FAL

Factorio Assembly Language

REFERENCE MANUAL
v0.5.7

CONTENTS

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 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*

0.1 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.

0.1 Napped 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.*

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 INSTUCTION 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.*

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 tick for an FMCU 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: MOV <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 MEM1.

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 MEM1.

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 MEM1.

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 MEM1.

3.5 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
```

3.6 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.*

3.7 DIG

Syntax: SWP <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 1st digit.

3.9 BITWISE AND

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

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

3.10 BITWISE OR

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

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

3.11 BITWISE XOR

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

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

3.12 BITWISE NOT

Syntax: BND <SRC:I/R>

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

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 MEM1.

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 MEM1.

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 MEM1.

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 MEM1.

4. ARITHMETIC 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 FMCUs call SYN.

6. 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.

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 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.

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 1st 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 FRAMM 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
tlt 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
tlt mem1 5        # Skip next line if MEM1 < 5
set 1 mem1        # Set MEM1 to 1
mov mem1 mem3     # Write MEM1 to MEM3
```