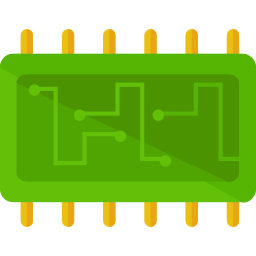
[](https://github.com/ZwerOxotnik/m-microcontroller) [FAL](https://mods.factorio.com/mod/m-microcontroller)

Factorio Assembly Language

|  |
| --- |
|  |

**REFERENCE MANUAL**

**v0.5.7**

# CONTENTS

|  |
| --- |
|  |

[0. OVERVIEW 4](#_Toc27316575)

[0.1 Registers 4](#_Toc27316576)

[0.1 Mapped Memory 4](#_Toc27316577)

[1. GLOSSARY 5](#_Toc27316578)

[1.1 Signal 5](#_Toc27316579)

[1.2 Type 5](#_Toc27316580)

[1.3 Value 5](#_Toc27316581)

[1.4 Move 5](#_Toc27316582)

[1.5 Set 5](#_Toc27316583)

[1.6 Register 5](#_Toc27316584)

[1.7 Clear 5](#_Toc27316585)

[1.8 NULL 5](#_Toc27316586)

[1.9 Label 5](#_Toc27316587)

[2. BASIC INSTRUCTION SET 6](#_Toc27316588)

[2.1 Comments 6](#_Toc27316589)

[2.2 Labels 6](#_Toc27316590)

[2.4 MOV 6](#_Toc27316591)

[2.5 SET 6](#_Toc27316592)

[2.6 SWP 7](#_Toc27316593)

[2.7 CLR 7](#_Toc27316594)

[2.8 Find In Green 7](#_Toc27316595)

[2.9 Find In Red 7](#_Toc27316596)

[2.10 JMP 7](#_Toc27316597)

[2.11 HLT 7](#_Toc27316598)

[3. ARITHMETIC INSTRUCTIONS 8](#_Toc27316599)

[3.1 ADD 8](#_Toc27316600)

[3.2 SUB 8](#_Toc27316601)

[3.3 MUL 8](#_Toc27316602)

[3.4 DIV 8](#_Toc27316603)

[3.5 MOD 8](#_Toc27316604)

[3.6 POW 9](#_Toc27316605)

[3.7 DIG 9](#_Toc27316606)

[3.8 DIS 9](#_Toc27316607)

[3.9 BITWISE AND 9](#_Toc27316608)

[3.10 BITWISE OR 9](#_Toc27316609)

[3.11 BITWISE XOR 9](#_Toc27316610)

[3.12 BITWISE NOT 9](#_Toc27316611)

[3.13 BITWISE LEFT SHIFT 9](#_Toc27316612)

[3.14 BITWISE RIGHT SHIFT 10](#_Toc27316613)

[3.15 BITWISE LEFT ROTATE 10](#_Toc27316614)

[3.16 BITWISE RIGHT ROTATE 10](#_Toc27316615)

[4. TEST INSTRUCTIONS 11](#_Toc27316616)

[4.1 TEST GREATER THAN 11](#_Toc27316617)

[4.2 TEST LESS THAN 11](#_Toc27316618)

[4.3 TEST EQUAL TO 11](#_Toc27316619)

[4.4 TEST TYPES EQUAL 11](#_Toc27316620)

[4.5 TEST TYPES NOT EQUAL 11](#_Toc27316621)

[5. BLOCKING INSTRUCTIONS 12](#_Toc27316622)

[5.1 SLP 12](#_Toc27316623)

[5.2 BKR 12](#_Toc27316624)

[5.3 BKG 12](#_Toc27316625)

[5.4 SYN 12](#_Toc27316626)

[6. INTERRUPT SIGNALS 13](#_Toc27316627)

[6.1 HLT 13](#_Toc27316628)

[6.2 RUN 13](#_Toc27316629)

[6.3 STP 13](#_Toc27316630)

[6.4 SLP 13](#_Toc27316631)

[6.5 JMP 13](#_Toc27316632)

[7. POINTERS 14](#_Toc27316633)

[7.1 MEM@N 14](#_Toc27316634)

[7.2 RED@N 14](#_Toc27316635)

[7.3 GREEN@N 14](#_Toc27316636)

[8. EXAMPLE PROGRAMS 15](#_Toc27316637)

[8.1 MULTIPLY INPUT 15](#_Toc27316638)

[8.2 ACCUMULATE INPUT 15](#_Toc27316639)

# OVERVIEW

|  |
| --- |
|  |

The Factorio Assembly Language is the future of automated manufacture. Designed ground-up for use in large-scale factories. The Factorio Assembly Language has over op-codes and the FMCU (Factorio MicroController Unit) can store 32 instructions. The FMCU has 4 internal read/write registers as well as 4 read-only registers.

|  |
| --- |
| Notes like this will appear in black outlined rectangles. These notes indicate more information on a topic |

## Registers

The FMCU has 4 internal read-write registers:

**MEM1 MEM2 MEM3 MEM4**

It also has 4 read-only registers:

**MEM5** or **IPT**: Instruction pointer index.

**MEM6** or **CNR**: Number of Signals on the Red Wire Input.

**MEM7** or **CNG**: Number of Signals on the Green Wire Input

**MEM8** or **CLK**: Monotonic clock.

## Mapped Memory

The FMCU can be extended with FRAMM (Factorio Random Access Memory Module). The FMCU has 4 external memory ports:

**North Port 01** is mapped to **MEM11-14**.

**South Port 01** is mapped to **MEM21-42**.

**North Port 02** is mapped to **MEM31-34**.

**South Port 02** is mapped to **MEM41-44**.

|  |
| --- |
| You can also connect an external FMCU to North and South Port 01. |

# GLOSSARY

|  |
| --- |
|  |

## Signal

A Type and a signed integer value.

## Type

Each signal contains a Type. The type could either refer to an item your factory consumes or produces or could be a ‘virtual’ type.

## Value

The integer part of a Signal.

## Move

Copy a Signal from one register to another.

## Set

Set the Value of a Signal to another Value.

## Register

Register A unit of memory that can store one Signal.

## Clear

Reset a Register to NULL.

## NULL

A Virtual Black Signal with a Value of 0.

## Label

A text identifier used for the jumps.

# BASIC INSTRUCTION SET

|  |
| --- |
|  |

<:I> specifies a parameter that takes a literal integer.

<:R> specifies a parameter that takes a register address.

<:W> specifies a parameter that takes a register address.

<:L> specifies a parameter that takes a register address.

|  |
| --- |
| The FMCU can only read one instruction per tick. |

## Comments

**Syntax:** #<COMMENT>

All text after the comment

## Labels

**Syntax:** :<LABEL>

Labels are used as identifiers for the jump instructions. A label is a colon followed by text. When using a label in a jump instruction you must also include the colon.

**Example:**

:LOOP

jmp :LOOP

* 1. **NOP**

**Syntax:** NOP

NOP stands for no-operation. It has no effect on the state of the internal registers. It will still take 1 tick for an FMCU to read a NOP instruction.

## MOV

**Syntax:** MOV <SRC:W/R> <DST:R>…

Takes the Signal at <SRC> and writes it to all <DST> Register(s).

## SET

**Syntax:** SET <SRC:I> <DST:R>

Takes the Value at <SRC> and write it to <DST>.

## SWP

**Syntax:** SWP <SRC:R> <DST:R>

Swaps the Signals in <SRC> and <DST>.

## CLR

**Syntax:** CLR < DST:R>…

Writes NULL to all <DST> Register(s).

## Find In Green

**Syntax:** FIG <SRC:R>

Looks for a Signal in the Green Wire Input where the Signal Type is equal to the type at <SRC>. If a signal is found it is written to MEM1.

**Example:**

fig mem21

mul mem1 2

mov mem1 out

## Find In Red

**Syntax:** FIR <SRC:R>

Looks for a Signal in the Red Wire Input where the Signal Type is equal to the type at <SRC>. If a signal is found it is written to MEM1.

## JMP

**Syntax:** JMP <SRC:I/R/L>

Jumps the instruction pointer to <SRC>. If <SRC> is a literal integer, the instruction pointer jumps to that line. If <SRC> is a Register, the instruction pointer jumps to line N where N is the value at the Register. If <SRC> is a Label, the instruction pointer jumps to the first declaration of that Label.

**Example:**

:LOOP

jmp :LOOP

## HLT

**Syntax:** HLT <SRC:R>

Halts the program

# ARITHMETIC INSTRUCTIONS

|  |
| --- |
|  |

## ADD

**Syntax:** ADD <SRC:I/R> <DST:I/R>

Adds the Value at <SRC> to the Value at <DST> and writes the result to MEM1.

## SUB

**Syntax:** SUB <SRC:I/R> <DST:I/R>

Subtracts the Value at <DST> from the Value at <SRC> and writes the result to MEM1.

## MUL

**Syntax:** MUL <SRC:I/R> <DST:I/R>

Multiplies the Value at <SRC> by the Value at <DST> and writes the result to MEM1.

## DIV

**Syntax:** DIV <SRC:I/R> <DST:I/R>

Divides the Value at <SRC> by the Value at <DST> and writes the result to MEM1.

## MOD

**Syntax:** MOD <SRC:I/R> <DST:I/R>

Executes <SRC> modulo <DST> and writes the result to MEM1.

**Example:**

:60 second clock.

add mem1 1

mod mem1 60

jmp 1

## POW

**Syntax:** POW <SRC:I/R> <DST:I/R>

Raises <SRC> to the power of <DST> and writes the result to MEM1.

|  |
| --- |
| Arithmetic instructions ignore Signal Type. |

## DIG

**Syntax:** SWP <SRC:I/R>

Reads the digit at position <SRC> from MEM1 and writes the result to MEM1.

## DIS

**Syntax:** DIS <SRC:I/R> <DST:I/R>

Writes <DST> to the digit at position <SRC> in MEM1.

If <DST> is more than 1 digit long, it writes the 1st digit.

## BITWISE AND

**Syntax:** BND <SRC:I/R> <DST:I/R>

Executes <SRC> AND <DST> then writes the result to MEM1.

## BITWISE OR

**Syntax:** BOR <SRC:I/R> <DST:I/R>

Executes <SRC> OR <DST> then writes the result to MEM1.

## BITWISE XOR

**Syntax:** BXR <SRC:I/R> <DST:I/R>

Executes <SRC> XOR <DST> then writes the result to MEM1.

## BITWISE NOT

**Syntax:** BND <SRC:I/R>

Executes NOT <SRC> then writes the result to MEM1.

## BITWISE LEFT SHIFT

**Syntax:** BLS <SRC:I/R> <DST:I/R>

Shifts bits in <SRC> by <DST> to the left, then writes the result to MEM1.

## BITWISE RIGHT SHIFT

**Syntax:** BRS <SRC:I/R> <DST:I/R>

Shifts bits in <SRC> by <DST> to the right, then writes the result to MEM1.

## BITWISE LEFT ROTATE

**Syntax:** BLR <SRC:I/R> <DST:I/R>

Rotate bits in <SRC> by <DST> to the left, then writes the result to MEM1.

## BITWISE RIGHT ROTATE

**Syntax:** BRR <SRC:I/R> <DST:I/R>

Rotate bits in <SRC> by <DST> to the right, then writes the result to MEM1.

# TEST INSTRUCTIONS

|  |
| --- |
|  |

Test instructions will skip the next instruction if the test is successful.

## TEST GREATER THAN

**Syntax:** TGT <SRC:I/R> <DST:I/R>

Tests if <SRC> Value is greater than <DST> Value.

## TEST LESS THAN

**Syntax:** TLT <SRC:I/R> <DST:I/R>

Tests if <SRC> Value is less than <DST> Value.

## TEST EQUAL TO

**Syntax:** TEQ <SRC:I/R> <DST:I/R>

Tests if <SRC> Value is equal to <DST> Value.

## TEST TYPES EQUAL

**Syntax:** TTE <SRC:R> <DST:R>

Tests if <SRC> Type is equal to <DST> Type.

## TEST TYPES NOT EQUAL

**Syntax:** TTN <SRC:R> <DST:R>

Tests if <SRC> Type is not equal to <DST> Type.

# BLOCKING INSTRUCTIONS

|  |
| --- |
|  |

Blocking instructions will pause the program until the operation is complete.

## SLP

**Syntax:** SLP <SRC:I/R>

Program will sleep for <SRC> ticks.

## BKR

**Syntax:** BKR <SRC:I/R>

Pause the program until there is at least <SRC> Signals on the Red Wire Input.

## BKG

**Syntax:** BKG <SRC:I/R>

Pause the program until there is at least <SRC> Signals on the Green Wire Input.

## SYN

**Syntax:** SYN

Pause the program until all other connected FMCUs call SYN.

# INTERRUPT SIGNALS

|  |
| --- |
|  |

There are 5 special signals that can be used to interrupt a program. When an FMCU receives an interrupt signal on either it's Green or Red Wire Input I will immediately execute the interrupt.

## HLT

Halts the program

## RUN

Runs the program

## STP

Steps the program (executes the current instruction then halts).

## SLP

Program will sleep for N ticks, where N is the Signal’s Value.

## JMP

Jumps the Program Instruction Pointer to N, where N is the Signal’s Value.

# POINTERS

|  |
| --- |
|  |

When specifying a memory address as a parameter to an instruction, you may also pass a memory pointer. A pointer is a special address were the literal address is evaluated at run-time.

Typically a memory address takes the form MEM1. This instructs the FMCU to access the 1st Register. A pointer takes the from MEM@1. This instructs the FMCU to read the Value at Register 1 and then read the Value at Register N, where N was the Value at Register 1.

## MEM@N

Access register X, where X is the Value at Register N.

## RED@N

Access Red Wire Input X, where X is the Value at Register N.

## GREEN@N

Access Green Wire Input X, where X is the Value at Register N.

# EXAMPLE PROGRAMS

|  |
| --- |
|  |

## MULTIPLY INPUT

This program takes the 1st Red Wire Input, doubles it and outputs the result.

|  |  |
| --- | --- |
| mov red1 mem1  mul mem1 2  mov mem1 out | # Write Red wire Input 1 to Register 1  # MEM1 = MEM1 \* 2  # Write Register 1 to Output |

## ACCUMULATE INPUT

This program takes the first 4 Signals on the Red Wire Input and accumulates them over time. It requires a FRAMM at North Port 01.

|  |  |
| --- | --- |
| clr  set 11 mem2  set 3 mem2  :loop  mov red@3 mem1  add mem1 mem@2  mov mem1 mem@2  add mem2 1  tlt mem1 15  set 11 mem1  mov mem1 mem2  add mem3 1  tlt mem1 5  set 1 mem1  mov mem1 mem3 | # Clear all registers  # Set MEM2 to 11  # Set MEM2 to 3  # Create a label  # Write RED[MEM3] to MEM1  # MEM1 = MEM1 + MEM[MEM2]  # Write MEM1 to MEM[MEM2]  # MEM1 = MEM2 + 1  # Skip next line if MEM1 < 15  # Set MEM1 to 11  # Write MEM1 to MEM2  # MEM1 = MEM3 + 1  # Skip next line if MEM1 < 5  # Set MEM1 to 1  # Write MEM1 to MEM3 |