>
<td></td>
</tr>
<tr>
<td>LBIT</td>
<td>74</td>
<td><code>action, register</code></td>
<td>Bit Clear, Set, Toggle, Test</td>
<td></td>
</tr>
<tr>
<td>SETIND</td>
<td>77</td>
<td>`type, {register</td>
<td>address</td>
<td>function, offset}`</td>
</tr>
<tr>
<td>ADDIND</td>
<td>78</td>
<td><code>register, unsignedByte</code></td>
<td>Add to indirect pointer</td>
<td></td>
</tr>
<tr>
<td>COPYIND</td>
<td>79</td>
<td><code>register1, register2, register3</code></td>
<td>Copy using indirect pointers</td>
<td></td>
</tr>
<tr>
<td>LOADIND</td>
<td>7A</td>
<td><code>register</code></td>
<td>Load reg[0</td>
<td>128] using indirect pointer</td>
</tr>
<tr>
<td>SAVEIND</td>
<td>7B</td>
<td><code>register</code></td>
<td>Save reg[A] using indirect pointer</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Code</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDA</td>
<td>7C</td>
<td><code>register</code> Select register A using reg[register] value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDX</td>
<td>7D</td>
<td><code>register</code> Select register X using reg[register] value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCALL</td>
<td>7E</td>
<td><code>function</code> Call user-defined function stored in Flash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT</td>
<td>7F</td>
<td><code>action</code> <code>{,function}</code> Background Events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>80</td>
<td>Return from user-defined function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRA</td>
<td>81</td>
<td><code>relativeOffset</code> Unconditional branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRA,cc</td>
<td>82</td>
<td><code>conditionCode, relativeOffset</code> Conditional branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>83</td>
<td><code>absoluteOffset</code> Unconditional jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP,cc</td>
<td>84</td>
<td><code>conditionCode, absoluteOffset</code> Conditional jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>85</td>
<td><code>tableSize, tableItem1... tableItemN</code> Table lookup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FTABLE</td>
<td>86</td>
<td><code>conditionCode, tableSize, tableItem1... tableItemN</code> Floating point reverse table lookup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTABLE</td>
<td>87</td>
<td><code>conditionCode, tableSize, tableItem1... tableItemN</code> Long integer reverse table lookup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POLY</td>
<td>88</td>
<td><code>count, float32Value1... float32ValueN</code> reg[A] = nth order polynomial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOTO</td>
<td>89</td>
<td><code>register</code> Computed GOTO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET,cc</td>
<td>8A</td>
<td><code>conditionCode</code> Conditional return from user-defined function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LWRITE</td>
<td>90</td>
<td><code>register, int32Value</code> Write 32-bit long integer to reg[register]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LWRITEA</td>
<td>91</td>
<td><code>int32Value</code> Write 32-bit long integer to reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LWRITEX</td>
<td>92</td>
<td><code>int32Value</code> Write 32-bit long integer to reg[X], X = X + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LWRITE0</td>
<td>93</td>
<td><code>int32Value</code> Write 32-bit long integer to reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>LREAD</td>
<td>94</td>
<td><code>register</code> <code>int32Value</code> Read 32-bit long integer from reg[register]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LREADA</td>
<td>95</td>
<td><code>int32Value</code> Read 32-bit long value from reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LREADX</td>
<td>96</td>
<td><code>int32Value</code> Read 32-bit long integer from reg[X], X = X + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LREAD0</td>
<td>97</td>
<td><code>int32Value</code> Read 32-bit long integer from reg[0</td>
<td>128]</td>
<td></td>
</tr>
<tr>
<td>LREADBYTE</td>
<td>98</td>
<td><code>byteValue</code> Read lower 8 bits of reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LREADWORD</td>
<td>99</td>
<td><code>wordValue</code> Read lower 16 bits reg[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOL</td>
<td>9A</td>
<td><code>string</code> Convert ASCII to long integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTOA</td>
<td>9B</td>
<td><code>format</code> Convert long integer to ASCII</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSET</td>
<td>9C</td>
<td><code>register</code> reg[A] = reg[register]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
</tr>
<tr>
<td>-------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>LCMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUDIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSET0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LADD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSUB0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMUL0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDIV0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCMP0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUDIV0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCMP0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTST0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSETI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LADDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSUBI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMULI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDIVI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCMP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUDIVI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCMP1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTST1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSTATUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSTATUSA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCMP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCMP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LNEG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LABS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LNOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LXOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSHIFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Summary

<table>
<thead>
<tr>
<th>Code</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LMIN</strong></td>
<td>C4</td>
<td>register</td>
</tr>
<tr>
<td><strong>LMAX</strong></td>
<td>C5</td>
<td>register</td>
</tr>
<tr>
<td><strong>LONGBYTE</strong></td>
<td>C6</td>
<td>signedByte</td>
</tr>
<tr>
<td><strong>LONGUBYTE</strong></td>
<td>C7</td>
<td>unsignedByte</td>
</tr>
<tr>
<td><strong>LONGWORD</strong></td>
<td>C8</td>
<td>signedWord</td>
</tr>
<tr>
<td><strong>LONGUWORD</strong></td>
<td>C9</td>
<td>unsignedWord</td>
</tr>
<tr>
<td><strong>LSHIFTI</strong></td>
<td>CA</td>
<td>unsignedByte</td>
</tr>
<tr>
<td><strong>LANDI</strong></td>
<td>CB</td>
<td>unsignedByte</td>
</tr>
<tr>
<td><strong>LORI</strong></td>
<td>CC</td>
<td>unsignedByte</td>
</tr>
<tr>
<td><strong>SETSTATUS</strong></td>
<td>CD</td>
<td>status</td>
</tr>
<tr>
<td><strong>SEROUT</strong></td>
<td>CE</td>
<td>action, {baud}{string}</td>
</tr>
<tr>
<td><strong>SERIN</strong></td>
<td>CF</td>
<td>action</td>
</tr>
<tr>
<td><strong>DIGIO</strong></td>
<td>D0</td>
<td>action, {mode}</td>
</tr>
<tr>
<td><strong>ADC Mode</strong></td>
<td>D1</td>
<td>mode</td>
</tr>
<tr>
<td><strong>ADCTRIG</strong></td>
<td>D2</td>
<td>A/D manual trigger</td>
</tr>
<tr>
<td><strong>ADCScale</strong></td>
<td>D3</td>
<td>channel</td>
</tr>
<tr>
<td><strong>ADC Long</strong></td>
<td>D4</td>
<td>channel</td>
</tr>
<tr>
<td><strong>ADC Load</strong></td>
<td>D5</td>
<td>channel</td>
</tr>
<tr>
<td><strong>ADCWait</strong></td>
<td>D6</td>
<td></td>
</tr>
<tr>
<td><strong>TIME Set</strong></td>
<td>D7</td>
<td>period</td>
</tr>
<tr>
<td><strong>TIME Long</strong></td>
<td>D8</td>
<td>action</td>
</tr>
<tr>
<td><strong>TICK Long</strong></td>
<td>D9</td>
<td></td>
</tr>
<tr>
<td><strong>DEVIO</strong></td>
<td>DA</td>
<td>device, action {,…}</td>
</tr>
<tr>
<td><strong>DELAY</strong></td>
<td>DB</td>
<td>action</td>
</tr>
<tr>
<td><strong>RTC</strong></td>
<td>DC</td>
<td></td>
</tr>
<tr>
<td><strong>SETARGS</strong></td>
<td>DD</td>
<td></td>
</tr>
<tr>
<td><strong>EXT Set</strong></td>
<td>E0</td>
<td></td>
</tr>
<tr>
<td><strong>EXT Long</strong></td>
<td>E1</td>
<td></td>
</tr>
<tr>
<td><strong>EXT Wait</strong></td>
<td>E2</td>
<td></td>
</tr>
<tr>
<td><strong>STR Set</strong></td>
<td>E3</td>
<td>string</td>
</tr>
<tr>
<td><strong>STR Sel</strong></td>
<td>E4</td>
<td>start, length</td>
</tr>
<tr>
<td><strong>STRINS</strong></td>
<td>E5</td>
<td>string</td>
</tr>
<tr>
<td><strong>STRCMP</strong></td>
<td>E6</td>
<td>string</td>
</tr>
<tr>
<td><strong>STR FIND</strong></td>
<td>E7</td>
<td>string</td>
</tr>
<tr>
<td><strong>STR PC</strong></td>
<td>E8</td>
<td>string</td>
</tr>
<tr>
<td><strong>STR Field</strong></td>
<td>E9</td>
<td>field</td>
</tr>
<tr>
<td><strong>STR TOF</strong></td>
<td>EA</td>
<td></td>
</tr>
<tr>
<td><strong>STR TOL</strong></td>
<td>EB</td>
<td></td>
</tr>
<tr>
<td><strong>READSEL</strong></td>
<td>EC</td>
<td>string</td>
</tr>
<tr>
<td><strong>STR Byte</strong></td>
<td>ED</td>
<td></td>
</tr>
<tr>
<td><strong>STRING</strong></td>
<td>EE</td>
<td></td>
</tr>
<tr>
<td><strong>STR DEC</strong></td>
<td>EF</td>
<td></td>
</tr>
<tr>
<td><strong>SYNC</strong></td>
<td>F0</td>
<td>5C</td>
</tr>
<tr>
<td><strong>READ STATUS</strong></td>
<td>F1</td>
<td>status</td>
</tr>
<tr>
<td><strong>READ STR</strong></td>
<td>F2</td>
<td>string</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>F3</td>
<td>string</td>
</tr>
<tr>
<td><strong>IEEE MODE</strong></td>
<td>F4</td>
<td></td>
</tr>
</tbody>
</table>

**Reg expressions**

- reg[A] = min(reg[A], reg[register])
- reg[A] = max(reg[A], reg[register])
- reg[0 | 128] = long(signedByte)
- reg[0 | 128] = long(unsignedByte)
- reg[0 | 128] = long(signedWord)
- reg[0 | 128] = long(unsignedWord)
- reg[A] = reg[A] shift unsignedByte
- reg[A] = reg[A] AND unsignedByte
- reg[A] = reg[A] OR unsignedByte
- Set status byte
- Serial output
- Serial input
- Digital I/O
- Set A/D trigger mode
- A/D manual trigger
- ADCscale[ch] = reg[0]
- reg[0] = ADCvalue[channel]
- reg[0] = float(ADCvalue[channel]) * ADCscale[channel]
- wait for next A/D sample
- time = reg[0]
- reg[0] = time (long integer)
- reg[0] = ticks (long integer)
- Device I/O
- Delay (in milliseconds)
- Real-time Clock
- Enable FCALL argument loading
- external input count = reg[0]
- reg[0] = external input counter
- wait for next external input
- Copy string to string buffer
- Set selection point
- Insert string at selection point
- Compare string with string selection
- Find string
- Set field separators
- Find field
- Convert string selection to floating point
- Convert string selection to long integer
- Read string selection
- Insert byte at selection point
- Increment string selection point
- Decrement string selection point
- Get synchronization byte
- Read status byte
- Read string from string buffer
- Copy version string to string buffer
- Set IEEE mode (default)
<table>
<thead>
<tr>
<th>PICMODE</th>
<th>F5</th>
<th>Set PIC mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKSUM</td>
<td>F6</td>
<td>Calculate checksum for uM-FPU code</td>
</tr>
<tr>
<td>BREAK</td>
<td>F7</td>
<td>Debug breakpoint</td>
</tr>
<tr>
<td>TRACEOFF</td>
<td>F8</td>
<td>Turn debug trace off</td>
</tr>
<tr>
<td>TRACEREG</td>
<td>F9</td>
<td>Turn debug trace on</td>
</tr>
<tr>
<td>TRACESTR</td>
<td>FA</td>
<td>string</td>
</tr>
<tr>
<td>READVAR</td>
<td>FC</td>
<td>item</td>
</tr>
<tr>
<td>SETREAD</td>
<td>FD</td>
<td></td>
</tr>
<tr>
<td>XOP</td>
<td>FE</td>
<td>xop_number,byte, byte</td>
</tr>
<tr>
<td>RESET</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

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, FIFO
DEVIO, MEM
DEVIO, SDFAT
FTABLE
LTABLE
RDIND
TABLE
WRIND
TRACESTR

New Instructions
DEVIO, loadable devices

Deleted Instructions
DEVIO, VDRIVE2

Release 408
Modified Instructions
FTOA
READVAR
SELECTMA
SELECTMB
SELECTMC
XOP

Release 407, 406
Modified Instructions
DEVIO, LCD
DEVIO, SPI
TABLE

Release 405
Modified Instructions
DEVIO, FIFO
DEVIO, I2C
DEVIO, LCD
DIGIO
RTC
TIMESET
TICKLONG

Release 404
Modified Instructions
MOP

Release 402
Modified Instructions
ADDIND
COPYIND
DELAY
DEVIO, COUNTER
<table>
<thead>
<tr>
<th>EVENT</th>
<th>LOADMB</th>
<th>SAVEIND</th>
<th>SELECTMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCALL</td>
<td>LOADMC</td>
<td>SAVEMA</td>
<td>SERIN</td>
</tr>
<tr>
<td>FFT</td>
<td>MOP</td>
<td>SAVEMB</td>
<td>SEROUT</td>
</tr>
<tr>
<td>FTOA</td>
<td>READVAR</td>
<td>SAVEMC</td>
<td>SETARGS</td>
</tr>
<tr>
<td>LEFT</td>
<td>RET</td>
<td>SELECTA</td>
<td>SETIND</td>
</tr>
<tr>
<td>LOADIND</td>
<td>RET,CC</td>
<td>SELECTMA</td>
<td></td>
</tr>
<tr>
<td>LOADMA</td>
<td>RTC</td>
<td>SELECTMB</td>
<td></td>
</tr>
</tbody>
</table>

**Release 401**

**New Instructions**

<table>
<thead>
<tr>
<th>ADDIND</th>
<th>DREAD</th>
<th>LBIT</th>
<th>RTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPYIND</td>
<td>DWRITE</td>
<td>LCOPYI</td>
<td>SETARGS</td>
</tr>
<tr>
<td>DELAY</td>
<td>EVENT</td>
<td>LORI</td>
<td>SETIND</td>
</tr>
<tr>
<td>DEVIO</td>
<td>FCOPYI</td>
<td>LSHIFTI</td>
<td>SETREAD</td>
</tr>
<tr>
<td>DIGIO</td>
<td>LANDI</td>
<td>RDIND</td>
<td>WRIND</td>
</tr>
</tbody>
</table>

**Modified Instructions**

<table>
<thead>
<tr>
<th>ADCLOAD</th>
<th>ADCSCALE</th>
<th>SAVEIND</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCLONG</td>
<td>LOADIND</td>
<td>SERIN</td>
<td></td>
</tr>
<tr>
<td>ADCMODE</td>
<td>READVAR</td>
<td>SEROUT</td>
<td></td>
</tr>
</tbody>
</table>