USER MANUAL.
MultiClet P1 multicellular processor software

Contents

1 User Manual on Software Installation and Uninstallation **6**

1.1 Installation of software and documentation 6

1.2 Software and documentation uninstallation. 10

2 User Manual on Development Environment **13**

2.1 Framework description 13

2.2 Project creation 15

2.3 Project compilation 15

2.4 Program start on the model 15

2.5 Documentation 16

2.6 Confines 16

3 User Manual on Compiler C **17**

3.1 Description 17

3.2 Preparation for work 18

3.3 Transferring and assembling by the driver 18

3.4 Transfer of arguments to link editor 19

3.5 Some other useful keys of compilation driver 19

3.6 Diagnostic messages 19

3.7 Features of the earlier version of the compiler 20

3.8 Program assembling 20

4 User Manual on Assembler **24**

4.1 General information about multicellular processor 24

4.1.1 Program memory (PM) 25

4.1.2 Data memory (DM) 26

4.1.3 Registers 27

4.1.3.1 General-purpose registers 27

4.1.3.2 Index registers . . . 27
4.1.3.3 Control registers . . . 28
4.1.3.4 PSW register to control calculation process . . . 28
4.1.3.5 INTR interruption register . . . 30
4.1.3.6 MSKR register of interruption mask . . . 31
4.1.3.7 ER error register . . . 32
4.1.3.8 IRETADDR register of return address . . . 33
4.1.3.9 STVALR register of system timer period . . . 33
4.1.3.10 STCR register of system timer control . . . 34
4.1.3.11 IHOOKADDR register of initial interrupt handler . . . 35
4.1.3.12 Interrupt number register INTNUMR . . . 36
4.1.3.13 MODR register of modification mask of index registers . . . 36
4.1.4 Commutator . . . 37
4.1.5 Command fetching . . . 38
4.2 General information about assembler . . . 39
4.2.1 Assembler start and command line options . . . 39
4.3 Basic terms of the language . . . 40
4.3.1 Comments . . . 40
4.3.2 Constants . . . 40
4.3.2.1 Numeric constants . . . 40
4.3.2.2 Symbolic (literal) constants . . . 41
4.3.3 Sections . . . 42
4.3.3.1 Subsections . . . 43
4.3.4 Symbols . . . 44
4.3.4.1 System of symbol names . . . 44
4.3.4.2 Labels . . . 44
4.3.4.3 Symbols with absolute values . . . 45
4.3.4.4 Symbol attributes . . . 45
4.3.5 Expressions . . . 45
4.3.5.1 Empty expressions . . . 46
4.3.5.2 Integer expressions . . . 46
4.3.5.3 Arguments . . . 46
4.3.5.4 Operators . . . 46
4.4 Command system of assembler . . . 49
4.4.1 Convention . . . 49
4.4.2 Types of operations . . . 50
4.4.3 General principle of command building in an assembler . . . 51

4.4.3.1 General rules of formation of command arguments and their interpretation . . . 51
4.4.3.2 Rules of formation of command results . . . 52
4.4.4 Description of commands . . . 53
4.4.4.1 abs . . . 53
4.4.4.2 adc . . . 56
4.4.4.3 add . . . 60
4.4.4.4 and . . . 63
4.4.4.5 cfsl . . . 64
4.4.4.6 clf . . . 67
4.4.4.7 cslf . . . 69
4.4.4.8 div . . . 71
4.4.4.9 exa . . . 74
4.4.4.10 get . . . 75
4.4.4.11 insub . . . 78
4.4.4.12 ja . . . 81
4.4.4.13 jae / jnc . . . 82
4.4.4.14 jb / jc . . . 83
4.4.4.15 jbe . . . 84
4.4.4.16 je . . . 85
4.4.4.17 jg . . . 86
4.4.4.18 jge . . . 87
4.4.4.19 jl . . . 88
4.4.4.20 jle . . . 89
4.4.4.21 jmp . . . 90
4.4.4.22 jne . . . 91
4.4.4.23 jno . . . 92
4.4.4.24 jns . . . 93
4.4.4.25 jo . . . 94
4.4.4.26 js . . . 95
4.4.4.27 madd . . . 96
4.4.4.28 max . . . 97
4.4.4.29 min . . . 98
4.4.4.30 mul . . . 99
4.4.4.31 norm . . . 102
4.4.4.32 not . . . 103
4.4.4.33 or . . . 104

4.4.4.34 pack . . . 105
4.4.4.35 patch . . . 106
4.4.4.36 rd . . . 107
4.4.4.37 rol . . . 110
4.4.4.38 ror . . . 111
4.4.4.39 sar . . . 112
4.4.4.40 sbb . . . 113
4.4.4.41 set . . . 114
4.4.4.42 sll / sal . . . 116
4.4.4.43 slr . . . 117
4.4.4.44 sqrt . . . 118
4.4.4.45 sub . . . 119
4.4.4.46 wr . . . 120
4.4.4.47 xor . . . 121
4.5 Instruction system of assembler . . . 122
4.5.1 .alias . . . 122
4.5.2 .align . . . 122
4.5.3 .ascii . . . 123
4.5.4 .asciiz . . . 123
4.5.5 .bss . . . 123
4.5.6 .byte . . . 123
4.5.7 .comm . . . 123
4.5.8 .data . . . 124
4.5.9 .else . . . 124
4.5.10 .elseif . . . 124
4.5.11 .end . . . 124
4.5.12 .endif . . . 124
4.5.13 .equ . . . 125
4.5.14 .equiv . . . 125
4.5.15 .eqv . . . 125
4.5.16 .err . . . 125
4.5.17 .error . . . 125
4.5.18 .fill . . . 125
4.5.19 .float . . . 126
4.5.20 .global, .globl . . . 126
4.5.21 .if . . . 126
4.5.22 .include . . . 128

4.5.23 .lcomm . . . 128
4.5.24 .local . . . 128
4.5.25 .long . . . 129
4.5.26 .p2align . . . 129
4.5.27 .print . . . 130
4.5.28 .quad . . . 130
4.5.29 .rept . . . 130
4.5.30 .set . . . 130
4.5.31 .short . . . 130
4.5.32 .single . . . 131
4.5.33 .size . . . 131
4.5.34 .skip . . . 131
4.5.35 .space . . . 131
4.5.36 .string . . . 132
4.5.37 .text . . . 133
4.5.38 .type . . . 133
4.5.39 .warning . . . 133
4.5.40 .weak . . . 133
4.6 Programming system in assembler for multicellular processor . . . 135
4.7 Interruptions and their processing . . . 138

5 User Manual on Link Editor **141**
5.1 General information on a link editor . . . 141
5.2 Usage of a link editor . . . 142

6 User Manual on Loader **143**
6.1 General information of a loader . . . 143
6.2 Loader usage . . . 143

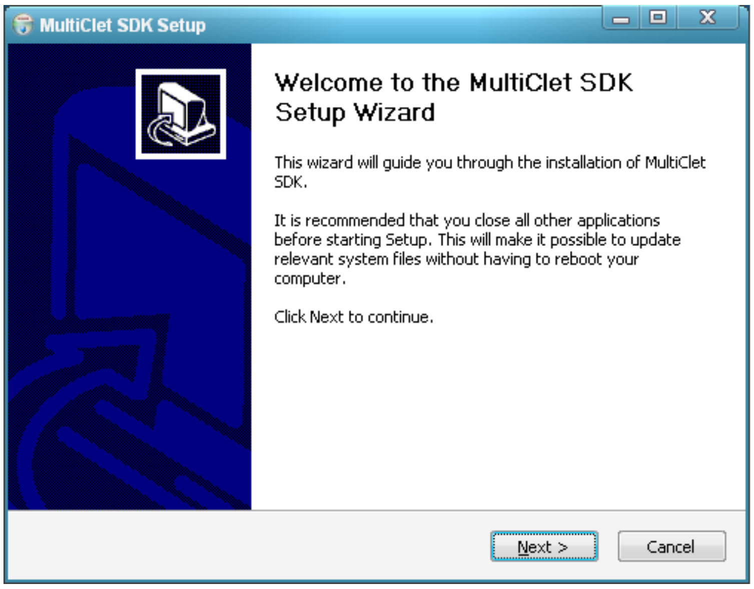
1. User Manual on Software Installation and Uninstallation

1.1 Installation of software and documentation

Installation of software and documentation (further on MultiCletSDK) is carried out by a specially developed installer.

Start the file MultiCletSDK.exe to begin installation.

After the installation started a hello message window appears with brief information about the product installed.



Welcome to the MultiClet SDK Setup Wizard

This wizard will guide you through the installation of MultiClet SDK.

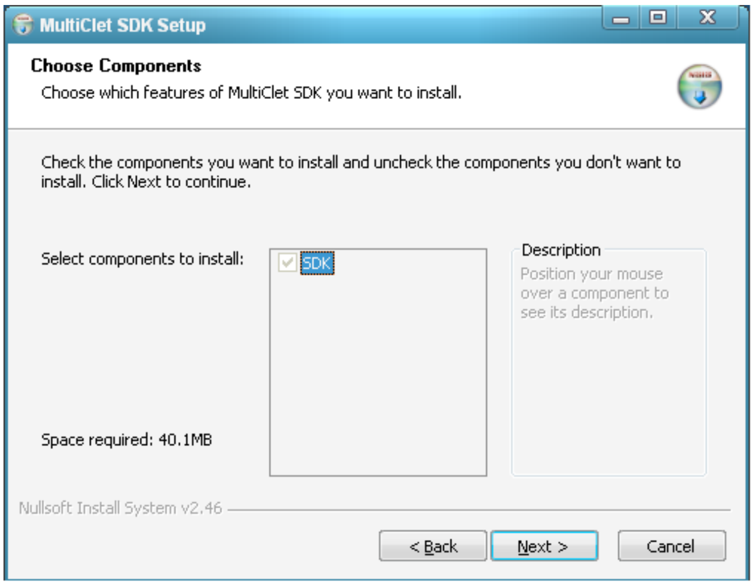
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Press “Next” button to continue installation.

On the second step appears a window with a list of available components for installation where a parameter “Shortcut on desktop” can be chosen. It is switched off on default.

In case of its activation a shortcut on desktop will appear showing the main folder with documentation, tests and MultiCletSDK projects.


MultiClet SDK Setup

Choose Components

Choose which features of MultiClet SDK you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install: ✓ SDK

Description

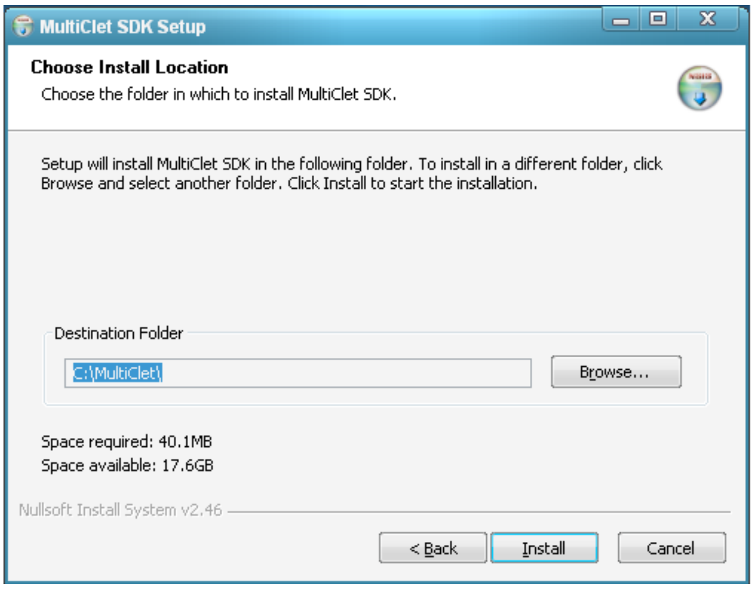
Position your mouse over a component to see its description.

Space required: 40.1MB

Nullsoft Install System v2.46

Then press “Next” button.

On the third step a catalogue can be chosen where MultiClet SDK files will be installed.


MultiClet SDK Setup

Choose Install Location

Choose the folder in which to install MultiClet SDK.

Setup will install MultiClet SDK in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\MultiClet\

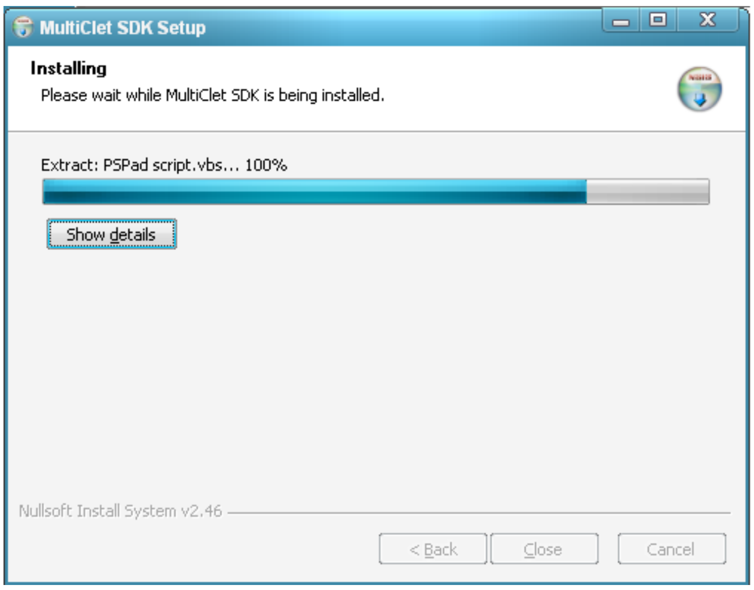
Space required: 40.1MB

Space available: 17.6GB

Nullsoft Install System v2.46

Then press “Install” button.

In the process of installation the progress of copying files will be displayed.

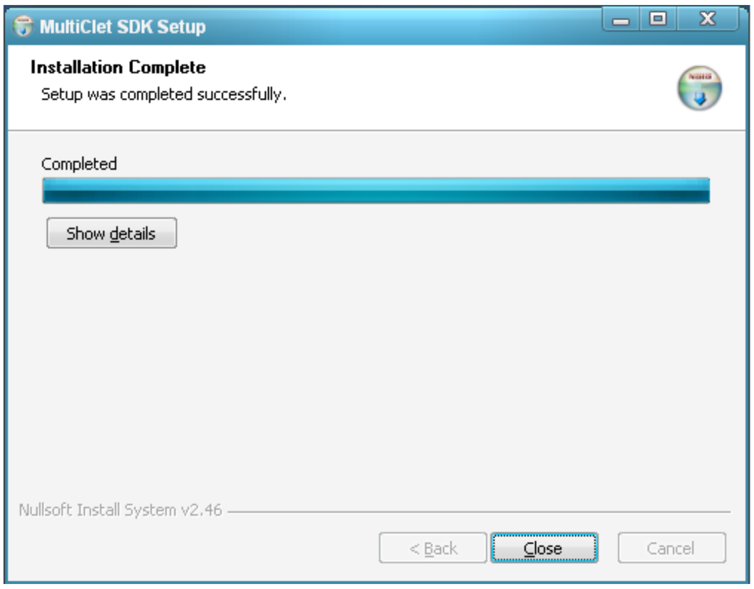

MultiClet SDK Setup

Installing

Please wait while MultiClet SDK is being installed.

Extract: PSPad script.vbs... 100%

On completion press “Close” button. The installation is supposed to be done at this point.

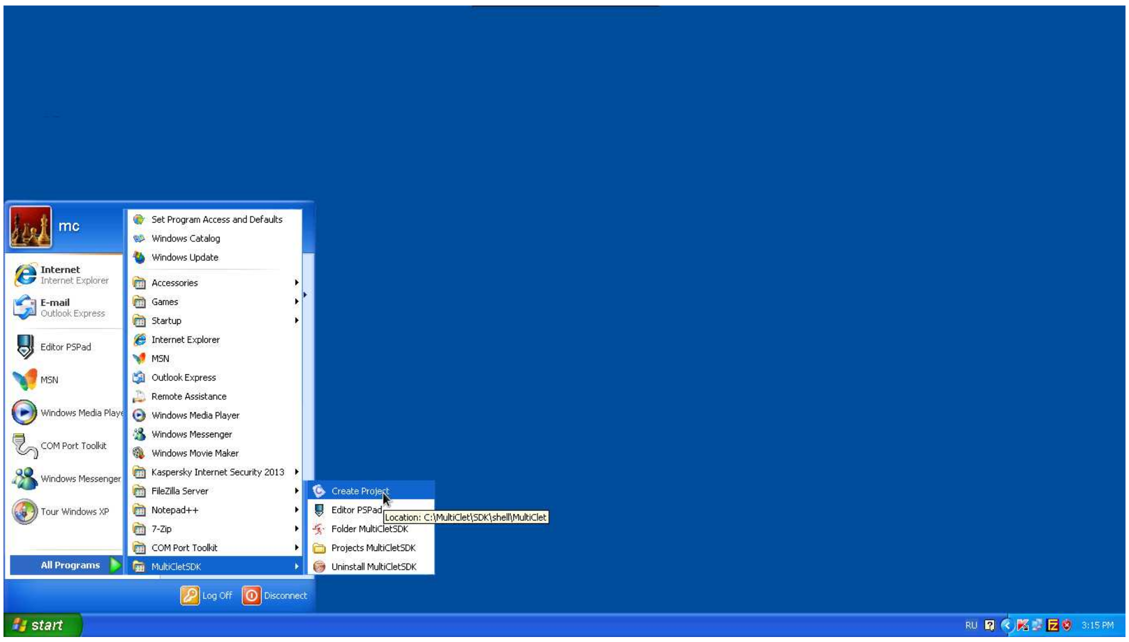

MultiClet SDK Setup

Installation Complete

Setup was completed successfully.

Completed

As a result of installation program termination a group of MultiCletSDK shortcuts will be added to “Start” menu.



To start work choose a necessary point in the menu:

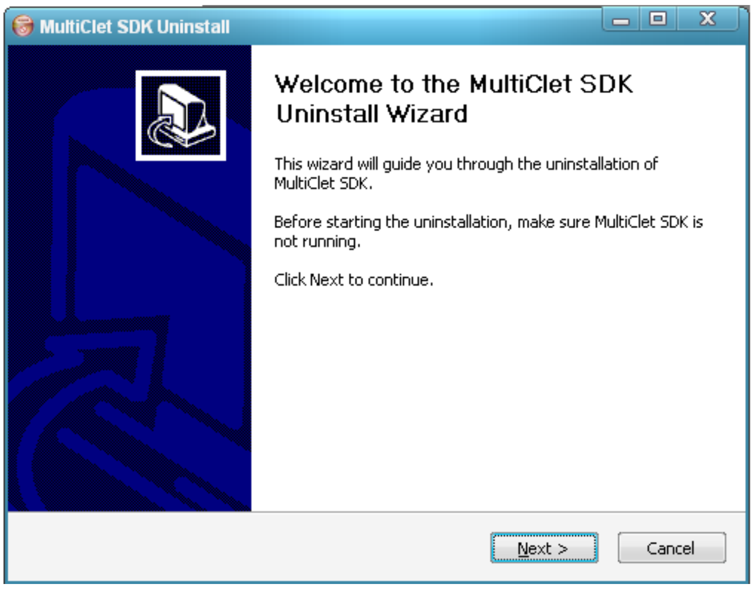
- MultiCletSDK documentation;
- MultiCletSDK folder;
- MultiCletSDK projects;
- Create a project;
- Remove SDK;
- PSPad editor (launching PSPad editor).

1.2 Software and documentation uninstallation.

Software and documentation uninstallation (further on MultiCletSDK) is carried out by a specially developed uninstaller.

To begin the uninstallation process start uninstall.exe file which is located in the MultiCletSDK installation directory or choose a point «MultiCletSDK\Remove MultiCletSDK» in the "Start" menu.

After starting the uninstallation process a hello message window appears with brief information about the product uninstalled.



Welcome to the MultiClet SDK Uninstall Wizard

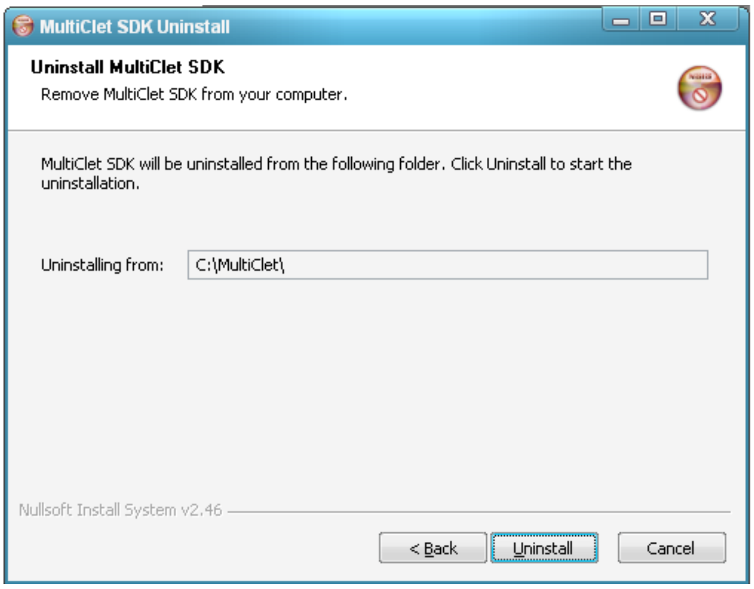
This wizard will guide you through the uninstallation of MultiClet SDK.

Before starting the uninstallation, make sure MultiClet SDK is not running.

Click Next to continue.

To continue installation press "Next" button.

On the second step a window of MultiCletSDK removal confirmation appears.


MultiClet SDK Uninstall

Uninstall MultiClet SDK

Remove MultiClet SDK from your computer.

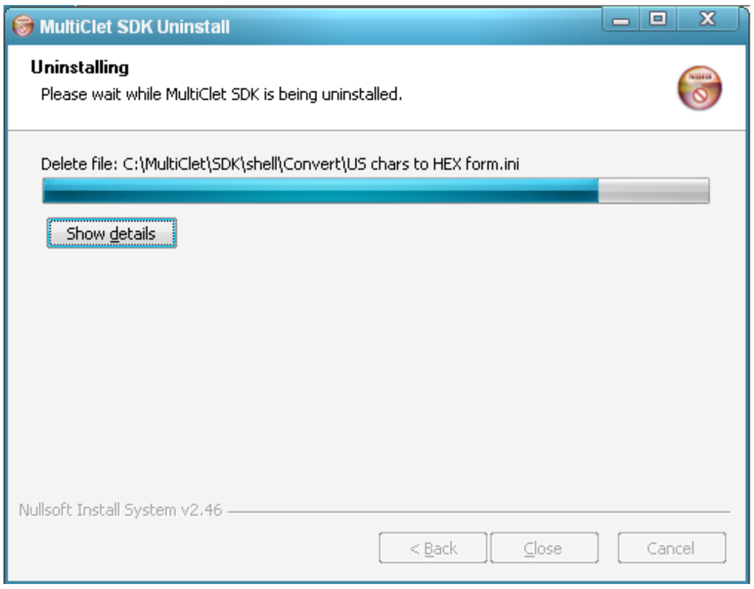
MultiClet SDK will be uninstalled from the following folder. Click Uninstall to start the uninstallation.

Uninstalling from: C:\MultiClet\

Nullsoft Install System v2.46

To confirm MultiClatSDK removal press “Remove” button.

In the process of removal the progress of files removal will be displayed.


MultiClet SDK Uninstall

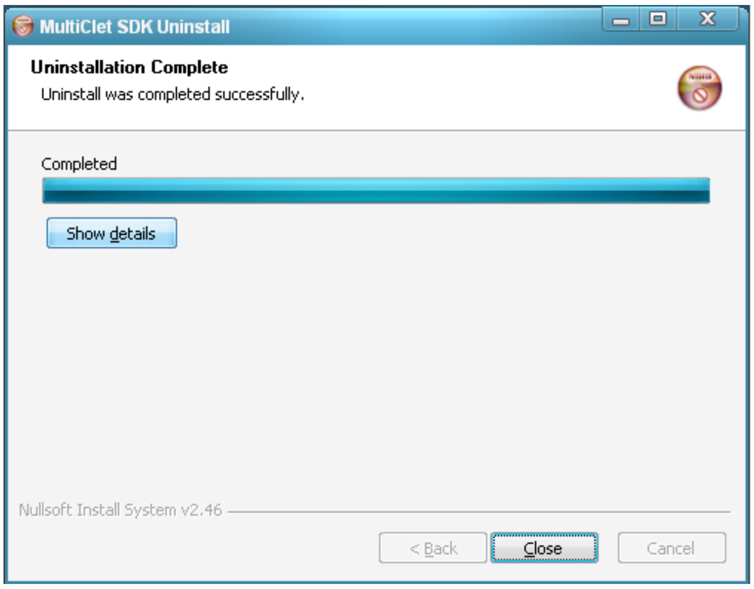
Uninstalling

Please wait while MultiClet SDK is being uninstalled.

Delete file: C:\MultiClet\SDK\shell\Convert\US chars to HEX form.ini

Nullsoft Install System v2.46

On completion press “Close” button. At this point the process of the product removal is supposed to be done.



Uninstallation Complete

Uninstall was completed successfully.

Completed

Nullsoft Install System v2.46

2. User Manual on Development Environment

2.1 Framework description

The framework is based on the basis of a freeware PSPad editor (http://www.pspad.com)with specific settings to work with MultiClet projects.

The editor has the following characteristics:

- work with projects;
- work with several documents simultaneously (MDI);
- saving of screen session i.e. at the next entry PSPad will automatically open all the files opened at the closing;
- FTP client - you can edit the files directly from web-server;
- script support: record, save and download scripts;
- search and replace in the files;
- comparison of texts with multicoloured highlighting of differences;
- patterns (HTML-tags, scripts, code patterns. . .);
- installation includes the patterns for HTML, PHP, Pascal, JScript, VBScript, MySQL, MS-Dos, Perl,...;
- syntax highlighting is applied automatically according to the file type;
- highlighting styles chosen by the user for exotic syntaxes;
- autocorrelation;

- intellectual in-built HTML-preview using IE and Mozilla;
- HEX editor of full value;
- activation of external programs, separately for every development environment;
- external compiler with interception of output, log window and parser logs for every environment create “IDE” effect;
- colour highlight of syntax for printing and preprinting preview;
- TiDy-library for formatting and checking HTML-code, conversion to CSS, XML, XHTML is integrated;
- in-built free version of CSS-editor TopStyle Lite;
- code export with highlight in RTF, HTML, TeX formats in file or clipboard;
- vertical highlight, bookmarks, marks, line numbering. . . ;
- reformatting and HTML-code compression, changing word register, tags, letters;
- sorting lines with an opportunity to sort in the column specified with a duplicate removal parameter;
- table of ASCII-symbols with reduction of HTML-mnemonics compatibility;
- code navigator for Pascal, INI, HTML, XML, PHP and many other in future;
- spell checking;
- in-built web-browser with APACHE support;
- matching parentheses highlight.

You can find a more detailed information about all the possibilities of the editor and the way it can be used from the in-built editor reference because it turns impossible to describe it all in the frames of this document.

Further only the properties inherent in MultiClet projects will be described.

In the process of installation the following file names extensions are secured for the editor:

- *.ppr - project file;
- * .asm - MultiClet assembler file.

*.c and *.h extensions are not registered not to overlay system settings on default because chances are high that the editor of these files has already been installed in the system.

2.2 Project creation

A project can be created in two ways:

- by launching the new project creation script;
- by means of the editor (see editor reference).

In the main menu and in Multiclet folder there is a link to project creation script: "Create a project". When launching the script a new project folder is created in the project folder which is located in "C:/Multiclet/Projects" folder on default. One pattern file is included in the project. Further file addition to the project is provided by means of the editor. A project can be created in the editor itself as well. The procedure of a new project creation is described in the in-built reference in detail.

2.3 Project compilation

The editor is adjusted for compilation of C projects and MultiClet assembler.

To start compilation open a relevant project.

To start compilation press **Ctrl+F9**, this will lead to compilation of all the C files and the assembler located in the directory and all the subdirectories of the project. .

The progress of compilation (log) is displayed at the bottom of the editor window.

On completion of compilation the result of compilation is displayed in the log.

If the compilation is effective, completion code is equal to 0 otherwise error code is displayed.

2.4 Program start on the model

After compilation the obtained file of memory image can be launched on the model.

To launch the model press **F9**.

In the process of model execution diagnostic messages or other prompts can appear.

2.5 Documentation

Documentation for SDK MultiClet is located in “C:/MultiClet/Docs” folder:

- User manual. ASM.pdf — User manual on Assembler compiler;
- User manual. LD.pdf — User manual on Link editor;
- User manual. Shell.pdf — User manual on Development environment.
- User manual. Shell.pdf — User manual on Installation and uninstallation of software and documentation MultiCletSDK.

2.6 Confines

Given version does not support program debugging.

After compilation completion all actions including program downloading into the processor or model startup are to be implemented manually.

3. User Manual on Compiler C

Compiler for MCp processor is developed on the basis of LCC (redirecting compiler for ANSI C - standard C89). LCC 4.2 used in the project provides compiler front: preprocessor, lexical analyzer, syntactic analyzer, some optimization of intermediate code (for example constant propagation), some libraries. "Multiclet" company has developed a generator of assembler code from intermediate representation LCC 4 and compilation process driver, which enables to obtain memory image for loading to debugging plate or to the processor model.

3.1 Description

At present the generator outputs a code corresponding to MCp0411100101 architecture and command system. Thus the operations of integer division and taking the remainder from the division are implemented through the activation of the functions written on the assembler. They are kept in crt0.s file in the system library.

The compiler contains three programs:

- driver (lcc в Linux / lcc.exe в Windows);
- preprocessor (cpp / cpp.exe);
- translator (rcc / rcc.exe).

The driver also uses the executable files:

- of assembler (as / as.exe);
- of link editor (ld / ld.exe);
- means of image preparation.

These programs are included in the assembler packet (as) and link editor (ld, convert), provided by MultiClet Corp. as well as in the integrated development environment.

3.2 Preparation for work

To enable the compiler driver to provide a full program compilation up to the acceptable memory image to load and execute, place the executable files in one directory:

lcc,

cpp,

rcc,

as,

ld.

In the systems with support of the links to files, the links to them can be put in this directory instead of the copies of the files.

3.3 Transferring and assembling by the driver

The current version of compiler requires indication of an option -lccdir = LCCDIR in the command line, where LCCDIR — is a directory with the executable files: lcc, cpp, rcc, as, ld.

Thus transfer of a set of source files C89 to the set of object files is implemented by the command:

- lcc -lccdir=LCCDIR -c f1.c ... fN.c

Assembling of the memory image prepared for execution can be implemented by the command:

- lcc -lccdir=LCCDIR f1.c ... fN.c asm1.s ... asmM.s o1.o ...
- oL.o CRTDIR/crt0.o -o image,

where CRTDIR — is a directory containing crt0.o. In this command it is demonstrated that the driver can assemble program image from:

- source texts on C89;
- source texts on assembler MCp;
- object files prepared beforehand.

If the option -o specifying the name of the image assembling is not used in the command line, this image will receive the name image.bin in the current directory.

3.4 Transfer of arguments to link editor

Sometimes it is useful to transfer some arguments of command line to link editor. For example, -M key, to which the editor responds with output of information on symbol location what is useful for debugging and memory analysis after program start in the debugging memory or in the processor program model.

The options can be transferred to the link editor with the help of key -Wm like it is done in the following example (in one of Linux-systems developers; \indicates transfer of the program to the next line)::

```
$ cd /tmp/flc/MC/bld
$ lcc \
       -lccdir=./ \
       -Wm-M \
       ../lcc/multiclet/symbolic/tst/factorial.c \
       ../CRTDIR/crt0.o \
       -o fact.img
```

3.5 Some other useful keys of compilation driver

-S driver will provide only transferring of source files to C to MCp processor assembler. Other construction stages of the executable will not be held.

3.6 Diagnostic messages

As the compiler front is built on the basis of LCC, the diagnostic messages connected with lexical, syntactic and semantic correctness of the source program are standard for this system. They are quite detailed and informative. For more detailed information see the instructions on the site http://www.cs.virginia.edu/ lcc-win32/.

Compiler code generator can also display diagnostic messages connected with the current particularities of its realization. The list of messages for the current version is as follows:

1. <coordinates in source texts>: expression is too complex; please, consider to decompose it — this message is displayed if the compiler did not manage to put the expression in the specified position in the source texts (file, line) within the restrictions imposed to the structure of the paragraph.

The problem will be solved in the next versions.

3.7 Features of the earlier version of the compiler

The current compiler version does not support operations of data block assignments. Thus the following constructions:

1. Initialization by constants of automatic arrays and structures.
2. Assignment of one structure by another one, i.e. constructions of the following type::

   ```
   struct S s1, s2;
   ...
   s1 = s2;
   ```

3. Activation of functions returning the structures or accepting them as arguments (just the structures, no restrictions are imposed on pointers). In the next compiler versions (to be issued after 22.07.2012) the problem will be solved.

3.8 Program assembling

Let us consider the example of the program written in C language, using the sorting algorithm by the bubble sort.

```
#define SIZE 256
int array[SIZE];

void main(void)
{
    int i;
    int j;
    int c = 0;
```

```
    for(i = 0; i < SIZE; i += 1)
    {
        c = 255 - i;
        array[i] = c;
    }

    for(j = SIZE-1; j; j -= 1)
    {
        for(i = 0; i < j; i += 1)
        {
            if(array[i] > array[i + 1])
            {
                int tmp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = tmp;
            }
        }
    }
}
```

Let us suppose that the given program is saved in the file under the name bubble.c. The process of program assembling or in other words acquisition of file, containing program memory images and data memory of the program executed, necessary for loading of debugging plate to ROM or for execution on the model, consists of the following stages:

1. processing of bubble.c file by C preprocessor. To do this execute the following command in the command line:

   ```
   mc-mcpp bubble.c bubble.i
   ```

   Preprocessor executes the removal from the source code of comments as well as performs processing of preprocessor directives beginning with the «#», such as #define, #include and others.

2. compilation of bubble.i file by C compiler. To do this execute the following command in the command line:

   ```
   mc-rcc -target=mcp/win32 bubble.i bubble.s
   ```

   C compiler executes transfer of the program in C language to the equivalent program in assembler language.

3. compilation of bubble.s file by the compiler of assembler code. To do this execute the following command in the command line:

   ```
   mc-as --arch=MCp0411100101 -obubble.o bubble.s
   ```

   Assembling of the source program code in assembler language to the object file containing machine code blocks and program data blocks with uncertain symbol addresses to the data and procedures in other object files as well as the list of its procedures and data.

   Option of assembler code compiler «-o» specify the name of output object file.

   Option of assembler code compiler «--arch» specify the multicellular processor architecture.

4. assembling of the file containing program memory images and images of data memory of the program executed from one or several object files. To do this execute the following command in the command line:

   ```
   mc-ld -M -oimage.bin crt0.o bubble.o
   ```

   Option of the link editor «-M» shows the necessity of output to the standard flow of output of information on the location of the given object files in the memory and on the values assigned to the symbols.

   option of the line editor «-o» specify the name of the output file containing program memory images and images of the data memory of the program executed.

   The object file crt0.o represents a set of starting executed procedures which perform the necessary initialization before activating the main function of the program. The given file is to be put first in the list of files transferred for the input of line editor. File crt0.o may not be used for the programs the source code of which is written only in assembler language.

The process of program assembling can be executed by mc-lcc assembling driver. To do this execute the following command in the command line:

```
mc-lcc -target=mcp/win32 -Wl-M -oimage.bin crt0.o bubble.s
```

In this case the driver will perform the sequence of actions described above by itself.

The obtained file of memory images of the program executed (image.bin) can be:

- loaded to ROM of the debugging plate. To do this execute the following command in the command line:

```
mc–ploader image.bin
```

- performed on the functional model. To do this execute, for example, the following command in the command line:

  ```
  mc–model \
      –dump–raw \
      –dump–addr \
      –dump–long \
      –dump–length 4 \
      –dump–from 0x00000088 \
      –dump–to 0x00000488 \
      image.bin
  ```

  Options «-dump-from» and «-dump-to» specify the initial and final addresses of data memory block the contents of which are to be sent to the printer. Beginning from the address $0x00000088$ in data memory, the line editor located the array, the elements of which as a result of executing the program will be sorted out by their growth.

  To find the description of all options of the functional model launch execute the command in the command line:

  ```
  mc–model ––help
  ```

4. User Manual on Assembler

4.1 General information about multicellular processor

A multicellular processor core is the first processor core with an unprecedented new multicellular architecture developed in Russia. A multicellular processor is intended for solving a wide range of control problems and digital processing of signals in applications requiring minimum energy consumption and high performance.

A multicellular processor consists of 4 cells (coherent processor blocks), united by the intellectual switch environment, with a separate memory of program and data.

The system of core commands of a multicellular processor has two formats (AA — word dimension (32 bits) and long word dimension (AV — 64 bits)).

Core commands work with the following data types:

- sign/unsigned byte, size 8 bits;
- sign/unsigned word, size 32 bits;
- unsigned double word, size 64 bits;
- sign tangible, size 32 bits;
- sign tangible loaded, size 64 bits;
- sign tangible complex, size 64 bits.

It is noteworthy that not every core command supports all the data types listed.

A multicellular core provides by hardware parallelism realization on the operator level in a "natural" way without solving the paralleling problem.

4.1.1 Program memory (PM)

Program memory represents independent blocks of statistic randomaccess-memory (PM0 - PM3) the number of which equals the number of processor blocks (cells). Every processor block has its own program memory accordingly. The memory blocks specified are not connected among each other and function independently.

For the user the program memory functions only in reading mode and is used only for storing program algorithm. For the constants an area singled out for this in data memory. Addressing is made to 64-bit long word.

Figure 4.1 shows a logical addressing of program memory for the processor consisting of 4 processor blocks.

PM0	**PM1**	**PM2**	**PM3**
$0x0000$	$0x0000$	$0x0000$	$0x0000$
$0x0001$	$0x0001$	$0x0001$	$0x0001$
...	...	...	...
$0x0fff$	$0x0fff$	$0x0fff$	$0x0fff$

Figure 4.1: Data memory structure

Processor program is viewed as a set of successively located paragraphs. Paragraph — is a group of sentences which is written successively, has one input and one output. In its turn sentence — is a group of operations connected by information. Every paragraph is located beginning from PM0. Paragraph commands are placed successively. Every next command is located in PM segment belonging to the next processor block. *AV* format command is fully placed in one segment. This is, for example, a following command succession formed by two paragraphs:

«av0,av1,aa2,aa3,aa4,av5» «aa6,aa7,av8,av9,aa10,av11,av12»

can be located in the way shown in fig. 4.2. The order of instruction read-out is demonstrated by the increase in colour background intensity.

Address	PM0		PM1		PM2		PM3	
$0x0000$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$0x0zzz + 0x00$	$av0$	$av0$	$av1$	$av1$	$aa2$	0	$aa3$	0
$0x0zzz + 0x01$	$aa4$	0	$av5$	$av5$	0	0	0	0
$0x0zzz + 0x02$	$aa6$	$aa10$	$aa7$	$av11$	$av8$	$av8$	$av9$	$av9$
$0x0zzz + 0x03$	0	0	$av11$	0	$av12$	$av12$	0	0
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$0x0fff$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

Figure 4.2: Paragraph location on PM

4.1.2 Data memory (DM)

Data memory represents independent blocks of static randomaccess-memory (DM0 - DM3) the number of which equals the number of processor blocks (cells), addressing is carried out on byte-serial principle. The particularity of data memory organization is that the cells with adjacent addresses are located in different data memory blocks. To cut the access time to the memory it is recommended to equalize the data to 8 byte.

Addressing of data memory for 4 segments is shown in fig. 4.3.

Address (basic)	DM0	DM1	DM2	DM3
	Address deposition			
$0x00000$	$0x00000 \dots 0x00007$	$0x00008 \dots 0x0000f$	$0x00010 \dots 0x00017$	$0x00018 \dots 0x0001f$
$0x00020$	$0x00000 \dots 0x00007$	$0x00008 \dots 0x0000f$	$0x00010 \dots 0x00017$	$0x00018 \dots 0x0001f$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$0x1ffe0$	$0x00000 \dots 0x00007$	$0x00008 \dots 0x0000f$	$0x00010 \dots 0x00017$	$0x00018 \dots 0x0001f$

Figure 4.3: Data memory structure

4.1.3 Registers

Multicellular processor contains the following registers:

- General Purpose Registers (GPR);
- Index Registers (IR);
- Control Register (CR).

All the registers listed above are 64-bit. Addressing to registers is done according to their number or name which are preceded by a sharp symbol "#". The total amount of registers is — 64. All commands of all processor devices when being decoded have a simultaneous reading access to all the registers. Recording to the registers is also executed simultaneously on completion of the current paragraph. Register numbering is continuous and starts from zero.

4.1.3.1 General-purpose registers

Used as a scratchpad memory. To address to any register of this type use numbers from 0 to 7. Registers of this type have no names. Interpretation of register value depends on the command type.

4.1.3.2 Index registers

Are used for indirect addressing. Logical structure of the index register is shown in fig. 4.4).

Bit numbers	63 . . . 48	47 . . . 32	31 . . . 0
	Index (*Index*)	Mask (*Mask*)	Base (*Base*)

Figure 4.4: Index register structure

To address to any of the registers of this type numbers from 32 to 39 are used. The registers of this type have no names. In general case (see exclusions in the description of a specific command in the chapter "Assembler command system") when using the register of the given type as an operation argument the value of this argument is formed according to the following algorithm:

1. calculation of the effective address according to the following formula:

$$Address = Index + Base$$

2. addressing to data memory at the effective address *Address* for reading/writing the argument value according to the type of the command used.

Modification of index register value is done by hardware on completion of the paragraph in case the appropriate MODR register bit of mask of switching index registers is set (see chapter "Control register") according to the following formula:

$$Index = ((Index \mid \sim Mask) + 1) \ \& \ Mask,$$

where | — is an operation of bit-by-bit "OR", & — is an operation of bit-by-bit "AND", ∼ — is an operation of bit-by-bit conversion.

Two formulae given above use integer 32-bit arithmetic. The values of 16 high-order bits (from 16 to 31) Index field (*Index*) and Mask field (*Mask*) are filled with zeroes.

4.1.3.3 Control registers

Processor contains the following control registers:

Register number	**Register name**	**Access rights**	**Description**
48	PSW	R/W	register to control calculation process
49	INTR	R/W	Interrupt register
50	MSKR	R/W	Maskable interrupt register
51	ER	R	Exception register
52	IRETADDR	R	Return address register
53	STVALR	R/W	System timer period register
54	STCR	R/W	System timer control register
55	IHOOKADDR	R/W	Primary interrupt handler register
56	INTNUMR	R	Interrupt number register
57	MODR	R/W	Register of index register's modification mask

4.1.3.4 PSW register to control calculation process

Calculation control register is used to control calculation process.

Fig. 4.5 shows the structure of calculation control register.

PSW	Control register PSW																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	R = 0																									RW	RW	RW	R	RW	R	RW
Initial state	0																									0	0	0	0	0	0	0
Description	Reserved																									RW	SWR	STOP	RESERVED	SUSPEND	RESERVED	ONIRQS

Figure 4.5: Structure of calculation control register

Function of stages of register:

For all the stages of register:

0 — absence of any feature or event

1 — presence of any feature or event

0	Handling of masked interruptions. The feature does not block the reception of interruptions by INTR register, it blocks only their further handling. Its is always installed by software, removed by software and hardware (when switching to the program of interruption handler).
1	Reserved
2	Switch to sleep mode. On completion of the paragraph where the given bit is set the core switches to sleep mode. It stays in this mode till the interruption. When interrupted this feature is switched off by hardware and control is passed on to interruption service program. On completion of interruption service, unless the programmer specifies any actions, the program suspended continues.
3	Reserved
4	Core stop. When set to 1 on completion of the current paragraph the core stops instruction fetching. Work renewal is possible only from without by hardware reset or sending to the processor «wake_up» input the voltage of logical level «1».
5	Software reset. When set to 1, on completion of the paragraph the processor undergoes full initialization, the core automatically begins work from the initial state, the registers are reset to the initial state. The content of main memory is not stored.

6	Order of reading/writing command execution: 0 — writing commands are executed only after completion all reading commands of this paragraph 1 — Order control is not executed
7 — 31	Reserved

4.1.3.5 INTR interruption register

Interruption register is used for signaling the occurrence of interruptions as well as initiation of program interruptions. Interruption register structure is shown in fig. 4.6

Interrupt register INTR

INTR																
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Operations	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial state	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Description	USB0_IRQ	ETHERNET0_IRQ	GPIOD_IRQ	GPIOC_IRQ	GPIOB_IRQ	GPIOA_IRQ	RTC_IRQ	PWM0_IRQ	GPTIM6_IRQ	GPTIM5_IRQ	GPTIM4_IRQ	GPTIM3_IRQ	GPTIM2_IRQ	GPTIM1_IRQ	GPTIM0_IRQ	I2S0_IRQ

INTR																
Bit number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial state	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Description	SPI2_IRQ	SPI1_IRQ	SPI0_IRQ	I2C1_IRQ	I2C0_IRQ	UART3_IRQ	UART2_IRQ	UART1_IRQ	UART0_IRQ	SWI	SWT	MPRGE	PPGE	PERE	ENMI	INMI

Figure 4.6: Interruption register structure

Purpose of stages of register:

For all stages of register
0 — absence of any feature or event
1 — presence of any feature or event

0	Internal nonmaskable interruption (INMI)
1	External nonmaskable interruption (ENMI)
2	Nonmaskable exception in hardware compone(PERE)
3	Nonmaskable program exception (PPGE)
4	Maskable program exception (MPRGE)
5	Interruption from system timer (SWT)
6	Software interruption (SWI)
7	Maskable interruption from UART0
8	Maskable interruption from UART1
9	Maskable interruption from UART2
10	Maskable interruption from UART3
11	Maskable interruption from I2C0

12	Maskable interruption from I2C1
13	Maskable interruption from SPI0
14	Maskable interruption from SPI1
15	Maskable interruption from SPI2
16	Maskable interruption from I2S0
17	Maskable interruption from GPTIM0
18	Maskable interruption from GPTIM1
19	Maskable interruption from GPTIM2
20	Maskable interruption from GPTIM3
21	Maskable interruption from GPTIM4
22	Maskable interruption from GPTIM5
23	Maskable interruption from GPTIM6
24	Maskable interruption from PWM0
25	Maskable interruption from RTC
26	Maskable interruption from GPIOA
27	Maskable interruption from GPIOB
28	Maskable interruption from GPIOC
29	Maskable interruption from GPIOD
30	Maskable interruption from ETHERNET0
31	Maskable interruption from USB0

4.1.3.6 MSKR register of interruption mask

Interruption mask register is applied to mask interruptions.

The structure of interruption mask register is shown in fig. 4.7

MSKR	**Mask interruption MSKR**																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$				*RW*																											
Initial state	0				0																											
Description	Reserved				Masked interruptions																											

Figure 4.7: Structure of interruption mask register

Purpose of register stages:

0 — 27	Mask for 4 — 31 bits of INTR register 0 — prohibition interrupt request 1 — permit for interrupt handling
28 — 31	Reserved

4.1.3.7 ER error register

Structure of error register fig. 4.8

ER	Error register ER																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$																RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	RC	$R = 0$	$R = 0$	RC	
Initial state	0																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Description	Reserved																II3	II1	II2	II0	AI3	AI2	AI1	AI0	DZ3	DZ2	DZ1	DZ0	RESERVED	RESERVED	AMBATR_ER	

Figure 4.8: Structure of error register

Purpose of register stages:

For all stages of register:
0 — absence of any feature or event
1 — presence of any feature or event

0 — 1	Bus errors
2	Reserved
3	Reserved
4	Error «divide-by-zero» in 0 core
5	Error «divide-by-zero» in 1 core
6	Error «divide-by-zero» in 2 core
7	Error «divide-by-zero» in 3 core
8	Error «wrong address» in 0 core, formulated wrong address
9	Error «wrong address» in 1 core
10	Error «wrong address» in 2 core
11	Error «wrong address» in 3 core
12	Error «wrong instruction» in 0 core, non-existent instruction is read from program memory
13	Error «wrong instruction» in 1 core

14	Error «wrong instruction» in 2 core
15	Error «wrong instruction» in 3 core
16 — 31	Reserved

4.1.3.8 IRETADDR register of return address

Address return register is used to save the return address of program memory which is formed only when interrupted. It is noteworthy that the actual drift to the initial interrupt handler (see 4.1.3.11) is executed only on completion of all operations of the current paragraph and when the address of the next paragraph is known which is formed by one of the operations of setting the address of the next paragraph. The return address register keeps the address of the next paragraph. If the address of transition to the next paragraph is not known, the processor will switch to wait state of «start» signal (external reset wait).

Structure of the initial interrupt handler register is shown in fig. 4.9

IRETADDR	Return address register IRETADDR																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$																				R											
Initial state	0																				0											
Description	Reserved																				IRET_ADDR											

Figure 4.9: Structure of return address register

Purpose of stages of register:

0 — 11	Return address formed only when interrupted
12 — 31	Reserved

4.1.3.9 STVALR register of system timer period

System timer period register is used to save the values of system timer period and is available only for reading. Structure of system timer period register is shown in fig. 4.10

STVALR	Counter period register STVALR																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	*R* = 0																*RW*															
Initial state	0																0															
Description	Reserved																CNTVAL															

Figure 4.10: Structure of system timer period register

Purpose of stages of register:

0 — 15	Values of counter period in the periods of system frequency after predivisor of system timer. (see 4.1.3.10)
16 — 31	Reserved

4.1.3.10 STCR register of system timer control

Register of system timer control is used to control a system timer. A system timer is applied to form the specified periodic or single time slots. The timer represents a decrement counter with the divisor of clock signal at the input. The initial value of the counter is recorded to STVALR register, control is executed through STCR register. When the specified time slots finishes the inquiry for interrupt handling is formed.

Structure of control register of system timer is shown in fig. 4.11

STCR	STCR register of system timer control																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	*R* = 0																*RW*								*R* = 0					*RW*	*RW*	*RW*
Initial state	0																0								0					0	0	0
Description	Reserved																PREDIV								Reserved					CNTCMP	ENCCNT	EN

Figure 4.11: Structure of control register of system timer.

Purpose of stages of register:

0	Permission of counter work
	0 — prohibited
	1 — permitted

1	Permission of timer cycling work (if the cycling work is prohibited the timer will switch off after one period, bit 0 of the given register will be set to «0»): 0 — prohibited 1 — permitted
2	Attribute of period count completion set in STVAL (in accordance with given attribute interruption (from timer) processing request is being implemeted): 0 — prohibited 1 — permitted
3 — 7	Reserved
8 — 15	Value of counter predivisor. System frequency is divided by the value specified in the given bits. The frequency divided is the clock frequency for the timer counter
16 — 31	Reserved

4.1.3.11 IHOOKADDR register of initial interrupt handler

Register of initial interrupt handler is used to keep the address of the program memory of the initial interrupt handler. Structure of register of initial interrupt handler is shown in fig. 4.12

IHOOKADDR	Address register of interrupt handler IHOOKADDR																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$																				RW											
Initial state	0																				0											
Description	Reserved																				IHOOK_ADDR											

Figure 4.12: Structure of register of initial interrupt handler

Register bits' assignation:

0 — 11	Initial interrupt handler's value
12 — 31	Reserved

4.1.3.12 Interrupt number register INTNUMR

Interrupt number register is designed for saving top priority unmasked interruption number at present moment and is read-only.

Structure of interrupt number register is shown on fig. 4.13

INTNUMR	Register of interrupt number INTNUMR																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$																										R					
Initial state	0																										0					
Description	Reserved																										INT_NUM					

Figure 4.13: Structure of interrupt number register

Purpose of stages of register:

0 — 5	Value of the address of the initial interrupt handler
6 — 31	Reserved

4.1.3.13 MODR register of modification mask of index registers

Register of modification mask of index registers is used to indicate the necessity of recounting the values of index register the number of which is defined according to the formula $32 + i$,where i — the bit number of MODR register on the completion of operations of the paragraph. The rule of changing the value of index register can be found in the chapter "Index registers, 4.1.3.2".

Structure of register of modification mask of index registers is shown in fig. 4.14

MODR	Modofocation register of index registers MODR																															
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operations	$R = 0$																								RW	RW	RW	RW	RW	RW	RW	RW
Initial state	0																								0	0	0	0	0	0	0	0
Description	Reserved																															

Figure 4.14: Structure of register of modification mask of index registers

Purpose of stages of register:

0	Permission of modification of index register number 32: 0 — prohibited 1 — permitted
1	Permission of modification of index register number 33: 0 — prohibited 1 — permitted
2	Permission of modification of index register number 34: 0 — prohibited 1 — permitted
3	Permission of modification of index register number 35: 0 — prohibited 1 — permitted
4	Permission of modification of index register number36: 0 — prohibited 1 — permitted
5	Permission of modification of index register number 37: 0 — prohibited 1 — permitted
6	Permission of modification of index register number 38: 0 — prohibited 1 — permitted
7	Permission of modification of index register number 39: 0 — prohibited 1 — permitted
8 — 31	reserved

4.1.4 Commutator

Commutator is used to exchange command results between the commands inside one paragraph.

Paragraph — is a group of sentences which is written successively, has one input and one output.

Sentence — is a group of operations connected by information.

All the processor commands inside one paragraph are numerated conditionally starting from zero.

The link to the result of textually preceding command is written as @N. It is calculated according to the following formula

$$N = N_{req} - N_{res},$$

where N_{req} — нthe number of command inquiring for the result of textually preceding command, N_{res} — the number of textually preceding command the result of which is inquired.

Maximum number of results of textually preceding commands which can be saved in the commutator is — 63.

Link to the result of the textually preceding command which does NOT return the result definitionally will lead to the error at the compiling stage.

4.1.5 Command fetching

The initial program memory address from which processor blocks start command fetching is 0.

The initial program memory address from which processor blocks start command fetching is 0. The necessary conditions of fetching and subsequent decoding of the next command by all processor blocks are termination of decoding of the preceding command by all processor blocks, reading and location of the next command in command register by all processor blocks (every processor block has its own command register).

Fetching and decoding of commands continues until the last command of the paragraph is fetched. The address of the new paragraph can come either at any moment of command fetching of the current paragraph or on completion of command fetching. It comes to all processor blocks simultaneously. If at the moment of fetching of the last command of the paragraph the address of the next paragraph is not calculated, fetching suspends until the address is received. If on completion of all commands of the paragraph the address of the next paragraph is not formed, processor switches to the wait state of «start» signal (external reset wait).

4.2 General information about assembler

The assembler given id used to code programs for multicellular processor. The assembler has the following characteristics:

- use of separate command memory for every processor block (cell) and of the general data memory;
- 32-bit addressing of command memory;
- 32-bit addressing of data memory;

Assembler can be started on IBM PC compatible computers having the following characteristics:

- operating system: Microsoft Windows 2000, XP, Vista, 7.

Minimum requirements to hardware:

- Processor not less than 500 MHz;
- main memory not less than 512 MB;
- video card should support minimum resolution 1024x768 with 256 colors and hardware acceleration of functions DirectX of 5.0 version or higher.

4.2.1 Assembler start and command line options

The assembler given is started from the command line by as command ***as***the argument of which is the file with the source code. The following options are supported:

-I, --include-path=DIR — to add the directory DIR to the list of directories used for searching the file connected by assembler «.include»

-o, --output=objfile — to place the output to the object file objfile

-V — to print the number of assembler version

-h, --help — to show this message and exit

4.3 Basic terms of the language

4.3.1 Comments

Assembler supports the following types of comments:

- single-line comment

  the given type of comment starts from c ';' or '//' and finishes by the end of the line

- multiline comment

  the given type of comment starts from '/*' and finishes by'*/', i.e. everything that is placed between these limiters is ignored; the enclosed comments are not allowed.

4.3.2 Constants

Assembler supports numeric and symbolic (literal) constants.

4.3.2.1 Numeric constants

The following variants of numeric constants representation are possible in the assembler:

1. in the form of hexadecimal number

   This number starts from prefix '0x' or '0X', followed by one or more hexadecimal numbers '0123456789abcdefABCDEF'. To change the sign use a prefix operator '-'.

2. In the form of octal number.

   This number starts from the zero number followed by one or more octal numbers '01234567'. To change the sign use a prefix operator '-'.

3. In the form of binary number

   This number starts from the prefix '0b' or '0B', followed by one or more binary numbers '01'. To change the sign use a prefix operator '-'.

4. In the form of decimal number

   This number starts from nonzero number followed by one or more decimal numbers '0123456789'. To change the sign use a prefix operator '-'.

5. In the form of real decimal number with floating point written in the following format:

(a) starting with the prefix '0f' or'0F',

(b) further optionally comes the number sign '+' or '-',

(c) further optionally comes the integer part of the number consisting of zero or more decimal numbers,

(d) further optionally comes the fractional part of the number starting from the dot sign '.' and consisting of zero or more decimal numbers.

(e) further optionally comes the exponential part of the number consisting of:

- 'e' or 'E'
- sign '+' or '-' sign of exponential part (optionally)
- one or more decimal numbers.

At least one of the integer or fractional parts should be specified.

4.3.2.2 Symbolic (literal) constants

The following variants of representation of symbolic (literal) constants are possible in an assembler:

1. In the form of a string (sequence of literals)

   The string constants (sequence of literals) are written in rabbit ears. They may contain any possible symbols (literals), as well as the following escape-sequences:

   - \b — cancel character (backspace); ASCII code in octal numeration system 010.
   - \f — new page (FormFeed); ASCII code in octal numeration system 014.
   - \n — line feed (newline); ASCII code in octal numeration system 012.
   - \r — carriage return (carriage-Return); ASCII code in octal numeration system 015.
   - \t — — horizontal tab (horizontal Tab); ASCII code in octal numeration system 011.
   - \ oct-digit oct-digit oct-digit — symbol code in octal numeration system. Symbol code consists of 3 octal digits. If a given number outnumbers the maximum possible octal value, eight least significant digits will be used.

- \x hex-digit hex-digit — symbol code in a sexadecimal numeration system. Symbol code consists of 2 sexadecimal digits. Literal register 'x' has no value. If none of sexadecimal digits is given, the value "zero" is used.
- \\ — corresponds to a literal '\'.
- \" — corresponds to a literal '"'.
- \anything-else — corresponds to any symbol, except for the abovementioned.

2. 2. In the form of a single symbol (literal)

   A single symbol (literal) may be represented in the form of single quotes «´», which are followed by a required symbol (literal) or escape-sequenceṪherefore for representation of a symbol (literal) «\» we shall write «´\\», where the first symbol (liretal) «\» escapes the second one «\». The symbol of line feed, which follows «´», is interpreted as a symbol (literal) «\n» and is not the end of an expression. The value of a symbol (literal) constant in an integer expression is a computer code with the size of 1 bite, of a symbol (literal). "as" uses ASCII symbol (literal) encoding: ´A has an integer value 65, ´B has an integer value 66, and so on.

4.3.3 Sections

Without giving specifics, the section is a continuous span of addresses, the data of which are interpreted in the same way for some particular actions.

As a result of compiling the source code of a part of a program, the assembler creates an object file, supposing that the given part of a program is placed from a zero address. Assembly of an executable program itself is performed by a builder using one or several object files, created by an assembler. As a result of the above, each object file is assigned with a final address in such a way that no object file is overlapped by another.

During program assembly the blocks of bites, as a whole, are moved to those addresses, which they will have during program execution; their length and order of bites do not change. Such a block of bites is called a section, and a procedure of address assignment to these sections is called — relocation. Aside from address assignment to sections, execution time and relocation, in assembling phase the values of an object file symbols are brought into accordance so that all references to those symbols would have correct addresses of execution time. It should be noted, that because of the special organization of program memory of a multicellular processor, where the executed program instructions, length of section .text, in which the assembler located executed program instructions, may increase because of the adjustment of paragraphs (for further details ref. to section “Program Memory (PM)”).

An object file, created by an assembler, includes at least 3 sections, each of which may be empty:

1. секция .text. In this section the executed program instructions are located, which in the process of assembly will be relocated in program memory.
2. секция .data. In this section the starting data of a program are located, which in the process of assembly will be relocated in the preset area of data memory.
3. секция .bss. This section contains bites with zero values prior to the beginning of program execution. It is used for storing uninitialized variables or a common block of data memory.

In the process of creation of an object file the section .text if relocated from address 0 of program memory; section .data is relocated from address 0 of program memory, which is followed by relocation of section .bss.

Sections .text и .data are located in an object file irrespective of the fact whether they contain any commands or not.

In order to inform the builder of which data are changed during storage relocation (section relocation) and in accordance with which rules they are changed, the assembler stores all necessary information into the object file into separate sections (usually for a section .text it stores into a section .rel.text, for a section .data it stores into a section .rel.data).

Besides sections .text, .data, .bss it is possible to use an absolute section. By assembly of a program addresses in the absolute section do not change.

Aside from the abovementioned sections there exists an undefined section as well. Any symbol which has a reference to, and which was not defined, on the stage of assembly is assigned to the undefined section. A common (commonly used) symbol , which addresses named common block, on the stage of assembly is assigned to the undefined section as well. Value of binding attribute of any undefined symbol equals «GLOBAL» by default.

4.3.3.1 Subsections

Assembled bites are relocated in 2 sections: .text and .data. For adjustment of different data groups inside of sections.text and .data., the subsection are used in a generated object file. The numbered subsections, beginning from zero, may be located inside of each section. The objects assembled into one and the same subsection, are relocated in an object file together with the objects of the same subsection.

Subsections are relocated in an object file in ascending order of their numbers. Information on subsections is not saved in an object file.

In order to specify in which subsection to assemble the below mentioned instructions, we shall specify the subsection number in the command .text/.data. If the subsections in a source code of a program are not used, all instructions are assembled to a subsection with a number 0.

Each section has “a counter of a current place”, which increases by 1 by assembling of each new bite into a section. Since the subsections are just for convenience and are used only inside of an assembler, there are no “ counters of a current place” in subsections.

“Counter of a current place” of a section, in which instructions are being assembled at the moment, is called active counter of a place.

Section bss is issued as a place for storing global variables. For this section, by using command .lcomm, it is possible to substract address space without specifying which data will be loaded into it prior to program execution. At the time of beginning of program execution the content of section .bss is filled with zeros.

4.3.4 Symbols

Symbols (identifiers, variables and so on) are used in an assembler for naming various contents.

4.3.4.1 System of symbol names

System of names in an assembler is built up according to the following principle — names may consist of capital and block Latin letters, digits, underlining symbol («_») and stop symbol («.») only, and the name may not begin with a digit. Capitalization of letters in a name is taken into account.

4.3.4.2 Labels

A label is defined as a symbol which is followed by a two-spot «:». This symbol represents current value of an active counter of place depending on the currentsection(.text, .data, .bss). Redefinition of labels in an assembler is not admissible.

4.3.4.3 Symbols with absolute values

These symbols may be defined with the help of the following commands of an assembler: .set. .eqv, .equ, .equiv. The values of these symbols are never changed by a builder. In general, the symbol, defined with the help of the abovementioned assembler commands may have no absolute value.

4.3.4.4 Symbol attributes

Aside from a name, each symbol has the attributes “Value”, “Type/Collecting”, “Size” and an attribute of appurtenance of a symbol to any section. By using a symbol without its defining, all its attributes have zero values, and a symbol itself is assigned to an undefined section.

Value of a symbol is 32-positioning. For symbols, which address location in sections .text, .data, .bss and absolute, the value is dislocation of addresses with relation to the starting position of a section prior to a label. In the process of program assembly the values of such symbols for section .text, .data, .bss change, because the starting positions of these sections change as well. Values of absolute symbols in the process of assembly are not changed.

Attribute of a symbol “Type/Collecting” defines the type and symbol viewability for a builder, as well as behavior of a builder in the process of changing the symbol value by section dislocation. For setting the a value like this attribute a command of an assembler .type is used, and for collecting — the commands .local, .global, .weak are used.

An attribute of a symbol “Size” on assembly stage is by default always equal to zero. Value of this attribute may be changed with the help of an assembler command .size.

The symbol appurtenance attribute to some section is set by an assembler automatically, depending on the current assembly section. In other words, this attribute shows in which section the symbol is defined.

4.3.5 Expressions

Expression defines an address and a numeric value. The result of a n expression shall be an absolute digit or a dislocation in a certain section.

4.3.5.1 Empty expressions

An empty expression has no value: it is just a space. In case if there is no expression in such a place of a source code where it is needed, an assembler uses zero value.

4.3.5.2 Integer expressions

Integer expressions — is one or more arguments separated by operators.

4.3.5.3 Arguments

Symbols (identifiers), digits or subexpressions may act as expression arguments.

A value of a symbol in any section secName is a dislocation in relation to the beginning of this section. Sections .text, .data, .bss and an absolute (*ABS*) and an undefined (*UND*) may act as a section secName. A value is a 32-positioning integer value with a digit in two's complement.

Digits usually are integer. If an expression consists of one argument, which is a digit with a floating point, then the result of calculation of an expression will be this very digit without any alternation. If an expression consists of several arguments, separated by operators, or consists of a prefix operator, which is followed by only one argument, then by calculation of an expression the argument value, which is a digit with a floating point, will be replaced by a zero value and a corresponding warning will be typed-out.

Subexpressions are the same expressions written in brackets "()", or a prefix operator, which is followed by an argument.

4.3.5.4 Operators

Operators are arithmetic functions and they are divided into prefix and infix.

Prefix operators are one-argument, An argument shall be absolute. The following prefix operators are available:

«−» Negation. Scale-of-two additional negation.

«∼» Addition. Bit-by-bit negation.

«!» Logic NOT. 1 returns if an argument is not zero, otherwise 0 returns.

Infix operators are two-argument. These operators have a priority, which defines their execution order. Operations of equal priority are executed from left to right. Arguments of all infix operators, except operators «+» and «−» shall be absolute. The following infix operators are available according to decrease of priority:

1. «∗» Multiplying.

   «/» integer division.

   «%» Remaining (taking of a remaining from an integer division).

   «≪» Shift to the left.

   «≫» Shift to the right.

2. «|» bit-by-bit OR.

   «&» bit-by-bit AND.

   «^» bit-by-bit exclusive OR

   «!» bit-by-bit OR NOT

3. «+» Addition. If one of the arguments is absolute, the result is assigned to the section of another argument. Addition of two arguments, defined in relation to different sections is not admissible.

   «−» Subtraction. If the right (second) argument is absolute, the result is assigned to the section of the left (first) argument. If two arguments are defined by one and the same section, the result is absolute. Subtraction of two arguments, defined by different sections is not admissible.

   «==» Comparison for equality

   «! =» or «<>» Comparison for inequality.

   «<» Comparison for less than.

   «>» Comparison for more than.

   «>=» Comparison for more than or equal.

   «<=» Comparison for less than or equal.

As a result of performance of any operation of comparison, -1 returns in case if the result is correct, and 0 returns if the result is not correct. Comparison operations interpret arguments as character.

4. «&&» Logic AND.

   «||» Logic OR.

These logic operations may be used for uniting the results of subexpressions. As a result of performance of any logic operation, -1 returns in case if the result is correct, and 0 returns if the result is not correct.

4.4 Command system of assembler

A command of a multicellular processor, as any another one, in general, is an encoded (according to a number of rules) instruction for a processor for execution of some operation with a number of operands.

In a multicellular processor there are short instructions, 32 bites, and long instructions, 64 bites. The instruction of a multicellular processor has (encoded inside of it):

- operation code, which defines the action which the processor shall execute;
- operation suffix, which defines the rules of formation of operation operands;
- type of operation, which defines the size of operands and interpretation of their values;
- a set of data (value of a reference to the result of a command, value of a reference to the case, raw value) necessary for formation of operands;
- other data necessary for execution of operation and actions connected with an operation.

4.4.1 Convention

$@S, @S1, @S2$ — stands for a reference to the result of a command, which is saved in a communicator, in relation to a current command (ref. to section “Commutator”).

$\#R$ — stands for a reference to the case (ref. to section “Cases”).

$\#GPR$ — stands for a reference to the general purpose register (ref. to section “General Purpose Register”).

$\#IR$ — stands for a reference to the index register (ref. to section “Index Registers”).

$\#IR_{index}$, $\#IR_{mask}$, $\#IR_{base}$ — stands for an index, template and a base of the index register correspondingly IR.

$\#CR$ — stands for a reference to control register (ref. to section “Control Registers”).

$V8$, $V32$ — stands for a raw value with a size of a bite (8 bits), a word (32 bits).

ARG, $ARG1$, $ARG2$ — general indication of command arguments

$DM(ADDR)$ — general indication of access to data memory at the address $ADDR$

$EXPR$ — general indication of an expression

$RES(EXPR)$ — general indication of a result of expression calculation

$(|b|l|q)$ — general indication of choosing one of possible values, i. e. either nothing, or b, or l, or q

4.4.2 Types of operations

In general, a multicellular processor may execute operations over the following operand types:

- signed / unsigned integer, with dimension of 1 bite (8 bits);
- signed / unsigned integer, with dimension of 4 bites (32 bits);
- unsigned integer, with dimension of 8 bites (64 bits);
- signed real-valued short precision, with dimension of 4 bites (32 bits);
- signed compressed, with dimension of 8 bites (64 bits), 4 high-order bites (from 32 to 63 bits) are a high part and 4 low bites (from 0 to 31 bits) are a low part (type of a high and a low part is — real-valued and short precision);
- signed complex, with dimension of 8 bites (64 bits), 4 high bites (from 32 to 63 bits) is a real part, and 4 low bites (from 0 to 31 bits) is an imaginary part (type of a real and imaginary part is — real-valued and short precision);

In an assembler the dependence from a code of data type is shown in a command mnemonics. Command mnemonics consists of 2 parts of a root, which corresponds to the code of operation, and a suffix, which corresponds to the type of operation.

in the table below 4.12 the mnemonics of suffixes of operation types are shown.

Table 4.12: Mnemonics of operation types

Operation type	**Unsigned**	**Signed**
Integer, size 1 byte	b	sb
Integer, size 4 bytes	l	sl
Integer, size 8 bytes	q	-
Signed real-valued single precision, size 4 bytes	-	f
Signed packed, size 8 bytes	-	p
Signed complex, size 8 bytes	-	c

4.4.3 General principle of command building in an assembler

On the fig. 4.15 a general syntax of commands of a multicellular processor is shown.

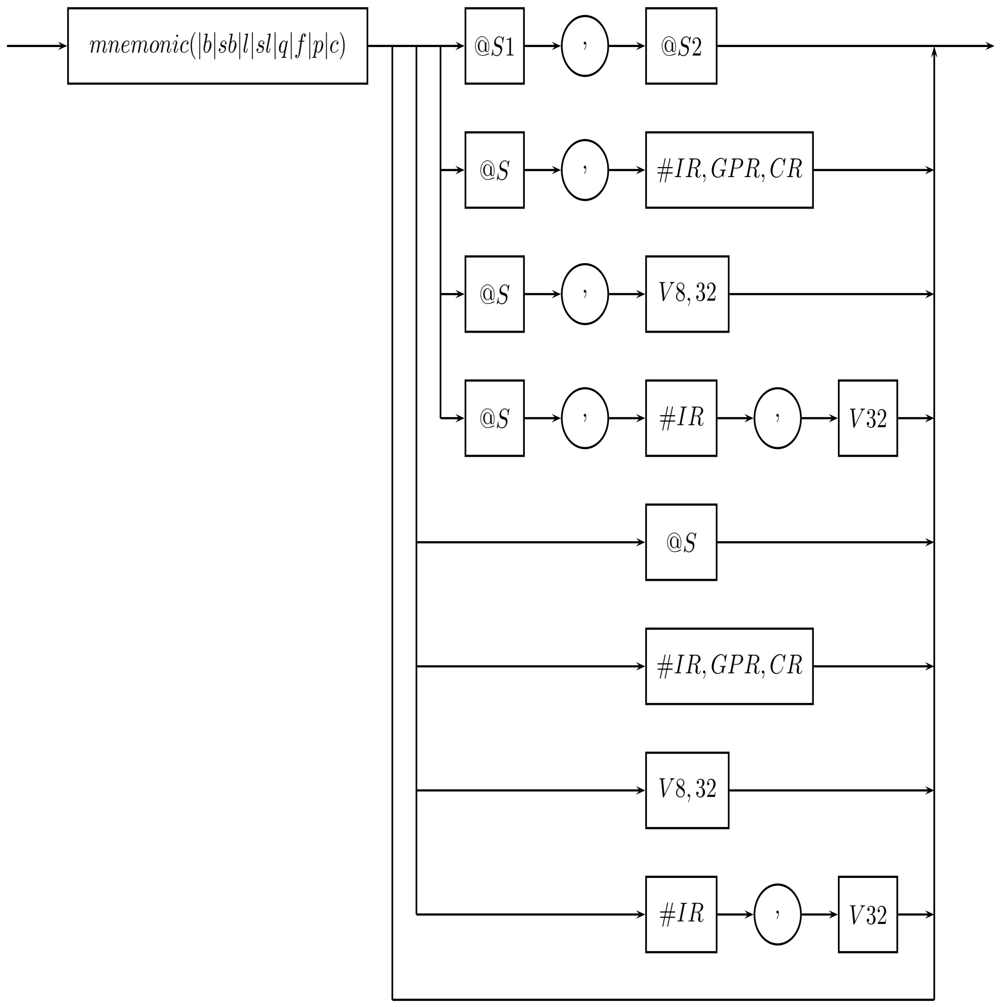


Figure 4.15: Syntax description of multicellular processor command

4.4.3.1 General rules of formation of command arguments and their interpretation

In general the commands may be two-argument, one-argument and without arguments.

According to the syntax description (рис. 4.15), as the first argument of a two-argument command the result is always used. This is the result of execution of one of 63 previous commands, which is located in a commutator and is specified with a reference to this value.

the second argument of a two-argument command or an argument of a one-argument command may be specified using one the following variants:

- using the result of execution of one of 36 previous commands, which are located in a commutator and specified with a reference to this value.
- using the value of a general purpose register ($\#GPR$) or a value of control register ($\#CR$)
- using a value, decoded from data memory by the address, calculated on the basis of a value of an index register ($\#IR$), according to the formulae

$$\#IR_{base} \ + \ \#IR_{index}$$

- using a raw value, specified in a command word, with dimension of bites ($V8$) and word ($V32$) according to the type of a command
- using a value, decoded from data memory by the address, calculated on the basis of a value of index register ($\#IR$) and a raw 32-positioning value, specified in a command word, according to a formulae

$$\#IR_{base} + \#IR_{index} + V32$$

For exceptions from the rules of formation of command arguments and their interpretation ref. to description of a certain command.

4.4.3.2 Rules of formation of command results

In general, command form a result, saved in a commutator. The size of a value of any command corresponds to the size of a commutator — 64 bits.

The result is terminated with values of indicators:

Table 4.13: Result flags.

Flag	**Description**
ZF (Zero Flag)	Is set in case when all bits of result command are equal to zero
OF (Overflow Flag)	Is set in case of significant bit's loss
CF (Carry Flag)	Is set in case when command's result goes beyond word length

SF (Sign Flag)	Is set equal to command result most significant(signed) bit's value

Result indicators are set depending upon the operation type and may have a value not for each command. More detail information you may find in description of a certain command.

4.4.4 Description of commands

4.4.4.1 abs (ABSolute value)

Absolute value

abs ARG

Assignment: operation of calculation of an absolute value of an argument

Syntax:

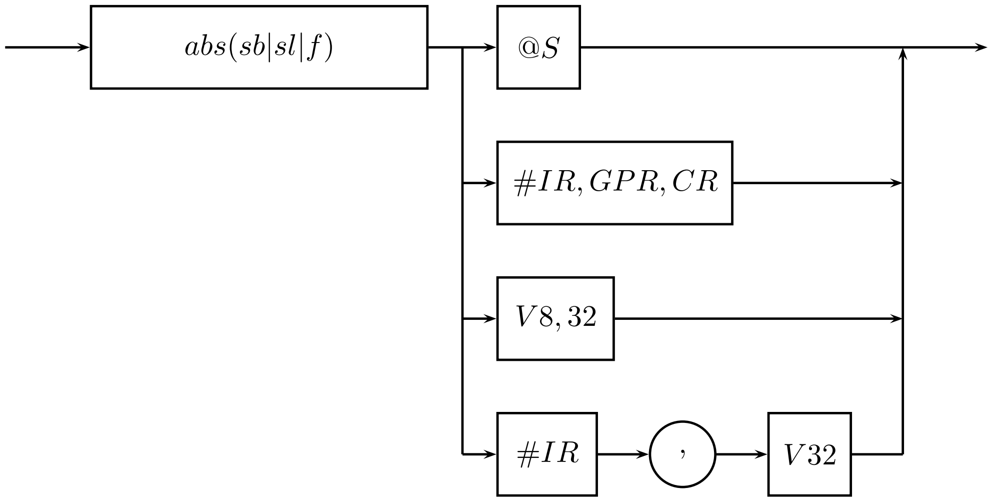


Figure 4.16: Syntax description of command *abs*

Occurrence of a result: yes

Algorithm of work:

– calculate the absolute value of an argument: ARG;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	r	r

Application: command abs is used for calculation of an absolute value. The result of calculation of an absolute value of minimum possible integer digit, depending on the type of operation, overruns word size, on which the indicator of overflow signalizes (OF).

Example:

```
1  .text
2
3  A:
4         getsb  -128
5         abssb  @1
6         abssl  0xF0010203
7         absf  0f-12.85
8  complete
```

Explanation of an example:

– in the line No. 1 a current section of assembly — section of executed instructions text is set as command of an assembler .text;

– in the line No. 3 a symbol (indicator) A is declared, which is an indicator in a current section of assembly (text), and is initialized by a current value of assembly address, and starts a new paragraph;

– in the line No. 4 an 8-positioning integer sign digit is placed in a commutator with a command getsb (constant −128);

– in the line No. 5 with a command abssb an absolute value of the execution result of a previous command is being calculated: @1 — the result of command execution extraction in the line No. 4; argument of a command abssb in accordance with suffix sb is interpreted as integer sign with the dimension of a bite; the result of command execution in interpreted as integer sign with the dimension of a bite and is located into a commutator with a set indication of overflow (OF), since the value 128 overruns word size of a sign integer digit with the dimension of a bite;

- in the lines No. 6, 7 other variants of usage of command abs are shown;
- in the line No. 8, with a command complete a current paragraph is finished.

4.4.4.2 adc (ADdition with Carry)

Addition with carry

*adc ARG*1, *ARG*2

Assignation: operation of integer-valued addition with consideration of carry indication (CF) of the result of previous addition by a command *add*

Syntax:

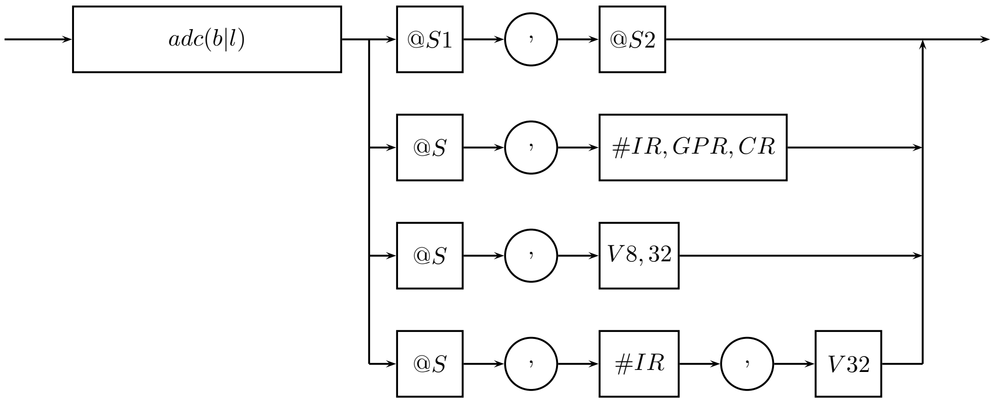


Figure 4.17: Syntax description of command *adc*

Occurrence of a result: yes

Algorithm of work:

- - perform addition of carry indicator values (CF) argument $ARG1$ with a value of argument $ARG2$;
- - set indicators;
- - place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

Application: command adc is used by addition of long integer digits. As distinct from similar commands of most other processors; a command adc of a multicellular processor adds a value of a second argument with 1, if a carry indicator (CF) of the first argument is set, otherwise just a value of the second argument returns as a result.

Example:

```
1  /*
2   * Program of two 64 bit numbers' addition
3  */
4
5  .data
6
7  A:
8          .long 0x00010203, 0xAABBCCDD
9  B:
10         .long 0x04050607, 0xEEFF0908
11 C:
12         .long 0x00000000, 0x00000000
13
14 .text
15
16 D:
17         rdl A
18         rdl A + 4
19         rdl B
20         rdl B + 4
21         addl @3, @1; result 0x99BAD5E5, OF == 1
22         addl @5, @3; result 0x0406080A
23         adcl @2, @1; result 0x0406080B
24         wrl @1, C
25         wrl @4, C + 4
26 complete
```

Explanation of an example:

- in the line No. 5 a current assembly section is established as a command of assembler .data — a section of data assembly data;
- in the lines No. 7, 9, 11 the symbols (indicators) A, B, C, which are indicators of a current assembly section (data) are declared, and initialized with current values of assembly address;
- in the lines No. 8, 10, 12 32-positioning digits: $0x00010203$, $0xAABBCCDD$, $0x04050607$, $0xEEFF0908$, $0x00000000$, $0x00000000$ are being written into an

assembler command .long into a current assembly section on current address of assembly;;

- in the line No. 14 a current assembly section - a section of executed instructions text is set with an assembler command;
- in the line No. 16 a symbol (indicator) D is declared, which is an indicator in a current assembly section (text) and is initialized with a current assembly address value and starts a new paragraph;
- in the lines No. 17, 18, 19, 20 with commands rdl four 32-positioning integer unsigned digits are being read from data memory by addresses A, $A+4$, B, $B+4$ correspondingly and are placed into a commutator;
- in the line No. 21 with a command addl the results of execution of two previous commands are added: @3 — the result of execution of a reading command in the line No. 18, @1 — the result of execution of a reading command in the line No. 20; both arguments of a command addl in accordance with siffix I are interpreted as 32-positioning integer unsigned digits; the result of execution of a command is also interpreted as 32-positioning integer unsigned digit and is placed into a commutator;
- in the line No 22 with a command addl the results of execution of 2 previous commands are addes: @5 — the result of execution of a reading command in the line No. 17, @3 — the result of execution of a reading command in the line No. 19; both arguments of a command addl in accordance with suffix I are interpreted as 32-positioning integer unsigned digits; the result of execution of a command is also interpreted as 32-positioning integer unsigned digit and is placed into a commutator;
- in the line No. 23 with a command adcl the results of execution of two previous commands are added: @2 — the result of execution of an adding command in the line No. 20 (in fact only the value of an overflow indicator is used OF), @1 — the result of execution of an adding command in the line No. 21; both arguments of a command addl in accordance with suffix I are interpreted as 32-positioning integer unsigned digits; the result of execution of a command is also interpreted as 32-positioning integer unsigned digit and is placed into a commutator;
- in the line No. 24 with a command wrl the recording into data memory is being performed by an address C of the result of execution of a previous command: @1 — the result of execution of an adding command with consideration of a carry

indicator in the line No. 24;

– in the line No. 25 with a command wrl the recording into data memory is being performed by an address $C + 4$ results of execution of a previous command: @4 — the result of execution of an adding command in the line No. 21;

– in the line No. 26 with a command complete a current paragraph is being completed.

4.4.4.3 add (ADDition)

Addition

*add ARG*1, *ARG*2

Application: operation of addition of two arguments

Syntax:

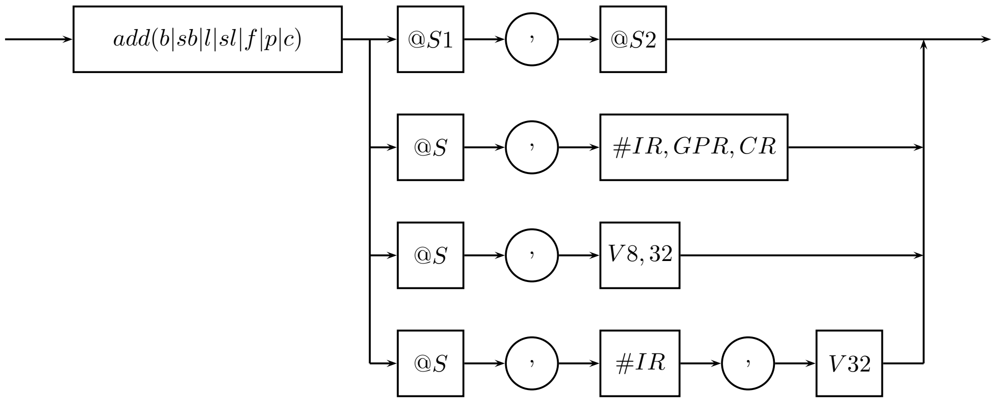


Figure 4.18: Syntax description of command *add*

Occurrence of the result: yes

Algorithm of work:

– perform addition of $ARG1 + ARG2$;

– set indicators;

– place a result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

Application: command add is used for addition of two commands, values of which are interpreted in accordance with the type of operation.

Example:

```
.data

B:
 4         .float\
 5              0f12.8, 0f-5.6,\
 6              0f-1.78, 0f0.19
 7
 8    .text
 9
10    A:
11         rdq B
12         rdq B + 8
13         addc @1, @2
14         wrq @1, B + 16
15    complete
```

Explanation of an example:

– in the line No. 1 with an assembler command .data a current assembly section is set — which is the section of initialized data;

– in the line No. 3 a symbol (indicator) B is declared, which is an indicator in a current assembly section (data), and is initialized with a current assembly address value;

– in the line No. 4 with an assembler command .float four 32-positioning real-valued digits are written into a current assembly section, starting from a current address of assembly (a symbol of a reverse slash \ in the end of a line is used to continue a line, i.e. the lines No. 4, 5, 6 are logically a one line);

– in the line No. 8 with an assembler command .text a current assembly section is set — a section of executed instructions text;

– in the line No. 12 a symbol (indicator) A, is declared, which is an indicator of a current assembly section (text), and is initialized with a current assembly address value, and starts a new paragraph;

– in the lines No. 11, 12 with commands rdq, two 64-positioning integer unsigned digits are being read from data memory by addresses B and $B+8$ and accordingly are placed into a commutator;

– in the line No. 13 with a command addc an adding operation of the results of execution of 2 previous commands is being performed: @1 — the result of an execution of a reading command in the line No. 11, @2 — the result of execution of a reading command in the line No. 12; both arguments of a command addc according to suffix are interpreted as complex digits with the dimension of 64 bits (32 high positions represent real parts of digits, and 32 low positions are imaginary). The result of a command execution is interpreted as a 64-positioning complex digit and is placed into a commutator;

- in the line No. 14 with a command wrq recording into data memory is being performed at the address $B+16$ of the result of execution of a previous command: @1 — the result of execution of an addition command in the line No. 13;
- in the line No. 15 with a command complete a current paragraph is being completed.

4.4.4.4 and (AND)

Logical multiplication

and $ARG1$, $ARG2$

Assignation: operation of logical multiplication of two arguments

Syntax:

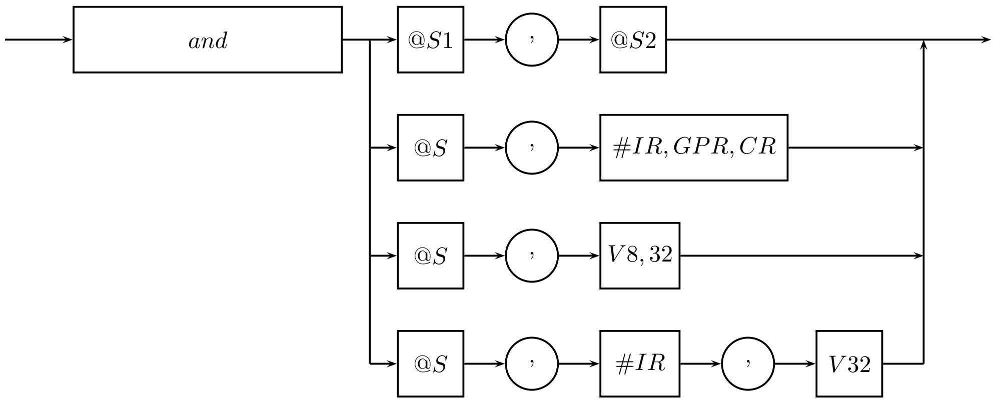


Figure 4.19: Syntax description of command *and*

Occurrence of the result: yes

Algorithm of work:

– perform AND operation $ARG1$ & $ARG2$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.5 cfsl (Convert Float to Signed Long)

Transformation of type

cfsl ARG

Assignation: Operation of type transformation

Syntax:

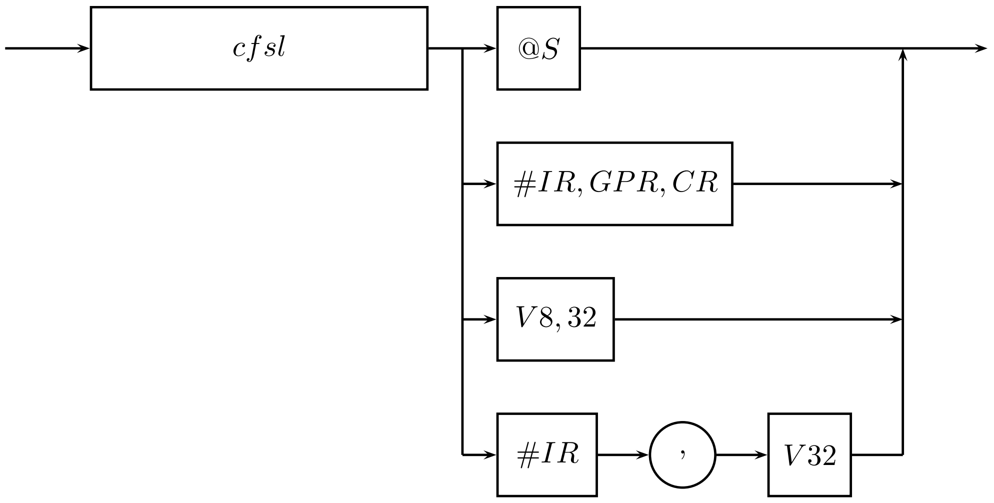


Figure 4.20: Syntax description of command *cfsl*

Occurrence of the result: yes

Algorithm of work:

– implement round off of a real number single precision, set by argument *ARG*2, to proximal integer;

– implement round off of a real number single precision to signed 32-bit integer number;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	r	r

Application: command cfsl is used for transformation with preliminary round-off to the nearest integer operand value, an interpreted as real-valued digit of short precision, to a signed integer 32-positioning digit.

Example:

```
1 .text
2
3 A:
4       clf 1394
5       wrl @1, 0
6 complete
```

Explanation of an example:

– in the line No. 1 with an assembler command .text a current assembly section is set — a section of executed instructions text;

– in the line No. 3 a symbol (indicator) A, which is an indicator in a current assembly section(text), and is initialized with a current assembly address value and starts a new paragraph;

– in the line No. 4 with a command cfsl an operation of conversion is being performed with integer unsigned 32-bit number 1394 to the real-valued short precision number;

– in the line No. 5 with a command wrl a value of the result of execution of a previous command is recorded into data memory at the address 0: @1 — the result of execution of a conversion command in the line No. 4;

– in the line No. 6 with a command complete a current paragraph is being completed.

Example:

```
1 .text
2
3 A:
4       cfsl 0f1.394E1
5       wrl @1, 0
6 complete
```

Explanation of an example:

– in the line No. 1 with an assembler command .text, a current assembly section is set — a section of executed instructions text;

– in the line No. 3 a symbol (indicator) A, is declared, which is an indicator in a current assembly section (text), and is initialized with a current assembly address value and starts a new paragraph;

– in the line No. 4 with a command cfsl an operation of conversion is being performed with preliminary round-off to the nearest integer real-valued digit of short precision 13.94 to an integer signed 32-positioning digit (the result of execution is an operation — 14);

– in the line No. 5 with a command wrl a value of the result of execution of a previous command is recorded into data memory at the address 0: @1 — the result of execution of a conversion command in the line No. 4;

– in the line No. 6 with a command complete a current paragraph is being completed.

4.4.4.6 clf (Convert Long to Float)

Type conversion

clf ARG

Assignation: operation of a type conversion

Syntax:

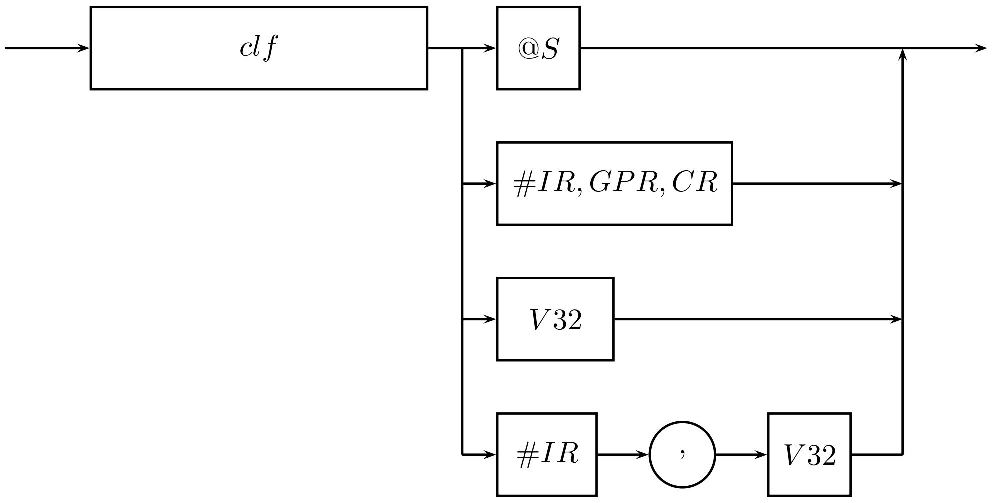


Figure 4.21: Syntax description of command *clf*

Occurrence of the result: yes

Algorithm of work:

– implement transformation of signed 32-bit integer number, set by argument *ARG*, into a number with floating point;

– - set indicators;

– - place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	0	r

Application: a command cfsl is used to convert (with preliminary round-off to the nearest integer operand value) an interpreted as real-valued digit of short precision, to a signed integer 32-positioning digit.

Example:

```
1 .text
2
3 A:
4         clf 1394
5         wrl @1, 0
6 complete
```

Explanation of an example:

- in the line No. 1 with an assembler command .text a current assembly section is set — a section of executed instructions text;
- in the line No. 3 a symbol (indicator) *A*, which is an indicator in a current assembly section(text), and is initialized with a current assembly address value and starts a new paragraph;
- in the line No. 4 with a command cfsl an operation of conversion is being performed with integer unsigned 32-bit number 1394 to the real-valued short precision number;
- in the line No. 5 with a command wrl a value of the result of execution of a previous command is recorded into data memory at the address 0: @1 — the result of execution of a conversion command in the line No. 4;
- in the line No. 6 with a command complete a current paragraph is being completed.

4.4.4.7 cslf (Convert Signed Long to Float)

Type conversion

cslf ARG

Assignation: operation of a type conversion

Syntax:

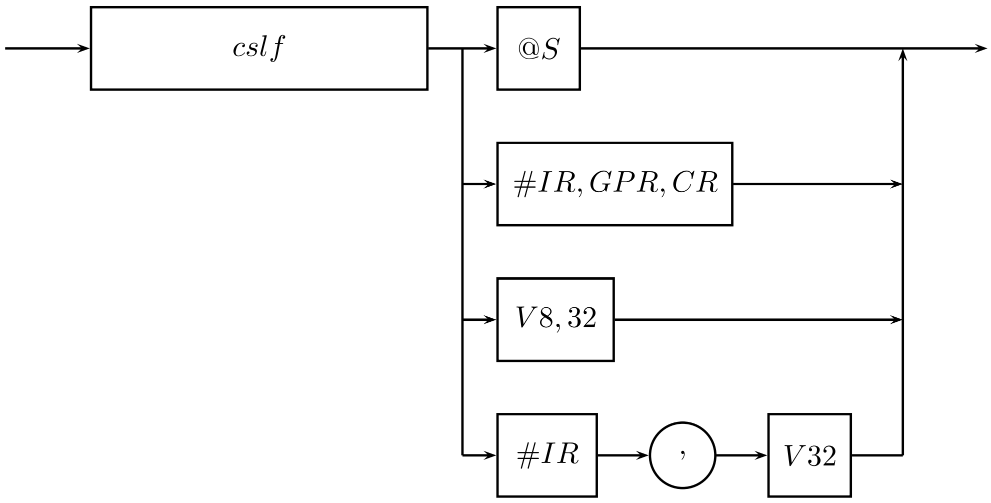


Figure 4.22: Syntax description of command *cslf*

Occurrence of the result: yes

Algorithm of work:

- implement transformation of signed 32-bit integer number, set by argument *ARG*, into a number with floating point;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	0	r

Application: command cfsl is used to convert (with preliminary round-off to the nearest integer operand value) an interpreted as real-valued digit of short precision, to a signed integer 32-positioning digit.

Example:

```
1  .text
2
3  A:
4          cslf  −1394
5          wrl  @1,  0
6  complete
```

Explanation of an example:

– in the line No. 1 with an assembler command .text a current assembly section is set — a section of executed instructions text;

– in the line No. 3 a symbol (indicator) *A* which is an indicator in a current assembly section(text), and is initialized with a current assembly address value and starts a new paragraph;

– in the line No. 4 with a command cfsl an operation of conversion is being performed with integer unsigned 32-bit number −1394 to the real-valued short precision number;

– in the line No. 5 with a command wrl a value of the result of execution of a previous command is recorded into data memory at the address : @1 — the result of execution of a conversion command in the line No. 4;

– in the line No. 6 with a command complete a current paragraph is being completed.

4.4.4.8 div (DIVide)

Division

div $ARG1,\ ARG2$

Assignation: Division operation of two arguments

Syntax:

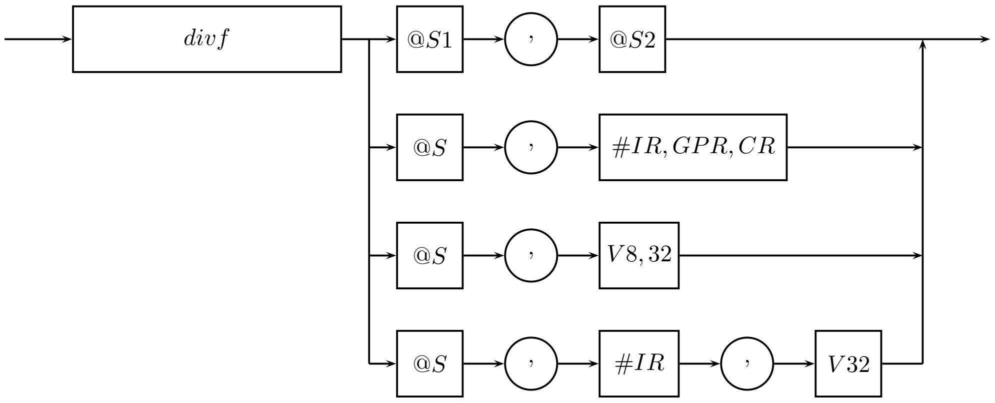


Figure 4.23: Syntax description of command *div*

Occurrence of the result: yes

Algorithm of work:

- implement division $ARG1/ARG2$;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	r	r

Application: command div is used for division of two operands, which value is interpreted in accordance with the operand type. During operation execution a spesific situation may occur — division on 0. In this case interrupt request is formed, that is, relevent bits of interrupt registers $INTR$ and exceptions ER are set.

Example:

```
1  .data
2
3  B:
4          .float  0f12.8e12 ,  0f-5.6e-4
5
6  .text
7
8  A:
9          rdl  B
10         rdl  B + 4
11         divf  @1,  @2
12         wrl  @1,  B + 8
13 complete
```

Explanation of an example:

- in the line No. 1 with an assembler command .data a current assembly section is set — which is the section of initialized data;
- in the line No. 3 a symbol (indicator) *B* is declared, which is an indicator in a current assembly section (data), and is initialized with a current assembly address value;
- in the line No. 4 with an assembler command .float into a current assembly section 2 32-bit real-valued numbers are set, beginning from current assembly address;
- in the line No. 6 with an assembler command .text a current assembly section is set — a section of executed instructions text;
- in the line No. 8 a symbol (indicator) *A* is declared, which is an indicator of a current assembly section (text), and is initialized with a current assembly address value, and starts a new paragraph;
- in the lines No. 9,10 with commands rdl, two 32-positioning integer unsigned digits are being read from data memory by addresses *B* and *B*+4 and accordingly are placed into a commutator;
- in the line No. 11 with a command divf a subtraction operation of the results of execution of 2 previous commands is being performed: @1 — the result of an execution of a reading command in the line No. 19, @2 — the result of execution of a reading command in the line No. 10; both arguments of command divf according to suffix f is interpreted as real-valued single precision numbers; the

result of program implementation is interpreted as real-valued single precision number and is placed in commutator;

– in the line No. 13 with command complete current paragraph is being completed.

4.4.4.9 exa (EXacutive Address)

Executive address

exa ARG

Assignation: form executive address of data memory

Syntax:

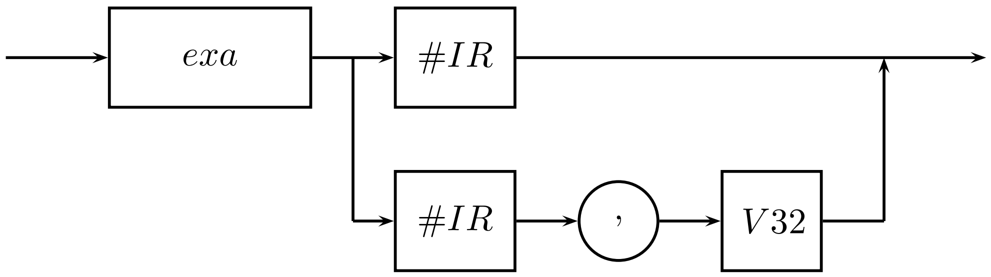


Figure 4.24: Syntax description of command *exa*

Occurrence of the result: yes

Algorithm of work:

- form executive address of data memory using values of index registers and depositions set by argument *ARG*;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
0	0	0	r

Application: command exa is used to form executive address of data memory and to save it in commutator. In other words, given command unlike most other processor commands does not implement reading of value from data mamory under formed address. Rules of forming an executive address are described in Section "Index registers" (4.1.3.2). In general, interpretation of the meaning saved by this command in commutator, depends on program algorithm.

4.4.4.10 get (GET value)

Extract of value

get ARG

Assignation: operation of getting value

Syntax:

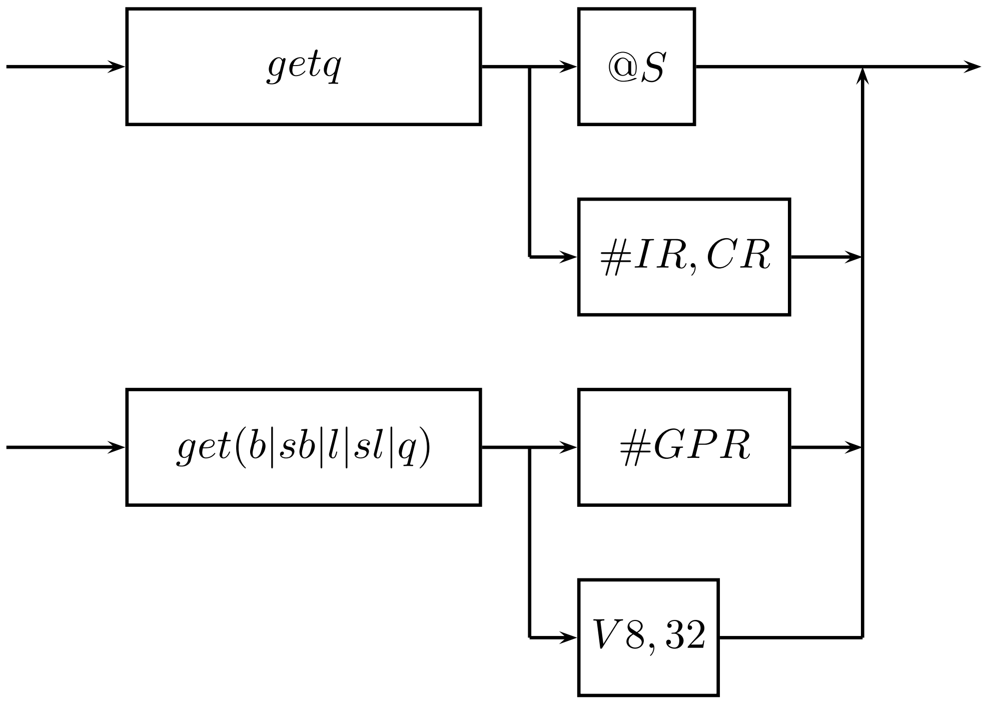


Figure 4.25: Syntax description of command *get*

Occurrence of the result: yes

Algorithm of work:

– get value, set by argument *ARG*;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	0	r

Application: command get is used

- to find value, earlier saved in commutator;
- for getting value from the register of all types and saving it in commutator, that is despite register type the value of exactly given register is got.
- for saving constatnts in commutator.

Example:

```
1 .text
2
3 A:
4         getsb  -128
5         getq  #0
6         getl  #32
7         getl  #PSW
8         getq  @4
9 complete
```

Explanation of an example:

- in the line No. 1 a current section of assembly — section of executed instructions text is set as command of an assembler;
- in the line No. 3 a symbol (indicator) *A*, is declared, which is an indicator in a current section of assembly (text), and is initialized by a current value of assembly address, and starts a new paragraph;
- in the line No. 4 an 8-positioning integer sign digit is placed in a commutator with a command getsb (constant −128);
- in the line No. 5 with a command getq general purpose register's value No. 0 is calculated and saved in commutator;
- in the line No. No. 6 with a command getl index register's base value No. 32 is calculated and saved in commutator(despite of in command getl index register is pointed as an argument, formation of executing data memory address, and addressing to data memory is not being performed,lowest 32 bits of index register are being calculated and saved in commutator);
- in the line No.7 with a command getl control register's value is calculated *PSW* and is saved in commutator;

- in the line No. 8 with a command getq earlier saved in commutator value is being saved, that is previous command's implementation result is being performed with flag values upgrade: @4 — the result of an extraction command in the line No4;
- in the line No. 9 with command complete current paragraph is being completed.

4.4.4.11 insub (INversion SUBtract)

Inverse subtraction

$insub\ ARG1,\ ARG2$

Assignation: operation of inverse subtraction of two arguments

Syntax:

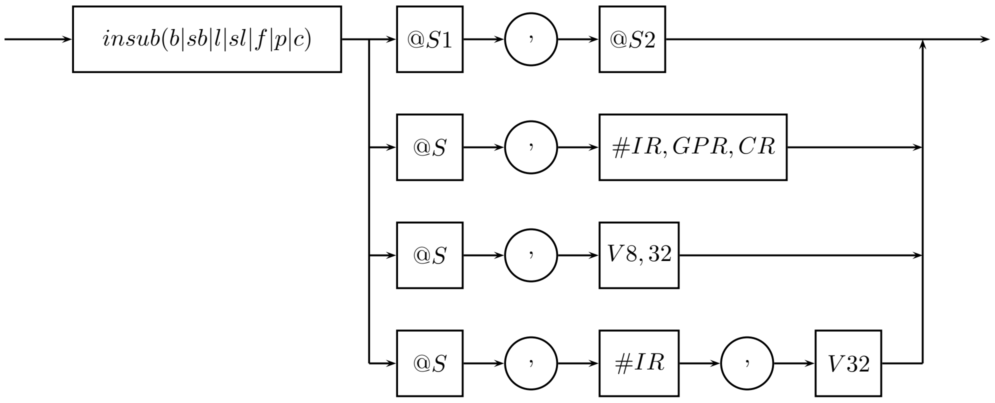


Figure 4.26: Syntax description of command *insub*

Occurrence of the result: yes

Algorithm of work:

- implement subtarction $ARG2 - ARG1$;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

Application: command insub is used for inverse subtraction of operands, meaning of which is interpreted in accordance with operation type.

Example:

```
.data

B:
```

```
4        .float\
5            0f12.8, 0f-5.6,\
6            0f-1.78, 0f0.19
7
8  .text
9
10 A:
11       rdq B
12       rdq B + 8
13       insubp @1, @2
14       wrq @1, B + 16
15 complete
```

Explanation of an example:

- in the line No. 1 with an assembler command .data a current assembly section is set — which is the section of initialized data;
- in the line No. 3 a symbol (indicator) *B* is declared, which is an indicator in a current assembly section (data), and is initialized with a current assembly address value;
- in the line No. 4 with an assembler command .float into a current assembly section 4 32-bit real-valued numbers are set, beginning from current assembly address (a symbol of a reverse slash \ in the end of a line is used to continue a line, i.e. the lines No. 4, 5, 6 are logically a one line);
- in the line No. 8 with an assembler command .text a current assembly section is set — a section of executed instructions text;
- in the line No. 10 a symbol (indicator) *A*, is declared, which is an indicator in a current assembly section (text), and is initialized with a current assembly address value, begins new paagraph;
- in the lines No. 11, 12 with commands rdq, two 64-positioning integer unsigned digits are being read from data memory by addresses *B* and *B*+8 and accordingly are placed into a commutator;
- in the line No. 13 with a command insubp reverse subtarction operation of the execution result of a previous command is being calculated: @1 — the result of an execution of a reading command in the line No.11, @2 — the result of an execution of a reading command in the line No.12; both arguments of command insubp in

accordance with suffix p are interpreted aspacked packed 64-bit numbers, that is why reverse subtraction operation is performed independently for highest and lowest group of numbers; the result of a command execution is interpreted as a 64-positioning packed digit and is placed into a commutator;

– in the line No. 14 with a command wrq recording into data memory is being performed at the address $B+16$ of the result of execution of a previous command: @1 — the result of execution of reverse subtarction command in the line No.13;

– in the line No. 15 with a command complete a current paragraph is being completed.

4.4.4.12 ja (Jump if Above)

next paragraph address setting, conditional

*ja ARG*1, *ARG*2

Assignation: operation of next paragraph address conditional setting

Syntax:

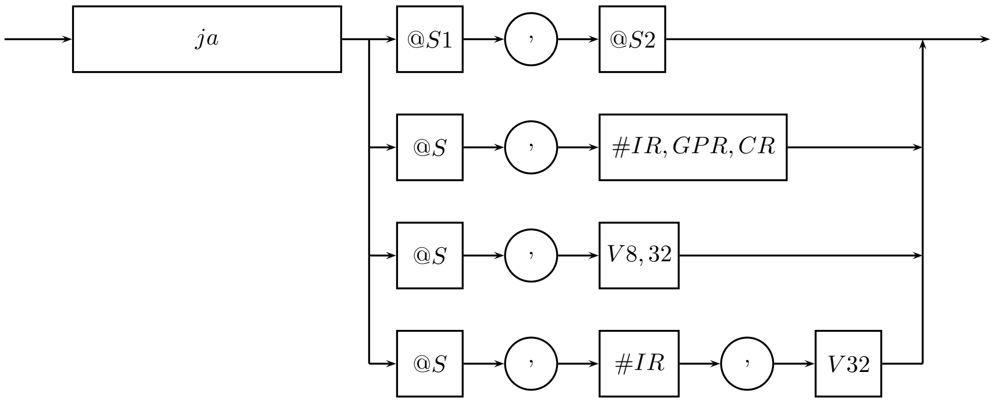


Figure 4.27: Syntax description of command *ja*

Occurrence of the result: no

Algorithm of work:

– set naxt paragraph address equal to argument value *ARG*2 (actual transition will be implemented in the end of current paragraph), if convert flag (CF) argument *ARG*1 is equal to zero and zero flag(ZF) argument *ARG*1 is equal to zero;

4.4.4.13 jae (Jump if Above and Equal) / jnc (Jump if Carry flag unset)

Next paragraph address setting, conditional

$(jae|jnc)\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

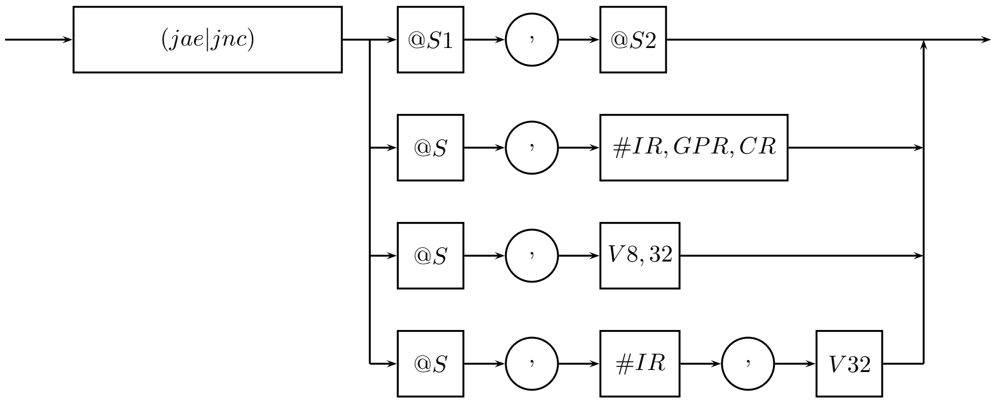


Figure 4.28: Syntax description of command *jae/jnc*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if convert flag (CF) argument $ARG1$ is equal to zero;

4.4.4.14 jb (Jump if Below) / jc (Jump if Carry flag set)

Next paragraph address setting, conditional

$(jb|jc)\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

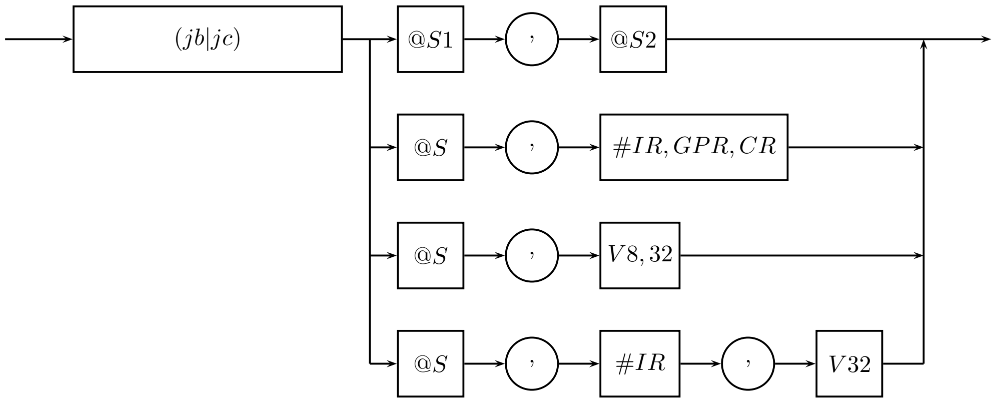


Figure 4.29: Syntax description of command jb/jc

Occurrence of the result: yes

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if convert flag (CF) argument $ARG1$ is equal to one;

4.4.4.15 jbe (Jump if Below and Equal)

Next paragraph address setting, conditional

jbe $ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

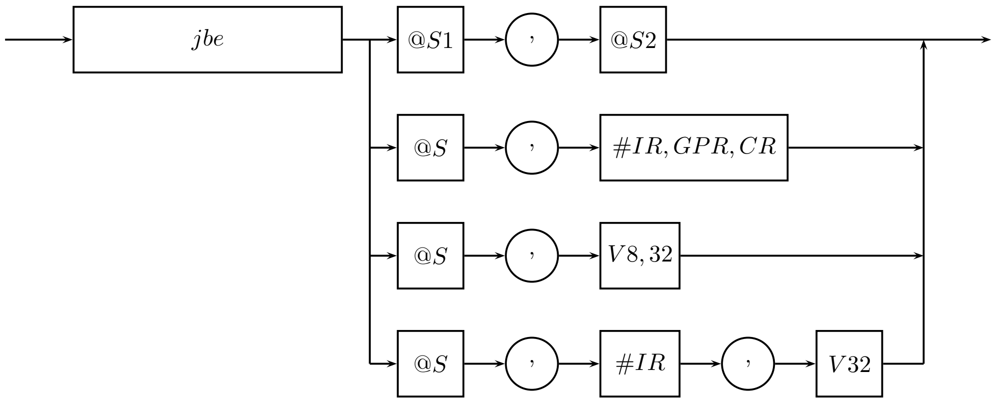


Figure 4.30: Syntax description of command *jbe*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if convert flag (CF) argument $ARG1$ is equal to one or zero flag (ZF) argument $ARG1$ is equal to one;

4.4.4.16 je (Jump if Equal)

Next paragraph address setting, conditional

$je\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

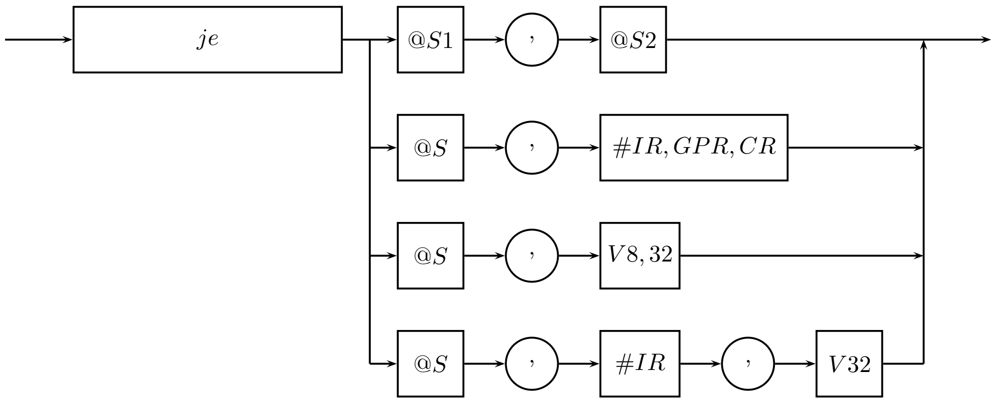


Figure 4.31: Syntax description of command je

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if zero flag (ZF) argument$ARG1$ is equal to one , that is argument $ARG1$ is equal to zero;

4.4.4.17 jg (Jump if Greater)

Next paragraph address setting, conditional

$jg\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

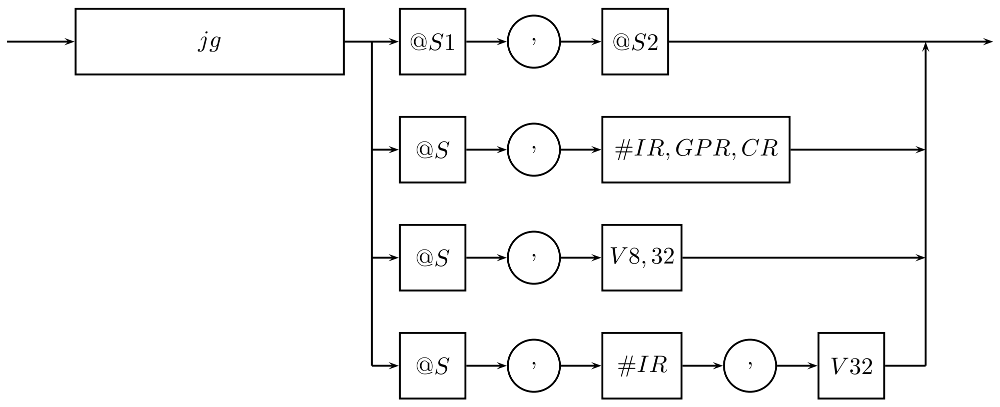


Figure 4.32: Syntax description of command jg

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value$ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flags (SF) and overload flags (OF) argument $ARG1$ are equal and zero flag (ZF) argument $ARG1$ is equal to zero;

4.4.4.18 jge (Jump if Greater and Equal)

Next paragraph address setting, conditional

$jge\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

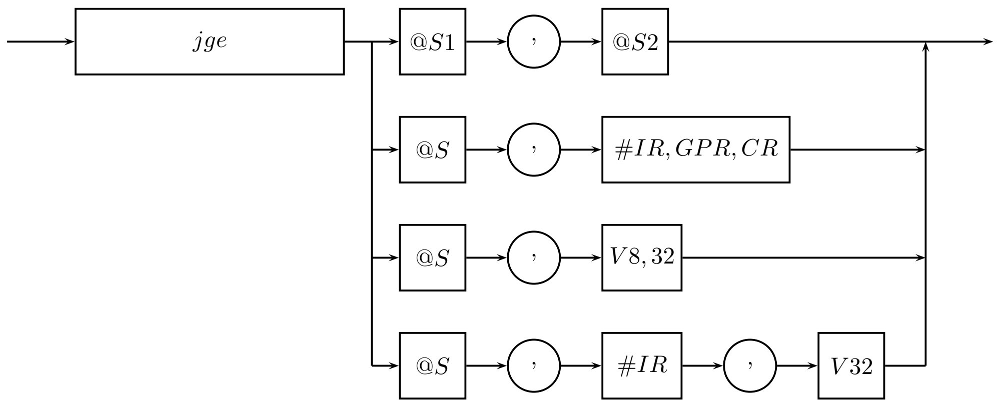


Figure 4.33: Syntax description of command jge

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flags (SF) and overload flags (OF) argument$ARG1$ are equal;

4.4.4.19 jl (Jump if Less)

Next paragraph address setting, conditional

$jl\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

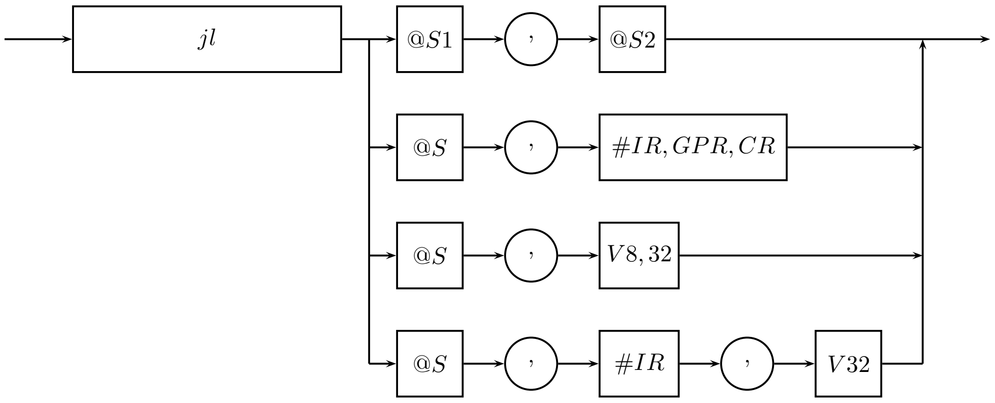


Figure 4.34: Syntax description of command *jl*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flags (SF) and overload flags(OF) argument $ARG1$ are not equal;

4.4.4.20 jle (Jump if Less and Equal)

Next paragraph address setting, conditional

jle ARG1, ARG2

Assignation: operation of conditional next paragraph setting

Syntax:

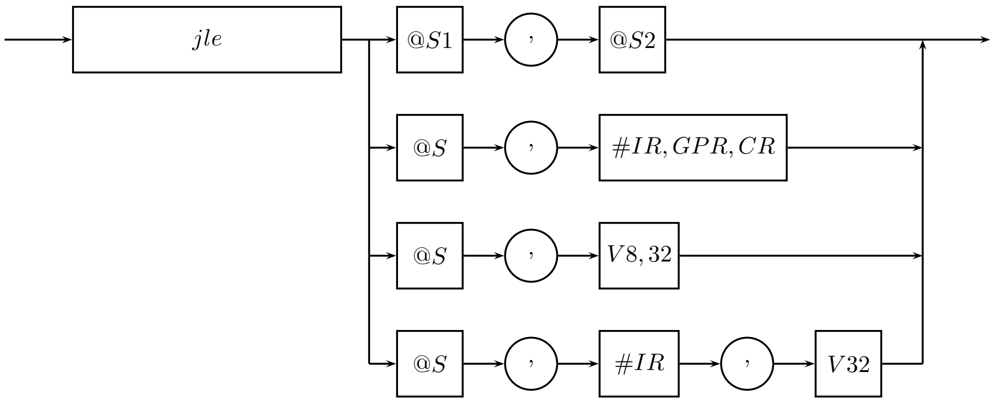


Figure 4.35: Syntax description of command *jle*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flags (SF) and overload flags (OF) argument $ARG1$ are not equal or zero flag (ZF) argument $ARG1$ is equal to one;

4.4.4.21 jmp (unconditional JuMP)

Next paragraph address setting, non-conditional

jmp ARG

Assignation: operation of non-conditional next paragraph setting

Syntax:

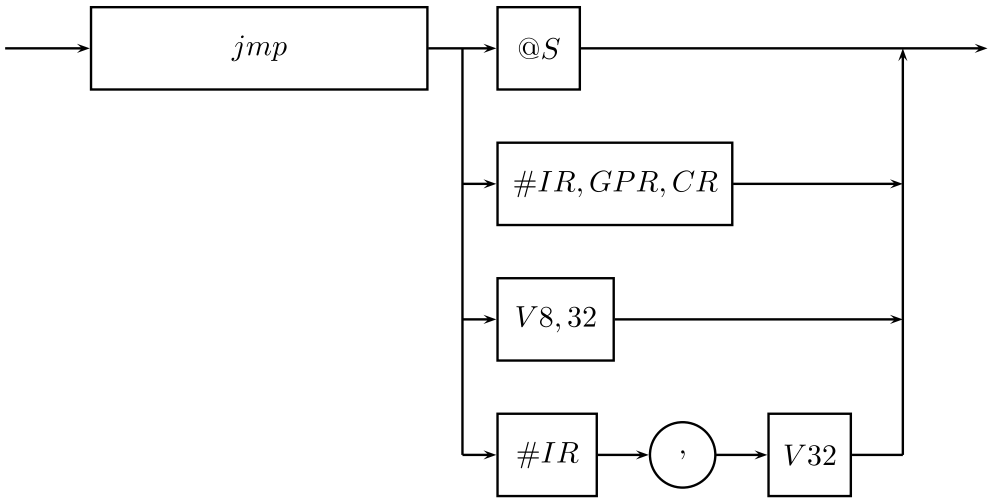


Figure 4.36: Syntax description of command *jmp*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value *ARG* (actual transition will be implemented in the end of current paragraph);

4.4.4.22 jne (Jump if Not Equal)

Next paragraph address setting, conditional

$jne\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

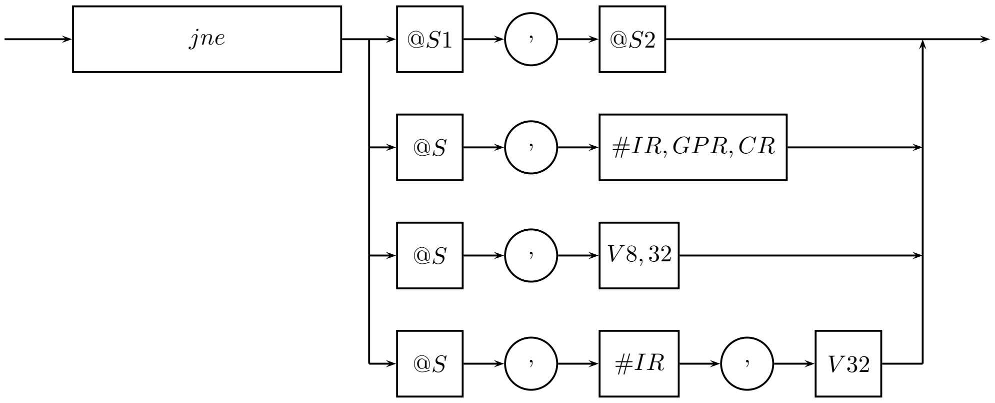


Figure 4.37: Syntax description of command *jne*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if zero flag (ZF) argument $ARG1$ isequal to zero, that is $ARG1$ is not equal to zero;

4.4.4.23 jno (Jump if Overflow flag unset)

Next paragraph address setting, conditional

$jno\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

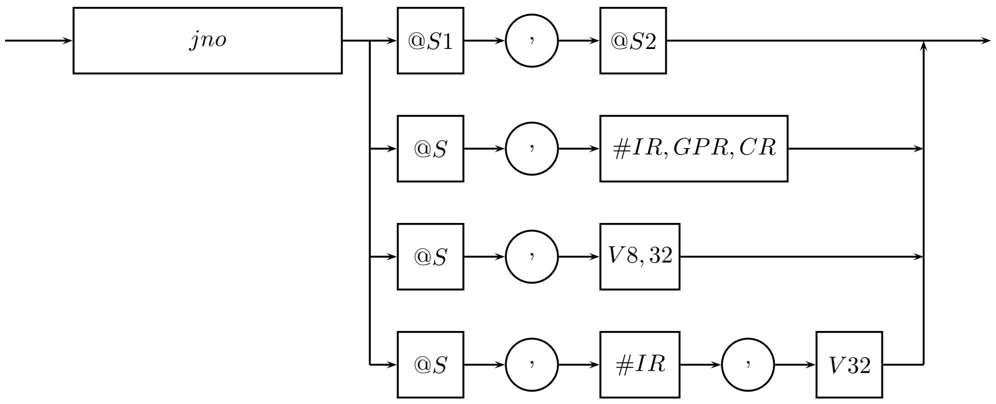


Figure 4.38: Syntax description of command *jno*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if overload flag (OF) argument $ARG1$ is equal to zero;

4.4.4.24 jns (Jump if Sign flag unset)

Next paragraph address setting, conditional

$jns\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

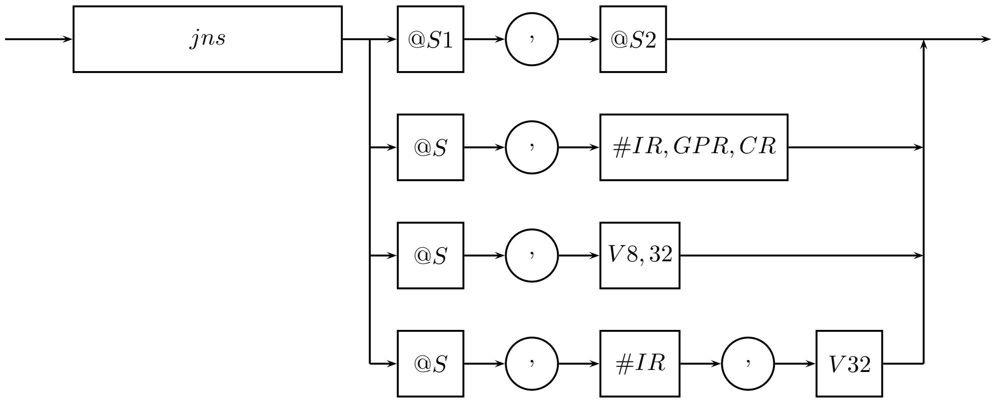


Figure 4.39: Syntax description of command *jns*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flag (SF) argument $ARG1$ is equal to zero;

4.4.4.25 jo (Jump if Overflow flag set)

Next paragraph address setting, conditional

$jo\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

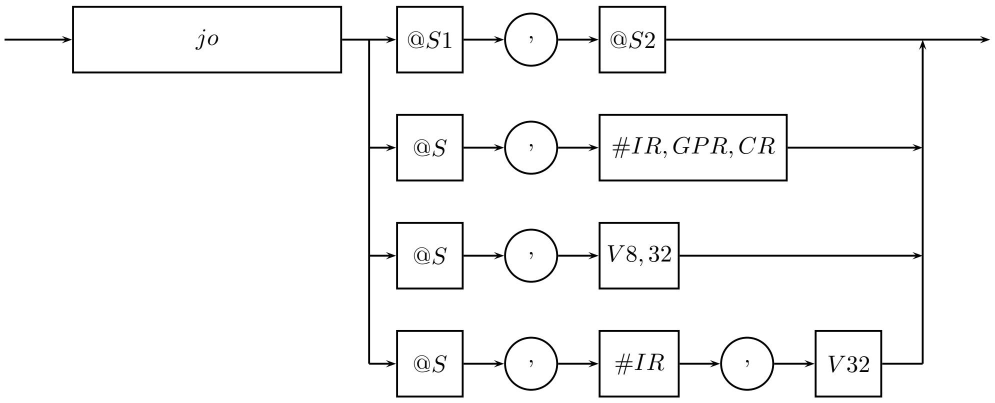


Figure 4.40: Syntax description of command *jo*

Occurrence of the result: no

Algorithm of work:

– set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if overload flag (OF) argument $ARG1$ is equal to one;

4.4.4.26 js (Jump if Sign flag set)

Next paragraph address setting, conditional

$js\ ARG1,\ ARG2$

Assignation: operation of conditional next paragraph setting

Syntax:

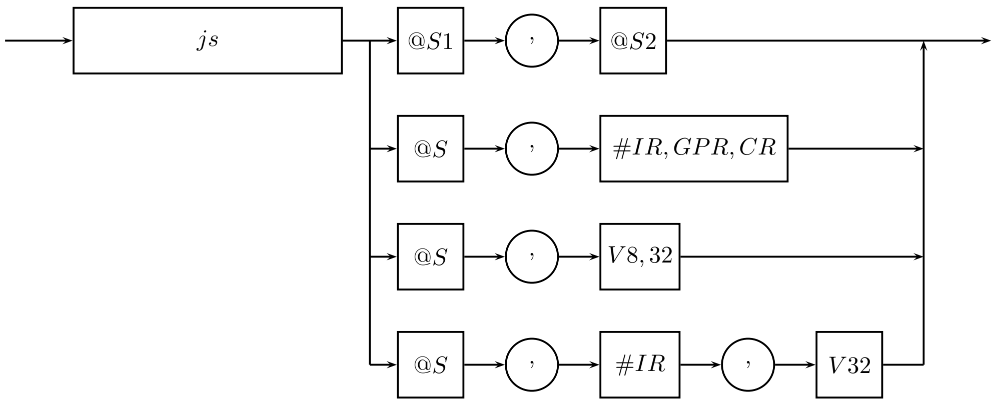


Figure 4.41: Syntax description of command js

Occurrence of the result: no

Algorithm of work:

- set next paragraph address equal to argument value $ARG2$ (actual transition will be implemented in the end of current paragraph), if sign flag (SF) argument $ARG1$ is equal to one;

4.4.4.27 madd (Multiplication with ADDition of packed arguments)

Multiplication with addition

madd ARG1, ARG2

Assignation: operation of packed multiplication with addition

Syntax:

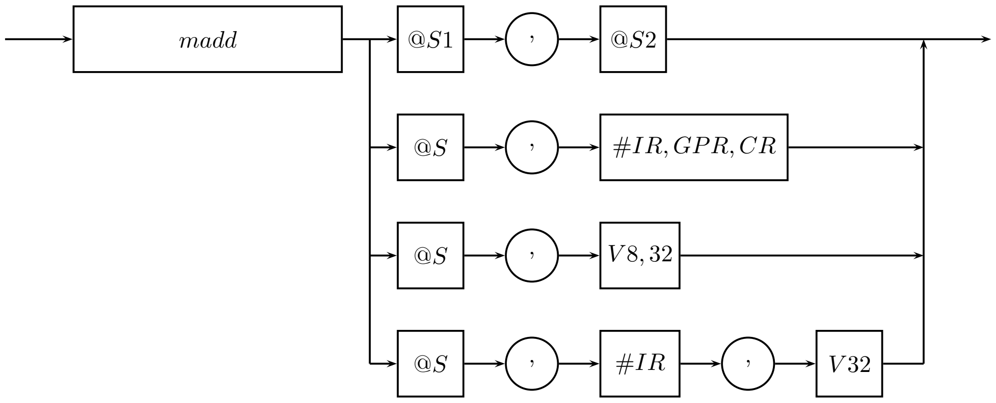


Figure 4.42: Syntax description of command *madd*

Occurrence of the result: yes

Algorithm of work:

- implement operation of addition packed numbers' highest and lowest parts composition (for example $ARG1 = (a, b)$, $ARG2 = (c, d)$, where a, c — highest parts of packed numbers, set by arguments $ARG1$ and $ARG2$ accordingly, and b, d — packed numbers' lowest parts, set by arguments $ARG1$ and $ARG2$ accordingly, then result is equal $a * c + b * d$);
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	r	r

4.4.4.28 max (MAXimum)

Maximum

*max ARG*1, *ARG*2

Assignation: operation of highest argument selection among two

Syntax:

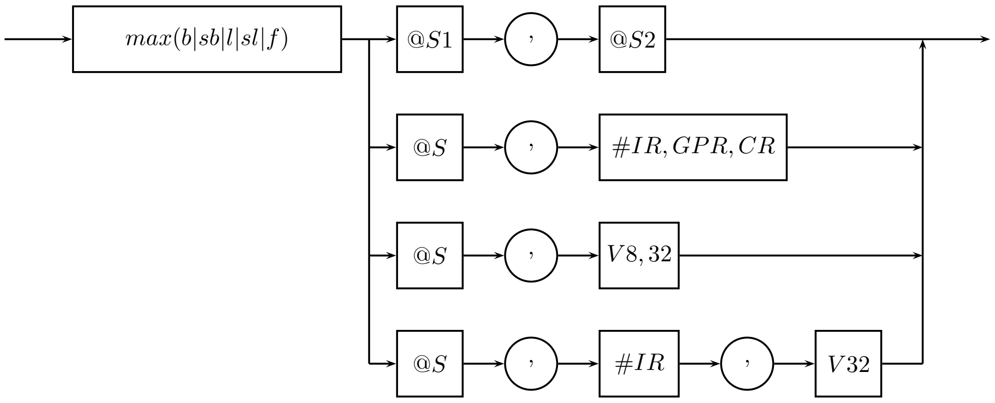


Figure 4.43: Syntax description of command *max*

Occurrence of the result: yes

Algorithm of work:

– implement operation $max(ARG1, ARG2)$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.29 min (MINimum)

Minimum

$min\ ARG1,\ ARG2$

Assignation: operation of lowest argument selection among two

Syntax:

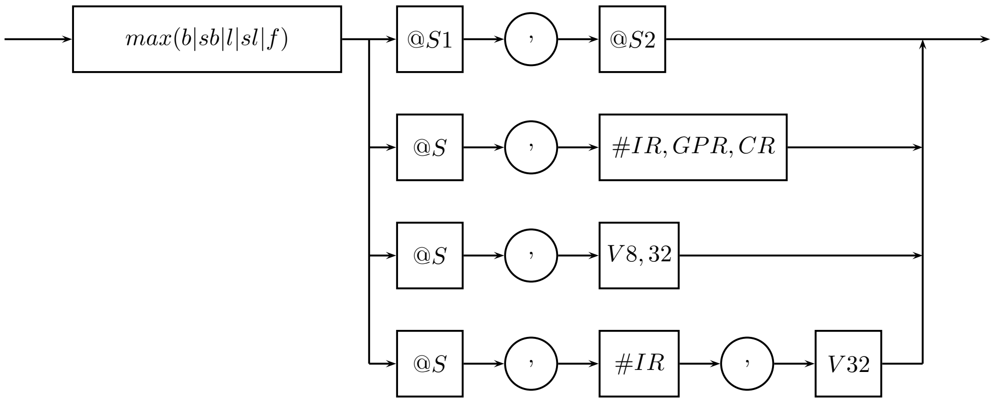


Figure 4.44: Syntax description of command *min*

Occurrence of the result: yes

Algorithm of work:

– implement operation $min(ARG1,\ ARG2)$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.30 mul (MULtiply)

Multiplication

mul ARG1, ARG2

Assignation: operation of two arguments multiplication

Syntax:

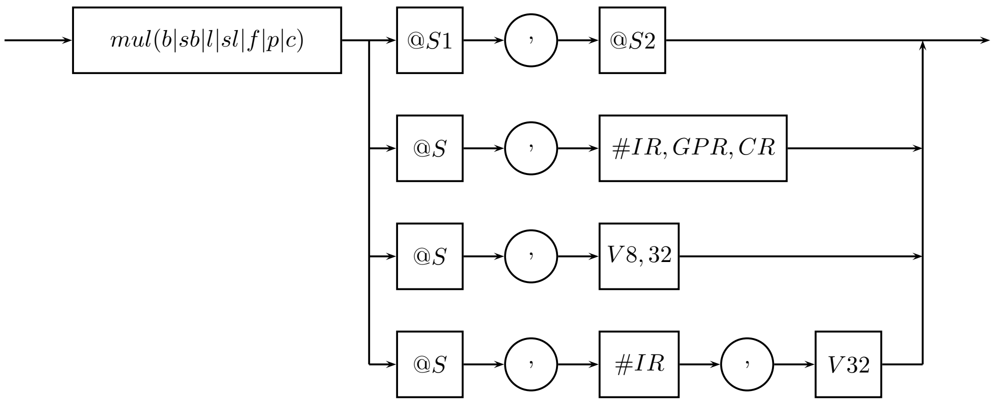


Figure 4.45: Syntax description of command *mul*

Occurrence of the result: yes

Algorithm of work:

- implement multiplication $ARG1 * ARG2$;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

Application: command mul is used for multiplication of two operands, which value is interpreted according to operattion type.

Example:

```
.data

B:
 4        .float\
 5            0f12.8, 0f-5.6,\
 6            0f-1.78, 0f0.19
 7
 8  .text
 9
10  A:
11        rdq B
12        rdq B + 8
13        mulc @1, @2
14        wrq @1, B + 16
15  complete
```

Explanation of an example:

- in the line No. 1 with an assembler command .data a current assembly section is set — which is the section of initialized data;
- in the line No. 3 a symbol (indicator) *B* is declared, which is an indicator in a current assembly section (data), and is initialized with a current assembly address value;
- in the line No. 4 with an assembler command .float into a current assembly section 4 32-bit real-valued numbers are set, beginning from current assembly address (a symbol of a reverse slash \ in the end of a line is used to continue a line, i.e. the lines No. 4, 5, 6 are logically a one line);
- in the line No. 8 with an assembler command .text a current assembly section is set — a section of executed instructions text;
- in the line No. 10 a symbol (indicator) *A* is declared, which is an indicator in a current assembly section (text), and is initialized with a current assembly address value, begins new paagraph;
- in the lines No. 11, 12 with commands rdq, two 64-positioning integer unsigned digits are being read from data memory by addresses *B* and *B*+8 and accordingly are placed into a commutator;
- in the line No. 13 with a command mulc multiplication operation of the execution result of a previous command is being calculated: @1 — the result of an execution of a reading command in the line No. 11, @2 — the result of an execution of a reading command in the line No. 12; both arguments of command

mulc in accordance with suffix c are interpreted as complex 64-bit numbers, that is why multiplication operation is performed in accordance with complex arithmetic rules; the result of a command execution is interpreted as a 64-positioning complex digit and is placed into a commutator;

– in the line No. 15 with a command complete a current paragraph is being completed.

4.4.4.31 norm (NORmalization)

Normalization

norm ARG

Assignation: calculation operation of displacements' reqired amount for fixed point number's normalization

Syntax:

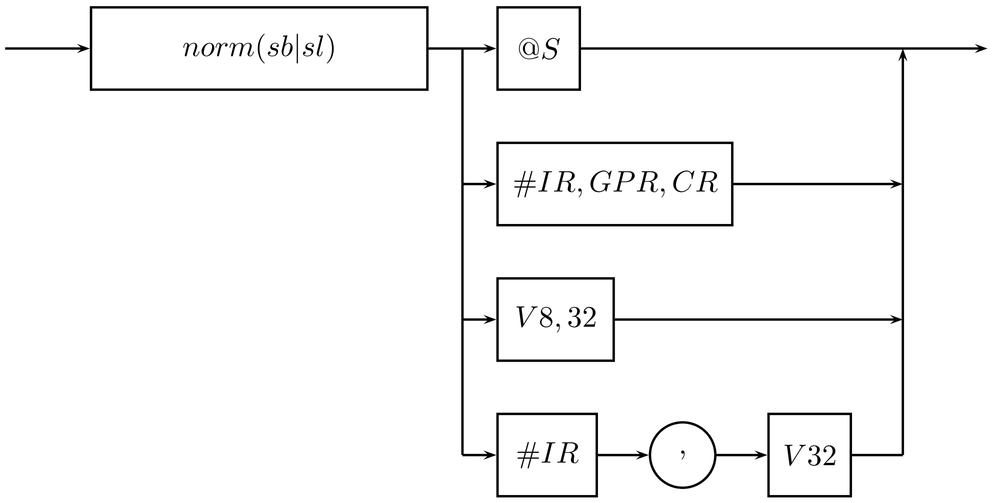


Figure 4.46: Syntax description of command *norm*

Occurrence of the result: yes

Algorithm of work:

– calculate required amount of displacements for fixed point number's normalization;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
0	0	0	r

4.4.4.32 not (NOT)

Logical negation

not ARG

Assignation: operation of arguments's logical negation

Syntax:

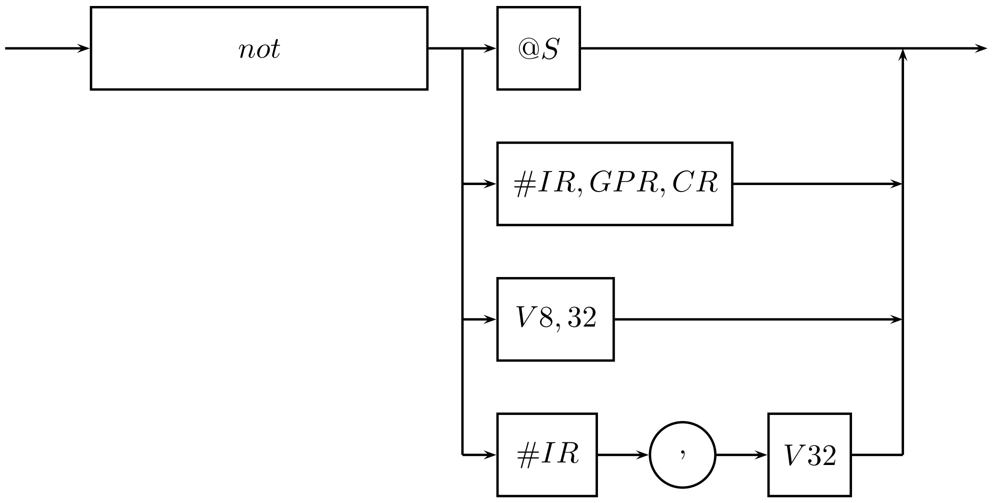


Figure 4.47: Syntax description of command *not*

Occurrence of the result: yes

Algorithm of work:

- implement logical negation $\sim ARG1$;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.33 or (OR)

Logical addition

or $ARG1,\ ARG2$

Assignation: operation of two arguments' logical addition

Syntax:

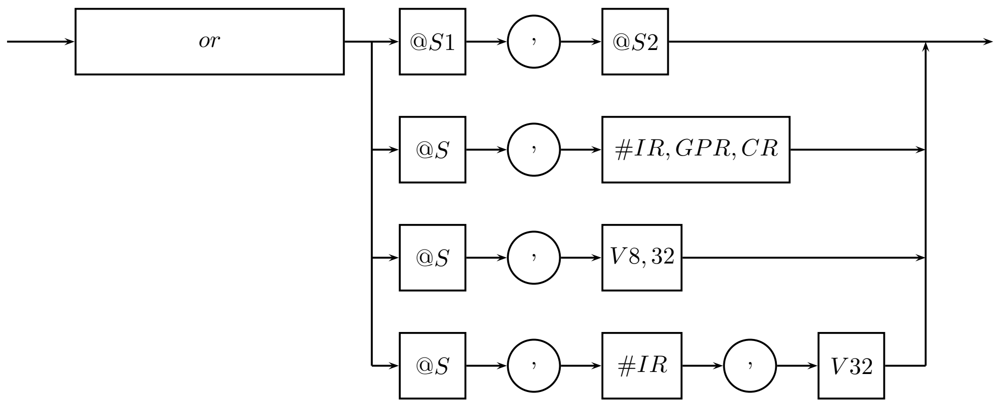


Figure 4.48: Syntax description of command *or*

Occurrence of the result: yes

Algorithm of work:

– implement logical addition $ARG1 \mid ARG2$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.34 pack (PACK)

Package

pack $ARG1,\ ARG2$

Assignation: operation of packed number formation

Syntax:

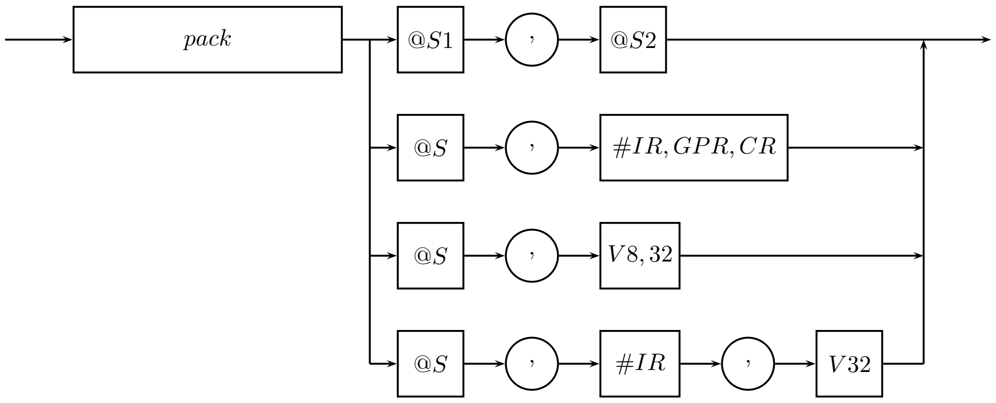


Figure 4.49: Syntax description of command *pack*

Occurrence of the result: yes

Algorithm of work:

– form 64-bit result value, highest 32 bit (from 32 to 63) of which are equal to lowest 32 bits (from 0 to 31) argument $ARG1$, and lowest 32 bits (from 0 to 31) — highest 32 bits (from 32 to 63) argument $ARG2$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.4.4.35 patch (PATCH)

Patching

patch ARG1, ARG2

Assignation: operation of patched number formation

Syntax:

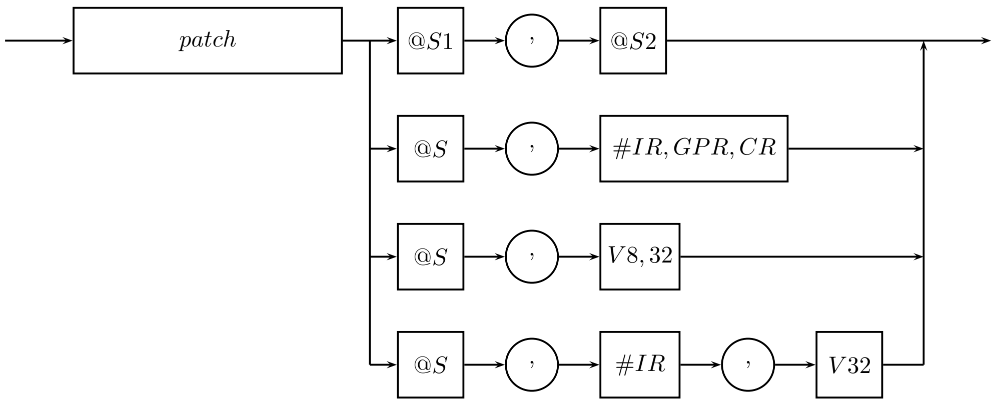


Figure 4.50: Syntax description of command *patch*

Occurrence of the result: yes

Algorithm of work:

– form result value, highest 32 bit (from 32 to 63) of which are equal to lowest 32 bits (from 0 to 31) argument $ARG1$, and lowest 32 bits (from 0 to 31) to lowest 32 bits (from 0 to 31) argument $ARG2$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	0	0	r

4.4.4.36 rd (ReaD)

Reading

rd ARG

Assignation: operation of data memory value reading

Syntax:

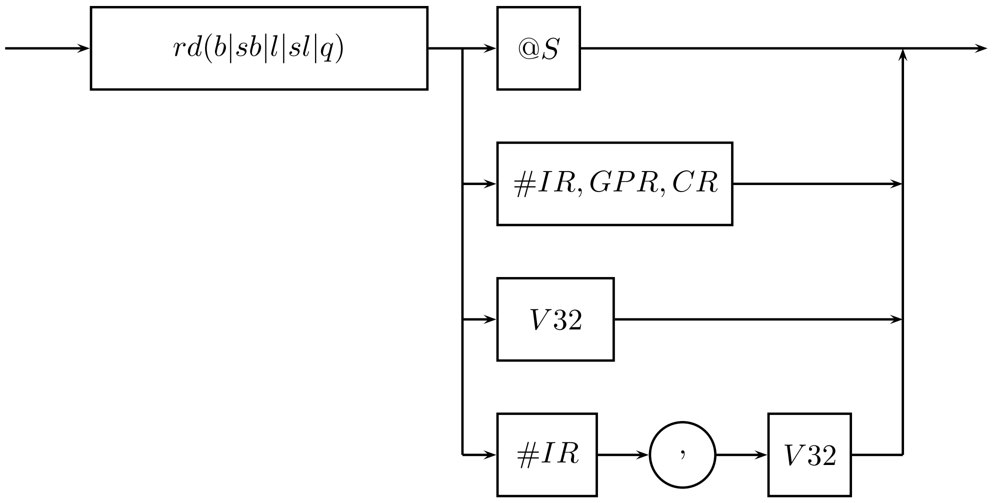


Figure 4.51: Syntax description of command *rd*

Occurrence of the result: yes

Algorithm of work:

- read depending on command type, 1 bite value (signed or unsigned), word (signed or unsigned), double word from data memory at the address set by argument *ARG*;
- in case of redin signed bite or word, multiply the sign;
- set indicators;
- place the result together with indicators into a commutator;

argument value interpretation: according to the work algorythm descried above, argument's value is always interpreted as data memory address, by means of which reding of a value is implemented, placed into commutator as a current command's result. In other words, given command always addresses to data memory despite the argument value's formation variant. If previous command's result is used to form argument's value, that is the link for comutator is used ($@S$) or values of general purpose register (#GPR) and control register (#CR), then highest 32 bits (from 32 to 63) are ignored.

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

Application: command rd is used for value reading, signed and unsigned, from data memory. During signed value reading multiplication of signed bit is implemented to reach the size of commutator. As far as possible given command should be found in the beginning of the paragraph.

Example:

```
 1  .data
 2
 3  A:
 4          .long  1,  2
 5
 6  .text
 7
 8  B:
 9          rdl  A
10          rdl  A + 4
11          addl  @1,  @2
12          wrl  @1,  A + 8
13  complete
```

Explanation of an example:

- in the line No. 1 with an assembler command .data a current assembly section is set — which is the section of initialized data;
- in the line No. 3 a symbol (indicator) A, is declared, which is an indicator in a current assembly section (data), and is initialized with a current assembly address value;
- in the line No. 4 with an assembler command .long into a current assembly section 32-bit digit 1 is set, followed by plus 4 byte 32-bit digit 2;

- in the line No. 6 with an assembler command .text a current assembly section is set — a section of executed instructions text;
- in the line No. 8 a symbol (indicator) B, which is an indicator in a current assembly section (text), and is initialized with a current assembly address value, and begins new paragraph;
- in the lines No. 9, 10 with commands rdl, two 32-positioning integer unsigned digits are being read from data memory by addresses A and $A+4$ and accordingly are placed into a commutator;
- in the line No. 11 with a command addl results of two previous commands are being added : @1 — the result of an execution of a reading command in the line No.10, @2 — the result of an execution of a reading command in the line No. 9; both arguments of command addl in accordance with suffix l are interpreted ungigned integer 32-bit numbers, that is why; the result of command implementation is interpreted as 32-bit unsigned integer digit and is being placed in commutator;
- in the line No. 12 with a command wrl recording into data memory is being performed at the address $A+8$ of the result of execution of a previous command: @1 — the result of execution of command in the line No11;
- in the line No. 13 with a command complete a current paragraph is being completed.

4.4.4.37 rol (ROtate Left)

argument's left cyclic rotation

rol ARG1, ARG2

Assignation: operation of argument's left cyclic rotation

Syntax:

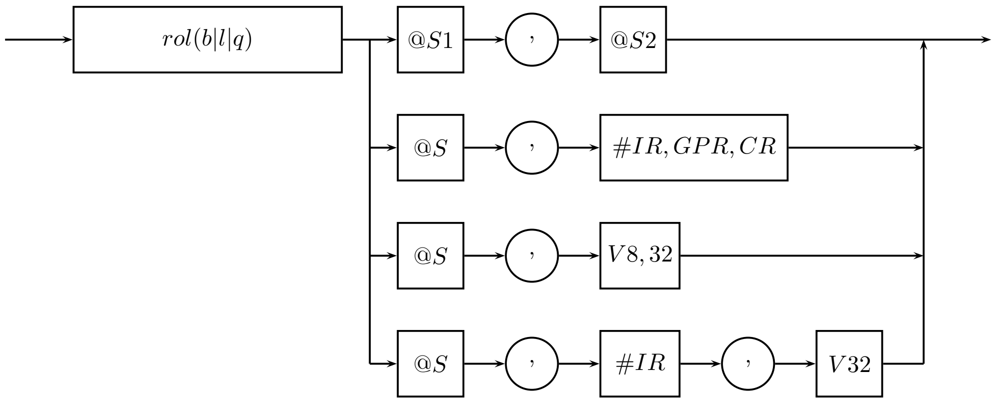


Figure 4.52: Syntax description of command *rol*

Occurrence of the result: yes

Algorithm of work:

– implement argument's cyclic rotation *ARG1* to the left for *ARG2* bit;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	r	r	r

4.4.4.38 ror (ROtate Right)

argument's right cyclic rotation

ror ARG1, ARG2

Assignation: operation of argument's right cyclic rotation

Syntax:

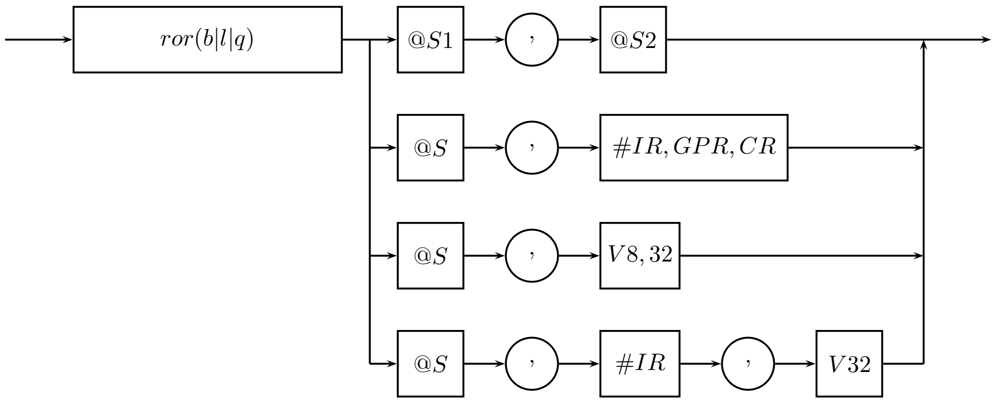


Figure 4.53: Syntax description of command *ror*

Occurrence of the result: yes

Algorithm of work:

– implement argument's cyclic rotation $ARG1$ to the right for $ARG2$ bit;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

4.4.4.39 sar (Shift Arithmetic Right)

right arithmetic shiftof the argument

$sar\ ARG1,\ ARG2$

Assignation: operation of argument's right shift

Syntax:

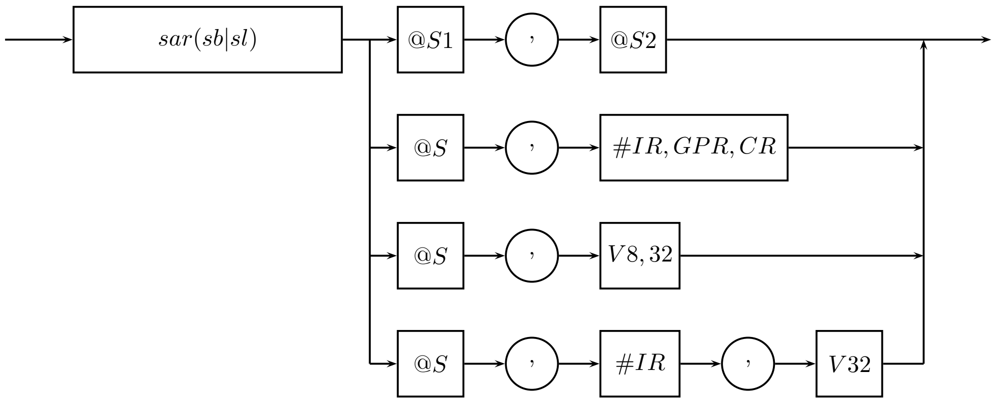


Figure 4.54: Syntax description of command *sar*

Occurrence of the result: yes

Algorithm of work:

- implement arithmetic shift of the argument $ARG1$ on the rigt to $ARG2$ bit;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

4.4.4.40 sbb (SuBtract with Barrow (Carry Flag))

Subtraction with borrow

sbb $ARG1,\ ARG2$

Assignation: operation of integer-valued subtraction with account of carry flag (CF) of the previous command's subtraction *sub*

Syntax:

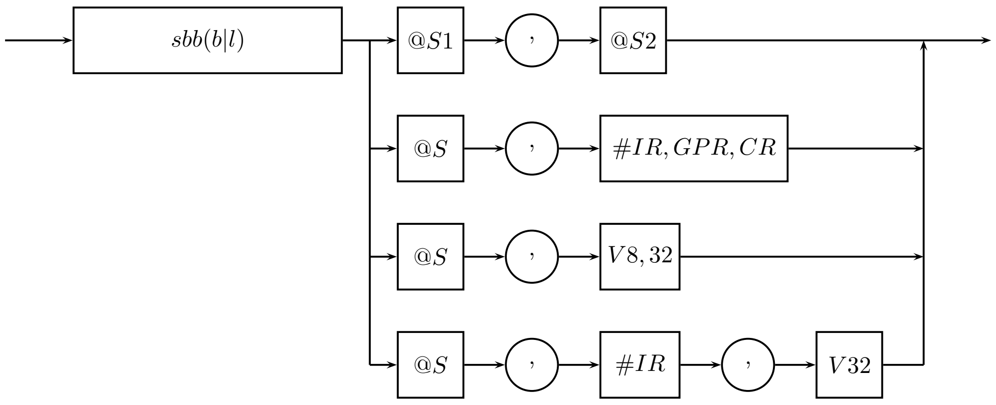


Figure 4.55: Syntax description of command *sbb*

Occurrence of the result: yes

Algorithm of work:

– implement subtraction from argument's value $ARG2$ carry flag value (CF) argument $ARG1$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

4.4.4.41 set (SET)

Installation

*set ARG*1, *ARG*2

Assignation: operation of register's value installation

Syntax:

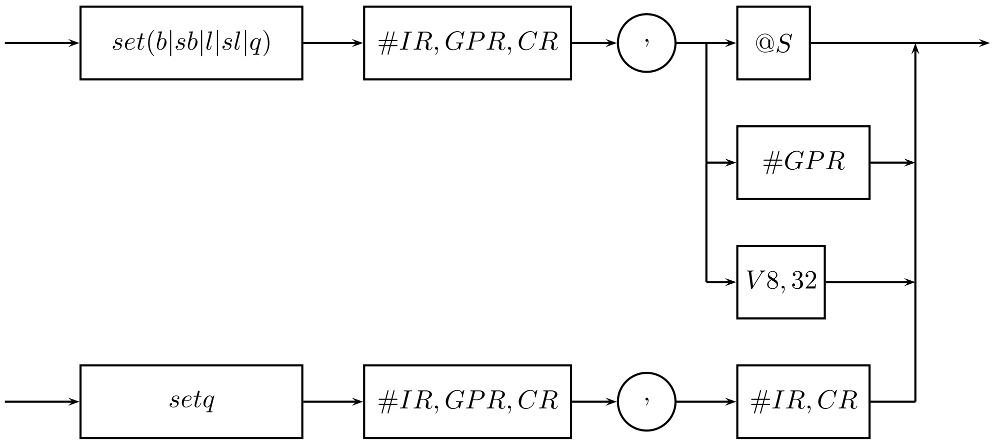


Figure 4.56: Syntax description of command *set*

Occurrence of the result: yes

Algorithm of work:

– install register's value , set by argument *ARG*1, equal to argument's value *ARG*2, size 1 bite (signed or unsigned), word (signed or unsigned), double word depending on command type;

Interpretation of argument' value: according to described above syntax and algorythm of work, first command argument determines refister's number and name(unlike most of other commands, which determines link number for the result of previous command), which value is to be set; second argument value, saved in pointed register, used directly (if the second argument contains the link to the index register, data memory access not occurs).

Application: command set is used to set 64-bit register value, equal to both signed and unsigned second argument's value. In case of signed command type (setsb, setsl) value of highest register bits are set equal to signed bit; for unsigned operation types (setb, setl, setq) values of highest register bits are set equal to zero.

Implementation peculiarities: actual set of register value is implem at the end of the paragraph. Accordingly, value of any register within the framework of one paragraph is to be set one time, using new register's value is possible only in the next paragraph. If one and the same register value is set several times, relevant warning will be given by the assembler, and register value at the end of the paragraph will be set into the value formed by the last **implemented** (command implementation is not ordered: command is implemented when its arguments are ready) installation command.

4.4.4.42 sll (Shift Logical Left) / sal (Shift Arithmetic Left)

Left shift of logical/arithmetic argument

$(sll|sal)\ ARG1,\ ARG2$

Assignation: operation of logical/arithmetic argument left shift

Syntax:

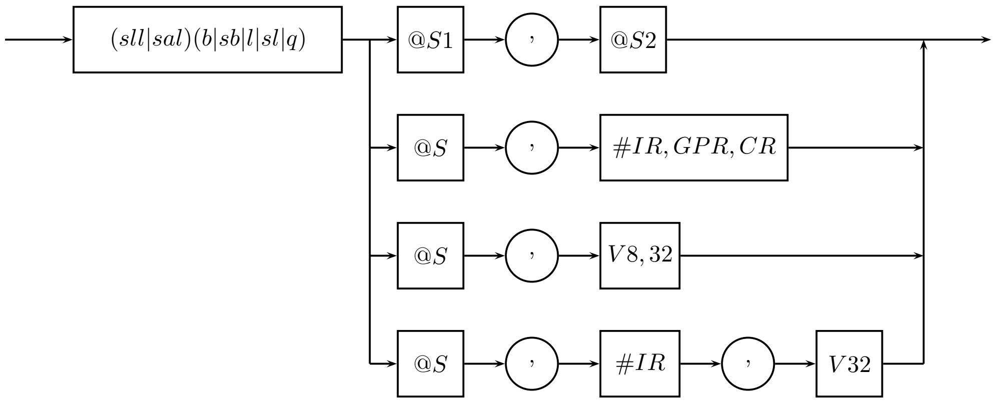


Figure 4.57: Syntax description of command *sll/sal*

Occurrence of the result: yes

Algorithm of work:

- implement logical/arithmetic shift of the argument $ARG1$ to the left at $ARG2$ bit;
- set indicators;
- place place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

4.4.4.43 slr (Shift Logical Right)

right shift of logical argument

slr ARG1, ARG2

Assignation: operation of logical argument right shift

Syntax:

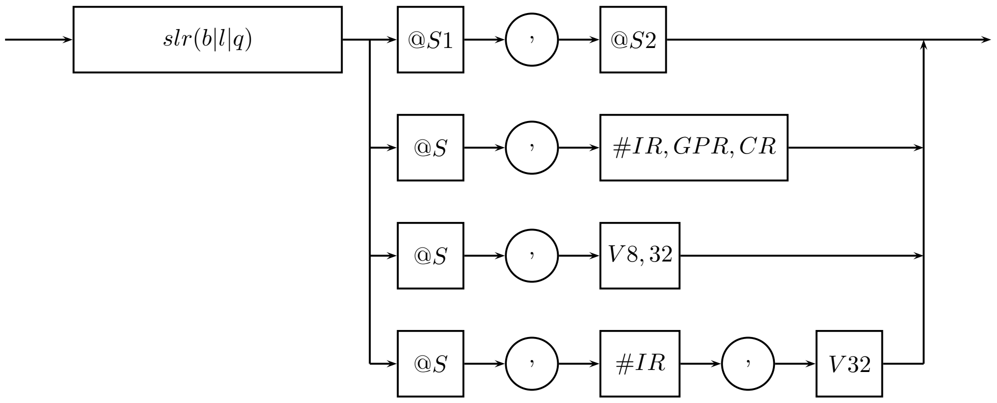


Figure 4.58: Syntax description of command *slr*

Occurrence of the result: yes

Algorithm of work:

- implement logical shift of the argument $ARG1$ to the right at $ARG2$ bit;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	r	r	r

4.4.4.44 sqrt (SQuare RooT)

Square root

sqrt ARG

Assignation: operation of real argument's square-rooting

Syntax:

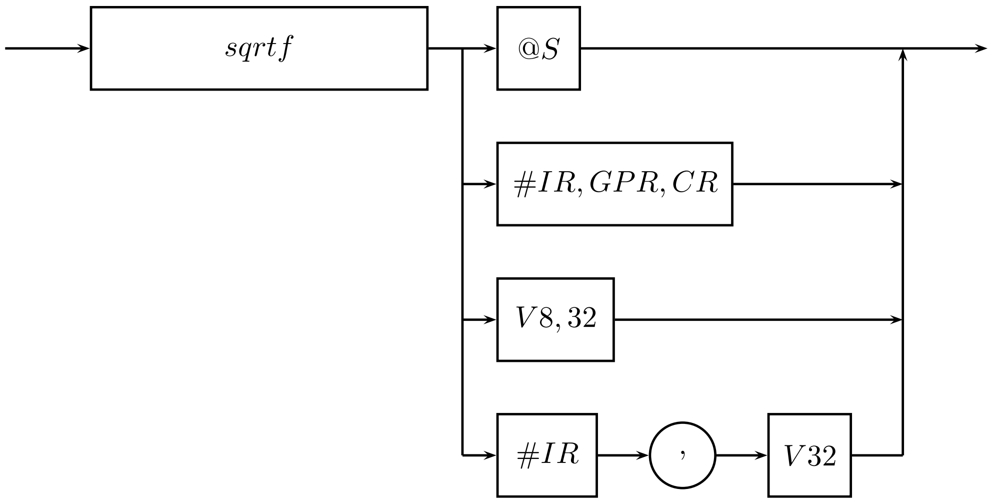


Figure 4.59: Syntax description of command *sqrt*

Occurrence of the result: yes

Algorithm of work:

– implement real argument's square-rooting $\sqrt{ARG}$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
0	0	0	r

4.4.4.45 sub (SUBtract)

Subtraction

$sub\ ARG1,\ ARG2$

Assignation: operation of two arguments' subtraction

Syntax:

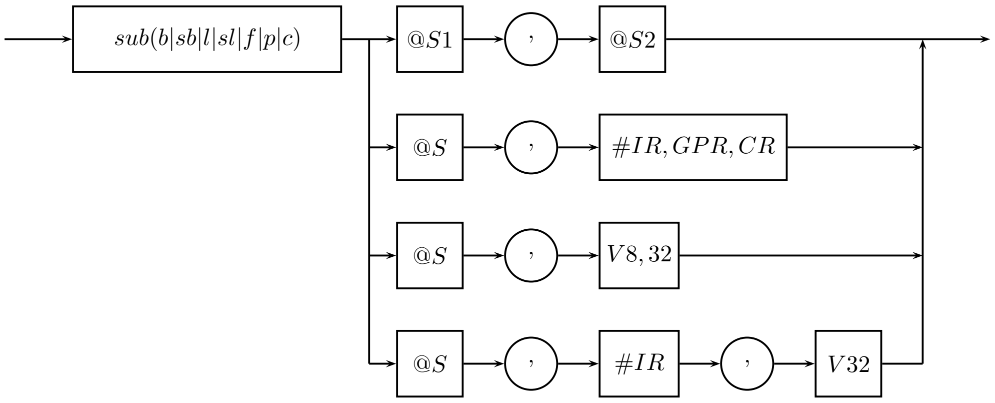


Figure 4.60: Syntax description of command *sub*

Occurrence of the result: yes

Algorithm of work:

– implement subtarction $ARG1 - ARG2$;

– set indicators;

– place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	*CF*	*OF*	*ZF*
r	r	r	r

Application: command sub is used for two operands' subtarction, which value is iterpreted in accordance with operation type.

4.4.4.46 wr (WRite)

Writing

wr ARG1, ARG2

Assignation: operation of value writing in data memory

Syntax:

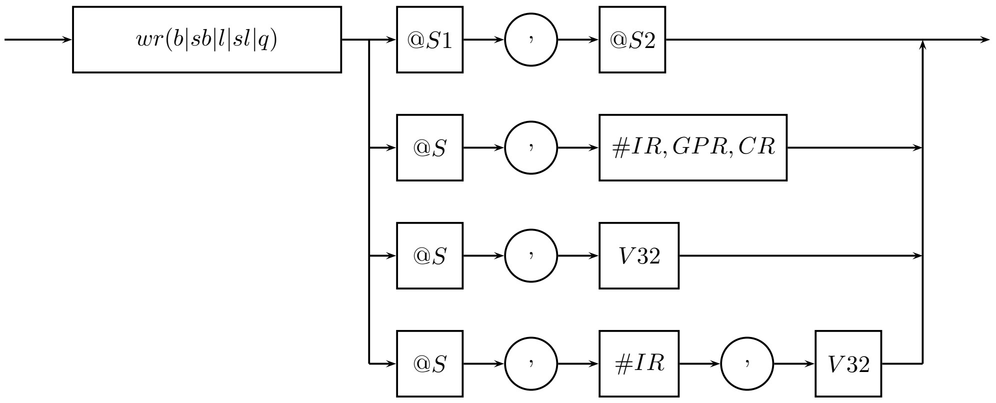


Figure 4.61: Syntax description of command *wr*

Occurrence of the result: yes

Algorithm of work:

– write depending on command type value size 1 byte (signed or unsigned), word (signed or unsigned), double word, set by argument *ARG1*, into data memory with the address set by argument *ARG2*;

Interpretation of arguments' values: in accordance with above described work algorythm, first argument's value is interpreted as it is (is directly used), second argument's value is always interpreted as data memory address, where first arguments' writing is implemented. In other words, given command always addresses to data memory inspite of formation type second argument value. If previous command's result is used to form argument's value, that is the link for comutator is used ($@S$) or values of general purpose register ($\#GPR$) and control register ($\#CR$), then highest 32 bits (from 32 to 63) are ignored.

Application: command wr is used for both signed and unsigned value writing into data memory. As far as possible given command is to be set at the end of the paragraph.

4.4.4.47 xor (XOR)

Logical addition

$xor\ ARG1,\ ARG2$

Assignation: operation of two argument's logical addition under mod2

Syntax:

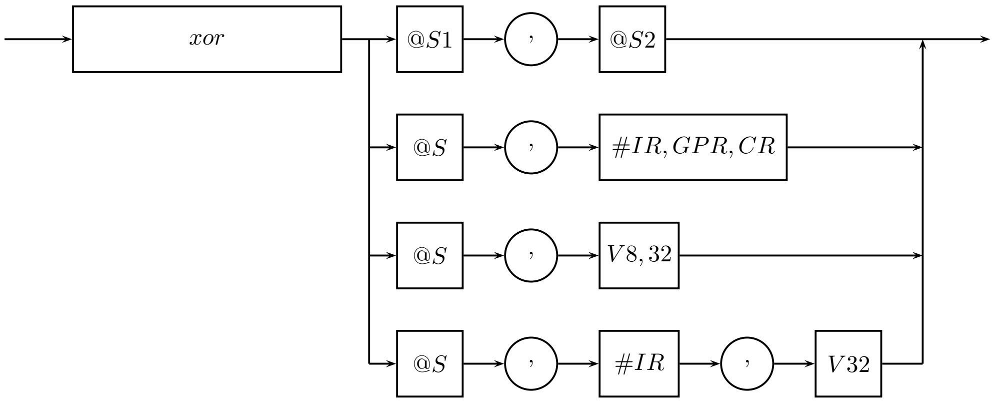


Figure 4.62: Syntax description of command *xor*

Occurrence of the result: yes

Algorithm of work:

- implement logical addition on mod2 $ARG1\ \hat{}ARG2$;
- set indicators;
- place the result together with indicators into a commutator;

Condition of result indicators after a command execution:

SF	CF	OF	ZF
r	0	0	r

4.5 Instruction system of assembler

All assembler commands have the names beginning with a dot symbol «.», which is followed by Latin symbols, in general, in lower case.

4.5.1 .alias name value

A command .alias serves for replacement of often used constants, key words, statements and expressions by some identifiers. Each alias shall be declared in one line prior to the first use. Alias value consists of any symbol set and begins with the first nonblank character, which follows the name, and ends either with the end of the line or with a symbol ';' or '//' (beginning of a single-line commentary). Redeclaration of alias value is admissible. This command replaces any further occurrences of identifier (name) to its value.

4.5.2 .align abs_expr, abs_expr, abs_expr

The command .align serves for increase of a current assembly address by a specified point by missing necessary number of addressing units.

The first expression, the result of calculation of which shall be an integer positive digit, sets necessary adjustment in bytes.

The second expression, the result of calculation of which shall be an integer positive digit, sets the filler value, which is used for initialization of ignored bytes. This expression (and a comma) may be ignored. In this case the value of a filler equals zero.

The third expression, the result of calculation of which shall be an integer positive digit, sets the value of a maximum number of bytes, which may be ignored by this command in order to achieve requested adjustment. If, in order to achieve requested adjustment it is necessary to ignore more bytes, than the specified maximum number, then ignore of bytes is not performed. In other words, perform adjustment only if the number of bytes is more or equals the specified maximum. This expression (and a comma) may be ignored. In this case the number of bytes necessary for adjustment will be ignored.

When using this command, ignore of filler (the second argument) value is also possible by means of specifying two commas after the first expression.

For example, '.align 8' will move the current assembly address forward up to the value multiple of 8. If the current assembly address is already multiple of 8, no action is performed.

4.5.3 .ascii “string” . . .

.ascii presupposes zero or more string literals separated with commas. This command may be located in sections .data or .bss. Each string without final zero byte is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. If a command is located in section .bss, only the value of the current assembly address changes, without actual data placement.

4.5.4 .asciiz “string” . . .

.asciiz presupposes zero or more string literals, separated with commas. This command may be located in sections .data or .bss. Each string without final zero byte is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. If a command is located in section .bss, only the value of the current assembly address changes, without actual data placement.

4.5.5 .bss subsection

.bss signals an assembler on the necessity to assemble all following instructions to the end of the subsection bss with the number of a subsection. A subsection value shall be an absolute expression. If a subsection value is not specified, the value 0 is presupposed.

4.5.6 .byte expressions

.byte presupposes zero or more string literals, separated by commas. This command may be located in sections .data or .bss. An 8-positioning digit is emitted for each expression. This digit in the process of performance is a result of expression calculation. The calculated 8-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. If a command is located in section .bss, only the value of the current assembly address changes, without actual data placement. If an expression value if not specified, the value 0 is presupposed.

4.5.7 .comm symbol, length, align

.comm defines a common (jointly used) with a number symbol (identifier). In the process of linkage a common symbol from one object file may be united with a specified or another common symbol from another object file with the same symbol name. If in the process of linkage no object file has a specified symbol with the name symbol, then for a symbol the length of bytes will be provided in an uninitialized area of data memory (section .bss), and also, if several common symbols having the same name are declared, the symbol with the maximum length value will be chosen. The length value shall be an absolute expression. The value "align" set the desired adjustment of a symbol and shall be an absolute expression.

4.5.8 .data subsection

.data signals an assembler on the necessity to assemble all following instructions to the end of the subsection "data" with the number of a subsection. A subsection value shall be an absolute expression. If a subsection value is not specified, the value 0 is presupposed.

4.5.9 .else

.else is a part of an assembly command .if, which provides sustention of conditional assembly code. .else marks the beginning of a code block, which is necessary to assemble in case if the condition of the preceding command .if/.elsief is false.

4.5.10 .elseif

.elseif is a part of an assembly command .if, which provides sustention of conditional assembly code. .elseif marks the beginning of a code block, which is necessary to assemble in case if the condition of the preceding command .if/.elsief is false, and a condition of this command is true.

4.5.11 .end

.end marks the end of an assembly file. Anything located under this command is ignored.

4.5.12 .endif

.endif is a part of an assembler command .if and marks the end of a code block, which is assembled conditionally.

4.5.13 .equ symbol, expression

.equ is a synonym of the command .set and sets the value symbol which is equal to the result of expression calculation, and changes the type attribute in accordance with a new value. Linkage attribute is not changed. With the help of this command it is possible to change a symbol value multiple times. The latest value will be saved in the table of object file symbols.

4.5.14 .equiv symbol, expression

.equiv sets the symbol value equal to the result of expression calculation, if a symbol was not defined earlier.

4.5.15 .eqv symbol, expression

.eqv sets the symbol value equal to the expression, if a symbol was not defined earlier. Calculation of the expression result is performed only by usage of a symbol.

4.5.16 .err

By assembly of this command an error message is output into the error output, and an object file is not generated. This command may be used for error signaling in a conditionally compilable code block (when using command of conditional compilation).

4.5.17 .error "string"

When assembling this command, an error message with the text "string" is output into the error output, and an object file is not generated. This command may be used for error signaling in a conditionally compilable code block (when using command of conditional compilation).

4.5.18 .fill repeat, size, value

Repeat, size and value are absolute expressions. For expression value, repeat of copies is emitted, each having the size of a byte. The size value shall be within the range of 0-8. If the size value is more than 8, a value of 8 is used as a size value. Size and value are optional. If the second comma and value are missing, value equals 0. If the first comma and following tokens are missing,the size equals 1.

4.5.19 .float flonums

.float is a synonym of a command .single and presupposes zero or more numbers with a floating point, separated with commas. This command may be located in sections .data and .bss. Each digit is a 32-positioning value and is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If a floatnum value is not specified, the value 0 is presupposed.

4.5.20 .global names, .globl names

.global sets linking attribute of each symbol, which are separated with commas, from the names list to the value «GLOBAL», as a result of which in the process of linkage the symbol will be viewable for all object files. If the symbol does not exist, it will be created.

This digit in the process of performance is a result of this expression calculation. The calculated 16-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.5.21 .if absolute_expression

.if provide sustention of conditional code assembly and marks the beginning of code block, which shall be assembled, if the result of calculation of expression absolute_expression is not equal to 0. The following variants of command .if are also sustained:

.ifdef symbol

assembles the next code block, if the symbol was not defined earlier.

.ifb text

assembles the next code block, if operand text is empty (not set).

.ifeq absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression is equal to 0. .

.ifeqs string1, string2

assembles the next code block, if the strings are similar (string1 == string2). Comparison of the strings is performed with regard to the case of string literals. The strings shall be in rabbit ears.

.ifge absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression is greater than or equal to zero.

.ifgt absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression is greater than zero.

.ifle absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression is less than or equal to zero.

.iflt absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression is less than zero.

.ifnb text

assembles the next code block, if operand text is not empty (set).

.ifndef symbol or **.ifnotdef symbol**

assembles the next code block, if the symbol was not defined earlier.

.ifne absolute_expression

assembles the next code block, if the result of expression calculation absolute_expression expression is not equal to 0.

.ifnes string1, string2

assembles the next code block, if the strings are different (string1 != string2). Comparison of the strings is performed with regard to the case of string literals. The strings shall be in rabbit ears.

4.5.22 .include “file”

..include provides insertion mode of auxiliary (additional) files in a certain position of a current file. A source code from a file is assembled in the way like it had followed in this file from position of command .include location. By the end of assembly of an auxiliary file the assembly of a current file continues. The ways of search of an auxiliary file may be specified using the option of a command line ’-I’.

sections .data or .bss. For each expression a 32-positioning digit is emitted. This digit in the process of performance is a result of this expression calculation. The calculated 32-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.5.23 .lcomm symbol, length

.Icomm reserves the length of bytes for a local common symbol in an uninitialized area of data memory (section .bss). Length shall be an absolute expression. An attribute of symbol linkage is set into the value «LOCAL», i.e. in the process of linkage the symbol will be invisible for other object files.

4.5.24 .local names

.local sets a linking attribute of each symbol, separated with commas, from names list into a value of «LOCAL», as a result of which these symbols will be invisible for other object files in the process of linkage. If the symbol does not exist, it will be created.

4.5.25 .long expressions

.long presupposes zero or more expressions, separated with commas. This command may be located in sections .data or .bss. For each expression a 32-positioning digit is emitted. This digit in the process of performance is a result of this expression calculation. The calculated 32-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.5.26 .p2align abs_expr, abs_expr, abs_expr

A command .p2align is aimed at increase of the current assembly address by the specified point, by means of ignore of the necessary number of addressing units.

The first expression, the result of calculation of which shall be an integer positive digit, sets the number of lower cases of assembly address, the values of which shall be zero after adjustment.

The second expression, the result of calculation of which shall be an integer positive digit, sets the filler value, which is used for initialization of ignored bytes. This expression (and a comma) may be ignored. In this case the value of the filler equals zero.

The third expression, the result of calculation of which shall be an integer positive digit, sets the value of a maximum number of bytes, which may be ignored by this command in order to achieve requested adjustment. If, in order to achieve requested adjustment it is necessary to ignore more bytes, than the specified maximum number, then ignore of bytes is not performed. In other words, perform adjustment only if the number of bytes is less or equals the specified maximum. This expression (and a comma) may be ignored. In this case the number of bytes necessary for adjustment will be ignored. When using this command, ignore of filler (the second argument) value is also possible by means of specifying two commas after the first expression.

For example, '.p2align 3' will move the current assembly address forward up to the value multiple of 8. If the current assembly address is already multiple of 8, no action is performed.

4.5.27 .print string

During assembly of this command an assembler will output the string into the standard output stream. A string shall be in rabbit ears.

4.5.28 .quad expressions

.quad presupposes zero or more expressions, separated by commas. This command may be located in sections .data or .bss. For each expression a 64-positioning digit is emitted. This digit in the process of performance is a result of this expression calculation. The calculated 64-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.5.29 .rept count

This command is used for organization of repetitions of instruction sequence. The instructions are located between this command (.rept) and a termination command .endr.

4.5.30 .set symbol, expression

.set is a synonym of a command .set and sets the symbol value which is equal to the result of expression calculation and changes a type attribute in accordance with a new value. A linking attribute is not changed. With the help of this command it is possible to change a symbol value multiple times. The latest value will be saved in the table of object file symbols.

4.5.31 .short expressions

sections .data or .bss. For each expression a 16-positioning digit is emitted. This digit in the process of performance is a result of this expression calculation. The calculated 16-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.5.32 .single flonums

.single is a synonym of a command .float and and presupposes zero or more expressions, separated with commas. This command may be located in sections .data or .bss. Each digit is a 32-positioning value and is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the floatnum value is not specified, the value 0 is presupposed.

4.5.33 .size name, expression

. size sets the size in bytes of a symbol name equal to the result of evaluating the expression.

4.5.34 .skip size, fill

.skip is a synonym of a command .size is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address, byte size, the value of which equals fill. Size and fill are absolute expressions. This command may be located in section .data or .bss. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If a comma and fill value isnot specified, the value 0 is presupposed.

4.5.35 .space size, fill

.space is a synonym of a command .skip and is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address, byte size, the value of which equals fill. Size and fill are absolute expressions. This command may be located in section .data or .bss. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If a comma and fill value isnot specified, the value 0 is presupposed.

4.5.36 .string “str”

.string presupposes zero or more string 8-bit literals, separated with commas. This command may be located in section .data or .bss. Each string literal and a final zero byte, are located sequentially in the initialized area of data memory (section .data) in 8 positions, starting from the current assembly address. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement.

The following variants of a command .string are also sustainable:

.string8 “str”

.string8 presupposes zero or more string 8-bit literals, separated with commas. This command may be located in section .data or .bss. Each string literal and a final zero byte, are located sequentially in the initialized area of data memory (section .data) in 8 positions, starting from the current assembly address. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement.

.string16 “str”

.string16 presupposes 0 or more string 8-bit literals, separated with commas. This command may be located in section .data or .bss. Each string literal and a final zero byte, are located sequentially in the initialized area of data memory (section .data) in 16 positions, starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement.

.string32 “str”

.string32 presupposes 0 or more string 8-bit literals, separated with commas. This command may be located in section .data or .bss. Each string literal and a final zero byte, are located sequentially in the initialized area of data memory (section .data) in 32 positions, starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement.

.string64 “str”

.string64 it presupposes 0 or more string 8-bit literals, separated with commas. This command may be located in section .data or .bss. Each string literal and a final zero byte, are located sequentially in the initialized area of data memory (section .data) in

64 positions, starting from the current assembly address. The order of bytes placement is little-endian. If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement.

4.5.37 .text subsection

.text signals an assembler on the necessity to assemble all following instructions into the end of subsection "text" with a number of subsection. Subsection value shall be an absolute expression. If a subsection value is not specified, a value of 0 is presupposed.

4.5.38 .type name, type

.type sets a type attribute of nem symbol which is equal to type value. Sustainable types (admissible type values):

STT_NOTYPE type of name symbol is not defined

STT_OBJECT type of name symbol is a data object (section .data, .bss)

STT_FUNC type of name symbol is a function (section .text)

STT_COMMON type of name symbol is a common data object (section .bss)

4.5.39 .warning “string”

When assembling this command a warning with the text "string" is output into the error output. This command may be used for error signaling in a conditionally compiled code block (when using the commands of conditional compilation)

4.5.40 .weak names

.weak sets a linking attribute of each symbol separated with commas from names into the value«WEAK», as a result of which in the process of linkage the symbol will be viewable for all object files. If the symbol does not exist. it will be created.

in section .data or .bss. For each expression a 32-positioning digit is emitted. This digit in the process of performance is a result of this expression calculation. The calculated 32-positioning digit is located sequentially in the initialized area of data memory (section .data), starting from the current assembly address. The order of bytes placement is little-endian.

If a command is located in a section .bss, only a value of the current assembly address is changed without actual data placement. If the expression value is not specified, the value 0 is presupposed.

4.6 Programming system in assembler for multicellular processor

The program in assembler is the source language with all files included, which is set at the input to the assembler.

Source language with all files included is called a translation unit. Including files into a source code is called assembler directive. include.

As a result of the compilation of such a translation unit assembler creates an object file, which is then possibly collected by a linker into an executable program with other object files.

The source language in assembler language consists of a sequence of instructions. Instruction in this case is processor command or assembler directive. Each statement must be placed in a separate line, that is, must end with a translation of a line or the beginning of a comment.

Separate program memory and the total memory are used in multicellular processor for each processor unit. data. To specify the memory area, into which the next instructions of source program text will be assembled , the following assembler directives are used:. data,. bss,. text.

Usage of directive .data in the source code of the program switches the current assembling section to the section data. In this section, the initial program data are stored, which in the process of program download are placed in data memory of processor. To initiate this section in the source code of the program the following assembler directives can be used:. ascii,. asciz,. byte,. float,. long,. short,. single,. string and its variants,. quad.

Usage of directive .bss in the program source code switches the current assembling section to the section bss. This section is used to backup the necessary size of uninitialized memory block of data, each byte of which is in the process of program download is initialized to zero. To select a block of uninitialized memory data in the source code of the program the following assembler directives can be used:.ascii,.asciz,.byte,.float, .long,.short,.single,.string and its variants,.quad without explicit initialization of value (if the initial value is specified, it will be ignored). To reserve a desired size uninitialized memory block without switching current assembling section the following assembler directives can be used:. lcomm and. comm.

Usage of directive .text in the source code of the program switches the current assembling section to the section text. In this section, the executable program instructions are stored, which are in the process of program download are located in the program memory of the processor. The distribution of executed instructions between processor units (Each processor block has its own program memory) is completed at the stage of program packaging by linker.

Tags can be installed in each of the above sections. Tag is defined as a character, followed by a colon «:». Tags are used as address of data memory or program memory. For example, tag set in section .data or. bss may be used in the read and write processor commands, and the tag set in the section .text — is used in next paragraph's address seek commands.

In addition, if the current assembling section is the section of executable instructions (.text), then the tag is interpreted as the beginning of the section. Tags are not allowed within a paragraph (if a tag is specified in a paragraph, it will be ignored). Each section ends with the command "complete". In other words, the section is macrocommand, all the instructions of which are to be followed; a paragraph may not be implented from the middle (not from first paragraph command)With no first-team section).

In general, the template version of the program in assembler can be represented as follows:

```
/*
  Given program is designed...
*/

/*
  Placement of the original data in the data memory
*/
.data; current assembly section

Da: // tag introduction
    /* data memory initialization command, for example */
    .long 1, 0xABCD
    .ascii "Some string"

Db: // tag introduction
    /* data memory initialization command */

/*
  Reservation of uninitialized data memory block of
     required size in bytes, without changing the current
     assembly section
*/
```

```
.lcomm Ba, 24

/*
  Reservation of uninitialized data memory block
*/
.bss; current assembly section

Bb: // tag introduction
    /* data memory initialization command, for example */
    .long ,,,
    .byte
    .skip 12

Bc: // tag introduction
    /* data memory initialization command */

/*
  Placement of the executed instructions in program memory
*/
.text; current assembly section

Ta: // tag and beginning of new paragraph introduction
    rdl Da
    rdl Da + 4
    mull @1, @2
    wrl @1, Bc
complete // paragraph completion

Tb: // tag and beginning of new paragraph introduction
    /* processor commands */
complete // paragraph completion
```

4.7 Interruptions and their processing

Interruption system of multicellular processor allows processing of 32 interruptions.

Number	Description
0	Internal non-maskable interruption (INMI)
1	External non-maskable interruption (ENMI)
2	Nonmaskable exception in hardware component (PERE)
3	Nonmaskable program exception (PPGE)
4	Maskable program exception (MPRGE)
5	System timer interruption (SWT)
6	Software interruption (SWI)
7	Maskable interruption from UART0
8	Maskable interruption from UART1
9	Maskable interruption from UART2
10	Maskable interruption from UART3
11	Maskable interruption from I2C0
12	Maskable interruption from I2C1
13	Maskable interruption from SPI0
14	Maskable interruption from SPI1
15	Maskable interruption from SPI2
16	Maskable interruption from I2S0
17	Maskable interruption from GPTIM0
18	Maskable interruption from GPTIM1
19	Maskable interruption from GPTIM2
20	Maskable interruption from GPTIM3
21	Maskable interruption from GPTIM4
22	Maskable interruption from GPTIM5
23	Maskable interruption from GPTIM6
24	Maskable interruption from PWM0
25	Maskable interruption from RTC
26	Maskable interruption from GPIOA
27	Maskable interruption from GPIOB
28	Maskable interruption from GPIOC
29	Maskable interruption from GPIOD
30	Maskable interruption from ETHERNET0

Continuation on the next page

Number	Description
31	Maskable interruption from USB0

Source with number «0» has the highest priority during interruption processing.

Nonmaskable interruptions:

Interruptions with number 0 — 3 are nonmaskable. Nonmaskable interruptions lead to immediate switch to interrupt handler. They can not be prohibited, they are permitted right after core's work start.

Maskable interruptions:

Interruptions with number 4 — 31 are maskable, they are globally permitted by bit ONIRQS in register PSW (4.1.3.4). Individual permission assigned by register MSKR (4.1.3.6).

The following registers are for functioning interruption system and interrupt handler: INTR (4.1.3.5), MSKR (4.1.3.6), ER (4.1.3.7), IRETADDR (4.1.3.8), IHOOKADDR (4.1.3.11), INTNUMR (4.1.3.12).

Registers INTR, MSKR, ER, INTNUMR refer to the control of interrupt controller. Registers IRETADDR, IHOOKADDR are used by program algorithm. In memory of multicellular processor fixed zone to place interrupt handler. Developer may place interrupt handler anywhere in address area of progra memory. In case of interruption the core will be trasmitted to the address written in register IHOOKADDR. At this addess developer may place initial interrupt handler, which can imlement dispatching and transmit the program to a needed address, where there is concrete interrupt handler. Return address is automatically saved in register IRETADDR, developer has a full access to it.

Interrupt hadling procedure:

In case of interruption, interrupt controller determines priority interrupt and forms a signal, which leads to installation of a relevant bit of register INTR. The whole program is devided into «paragraphs». At the time of «paragraph» execution all interruptions are prohibited, except nonmaskable, which may interrupt the work of processor core at any time. After «paragraph» completion, transmission to the intial address of interupt handler may occur, if there was an interrupt request, interruption is not maskable, (relevant bit of register MSKR is set) and there is no global interrupt prohibition (bit ONIRQS in register PSW). The address of the next «paragraph» is automatically written into register IRETADDR, into register INTNUMR — the number of requested interruption, and global interrupt prohibition is installed. The core is transmitted to algoriphm execution, placed at the address written in register IHOOKADDR. After the work of interrupt handlers is completed, algorithm should set global interrupt permission (bit ONIRQS in register PSW) if required and complete the exit from interruption.

5. User Manual on Link Editor

5.1 General information on a link editor

Builder of a link editor for a multicellular processor performs linkage of the executable image by means of uniting data of one or more object files and, if necessary, object files of static-link libraries, as well as binding of object file symbols so that all references to those symbols had correct addresses of execution time.

Object files present the result of compiling of a program source code by an assembler.

Static-link libraries present regular archives of object files, for creation of which a command ar is used.

5.2 Usage of a link editor

The above mentioned link editor starts from a command line with a command ***ld***. The arguments of this command are one or more object files. The following options are also available:

-l, --library=namespec — add a file of an archive namespec to the list of files, used for linkage. This option may be used manyfold. A link editor would search for a way to the library for a filename libnamespec.a. The search of an archive file is performed by a link editor only once, in the place where it is specified in a command line. If in an archive a symbol, which was undefined in a certain object file and this object file is specified in a command line prior to archive file, is being defined, a builder would add corresponding archive object files into a final image file. At the same time, occurrence of an undefined symbol in any of the object files, specified in a command line after an archive file, would not result in repeated search of a symbol in the earlier specified archive file. Any archive file may be specified in a command line several times.

-L, --library-path=searchdir — add a way searchdir to the list of the ways, where a link editor would search static libraries (archive files, specified by an option -l). This option may be used manyfold. The command are used in such an order as they are specified in a command line. All values of -L option are applied to all values of -I option, irrespective of the order in which these options were specified.

-e, --entry=entry_point — use a value of a symbol entry_point as a starting address of start of program execution. If the symbol entry_point was not found, an attempt of distribution of entry_point will be tried as well as an attempt to convert it into a digit. In case of success an attempt to use this digit as a starting address of program execution start will be tried.

-o, --output=FILE — place a rollout into an image file FILE. If this option is not used, the name of an output file will be by default — image.bin.

-M, --print-map — output into a standard output stream information on dislocation of object files data in memory and on values, specified by symbols.

-h, --help — show this message and exit.

6. User Manual on Loader

6.1 General information of a loader

Program loader for a multicellular processor performs loading of memory images of an executed program in ROM of a debug board.

A file of memory images of an executed program presents the result of linkage of a program by a link editor: a file of memory images and data memory of an executed program.

6.2 Loader usage

A program loader starts from a command line with a command ***ploader***, an argument of which is a file of memory images of an executed program. The following options are also available:

-l, --list — show the list of available ftdi devices.

-d, --device=deviceName — set the name ftdi of a device into deviceName, used for loading.

-f, --frequency=frequencyValue — set the frequency ftdi of a device into frequencyValue, used for loading (a value by default is 10000 khz).

-h, --help — show this message and exit.

Supposing that image.bin is a file of memory images of an executed program. In this case in order to load this image into ROM of a debug board it is necessary to execute the following command in a command line:

```
ploader image.bin
```

In this case the first found suitable ftdi device will be used to load the file image.bin into ROM of a debug board with frequency of 10000 khz.

The following command may be used to use a certain ftdi device with a specified frequency:

```
ploader image.bin -d"PicoTAP A" -f20000
```