



# FAL

Factorio Assembly Language

---

**REFERENCE MANUAL**  
**v0.10.0**

# CONTENTS

---

0. OVERVIEW	4
0.1 Registers	4
0.1 Mapped Memory	4
1. GLOSSARY	5
1.1 Signal	5
1.2 Type	5
1.3 Value	5
1.4 Move	5
1.5 Set	5
1.6 Register	5
1.7 Clear	5
1.8 NULL	5
1.9 Label	5
2. BASIC INSTRUCTION SET	6
2.1 Comments	6
2.2 Labels	6
2.4 MOV	6
2.5 SET	6
2.6 SWP	7
2.7 CLR	7
2.8 Find In Green	7
2.9 Find In Red	7
2.10 JMP	7
2.11 HLT	7
3. ARITHMETIC INSTRUCTIONS	8
3.1 ADD	8
3.2 SUB	8
3.3 MUL	8
3.4 DIV	8
3.5 MOD	8
3.6 POW	9

3.7 DIG	9
3.8 DIS	9
3.9 BITWISE AND	9
3.10 BITWISE OR	9
3.11 BITWISE XOR	9
3.12 BITWISE NOT	9
3.13 BITWISE LEFT SHIFT	9
3.14 BITWISE RIGHT SHIFT	10
3.15 BITWISE LEFT ROTATE	10
3.16 BITWISE RIGHT ROTATE	10
4. TEST INSTRUCTIONS	11
4.1 TEST GREATER THAN	11
4.2 TEST LESS THAN	11
4.3 TEST EQUAL TO	11
4.4 TEST TYPES EQUAL	11
4.5 TEST TYPES NOT EQUAL	11
5. BLOCKING INSTRUCTIONS	12
5.1 SLP	12
5.2 BKR	12
5.3 BKG	12
5.4 SYN	12
6. INTERRUPT SIGNALS	13
6.1 HLT	13
6.2 RUN	13
6.3 STP	13
6.4 SLP	13
6.5 JMP	13
7. POINTERS	14
7.1 MEM@N	14
7.2 RED@N	14
7.3 GREEN@N	14
8. EXAMPLE PROGRAMS	15
8.1 MULTIPLY INPUT	15
8.2 ACCUMULATE INPUT	15

# 0. 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 each MicroController can store 32 instructions.

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

## 0.1 Registers

MicroControllers 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.

## 0.1 Mapped Memory

The MicroController can be extended with RAM module. The MicroController 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 a MicroController to North and South Port 01.*

# 1. GLOSSARY

---

## **1.1 Signal**

A Type and a signed integer value.

## **1.2 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.

## **1.3 Value**

The integer part of a Signal.

## **1.4 Move**

Copy a Signal from one register to another.

## **1.5 Set**

Set the Value of a Signal to another Value.

## **1.6 Register**

Register A unit of memory that can store one Signal.

## **1.7 Clear**

Reset a Register to NULL.

## **1.8 NULL**

A Virtual Black Signal with a Value of 0.

## **1.9 Label**

A text identifier used for the jumps.

# 2. 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 wire Input address.  
<:L> specifies a parameter that takes a label.

*Each MicroController can only read one instruction per Nth tick.  
(customizable)*

## 2.1 Comments

**Syntax:** #<COMMENT>

All text after the comment

## 2.2 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
```

## 2.3 NOP

**Syntax:** nop

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

## 2.4 MOV

**Syntax:** mov <SRC:W/R> <DST:R>...

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

## 2.5 SET

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

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

## 2.6 SWP

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

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

## 2.7 CLR

**Syntax:** clr < DST:R>...

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

## 2.8 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
```

## 2.9 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.

## 2.10 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
```

## 2.11 HLT

**Syntax:** hlt <SRC:R>

Halts the program

# 3. ARITHMETIC INSTRUCTIONS

---

## 3.1 ADD

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

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

## 3.2 SUB

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

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

## 3.3 MUL

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

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

## 3.4 DIV

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

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

## 3.5 MOD

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

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

**Example:**

```
:60 second clock.
```

```
add mem1 1
```



mod mem1 60

jmp 1

### 3.6 POW

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

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

*Arithmetic instructions ignore Signal Type.*

### 3.7 DIG

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

Reads the digit at position <SRC> from mem1 and writes the result to mem1.

### 3.8 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 1<sup>st</sup> digit.

### 3.9 BITWISE AND

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

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

### 3.10 BITWISE OR

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

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

### 3.11 BITWISE XOR

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

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

### 3.12 BITWISE NOT

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

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

### 3.13 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 memN.

### 3.14 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 memN.

### 3.15 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 memN.

### 3.16 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 memN.

# 4. TEST INSTRUCTIONS

---

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

## 4.1 TEST GREATER THAN

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

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

## 4.2 TEST LESS THAN

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

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

## 4.3 TEST EQUAL TO

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

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

## 4.4 TEST TYPES EQUAL

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

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

## 4.5 TEST TYPES NOT EQUAL

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

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

# 5. BLOCKING INSTRUCTIONS

---

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

## 5.1 SLP

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

Program will sleep for <SRC> ticks.

## 5.2 BKR

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

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

## 5.3 BKG

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

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

## 5.4 SYN

**Syntax:** syn

Pause the program until all other connected MicroControllers call SYN.

# 6. INTERRUPT SIGNALS

---

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

## **6.1 HLT**

Halts the program

## **6.2 RUN**

Runs the program

## **6.3 STP**

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

## **6.4 SLP**

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

## **6.5 JMP**

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

# 7. 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 where the literal address is evaluated at run-time.

Typically a memory address takes the form `mem1`. This instructs the MicroController to access the 1<sup>st</sup> Register. A pointer takes the form `mem@1`. This instructs the MicroController to read the Value at Register 1 and then read the Value at Register N, where N was the Value at Register 1.

## 7.1 `mem@n`

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

## 7.2 `red@n`

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

## 7.3 `green@n`

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

# 8. EXAMPLE PROGRAMS

---

## 8.1 MULTIPLY INPUT

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

```
mov red1 mem1      # Write Red wire Input 1 to Register 1
mul mem1 2          # mem1 = mem1 * 2
mov mem1 out        # Write Register 1 to Output
```

## 8.2 ACCUMULATE INPUT

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

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