Solar Orbiter Level 3 Trigger Instruction Set

Stephan I. Böttcher

Revision 5980
Date 2017-03-23 14:31:25 +0100 (Do, 23 Mär 2017)

Contents

1 Architecture

1.1 Instruction Memory

The Instruction Memory has room for 1024 opcodes with size 32 bits. The Processor can start executing at any address that is a multiple of 256, i.e., at four entry points hex 0x000, 0x100, 0x200, and 0x300.

The Level 2 trigger shall choose the entry point for each trigger event.

1.2 Registers

The L3 processor has 256 registers. Each command stores its result into the register identified by the eight LSB of the command address. The only exception is the POKE opcode[1], which stores the result into R_d.

An opcode may use the values of up to two registers, R_x and R_y.

If a register is required that is located one or two addresses before the current opcode address, the value will not be fetched from the register file, but uses fastpath state stored in the execution unit.

This may lead to undefined behaviour if the opcode was reached by a recent GOTO, so that the state in the execution unit does not match the register contents. Don't do that.[2]

The data word size of the registers is 29 bits.[3]

1.3 Status Bits

In addition to the registers, the processor maintains state in two status bits.

The status bit C is changed by the CMP opcodes and by the BITC opcode. The status bit S can only be changed by the BITS opcode.

1.4 Conditional Execution

Every opcode is executed conditionally, depending on the c_2 condition bits in the opcode and the status of C and S. The conditions are

[1] or any future extension derived by defining reserved bits of the POKE opcode

[2] It may be safe and well defined if the target of a GOTO refers to $R_{[.-1]}$ to get the result of the opcode executed just before the GOTO.

[3] The physical memory size is 36 bits, with 7 bits used for EDAC.

c_2	mnm	condition
00		always
01	`ifS`	S is set
10	`ifN`	C is not set
11	`ifC`	C is set

The result of an opcode that does not meet the condition is the result of the previously executed opcode. This is also true after a GOTO. A POKE opcode that does not execute stores its result into the registers identified by its address, not to the poke destination.

1.5 Input

The event data is loaded into the last 32 registers from address hex 0xe0 to 0xff before the execution is started.

1.6 Output

Two opcodes produce output. HIST is equivalent to an ADD or SUB, and the result will be provided to the outside, to be interpreted as an address to the histogram memory that shall be incremented.

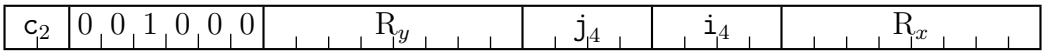
PHA is similar to an ADDI, and the result shall initiate and guide the storage of the raw event data.

2 Opcodes

STOP		cc00 0000 0--- ---- ---- ---- ---- ----
NOP		cc00 0000 10-- ---- ---- ---- ---- ----
GOTO	$\mathtt{u}_{10}$	cc00 0000 11-- --uu uuuu uuuu ---- ----
LOG	R_x	cc00 0001 ---- ---- ---- ---- xxxx xxxx
POKE	$\mathrm{R}_d = \mathrm{R}_x$	cc00 0010 ---- ---- dddd dddd xxxx xxxx
BITC	$\mathrm{R}_x\{\mathtt{u}_5\}$	cc00 0011 n0SC ---- ---u uuuu xxxx xxxx
BITS	$\mathrm{R}_x\{\mathtt{u}_5\}$	cc00 0011 n1SC ---- ---u uuuu xxxx xxxx
BRNG	$\mathrm{R}_x\{\mathtt{v}_5 : \mathtt{u}_5\}$	cc00 0100 ---- --vv vvvu uuuu xxxx xxxx
TRIM	$\mathrm{R}_x, \mathtt{u}_8, \mathtt{v}_8$	cc00 0101 vvvv vvvv uuuu uuuu xxxx xxxx
MULI	$\mathrm{R}_x * \mathtt{m}_{12} \gg \mathtt{e}_4$	cc00 0110 eeee mmmm mmmm mmmm xxxx xxxx
PHA	$\mathrm{R}_x + \mathtt{u}_{16}$	cc00 0111 uuuu uuuu uuuu uuuu xxxx xxxx
ADD	$\mathrm{R}_x \gg \mathtt{i}_4 + \mathrm{R}_y \gg \mathtt{j}_4$	cc00 1000 yyyy yyyy jjjj iiii xxxx xxxx
SUB	$\mathrm{R}_x \gg \mathtt{i}_4 - \mathrm{R}_y \gg \mathtt{j}_4$	cc00 1001 yyyy yyyy jjjj iiii xxxx xxxx
HIST	$\mathrm{R}_x \gg \mathtt{i}_4 + \mathrm{R}_y \gg \mathtt{j}_4$	cc00 1100 yyyy yyyy jjjj iiii xxxx xxxx
HIST	$\mathrm{R}_x \gg \mathtt{i}_4 - \mathrm{R}_y \gg \mathtt{j}_4$	cc00 1101 yyyy yyyy jjjj iiii xxxx xxxx
CMP	$\mathrm{R}_x + \mathtt{u}_8 <=> \mathrm{R}_y$	cc01 0ooo yyyy yyyy uuuu uuuu xxxx xxxx
CMP	$\mathrm{R}_x <=> \mathrm{R}_y + \mathtt{u}_8$	cc01 1ooo yyyy yyyy uuuu uuuu xxxx xxxx
ADDI	$\mathrm{R}_x + \mathtt{i}_{21}$	cc1i iiii iiii iiii iiii iiii xxxx xxxx

$\mathtt{c}_2$ is a condition, depending on two status bits C and S. $\mathtt{u}_n$, $\mathtt{v}_n$, $\mathtt{m}_n$, and $\mathtt{e}_n$ are unsigned integers. $\mathtt{i}_n$ and $\mathtt{j}_n$ are signed 2s-complement numbers. Negative shifts shift into the opposite direction. <=> is any comparison operator, encoded in $\mathtt{o}_3$. Unassigned - bits are reserved and should be zero. Future extensions may define additional opcodes where some of the reserved bits are non-zero.

2.1 ADD



Syntax:

ADD $\mathrm{R}_x \gg \mathtt{i}_4 + \mathrm{R}_y \gg \mathtt{j}_4$

Result:

$\mathrm{R}_x \cdot 2^{-\mathtt{i}_4} + \mathrm{R}_y \cdot 2^{-\mathtt{j}_4}$

Effect:

Add two register values, each may be shifted before addition by up to eight bits to the left or seven bits to the right.

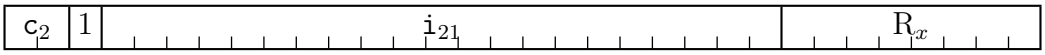
Caveats:

The result is properly sign extended and overflow is prevented.

Parser:

ADD ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩] '+' ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩]
⟨*BSHIFT*⟩ : {'<<'|'>>'} ⟨*MEXPR*⟩

2.2 ADDI



Syntax:

ADDI $\mathrm{R}_x + \mathtt{i}_{21}$
ADDI $\mathrm{R}_x - \mathtt{i}_{21}$

Result:

$\mathrm{R}_x + \mathtt{i}_{21}$

Effect:

Compute the sum of a signed constant and a register value.

Caveats:

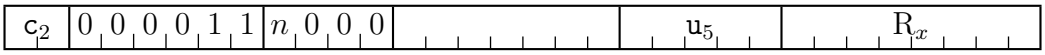
The result is properly sign extended and overflow is prevented.

Parser:

ADDI ⟨*RSPEC*⟩ [{'+'|'-'} ⟨*EXPR*⟩]

Operators in ⟨*EXPR*⟩ bind stronger than the initial {'+'|'-'}.

2.3 BITC



Syntax:

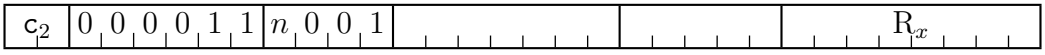
BITC ['~'] R_x{u_5}

Result:

R_x

Effect:

Set the status bit C to the [inverted] value of a bit u_5 from R_x. The value is inverted, indicated by ~, when the bit n is set in the opcode.

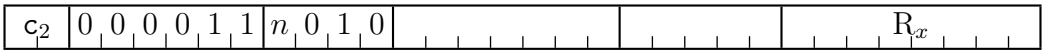


Syntax:

BITC [~] C

Effect:

Set the status bit C to the [inverted] value of C.

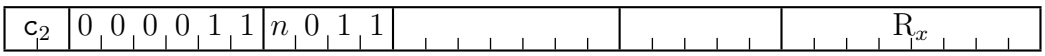


Syntax:

BITC [~] S

Effect:

Set the status bit C to the [inverted] value of S.



Syntax:

BITC [~] 0|1

Effect:

Set the status bit C to 0, or [1].

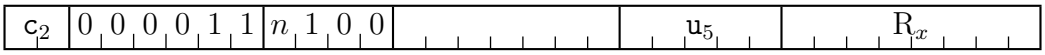
Caveats:

There is no syntax support to select R_x when testing the status bits or constants.

Parser:

BITC ['~'] ⟨*RSPEC*⟩ '{' ⟨*EXPR*⟩ '}'
BITC ['~'] { 'C' | 'S' | '0' | '1' }

2.4 BITS



Syntax:

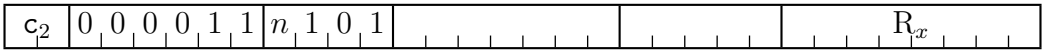
BITS [~] R_x{u_5}

Result:

R_x

Effect:

Set the status bit S to the [inverted] value of a bit u_5 from R_x. The value is inverted, indicated by ~, when the bit n is set in the opcode.

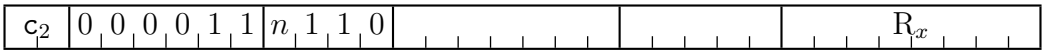


Syntax:

BITS [~] C

Effect:

Set the status bit S to the [inverted] value of C.

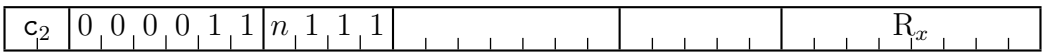


Syntax:

BITS [~] S

Effect:

Set the status bit S to the [inverted] value of S.



Syntax:

BITS [~] 0|1

Effect:

Set the status bit S to 0, or [1].

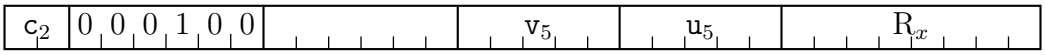
Caveats:

There is no syntax support to select R_x when testing the status bits or constants.

Parser:

BITS ['~'] ⟨*RSPEC*⟩ '{' ⟨*EXPR*⟩ '}'
BITS ['~'] { 'C' | 'S' | '0' | '1' }

2.5 BRNG



Syntax:

BRNG $R\{\mathtt{v}_5 : \mathtt{u}_5\}$

Result:

when $\mathtt{v}_5 \geq \mathtt{u}_5$: R_x with all bits except $\{\mathtt{v}_5 : \mathtt{u}_5\}$ cleared
when $\mathtt{v}_5 < \mathtt{u}_5$: R_x with bits $\{\mathtt{u}_5 - 1 : \mathtt{v}_5 + 1\}$ cleared

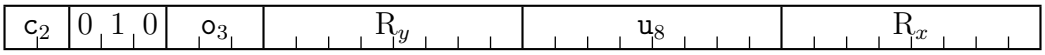
Effect:

Return the value of R_x with some bits masked out.

Parser:

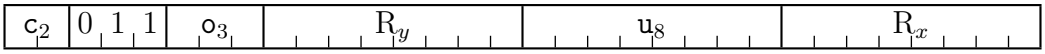
BRNG ⟨*RSPEC*⟩ '{' ⟨*EXPR*⟩ [':' ⟨*EXPR*⟩] '}'

2.6 CMP



Syntax:

CMP $\mathrm{R}_x + \mathtt{u}_8 < \mathrm{R}_y$
CMP $\mathrm{R}_x + \mathtt{u}_8 = \mathrm{R}_y$
CMP $\mathrm{R}_x + \mathtt{u}_8 \leq \mathrm{R}_y$
CMP $\mathrm{R}_x + \mathtt{u}_8 > \mathrm{R}_y$
CMP $\mathrm{R}_x + \mathtt{u}_8 \neq \mathrm{R}_y$
CMP $\mathrm{R}_x + \mathtt{u}_8 \geq \mathrm{R}_y$



Syntax:

CMP $\mathrm{R}_x < \mathrm{R}_y + \mathtt{u}_8$
CMP $\mathrm{R}_x = \mathrm{R}_y + \mathtt{u}_8$
CMP $\mathrm{R}_x \leq \mathrm{R}_y + \mathtt{u}_8$
CMP $\mathrm{R}_x > \mathrm{R}_y + \mathtt{u}_8$
CMP $\mathrm{R}_x \neq \mathrm{R}_y + \mathtt{u}_8$
CMP $\mathrm{R}_x \geq \mathrm{R}_y + \mathtt{u}_8$

Result:

R_x

Effect:

Set status bit C to the result of the comparison.

$\mathtt{o}_3 = 0b\,000$:	0	$\mathtt{o}_3 = 0b\,100$:	$>$
$\mathtt{o}_3 = 0b\,001$:	$<$	$\mathtt{o}_3 = 0b\,101$:	$\neq$
$\mathtt{o}_3 = 0b\,010$:	$=$	$\mathtt{o}_3 = 0b\,110$:	$\geq$
$\mathtt{o}_3 = 0b\,011$:	$\leq$	$\mathtt{o}_3 = 0b\,111$:	1

Caveats:

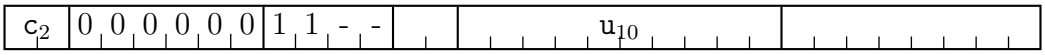
$\mathtt{o}_3 = 0b\,000$ is always false, $\mathtt{o}_3 = 0b\,111$ is always true. There is no syntax to support these cases. Use BITC.

Parser:

CMP ⟨*RSPEC*⟩ [{'+'|'-'} ⟨*EXPR*⟩] ⟨*COP*⟩ ⟨*RPEC*⟩
CMP ⟨*RSPEC*⟩ ⟨*COP*⟩ ⟨*RPEC*⟩ [{'+'|'-'} ⟨*EXPR*⟩]
⟨*COP*⟩ : { '<' | '==' | '<=' | '>' | '!=' | '>=' }

Operators in ⟨*EXPR*⟩ bind stronger than the explicit operators.

2.7 GOTO



Syntax:

GOTO u_{10}

Result:

The previous opcodes result.

Effect:

Continue execution at address u_{10}.

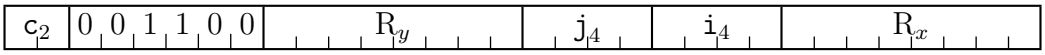
Caveats:

A taken GOTO causes a pipeline flush. GOTO target opcodes may get confused by the register fastpath.

Parser:

GOTO ⟨*RSPEC*⟩

2.8 HIST

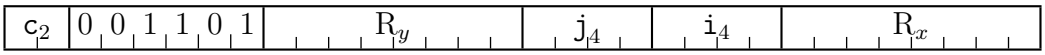


Syntax:

HIST $\mathrm{R}_x \gg \mathtt{i}_4 + \mathrm{R}_y \gg \mathtt{j}_4$

Result:

$\mathrm{R}_x \cdot 2^{-\mathtt{i}_4} + \mathrm{R}_y \cdot 2^{-\mathtt{j}_4}$



Syntax:

HIST $\mathrm{R}_x \gg \mathtt{i}_4 - \mathrm{R}_y \gg \mathtt{j}_4$

Result:

$\mathrm{R}_x \cdot 2^{-\mathtt{i}_4} - \mathrm{R}_y \cdot 2^{-\mathtt{j}_4}$

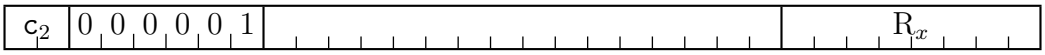
Effect:

Same as ADD or SUB. The result is transferred to the histogram memory as the address of a counter to be incremented.

Parser:

HIST ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩] {'+'|'-'} ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩]
⟨*BSHIFT*⟩ : {'<<'|'>>'} ⟨*MEXPR*⟩

2.9 LOG



Syntax:

LOG R_x

Result:

$16 \log_2(2R_x + 0.99)$. If $R_x < 0$ return 0.

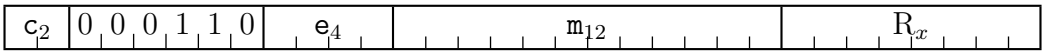
Effect:

Computes the logarithm of the value of R_x.

Parser:

LOG $\langle RSPEC \rangle$

2.10 MULI



Syntax:

MULI $R_x * \mathtt{m}_{12} \gg \mathtt{e}_4$
MULI $R_x * \langle \textit{float} \rangle$

Result:

$R_x * \frac{\mathtt{m}_{12}}{2^{\mathtt{e}_4}}$

Effect:

Multiply R_x with an unsigned constant.

Caveats:

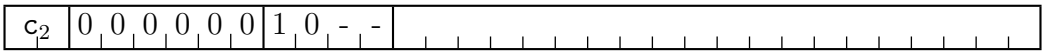
No overflow checks. A $\langle \textit{float} \rangle$ may be given instead of the explicit mantissa and exponents of the factor. The assembler will compute the best representation of the $\langle \textit{float} \rangle$ in terms of $\mathtt{m}_{12}$ and $\mathtt{e}_4$.

Parser:

MULI $\langle \textit{RSPEC} \rangle$ '*' $\langle \textit{MEXPR} \rangle$ ['>>' $\langle \textit{AEXPR} \rangle$]

The explicit toplevel operator '*' binds stronger than the explicit '>>'.

2.11 NOP



Syntax:

NOP

Result:

The previous opcodes result.

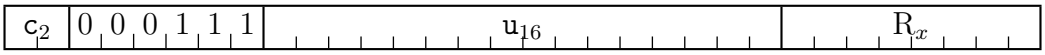
Effect:

None. May be useful as a GOTO target.

Parser:

NOP

2.12 PHA



Syntax:

PHA $R_x + u_{16}$

Result:

$R_x + u_{16}$

Effect:

Like ADDI, the result is transmitted to the event data storage unit, to guide the storage of *Pulse Height Analysis* records.

Caveats:

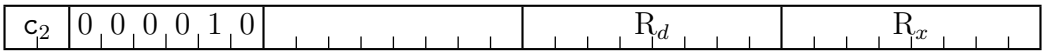
The constant u_{16} is unsigned.

Parser:

PHA ⟨*RSPEC*⟩ [{'+'|'-'|','} ⟨*EXPR*⟩]

Operators in ⟨*EXPR*⟩ bind stronger than the initial {'+'|'-'}.

2.13 POKE



Syntax:

POKE $R_d = R_x$

Result:

None

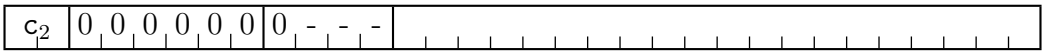
Effect:

Stores the value of R_x into register R_d, but not into the normal result register identified by the instruction address. If the condition is not true, the last opcodes result *will* be stored into the normal result register.

Parser:

POKE ⟨*RSPEC*⟩ '=' ⟨*RSPEC*⟩

2.14 STOP



Syntax:

STOP

Result:

The previous opcodes result, but who cares?

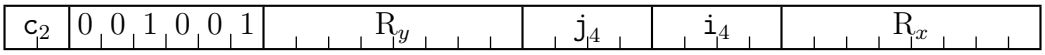
Effect:

The execution of the trigger processor stops. The processor becomes ready for a new event.

Parser:

STOP

2.15 SUB



Syntax:

SUB $\mathrm{R}_x \gg \mathtt{i}_4 - \mathrm{R}_y \gg \mathtt{j}_4$

Result:

$\mathrm{R}_x \cdot 2^{-\mathtt{i}_4} - \mathrm{R}_y \cdot 2^{-\mathtt{j}_4}$

Effect:

Subtract two register values, each may be shifted before subtraction by up to eight bits to the left or seven bits to the right.

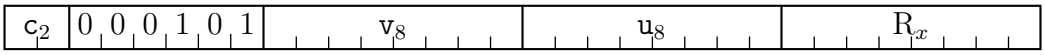
Caveats:

The result is properly sign extended and overflow is prevented.

Parser:

SUB ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩] `'-'` ⟨*RSPEC*⟩ [⟨*BSHIFT*⟩]
⟨*BSHIFT*⟩ : {`'<<'`|`'>>'`} ⟨*MEXPR*⟩

2.16 TRIM



Syntax:

TRIM R_x, u_8, v_8

Result:

when $R_x \leq u_8$: 0
when $u_8 \leq R_x \leq u_8 + v_8$: $R_x - u_8$
when $v_8 \leq R_x$: v_8

Effect:

Return R_x, limited to a range, offset to zero.

Parser:

TRIM ⟨*RSPEC*⟩ ',' ⟨*EXPR*⟩ ',' ⟨*EXPR*⟩

2.17 Syntax

Optional parts are set in square brackets [...]. Parts set in double square brackets may appear any number of times. Alternatives are set in curly brackets and separated by bars {...|...}.

⟨*LINE*⟩ : [{ ⟨*ASSIGN*⟩ | ⟨*COMMAND*⟩ | ⟨*DIRECTIVE*⟩ }] [{'#' | '"""' } ⟨*COMMENT*⟩]
⟨*ASSIGN*⟩ : ⟨*ID*⟩ '=' ⟨*EXPR*⟩
⟨*COMMAND*⟩ : [⟨*ID*⟩ '='] [⟨*COND*⟩] ⟨*INSTR*⟩
⟨*COND*⟩ : [{ 'ifC' | 'ifS' | 'ifN' | 'if1' }]
⟨*INSTR*⟩ : ⟨*MEMONIC*⟩ [⟨*PARAMETERS*⟩], see previous sections
⟨*EXPR*⟩ : ⟨*AEXPR*⟩ [{'<<' | '>>'} ⟨*AEXPR*⟩]
⟨*AEXPR*⟩ : ⟨*MEXPR*⟩ [[{'+'|'-'} ⟨*MEXPR*⟩]]
⟨*MEXPR*⟩ : ⟨*EEXPR*⟩ [[{'∗' | '/'} ⟨*EEXPR*⟩]]
⟨*EEXPR*⟩ : [[{'+'|'-'|'!'}]] {⟨*ID*⟩ | ⟨*NUM*⟩ | ⟨*REG*⟩ | ⟨*FUNC*⟩ | '(' ⟨*EXPR*⟩ ')' }
⟨*RSPEC*⟩ : ⟨*EEXPR*⟩
⟨*REG*⟩ : 'R' '[' ⟨*EXPR*⟩ ']'
⟨*ID*⟩ : '/[.a-zA-Z][.a-zA-Z0-9_]+/'
⟨*NUM*⟩ : float(·) | int(·, 0)
⟨*FUNC*⟩ : '$' ⟨*ID*⟩ '(' ⟨*EXPR*⟩ [[',' ⟨*EXPR*⟩]] ')'
⟨*CEXPR*⟩ : ⟨*EXPR*⟩ [[⟨*COP*⟩ ⟨*EXPR*⟩]]

Expressions can have three different types, integer, float, and register. An ⟨*RSPEC*⟩ must be of type register. All other expressions in ⟨*PARAMETERS*⟩ must be integer, except for a MULI factor without explicit shift, which may be a float.

3 Assembler

The assembler `l3.py` parses a program and yields the opcodes in hexadecimal notation. A program consists of a stream of lines. A line can be

- a comment,
- an assignment,
- a directive,
- an opcode statement.

A comment is a line that contains only whitespace up to the first `#` character. Other lines may contain comments after a `#`, except for some directives.

A sequence of three double quotes is also treated as a comment, up to the end of the line. This allows to hide assemby statements from Python, for example to load constant definitions into a python script.

3.1 Assignments

An assignment has the form

⟨*identifier*⟩ `=` ⟨*expression*⟩

An ⟨*identifier*⟩ is a name composed of letters, digits, underscore and periods. It must not start with a digit. Case is significant.

The period by itself is an identifier that represents the current instruction memory address. This can and should be the target of an assignment. The address increments after each opcode instruction.

Identifiers may be redefined. The last definition preceding the current line is used. Identifiers subject to a `.forward` directive shall not be explicitly defined multiple times.

3.2 Expressions

An ⟨*expression*⟩ can be composed of identifiers, numbers, function calls and register addresses, combined with operators and parenthesis.

Numbers can be any Python integer constants or floats. Floats are not supported everywhere. Floats are mostly useful as arguments for the MULI opcode, directly or via identifiers.

3.2.1 Register expressions

A register value represents a command address and its associated result register. An expression of type register is any identifier that represents such a value, or a term of the form `R[`⟨*expression*⟩`]`. The difference of two register values is an integer. The sum of a register and an integer is a register. No further math is allowed with register values.

To cast any expression to a register, put it into an `R[]`. To cast a register expression to an integer, subtract `R[0]`.

3.2.2 Functions

An expression can call functions. Function names must be preceeded with a `$` sign.

`$LOG():` compute a logarithm base 2, just the same as the ⟨*LOG*⟩ instruction, yield an integer.

`$log2():` compute the logarithm to base 2, yield a float.

`$floor():` yield an integer.

`$ceil():` yield an integer.

`$power(b,x):` compute the power b^x.

3.3 Directives

A directive starts with one of the following reserved identifiers:

`.print` ⟨*text*⟩
: Print the remaining line to the diagnostic output. Substrings of the form `$IDENDIFIER` or `${IDENTIFIER}` are substituted by the value of the identifier, which must be defined.

`.include` ⟨*filename*⟩
: The rest of the line must be a filename. The directive is replaced by the contents of the file. Includes can be nested to any depth.

`.forwardfile` ⟨*filename*⟩
: Open an auxiliary file for forward declarations.

  The rest of the line must be a filename that may not necessarily exist and that may be written or replaced by the assembler at the end of the compilation.

The contents of the file is included, if it exists. A new file is opened for writing, with `.new` appended to the filename.

At the end of the compilation the new file replaces the old one when any `.forward` mismatches were detected.

`.forward` ⟨*identifier*⟩
: Declare a identifier that is used before it is defined.

  A `.forwardfile` directive must precede any `.forward` directive.

  The next token on the line must be an identifier. If the identifier is not defined, it will be defined with the value 0. The identifier is then marked as a forward declaration. If the identifier is later redefined, an assignment will be written to the forwardfile. If the new value differs from the old value, a warning is emitted, and the forwardfile is marked for replacement.

  In the first assembly run, the forwardfile will not exist, most forward declarations will mismatch, and the new forwardfile will be saved. In the following runs, the forward declarations will be defined by the inclusion of the existing forwardfile. Unless the program changed, no mismatches will happen, and the program will assemble properly. The forwardfile will not be replaced, to not confuse the Makefile.

`.name` ⟨*identifier*⟩
: Set the name of the current source file, to assign a version to this source.

`.version` ⟨*text*⟩
: Set the version string for this source file. Put the Subversion `$Revision$` on this line.

`.if` ⟨*expression*⟩

`.ifdef` ⟨*identifier*⟩

`.ifndef` ⟨*identifier*⟩

`.elseif` ⟨*expression*⟩

`.else`

`.endif`
: Conditional assembly.

3.4 Opcode Statements

An assembly statement has the form

[⟨*identifier*⟩ `=`] [⟨*condition*⟩] ⟨*mnemonic*⟩ [⟨*arguments*⟩]

with optional items in square brackets. The optional assignment defines the identifier with the value of the instruction address. It is shortcut for an assignement

⟨*identifier*⟩ `=.`

preceding the assembly statement. The identifier can later be used as a register specification or GOTO target.

The ⟨*condition*⟩ can be `if1`, `ifC`, `ifN`, or `ifS`, and defaults to `if1` (unconditional).

The ⟨*mnemonic*⟩ is a reserved identifier, all uppercase letters, any of the names of the opcodes described in the previous section.

The ⟨*argument*⟩, if any, contains constants, register specifications, and punctuation according to the syntax described above for each opcode.

A register can be specified as an identifier, or as

`R[`⟨*expression*⟩`]`

with literal uppercase `R` and square brackets. The expression must resolve to the register address. Two higher bits are ignored.

Any constant can be specified as an ⟨*expression*⟩.

Most constants with the associated operators in the syntax definitions are optional and default to a neutral value.

Some register specifications are optional and default to the identifier `Z`.

3.5 Conventions

Programs should start with

```
Z = SUB .-.
```

to initialize a register with value zero.