# μCore Instruction Set

In the table below opcodes are grouped by functionality.

- **names** are the opcode names used in the uForth cross-compiler.
- **stack effect** indicates the effect of each opcode on the stacks. It shows a list of data stack input parameters up to the "--" follwed by the list of output parameters. Sometimes the return stack is affected as well, which is indicated by a second line with the **rs:** prefix. In a few cases there are input or output list alternatives (e.g. ?dup) - these are separated by the **|** character.

  The following conventions are used for the input and output list elements:

| **name** | **semantics** |
|---|---|
| n | 2s-complement number |
| u | unsigned number |
| flag | 0 = false, true otherwise. When flag is an output, true has all bits set. |
| addr | data memory address |
| paddr | program memory address |
| d | 2s-complement double number occupying two stack elements. The most significant word is on top of the least significant word. |
| ud | unsigned double number. The most significant word is on top of the least significant word. |
| mask | a bit mask. Usually only one single bit will be set. |
| op | 8 bit instruction |
| float | Floating point number, data_width wide. The mantissa takes the more significant bits, the exponent the less significant bits. Meaningful floating point operations are possible for data_widths >= 24. |
| man | 2s-complement mantissa. |
| exp | 2s-complement exponent. The significant number of bits is defined by **exp_width** in architecture_pkg.vhd. |

- **type** is either a **core**, an **ext.** (extended), or a **float** (floating point) opcode.
- **cycles** are the number of μCore cycles needed to execute an opcode. The prefix **u** is used to indicate uninterruptible sequences of cycles. Multi-cycle opcodes without **u** are interruptible after each cycle.

| **names** | **stack effect** | **type** | **cycles** | **description** |
|---|---|---|---|---|
| noop | -- | core | 1 | no operation |
| **Data Stack** | | | | |
| drop | n -- | core | 1 | drop top stack item |
| dup | n -- n n | core | 1 | duplicate top stack item |
| ?dup | n -- 0 \| n n | core | 1 | duplicate top stack item if it is not zero |
| swap | n1 n2 -- n2 n1 | core | 1 | exchange the top two stack items |
| over | n1 n2 -- n1 n2 n1 | core | 1 | push 2nd stack item on the stack |
| rot | n1 n2 n3 -- n2 n3 n1 | core | 1 | move 3rd stack item to the top |
| -rot | n1 n2 n3 -- n3 n1 n2 | core | 1 | move top stack item to 3rd position on the stack |
| nip | n1 n2 -- n2 | ext. | 1 | drop the 2nd stack item |
| tuck | n1 n2 -- n2 n1 n2 | ext. | 1 | equivalent to swap over |
| under | n1 n2 -- n1 n1 n2 | ext. | 1 | equivalent to over swap |

# µCore Instruction Set

| names | stack effect | type | cycles | description |
|---|---|---|---|---|
| **Return Stack** | | | | |
| >r | n --<br>rs: -- n | core | 1 | pop the top stack item and push it on the return stack |
| r> | -- n<br>rs: n -- | core | u 2 | pop the top return stack item and push it on the stack |
| r@ | -- n<br>rs: n -- n | core | 1 | push the top return stack item on the stack |
| local | offset -- addr | core | 1 | Converts an offset into the return stack into its equivalent data memory address. |
| rdrop | rs: n -- | ext. | u 2 | drop the top return stack item |
| I | -- n2<br>rs: n1 u -- n1 u | ext. | u 2 | Loop index for DO ... LOOPs |
| **Branches** | | | | |
| branch | n -- | core | 1 | If the lit flag is set, the next instruction is fetched from relative address PC+n,<br>else it is fetched from absolute address n. |
| 0=branch | flag n -- | core | u 2 | If the flag is set, continue execution at the next sequential instruction,<br>else execute the branch instruction. |
| next | rs: u -- u-1<br>rs: 0 -- | core | 1 \| u 2 | Primitive for the FOR ... NEXT loop.<br>If u > 0, u is decremented and the branch instruction is executed,<br>else the top return stack item is dropped and execution continues at the next sequential instruction. |
| call | n --<br>rs: -- paddr | core | 1 | The PC is pushed on the return stack and the branch instruction is executed. |
| exit | rs: paddr -- | core | u 2 | The top return stack item is dropped and execution continues at paddr. |
| iret | u --<br>rs: paddr -- | core | u 2 | The status register is restored from u, and the exit instruction is executed. |
| ?exit | flag --<br>rs: paddr --<br>rs: paddr -- paddr | ext. | 1 \| u 2 | Drop the top stack item.<br>When flag is true, execute the exit instruction,<br>else execution continues at the next sequential instruction. |
| **Data Memory** | | | | |
| ld | addr -- n addr | core | u 2 | LoaD pushes the content of the data memory location at addr on the stack as 2nd item.<br>`: @ ( addr -- n ) ld drop ;` |
| st | n addr -- addr | core | 1 | STore pops the 2nd item of the stack and stores it at data memory location addr.<br>`: ! ( n addr -- ) st drop ;` |
| @ | addr -- n | ext. | u 2 | Fetch the content of the data memory location at addr. |
| +! | n addr -- | ext. | u 2 | Add n to the content of the data memory location at addr and pop two stack items. This is an uninterruptible read-modify-write instruction. |

# μCore Instruction Set

| names | stack effect | type | cycles | description |
|---|---|---|---|---|
| **Unary Arithmetic** | | | | |
| not | u1 -- u2 | core | 1 | u2 is the bit-wise not of u1. |
| 0= | u -- flag | core | 1 | flag is true when u equals zero. |
| 0< | n -- flag | core | 1 | flag is true when n is negative. |
| **Shifting with Hardware Multiplier** | | | | |
| mshift | u1 n -- u1' u2 | core | 1 | Logical single-cycle barrel shift of u1 by n bit positions. u1' is the shift result, and u2 are the remaining bits that have been shifted out of u1.<br>If n is negative, a right shift is executed, the most significant bit position(s) are filled with zeros, and u2 is filled from the MSB on downwards. The carry flag is set to the MSB of u2.<br>If n is positive, a left shift is executed, the least significant bit position(s) are filled with zeros, and u2 is filled from the LSB on upwards. The carry flag is set to the LSB of u2.<br>If n is zero, u1' = u1 and u2 = 0.<br>`: shift  ( u1 n -- u1' ) mshift drop ;`<br>`: u2/    ( u1 -- u1' )   -1 shift ;`<br>`: rotate ( u1 n -- u1' ) mshift or ;` |
| mashift | n1 n2 -- n1' n3 | core | 1 | Arithmetic single-cycle barrel shift of n1 by n2 bit positions. n1' is the shift result, and n3 are the remaining bits that have been shifted out of n1.<br>If n2 is negative, a right shift is executed, the most significant bit position(s) are filled with the sign bit of n1, and n3 is filled from the MSB on downwards. The carry flag is set to the MSB of n3.<br>If n2 is positive, a left shift is executed, the least significant bit position(s) are filled with zeros, and n3 is filled from the LSB on upwards. The carry flag is set to the LSB of n3.<br>IF n2 is zero, n1' = n1 and n3 = 0.<br>`: ashift ( n1 n2 -- n3 ) mashift drop ;`<br>`: 2/     ( n1 -- n2 )    -1 shift ;` |
| **Shifting without Hardware Multiplier** | | | | |
| shift | u n -- u' | core | \|n\| | Logical shift of u by n bit positions.<br>If n is negative, a right shift is executed and the most significant bit position(s) are filled with zeros.<br>If n is positive, a left shift is executed and the least significant bit positions are filled with zeros.<br>The carry flag is set to the last bit shifted out of u. |

# µCore Instruction Set

| names | stack effect | type | cycles | description |
|---|---|---|---|---|
| ashift | n1 n2 -- n1' | core | \|n\| | Arithmetic shift of n1 by n2 bit positions.<br>If n2 is negative, a right shift is executed and the most significant bit position(s) of n1' are filled with the sign bit of n1.<br>If n is positive, a left shift is executed and the least significant bit position(s) of n1' are filled with zeros.<br>The carry flag is set to the last bit shifted out of n1. |
| c2/ | u -- u' | core | 1 | Shift right through carry. Used for multi-precision shift operations.<br>u is shifted right by one bit position and the content of the carry flag is shifted into the most significant bit position of u'. The carry flag is set to the least significant bit of u. |
| c2* | u -- u' | core | 1 | Shift left through carry. Used for multi-precision shift operations.<br>u is shifted left by one bit position and the content of the carry flag is shifted into the least significant bit position of u'. The carry flag is set to the most significant bit of u. |
| **Binary Arithmetic** | | | | |
| + | n1 n2 -- n3 | core | 1 | n3 is the 2s-complement sum of n1 + n2. |
| +c | n1 n2 -- n3 | core | 1 | n3 is the 2s-complement sum of n1 + n2 + carry flag |
| - | n1 n2 -- n3 | core | 1 | n3 is the 2s-complement difference of n1 - n2. |
| swap- | n1 n2 -- n3 | core | 1 | n3 is the 2s-complement difference of n2 - n1. |
| and | n1 n2 -- n3 | core | 1 | n3 is the logical and of n1 and n2. |
| or | n1 n2 -- n3 | core | 1 | n3 is the logical or of n1 or n2. |
| xor | n1 n2 -- n3 | core | 1 | n3 is the logical xor of n1 xor n2. |
| um* | u1 u2 -- ud | core | 1<br>dw+3 | ud is the double precision unsigned product of u1 * u2.<br>When no hardware multiplier is available, it takes data_width+3 cycles to execute. |
| * | n1 n2 -- n3 | core | 1<br>dw+3 | n3 is the signed product of n1 * n2. The overflow flag is set when the product does not fit into single precision n3.<br>When no hardware multiplier is available, it takes data_width+3 cycles to execute. |
| um/mod | ud u -- urem uquot | core | dw+2 | Quotient uquot and remainder urem are the unsigned results of dividing double number ud by unsigned divisor u.<br>It takes data_width+2 cycles to execute. |
| +sat | n1 n2 -- n3 | ext. | 1 | n3 is the 2s-complement sum of n1 + n2. In case of an overflow, n3 is set to the largest 2s-complement number. In case of an underflow it is set to the smallest 2s-complement number.<br>+sat is used to prevent control loops from oscillating in case of over/underflows. |
| 2dup_+ | n1 n2 -- n1 n2 n3 | ext. | 1 | See + |
| 2dup_+c | n1 n2 -- n1 n2 n3 | ext. | 1 | See +c |
| 2dup_- | n1 n2 -- n1 n2 n3 | ext. | 1 | See - |

# µCore Instruction Set

| names | stack effect | type | cycles | description |
|---|---|---|---|---|
| 2dup_swap- | n1 n2 -- n1 n2 n3 | ext. | 1 | See swap- |
| 2dup_and | n1 n2 -- n1 n2 n3 | ext. | 1 | See and |
| 2dup_or | n1 n2 -- n1 n2 n3 | ext. | 1 | See or |
| 2dup_xor | n1 n2 -- n1 n2 n3 | ext. | 1 | See xor |
| m* | n1 n2 -- d | ext. | 1<br>dw+38 | d is the signed double precision product of n1 * n2.<br>When no hardware multiplier is available, it takes data_width+38 cycles to execute. |
| m/mod | d n -- rem quot | ext. | dw+2 | Quotient quot and remainder rem are the signed results of dividing double number d by signed divisor n.<br>Quot is floored and rem has the same sign as divisor n.<br>It takes data_width+2 cycles to execute. |
| sqrt | u -- urem uroot | ext. | dw/2 +4 | uroot and urem are the root and the remainder after taking the square root of u.<br>It takes data_width/2+4 cycles to execute.<br>**Note:** Only defined for even data_widths. |
| **Flags** | | | | |
| status-set | mask -- | core | 1 | Status flags Carry, OVerFLow, InterruptEnable, and InterruptInService can be set or reset explicitly.<br>E.g. the carry is set using the phrase `#c status-set`, it is reset using `#c status-reset` without modifying other flags of the status register. |
| ovfl? | -- flag | core | 1 | Flag is true when the overflow flag is set. |
| carry? | -- flag | core | 1 | Flag is true when the carry flag is set. |
| time? | n -- flag | core | 1 | Flag is true when the auto-incrementing time register is larger than n.<br>The time register increments every 1/**ticks_per_ms** (see: architecture_pkg.vhd).<br>**Note:** The time register is a wrap-around counter and therefore, the maximum difference between the time register and n can be $2^{data\_width-1}$, which is the upper limit for time delays. |
| < | n1 n2 -- flag | core | 1 | Flag is true when n1 is less than n2 in 2s-complement representation. |
| flag? | mask -- flag | ext. | 1 | Flag is true when the bit selected by mask is set in the flags register. |
| **Program Memory** | | | | |
| pst | op paddr -- paddr | ext. | u 2 | ProgramSTore. Op is stored into the program memory at paddr.<br>This instruction will only be available during the cold boot phase.<br>`: p! ( op paddr -- ) pst drop ;` |

# μCore Instruction Set

| names | stack effect | type | cycles | description |
|---|---|---|---|---|
| pld | paddr -- op paddr | ext. | u 2 | ProgramLoaD. Op is fetched from program memory address paddr and stored as 2nd item on the stack.<br>This instruction may only be available during the cold boot phase.<br>`: p@ ( paddr -- op ) pld drop ;` |
| **Floating Point** | | | | |
| *. | n1 u -- n2 | float | 1 | Fractional multiply. 2s-complement n1 is multiplied by coefficient u. n2 is the most-significant word of the signed double product.<br>This operator is used to compute polynomial expressions using the Horner scheme. |
| log2 | u -- u' | float | dw+3 | u must be in the range [1 .. 2[ * $2^{data\_width-1}$.<br>u' is the logarithm dualis of u. |
| normalize | man exp -- man' exp' | float | < dw-2 | This is an auto-repeat instruction.<br>In each step, man is shifted one position to the left and exp is decrement by one until<br>a) the mantissa's sign bit and its second most significant bit have different values, or<br>b) exp has reached its minimum value.<br>In the end, man', exp' is a normalized version of man and exp.<br>It takes at most data_width-2 cycles to execute. |
| >float | man exp -- float | float | 2 | After normalization the pair man, exp is packed into floating point number float. |
| float> | float -- man exp | float | 1 | Unpack floating point number float into man and exp. |
| **Traps** | | | | |
| reset | -- | core | 1 | Hardware trap |
| interrupt | -- | core | 1 | Hardware trap |
| pause | -- | core | 1 | Hardware trap |
| break | -- | core | 1 | Soft trap for single-step tracing |
| dodoes | addr -- addr+1 | core | 1 | Soft trap compiled by DOES> |
| data! | addr n -- addr' | core | 1 | Soft trap used for data memory initialization |