PopAsm — The Popular Assembler User's Manual

Helcio B. de Mello

February 20, 2005

NOTE: This is but a draft of the *PopAsm* User's Manual. The contents of this document may change as needed, until the release of the first official full version. Comments on English mistakes or contents are welcome, given the author's native language is not English. Please see appendix C for contacting info.

About the author

Hélcio Mello was born in Brazil, in 1979. He has a Bachelor Degree in Computer Engineering from Federal University of Espirito Santo – ES – Brazil [1], a Master Degree in Computer Science in Pontificial Catholic University in Rio de Janeiro – RJ – Brazil [2]. He is currently taking the PhD course in Computer Science, which is expected to finish in 2008.

He joined *Sourceforge* [3] in December 2001, and then registered *PopAsm* as a *SourceForge* project. The project was finally open to the Free Software community. In July 2002, he joined *Advogato* [4, 5] and *FreshMeat*[6]

Acknowledgements

PopAsm **is the result of many months of work. Despite being the sole author of this project, I would like to thank everybody that somehow contributed to this project. Among those are the hundreds of people who have already visited *PopAsm* home page, talked about it with friends, mailed me for comments, and so on.**

I would also like to thank the ones who use *PopAsm* and trust it. I know I can always count on you for feedback about this project. *Sourceforge* support was crucial, as it hosts *PopAsm* CVS repository, home page, and so on. Thank you very much.

Special thanks go to my first university, UFES[1], where I learned the bulk of my computering skills, and to those classmates and teachers who believed *PopAsm* project, such as professor and PhD Sérgio Freitas[7], who decided to adopt *PopAsm* as the assembler for his graduate classes about assembly language.

Last, but not least, I would like to thank my family, friends and God for what I am now and what I have done so far.

Contents

List of Tables

List of Figures

Part I

Getting Started

The first chapters of this manual explain what *PopAsm* is, how to compile and install it, and how to run it, changing its default behavior as needed.

Chapter 1

Introduction

PopAsm **stands for "Popular Assembler". It is an assembler, that is, and assembly language compiler[1]. Its objective is to convert human readable code into machine instructions. These instructions will then be either executed as binary programs or linked with other modules (possibly written in other languages) to yield a computer program.**

Many assemblers already exist. Some are free and open source, others are not. Some will offer you features that others will not. Some will be suited for your needs, but others will not. *PopAsm* was designed to gather in a single assembler the best features of the existing assemblers, yet adding its own improvements and remaining compatible with existing code as well. As a result, most of your legacy code can be compiled under *PopAsm* without any modifications at all.

Besides the benefits discussed so far, *PopAsm* is a free open source project written in ANSI C++, which means that anyone can read its source code, modify, and compile it anywhere an ANSI C++ compiler is available. Its peculiar features make it suited for nearly any assembly programming project:

- **Huge numbers internal representation allows assembly-time expression evaluation in both integer and floating point format without any practical limit;**

[1]Technically speaking, assemblers are not compilers. They translate lines of code into machine language in a one-to-one basis. Compilers translate each line of code into several machine instructions. However, from this point on, this document will use both terms (compiler and assembler) indistinctly.

- **Smart default options make your code cleaner, without the redundancy demanded by some other assemblers;**
- **Top flexibility gives the developers the choice to use the infamous "red tape" present in some assemblers or just write the good old raw assembly code;**
- **Compatibility with existing code eases migration to *PopAsm* from other assemblers. There is no need to edit your code; *PopAsm* is compatible with *TASM* and *NASM*.**
- **And more...**

Due to the reasons discussed above, this assembler was named the "Popular Assembler". The next sections comment the two main assemblers *PopAsm* is compatible with.

1.1 Notes on *TASM*

***TASM* is not free software. It is also DOS/Windows only, and took some time to get the instructions of the new processors included. Such delay possibly contributed the creation of macros that assembled those missing instructions. Nevertheless, there seems to be lots of code written for *TASM* around.**

***PopAsm* is an attempt to offer the advantages discussed throughout this document without forcing people to rewrite their code by hand or throw it away. This means that *PopAsm* can be seen as a superset of *TASM*[2]. Such major feature may boost people to give *PopAsm* a try. If it is not of their liking, trying *PopAsm* did not change their legacy code anyway, so there's very little to lose.**

1.2 Notes on *NASM*

***NASM* is a free and portable assembler, and that's good. It looks like many people use *NASM* today, but there are several points where *NASM* can be questioned[3].**

[2] Actually, only the most used features are supported. Additional *TASM* features may be added as users request them.

[3] The objective of this section is to comment *NASM* under technical criteria, not to hurt anyone's feelings. All free software must be respected, regardless of its suitability to any project requirements.

Besides being incompatible with existing code, *NASM* does not provide (due its design) any typechecking feature for variables; the developer is obliged to remember the type of each variable himself. Some people may argue that this is not a big problem, but as a project grows in size, remembering each variable type may become nearly impossible.

It can be even worse when a team of developers is working on the same project. Unless they use a rigid (and redundant) naming scheme, things will get messy quickly. Using such assembler for small projects (say, less than 1,000 lines of code) may be an acceptable idea; doing so for a large project may lead to disaster. C++ or Java are strongly typed languages; just imagine how it would be if they did not perform type checking...

Another key point is the absence of good assembly-time arithmetics features, such as huge numbers arithmetics (both integer and floating point). *PopAsm* overcomes such problems as described in this document. Other points could be mentioned, but the purpose of this manual is solely to document *PopAsm* features, not to point out problems in other assemblers.

Chapter 2

Compiling and Installing *PopAsm*

This chapter explains how to compile and install *PopAsm* under Linux or other UNIX-like platforms and DOS/Windows. If you have already done that you may skip to the next chapter.

2.1 UNIX environments

This section describes compiling & installing info under UNIX-like operating systems (such as Linux).

2.1.1 Compiling *PopAsm* sources

In order to compile *PopAsm* sources, you will need an ANSI C++ compiler and its standard libraries[1]. You will also need *PopAsm* source code, which you should already have. If not, please go to *http://popasm.sourceforge.net/* and download it. You will get an archive containing the source code to be built.

Unpack it anywhere you like, using the appropriate software (e.g. if you downloaded a .tar.gz file, you should run *tar xvzf popasm-x.y.z.tar.gz*, where x, y and z are the version numbers of the package you got). At this point, a directory containing the source code will be created.

Now, you should enter the directory containing the sources. e.g. */tmp/popasm-0.0.1*. Please do not mistake it for something like

[1]For the curious, I use egcs 2.91.66

/tmp/popasm-0.0.1/src. You should be in ***PopAsm*** source code root directory, as in the example above.

PopAsm relies on ***Autoconf***[8] and ***Automake***[9] to probe your system for the necessary resources and generate the resulting ***Makefile***. This file will tell the ***Make***[10] utility how to compile ***PopAsm*** sources. First, type

```
./configure
```

to perform the necessary checks. If everything is ok, you should have a ***Makefile***. The next step is to compile the sources. Just type

```
make
```

and the ***Make*** utility will do the rest, but might take a few minutes. There should be no warnings and no errors. If you got any, please check whether your compiler is ANSI compliant. If it is, please let me know (contacting information can be found in appendix C).

2.1.2 Installing *PopAsm*

There is basically two ways of installing ***PopAsm***: either from the source code or from a RPM file.

Installing from Source Code

You should compile ***PopAsm*** as in section 2.1.1. Remain in the directory where you built ***PopAsm*** and issue the command (you will need to be ***root*** to do that):

```
make install
```

This will install the binary file, documentation, etc.

Installing from RPM files

The easiest way to install ***PopAsm*** is to use RPM files. Get one from ***http://popasm.sourceforge.net/*** if you have not done so yet. Login as ***root*** and type:

```
rpm -ivh popasm-x.y.z.rpm
```

replacing x, y and z for the appropriate version numbers. That's it, you're done.

2.2 DOS and Windows environments

To be written.

2.2.1 Compiling *PopAsm* sources

To be written.

2.2.2 Installing *PopAsm*

To be written.

Chapter 3

Running *PopAsm*

PopAsm **accepts a variety of options, which modify *PopAsm* default behavior in several ways. These include output formats and file names, generation of listings, compatibility options, etc.**

Such options can be passed to *PopAsm* via command line, environment variables, configuration files or any combination of the above methods.

3.1 Command line options

The simplest way to invoke popasm is

```
./popasm filename.asm
```

where `filename.asm`[1] is the source file to be assembled. Assuming the assembly process is successful, its output will be written to `filename.xxx` by default, where `xxx` is the extension for the output format being used (e.g. *bin* for plain binary format). See section 3.1.7 for details about output formats.

The next subsections describe the options currently accepted by *PopAsm* (in alphabetic order, capital letters first). They have been chosen as to be familiar to *NASM* users (that makes them case-sensitive as well).

[1] It is important to make clear that *PopAsm* does not require the source file to use the .asm extension; the developer may use whatever extension he wishes (or even no extension at all).

3.1.1 -@ — Using configuration file

This option makes *PopAsm* read additional options from the specified file. See section 3.3 for details. Example:

```
./popasm -@ my_proj.cnf src_file.asm
```

The command line above assembles `src_file.asm` according to the command line options loaded from `my_proj.cnf`.

3.1.2 -D — Predefining macros

This option can be used to define a macro at command line. This is often useful for conditional assembly. A value can optionally be assigned to the macro. Examples:

```
./popasm -DFOO=5 src_file.asm
./popasm -DFOO src_file.asm
```

The first line assembles the `src_file.asm` but it first defines `FOO` to be 5 (it is not necessary to issue the -D option before specifying the file name). The source code inside `src_file.asm` will then be able to use that macro.

The second line does the same, but `FOO` is defined to no initial value. Doing so can be useful if the code being assembled needs to check just whether or not a macro has been defined, regardless of its value.

TO DO: Mention %if and %ifdef here!!!

3.1.3 -E — Redirecting error messages to a file

By means of this option the user can redirect error messages to a file. See the -s option (subsection 3.1.13) if it is desired to send them to the standard output instead. Example:

```
./popasm -E src_err.txt src_file.asm
```

The errors generated by assemblying `src_file.asm` would be redirected to `src_err.txt` file instead.

3.1.4 -F — Choosing debug format

The -F option allows the user to select the desired debug info format. If the generation of debug info is not enabled this option will enable it automatically. If the debug format is not specified but the generation of debug info is enabled a default format will be used.

TO DO: Describe available formats and the default ones. TO DO: Place an example here.

3.1.5 -I — Setting include path

This option adds a directory to *PopAsm* include path. The include path is a list of directories where *PopAsm* searches include files for when the source code contains *INCLUDE* statements (see section A.2). If two or more directories are to be listed, the user should use the -I option multiple times (see the examples below).

Note that the current directory is always in the include path and it is where *PopAsm* will search the include files first. Moreover, the order which the -I options are issued *is* important (the include path is scanned from the last to the first directory)[2]. Examples:

```
./popasm -IC:\INCLUDE src_file.asm
./popasm -IC:\INCLUDE -I..\INCLUDE src_file.asm
./popasm -I. src_file.asm
```

The first line makes *PopAsm* add `C:\INCLUDE` to its include path and assemble `src_file.asm`. The second line does the same, except that *PopAsm* will also look for include files at `..\INCLUDE`. As already mentioned, in such case the `..\INCLUDE` directory would be examined first. In the third example the use of the -I option is redundant because *PopAsm* always check the current directory for include files.

3.1.6 -U — Undefining macros

By using this option the user can undefine a *previously* defined macro. It has no effect on macros defined within the source code to be assembled as well as on the ones defined at command line afterwards. This option is particularly useful for undefining macros

[2]Check section A.2.1 if you wonder why.

defined other ways, such as in the environment string (see section 3.2). Examples:

```
./popasm -DFOO -UFOO src_file.asm
./popasm -UFOO -DFOO src_file.asm
```

In the first example the -U option cancels the definition of `FOO`, but note how important the command line ordering is important, for in the second example the usage of -U has no effect (it cannot affect macros defined later in the command line).

3.1.7 -f — Choosing output format

This option selects the desired output format among the ones suported by *PopAsm* (see table 3.1). The default output file name will be the same as the source file, except that *PopAsm* will replace the extension for the one of the chosen output format. Example:

```
./popasm -f elf src_file.asm
```

The example above assembles `src_file.asm` to ELF32 format. The default output file name is `src_file.elf`.

If no output format is specified then *com* will be used by default. The only difference between *com* and *bin* formats are the usual checks good assemblers perform for `.COM` files (e.g. entry point at 100h).

Format	Extension	Description
aout	out	Linux a.out format
bin	bin	Raw binary output format
com	com	MS-DOS COM file format (default format)
elf	elf	ELF32 object file format
obj	obj	MS-DOS 16-bit object file format
win32	obj	Windows 32-bit object file format

Table 3.1: Suported output formats

3.1.8 -g — Enabling debug information

This option makes *PopAsm* include debugging information to the output file. The user should specify one of the available formats (see section 3.1.4), otherwise a default one will be used.

Example:

```
./popasm -g src_file.asm
```

3.1.9 -h — Getting help

Use this option to make *PopAsm* show a summarized help screen. It cannot be used with any other option.

3.1.10 -l — Generating listings

This option instructs *PopAsm* to generate a listing based on the assembling of the source file. A file name must be supplied after the option. *PopAsm* will overwrite the specified file if it already exists. Example:

```
./popasm -l src_file.lst src_file.asm
```

The command above assembles the file `src_file.asm` and generates a listing at `src_file.lst`.

3.1.11 -o — Setting output file name

This option tells *PopAsm* to send its output to the specified file. If no output file is given a default one will be used. The default output file has the same name of the source file, but the extension changes according to the table 3.1. If the source file has no extension *PopAsm* will append one to the output file. Example:

```
./popasm -f elf -o src_file.o src_file.asm
```

The example above assembles `src_file.asm` and outputs the resulting object code to `src_file.o`.

3.1.12 -r — Getting release information

When this option is used, *PopAsm* prints its release information (version number) in the screen. It cannot be used with any other option.

3.1.13 -s — Redirecting error messages to *stdout*

By default, *PopAsm* writes all error messages to *stderr* (the standard error output) which would make it impossible to redirect them (say, to a file). This option makes *PopAsm* write them to *stdout* (the standard output) so they can be redirected. Example:

```
./popasm -s src_file.asm > src_file.err
```

The example above assembles `src_file.asm` and redirect all messages (including the error messages) to `src_file.err`.

3.1.14 -w — Switching warnings on/off

This option enables/disables particular *PopAsm* warnings. In order to enable a warning, use `-w+warning_name`, where `warning_name` is the warning to be enabled (it must one of the warnings described in this section). Similarly, `-w-warning_name` disables a given warning. Examples:

```
./popasm src_file.asm -w+orphan-labels
```

That example turns the *orphan-labels* warning on and assembles `src_file.asm`. The next subsections list the suported warnings.

orphan-labels

When a label is left alone in a line without a trailing colon we have a so-called *orphan-label*. The main problem with an orphan-label is that it may actually be a mispelled command. For example, the *LODSB* instruction could be mistaken for "*LOADSB*", yielding an orphan-label instead of an instruction.

To avoid this possible headache developers are advised to always use a colon when defining a label (but not when defining data). Sections 4.1 and 11.5 also discuss these matters.

all

This special word refers to all suported warnings, including the ones that will be added in future versions of *PopAsm*. Because warnings may indicate actual errors, enabling all warnings is a good idea, despite not being the default configuration.

3.1.15 -y — Enabling special options

As already mentioned, compatibility with existing code is one of *PopAsm* main goals. Unfortunatelly, however, different assemblers may interprete identical commands differently, making it impossible for *PopAsm* to figure out the intended use for such commands.

In order to resolve the conflicts inherent in its compatibility endeavor, *PopAsm* uses the `-y` option followed by either a plus or minus sign (depending on whether a feature is to be enabled or disabled) and one of the suboptions described next[3].

For example, to run *PopAsm* making it case sensitive when dealing with user defined symbols and reverse non-byte strings, one would write:

```
./popasm src_file.asm -y+cs -y+rs
```

cs — Case Sensitive symbols

This suboption makes *PopAsm* distinguish uppercase and lowercase characters when dealing with user defined symbols. Note that *PopAsm* keywords such as registers and instructions remain unaffected. For example:

```
Next:           LODsw
                cmp     aX,Bx
                Jne     NEXT
```

If *PopAsm* is set to be case sensitive the code fragment above will not be able to be assembled because "`Next`" and "`NEXT`" would be treated as different names. Nevertheless all instructions and registers would be recognized properly.

By default *PopAsm* is case insensitive.

mn — Merge Namespaces

This suboption tells *PopAsm* whether or not it should merge all namespaces into a single one. See section 12.5 for details.

By default *PopAsm* does *not* merge namespaces.

[3]The only exception to this rule is the *sp* suboption

rs — Reverse non-byte Strings

Another point where *NASM* and *TASM* differ is the storing order of non-byte strings. For example, if someone writes

```
DW      'AB'
```

which character comes first (i.e. in the lowest memory address)? *TASM* would place 'A' first, while *NASM* would rather write 'B' to the lowest memory address (apparently reversing the string).

By default *PopAsm* behaves like *TASM*, but the *rs* suboption has been added to make *PopAsm NASM* compatible. In order to do so, simply enable this suboption (which is off by default) by adding "-y+rs" to the command line.

For more on this issue see section 11.1 and chapter 6.

wo — convert Weak memory to Offsets

As mentioned in section 10.2.1, *PopAsm* may interprete the name of a variable as either its contents or its offset. For example:

```
FOO             DW      1234h

                MOV     AX,FOO
                MOV     AX,OFFSET FOO
                MOV     AX,[FOO]
```

By default *PopAsm* interpretes the first *MOV* instruction exactly like the third one (i.e. the name of a variable refers to its contents), but it may be configured to act as in the second *MOV* instruction instead by means of this suboption.

Then, all one should do to make *PopAsm* behave like *NASM* regarding this point is to add "-y+wo" to the command line.

sp — Segment Prefix optimization

This is currently the only suboption that must not be preceeded by a plus or minus sign. Instead, the user should supply a number that specify how *PopAsm* will optimize segment prefixes. Allowed values are:

- **0 — Do not optimize at all. Segment prefixes are left as they are found, even if they are redundant.**

- **1 — Optimize redundant segment prefixes without exchanging the order registers appear inside brackets. For instance, "DS:[EAX]" will be optimized to "[EAX]" but "DS:[EBP+EAX]" will not be optimized to "[EAX+EBP]".**
- **2 — Like level 1, but the order registers are used inside brackets will be altered if such change allows for any optimization. Hence, "DS:[EBP+EAX]" *will* be optimized to "[EAX+EBP]". Performing such optimization generates no warnings.**

By default *PopAsm* uses maximum optimization level regarding segment prefixes[4], even those that may be added in a later version of *PopAsm*. Because of that, any code that relies on the presence of a segment prefix should use it in a separate line, as shown below:

```
DS:
MOV     AX,[BX]
```

3.2 Environment string

In order to be compatible with existing assemblers, *PopAsm* must suport lots of options as discussed so far. In order to make things easier to the user, *PopAsm* can also read command line options from the environment variable `POPASM_OPTIONS`. For example, if the user sets that variable to "-f win32" then *PopAsm* will generate its output as win32 object files. The following commands under Windows would do it:

```
C:\>set POPASM_OPTIONS=-f win32
C:\>popasm my_file.asm
```

The resulting file would be `my_file.obj`, instead of `my_file.com`[5]. Once the user decides which options to use to customize *PopAsm* he can set the `POPASM_OPTIONS` variable at the `AUTOEXEC.BAT` file or similar.

Whenever run, *PopAsm* checks whether or not `POPASM_OPTIONS` has been set. If so, its contents are *prepended* to the actual command line issued by the user, in order to be parsed first. Such

[4] see section 10.2.2 for a discussion about that

[5] Recall that the default output format is .COM

behavior allows the user to override any environment options as needed. For example, if POPASM_OPTIONS were set to "-f obj" and the command line contained "-f win32" the latter would prevail, that is, the output format would be *win32*.

3.3 Configuration files

Using the POPASM_OPTIONS environment variable helps a lot, but it has a major drawback: the same options would apply to all projects of a given user. Because different projects are likely to require different options ***PopAsm*** can also load them from individual files, using the -@ option (see section 3.1).

A configuration file is an ASCII file that holds command line options for ***PopAsm***. When the -@ option is used to load such a file, its contents are merged with the options specified from the environment variable and the command line itself. Options loaded from the configuration file override any others previously specified in the command line or in the environment variable POPASM_OPTIONS.

In short, ***PopAsm*** reads its environment variable first (if it has been defined), then the configuration file (if any) and finally the command line itself. If two or more options conflict (say, "-f elf" and "-f bin") the one specified last always prevails, in order to allow the user to override any previously specified options as needed.

Part II

PopAsm Syntax from the Beginning

This part documents *PopAsm* syntax, starting from the simplest concepts. Experienced assembly-language programmers are likely to feel compelled to skip the next few chapters, but are hereby advised not to do so before reading at least the their introductory paragraphs.

After presenting the basic syntax, its elements are described in great detail, each in a subsequent chapter. After reading this part, the average devoloper should be able to use the most common features of *PopAsm* easily.

Chapter 4

Basic Syntax

One of *PopAsm*'s major goals is compatibility with existing code. *PopAsm* syntax is very similar to the ones supported by *TASM* and *NASM*. This chapter discusses *PopAsm* basic syntax. The next chapters go into more detail in each syntax component (numbers, registers, etc.).

As stated in chapter 3, *PopAsm* takes one or more source files as inputs, each one containing assembly-language statements. These statements, in turn, have the following general form:

```
label:          CMD     arguments               ; comment
```

where:

- *label* is an identifier that marks an offset into the current memory segment, so it can be referenced elsewhere in the code. The colon (:) after the label is optional, but recommended, as discussed in section 4.1.
- *CMD* is either a x86 instruction (such as MOV, ADD, etc.) or a *PopAsm* internal command (ORG, DB, etc.). Command names are case-insensitive (MOV, mov and Mov are all accepted).
- *arguments* is a comma-separated list of arguments for the CMD command used in this line. An argument is any sort of *expression* (see chapter 10). Some commands neither require nor accept arguments.
- *comment* is whatever you want to write to make your code easier to read and maintain. Comments are placed after a semicolon, and *PopAsm* ignores them altogether.

For example:

```
MULTIPLY:       IMUL    CX,BX,5                 ; CX = BX * 5
```

is a valid statement. MULTIPLY is a label (followed by the optional colon), IMUL is a x86 instruction, and CX, BX and 5 are arguments for the IMUL instruction. The string following the semicolon is a comment, and is thus ignored.

None of the items above are mandatory, except that you cannot specify arguments for a command without issuing it first. So,

```
                MOV     AX,BX
```

is a valid statement, but

```
                AX,BX
```

is obviously not.

PopAsm imposes no tabbing restrictions; white spaces and tabs are ignored. This document uses a 1–17–25–49 tabbing scheme (that is, labels at column 1, commands at column 17, arguments at column 25 and comments at column 49).

4.1 Notes on labels

Valid labels are strings of letters (a–z or A–Z), digits (0–9) or any of the following characters:

_ @ $?

except that the first character cannot be a digit, otherwise the label would be treated as a number, as discussed in chapter 5. Also, note that it is not allowed to use ***PopAsm*** reserved words (such as command names, registers, etc.) as labels.

PopAsm is by default case-insensitive[1]. Such behavior may be changed by command line options and environment variables (see section 3.1.15). This document uses uppercase letters.

As already mentioned, the use of a colon after a label is optional, but recommended. For example, suppose you issue a LODSB command, but mistype it for LOADSB instead. As a result, the

[1]Due to compatibility with *TASM*.

mistyped command would be treated as an ordinary label, and thus not compiled properly. ***NASM*** calls these "orphan labels", and offers a command line option to warn you about that.

But let's check the code below:

```
I_LOVE_USING_LONG_LABELS
                LOADSB
                OR      AL,AL
                JNZ     I_LOVE_USING_LONG_LABELS
```

The first line is a rather long label, and the developer decided to place it alone in its line of code. The next line is the mistyped command. If the developer decides to enable the orphan-label warning, an annoying message will always bother him, saying the long label is orphan. Using the colon, this problem is solved.

Another good reason to use the optional colon is to have an additional mark to make it clear, to the ones reading your code, that the label is, indeed, a label, instead of, say, a macro name. Of course, this can also be achieved with an indentation pattern.

Special care should be taken when using a colon after a label that defines data. The colon should not appear in such circumstances. Please check chapter 11 for a detailed description of this issue.

Chapter 5

Numbers

PopAsm **accepts numbers in a variety of ways. A number is a sequence of as many digits as you want[1] or separators, in any order, except that:**

- **The first digit must be a 0–9 digit[2]**
- **All digits must be valid in the number base. For example, "3" is not a valid digit in binary notation.**

A separator is an underline (_) character that may appear anywhere among the number digits, except that it cannot be the first character. They are ignored by *PopAsm*, but help people read long numbers.

Examples:

- **12345678 is a valid number, because all of its digits are in the range 0–9**
- **12_345_678 is the same as above. Note how separators play the same role as a comma (that is, as a human being would write 12,345,678 to make it more readable)**
- **12_34_5_67___8 also works, and is *exactly* as the other two numbers above, despite looking weird and not being very useful.**

[1] There is no practical limit to number sizes in *PopAsm*. If you want to write a 1,000,000-digits long number, *PopAsm* will accept it gladly (well, if you have patience to type such number...). Some other assemblers are limited to 32-bit integer operations.

[2] Note that if the first digit is not decimal, *PopAsm* (and other assemblers) will treat the string as a symbol, not a number. For instance, AH is a register, but 0AH is a number (10 in hex notation).

- **_12_345_678 is not a number because the first character is not a digit.**
- **3241_7779q is not an acceptable octal number because "9" is not a valid digit in octal notation ("q" here stands for octal. Explained later).**

Typical uses for separators include separating binary fields of bit records. As an example, let's suppose that a simple communication protocol uses a byte to encode information about a transmission as follows:

Bits	Description
6–7	Package priority (0–3 range)
3–5	Transmitter's ID (0–7 range)
0–2	Receiver's ID (0–7 range)

Table 5.1: Encoding of a simple transmission protocol header

Thanks to the separators, the programmer can now write:

```
                ; p  snd recv
MOV     AL,11_000_010_B
```

That is, a priority $p = 11(binary) = 3$**, a sender whose ID is** $snd = 000(binary) = 0$ **and a receiver whose ID is** $recv = 010(binary) = 2$ **are clearly encoded within AL. More complicated examples may look messy without separators. The B suffix appended to that number stands for "binary", as discussed in the next section.**

5.1 Using other bases

As can be seen in the last example, *PopAsm* allows one to specify a number in many bases simply by appending a suffix. Bases and suffixes supported by *PopAsm* are:

There are two other ways of declaring a number as hex:

- **Using the 0x or 0X prefix, like in C language. If you use this prefix, the hex number that follows it does not need to begin with a decimal digit, that is, both 0xA and 0x0A are ok.**

[3]Note, however, that the first digit of an hex number *must* be in 0–9 range, as explained before.

Prefix	Base	Valid digits
b, B, y or Y	Binary (base 2)	0–1
o, O, q or Q	Octal (base 8)	0–7
d, D, t or T	Decimal (base 10)	0–9
h or H	Hex (base 16)	0–9, A–F and a–f[3]

Table 5.2: Prefixes and bases

- **Using the $ prefix, which may be dangerous. Mistaking -$70 (-70h) with $-70 (current offset minus 70) will surely lead to disaster.**

It is important to note that if you do not explicitly specify a number as binary, octal, decimal or hex, *PopAsm* will use the current radix value, which defaults to 10. That is, unless you specify otherwise, all numbers will be read in decimal notation. See section A.1 if you want to change the default radix.

5.2 Real numbers

Real numbers can be written in two forms:

- **The usual dot syntax, like in 1.23, 3.1416, etc. Note that you can use alternate bases to write real numbers as well. Eg.: 101.01B (5.25 in binary)**
- **Base and exponent, as in 6.02e-23 (Avogadro´s constant), 8.13e4 (8130), etc.**

Unlike some other assembers, *PopAsm* can perform assembly-time operations on both integer and real numbers. Real numbers have no storage limit either. *PopAsm* has also the advantage of neither rounding nor truncating real numbers. They have their exact values stored as long as possible.

Another key point is that *PopAsm* distinguishes between 0.0 and -0.0, because they have different encodings.

5.2.1 Special numbers

When converted to binary format, real numbers are encoded in IEEE format, which is the one used by x86 FPU's. However, some

bit patterns have special meaning, such as NaN's and infinities. The ability of defining such values is very useful if FPU programming is concerned.

In order to do so, ***PopAsm*** uses the folowing keywords:

- **INFINITY** — returns the IEEE encoding for $+\infty$. Because ***PopAsm*** supports arithmetics even on such special symbols, **-INFINITY** can be used to get $-\infty$ if needed.
- **QNAN n** — defines a "quiet NaN" that hold value n as its fraction. n must not be zero (see [11] for details).
- **SNAN n** — defines a "signaling NaN" that hold value n as its fraction. n must not be zero (see [11] for details).

Note that all of the above symbols, as well as all real numbers, are encoded differently, depending on the size of the variable that will hold them. For example, the 32-bit representation for $+\infty$ is obviously different from its 64-bit counterpart. ***PopAsm*** detects which size to use based on commands issued previously (eg. if an INFINITY keyword appears after a DQ command, then its 64-bit encoding will be used).

The next example shows how those keywords can be used:

```
DD      INFINITY                ;  7F80_0000h
DQ      -INFINITY               ; 0FFE0_0000_0000_0000h
DD      QNAN 5                  ;  7FC0_0005h
DQ      -SNAN 18                ; 0FFE0_0000_0000_0012h
DD      QNAN 0                  ; Error! QNaN being zero
```

Chapter 6

Strings

Another way of specifying numeric constants is to refer to the ASCII value of a character (or sequence of characters). This chapter shows how *PopAsm* translates quoted strings to such values.

6.1 General rules

A valid string must conform to a few general rules. First, it must be either single-quoted or double quoted (*PopAsm* treats them the same way). Whatever the choice is, the opening and closing quotes must match: 'abc' and "abc" are both the same, but neither 'abc" nor "abc' are accepted. Note that this rule allows one to quote the quote characters themselves! Yes, ' " ' and " ' " are accepted (the blank spaces have been added to make them easier to read).

An interesting example of use of that feature is to write "I don't like windows" or 'I said: "I hate windows".', but it is currently not possible to write 'I said: "I don't like windows".', because the single quote within "don't" would close the single quote that marks the start of the string.

Also, strings can only hold single line texts and they contain whatever is found between the opening and closing quotation marks, including tabs, commas and blank spaces.

6.2 Using strings as constants

As already mentioned, a quoted character is interpreted as its ASCII value. For example:

```
        MOV     AL,'a'
```

loads AL with 61h (which is the ASCII value of the lower-case "a").

Strings containing several characters are also allowed, and the resulting number will be 8-bits wide for each character within the string. For example, "hello" would be converted to a 40-bit number, and as such would not be able to fit EAX (a 32-bit register). Conversely, "ab" would be only 16-bit wide, hence it would fit any 16-bit destination (like DX) or larger ones (like EDX).

6.2.1 Compatibility issues

Unfortunatelly, the meaning of the line below is assembler-dependant:

```
        MOV     EBX,'abc'
```

NASM would translate 'abc' to 636261h, while ***TASM*** would interprete the same string as 61626300h. ***PopAsm*** default behavior is to act like ***NASM***, but for compatibility reasons there is an option to use ***TASM*** convention (see section 3.1.15).

The reason why ***PopAsm*** behaves like ***NASM*** is to allow fragments of code like the one below to be easier to read and write:

```
        CMP     DWORD [FOO],'ABCD'
        JE      THEY_ARE_EQUAL

FOO     DB      'A', 'B', 'C', D'
```

Note that if ***PopAsm*** behaved like ***TASM*** the developer would need to rewrite example above as

```
        CMP     DWORD [FOO],'DCBA'
        JE      THEY_ARE_EQUAL

FOO     DB      'A', 'B', 'C', D'
```

which is too awkward. See chapter 11 for more information on DB command.

Chapter 7

Operators

PopAsm **can perform several arithmetic operations on numbers and symbols. The operators currently supported, in increasing order of precedence, are listed in table 7.1.**

Operator precedences are discussed in section 10.1.

It is important to note that *PopAsm* works with both integer and real numbers. An operator returns an integer only if all of its arguments are integers. If at least one of them is real, the result will be real. All possible combinations are summarized in table 7.2.

The next sections provide additional details about each operator.

7.1 Size specifiers

Also known as size qualifiers, these are the BYTE, WORD, DWORD, etc. operators. They allow the developer to specify or override the size (in bits) of an expression. For example:

```
ADD     [BX],7                  ; Error! Undefined operand size.
ADD     BYTE [BX],7             ; Add 7 to the byte pointed to by BX
ADD     WORD [BX],7             ; Add 7 to the word pointed to by BX
```

The first line generates an error because *PopAsm* has no means to figure out whether the user wants to refer to the byte, word or dword pointed to by BX. There would be three different instructions depending on which size specifier were used.

[1] Due to compatibility with *NASM*.

The next example shows how the size specifiers can be used to override the size of a variable:

```
FAR_POINTER     DW      5678h, 1234h

                LDS     SI,DWORD [FAR_POINTER]  ; DS:SI = 1234h:5678h
```

As can be seen, the FAR_POINTER variable defines two words. Because ***PopAsm*** provides a type checking mechanism, the LDS instruction will check whether its second argument is a dword variable. In the example above, FAR_POINTER is defined as a word, and thus there would be a type mismatch. To tell ***PopAsm*** you are doing so intentionally, use the size specifier as in the example. This will instruct ***PopAsm*** to treat FAR_POINTER as a dword.

The reason why ***PopAsm*** behaves like that is to allow the developer to concentrate on his programming task instead of trying to remember sizes of variables. Of course, because ***PopAsm*** never forces people to write code in any particular way, anyone who likes specifying variable sizes are allowed to do so anytime.

Note that these size specifiers can be used to control the size of nearly *any* expressions, as shown below.

```
ADD     EDX,18                  ; Adds 18 to EDX
ADD     EDX,BYTE 18             ; Same as above
ADD     EDX,DWORD 18            ; 18 encoded as a dword
```

The first line can be assembled in two ways. One of them encodes 18 as a byte, while the second form encodes 18 as a dword. Because the first form is more efficient than the second one, and ***the developer did not explicitly choose any in particular***, ***PopAsm*** uses the former as default.

In fact, as a general rule, everytime the developer does not require a particular encoding for an instruction, ***PopAsm*** will use its most efficient form. That's because we understand that if someone needs an specific encoding, than this need should be emphasised by the size specifier. Such approach has the following advantages:

- The developer does not care about *which* encoding will be used, except under special circumstances where the instruction format is critical. Because such cases are very rare, the user can benefit from the efficiency without having to tell the assembler to generate the short encoding *everytime*.

- **The developer still has full control over his code, because the longer encoding can be specified, as in the third line of the example above. It's better to make people use the specifiers only in those particular cases instead of requiring them to do so everytime (since most people prefer the efficient form of the instructions).**

- **The code will not depend on command line options to be assembled correctly, since all critical instructions will be shielded from any such influence by the size specifiers.**

It's important to note that *PopAsm* will *never* replace an instruction for another one. For instance, `ADD EAX,1` will *never* be replaced by an `INC EAX` instruction.

Due to compatibility reasons, *PopAsm* will accept the qualifiers "BYTE PTR" as a sinonym for just "BYTE", "WORD PTR" as well as just "WORD" and so on.

Note that the size specifiers have a slightly different meaning than their *NASM* counterparts. If one assembles the last example using *NASM*, *even using the size specifiers* the encodings are not guaranteed to be size-specific. That's because *NASM* supports optimization levels in command line which may ignore such size specifiers altogether, unless the developer uses the "STRICT" keyword. Such keyword is useless in *PopAsm* because if someone cares about specifying the operand size manually that's because this someone *wants* that particular encoding, regardless of any optimization level (may be some critical instruction, for example). Due to compatibility reasons, however, the "STRICT" keyword is recognized, but ignored. Because of the stronger nature of *PopAsm* size specifiers there will be no problem in doing so.

7.2 + and -

These operators may be either unary or binary; their unary form have greater precedence. Unary minus change the sign of an expression. Unary plus performs no operation and exists for completeness.

7.3 AND, OR and XOR

Bitwise boolean operators. They can only be used with integer arguments (positive or negative). Because numbers have, in theory, infinite bits to their left (positive numbers may be left padded with zeroes and negative numbers can be sign-extended with ones), these operators are affected by the sign of their arguments. Table 7.3 shows an example of sign influence on bitwise operations. Tables 7.4–7.6 show all possible combinations of signs, boolean bitwise operations and the way they affect the resulting sign.

Due to compatibility with *NASM*, "&", "|" and "^" are also supported.

7.4 / and MOD

It is not possible to divide by or get the remainer by zero. Also, the MOD operator *always* return the result with the same sign of the first argument. For example, `-23 MOD 7 = -2` **and** `23 MOD -7 = 2`.

***NASM* operators //, % and %% are supported for compatibility reasons.**

7.5 SHL and SHR

These operators always perform binary shifts (i.e. they treat their arguments as unsigned numbers in two's complement notation). Do not mistake an SHL operator for the SHL instruction. In the examples below, the bits shifted in appear delimited by a separator:

```
MOV     AX,101B                 ; AX = 101B here
SHL     AX,3                    ; SHL instruction
; AX SHL 3 = 101B SHL 3 = 101_000B

MOV     AX,101B SHL 3           ; same as above

MOV     BL,-50 SHR 3            ; BL = 25
; 11001110 SHR 3 = 000_11001
```

NASM* counterparts for these operators (<< and >>) are supported for compatibility reasons. Note, however, that, like *NASM

and unlike ANSI C, such operators in *PopAsm* always perform binary shifts. If you need their arithmetic versions, see next section.

Another point to consider is that binary right shifting negative numbers require their two's complement value to be known. It is not possible to perform such calculation without fixing the number size in bits. For example:

```
MOV     AL,-1 SHR 1             ; AL = 127
MOV     AX,-1 SHR 1             ; AX = 32767
```

At first glance (without reading the comments), both instructions load the same value to their destination registers, but because -1 is encoded as $0FFh$ in 8 bits and as $0FFFFh$ in 16 bits, $-1\ SHR\ 1$ will yield different results. The second line can be rewritten as:

```
MOV     AX,BYTE -1 SHR 1        ; AX = 127 now
```

in order to get the same result for the first line. If the operand size is omitted, then *PopAsm* will get it from the context. As in the example before, because AX is a 16-bit register -1 is encoded in 16 bits as well.

7.6 SAL and SAR

These operators always perform arithmetic shifts. Do not mistake an SAR operator for the SAR instruction. In the examples below, the bits shifted in appear delimited by a separator:

```
MOV     AX,5                    ; AX = 5 = 101B here
SAL     AX,3                    ; SAL instruction
; AX SAL 3 = 101B SAL 3 = 101_000B

MOV     AX,5 SAL 3              ; same as above

MOV     BL,-50 SAR 3            ; BL = -7
; 11001110 SAR 3 = 111_11001
```

The cautious reader should note that SHL and SAL produce the same results. SAL is supported for completeness only. Note also that SAR does not depend on the operand size in bits, as SHR does. Repeating the example of the last section we have:

```
MOV     AL,-1 SAR 1             ; AL = -1 = 255
MOV     AX,-1 SAR 1             ; AX = -1 = 65535
```

7.7 NOT

Performs the ones complement operation (that is, it toggles all bits of its argument). `NOT` is both an instruction and an operator. For example:

```
MOV     AL,5                    ; AL = 5 = 101B here
NOT     AL                      ; NOT instruction
; AL = NOT 5 = NOT 101B = 11111_010B

MOV     AL,NOT 5                ; same as above
```

***NASM* counterpart for this operator, "~", is supported for compatibility reasons.**

Precedence	Operator	Description
	BYTE	8-bit qualifier
	WORD	16-bit qualifier
	DWORD	Double WORD — 32-bit qualifier
	PWORD	triPle WORD — 48-bit qualifier
	FWORD	Far WORD — Same as PWORD
	QWORD	Quad WORD — 64-bit qualifier
Lowest	OWORD	Oct WORD — 128-bit qualifier
	TBYTE	Ten-BYTE — 80-bit qualifier
	TWORD	Ten-WORD — Same as TBYTE[1]
	SHORT	8-bit relative displacement
	NEAR	16- or 32-bit relative displacement
	FAR	32- or 48-bit relative displacement
	OR or |	Inclusive or
	XOR or ^	Exclusive or
	AND or &	Boolean AND
	+	Addition
	- (binary)	Subtraction
	*	Multiplication
	/	Division
	MOD or %	Remainer (modulus)
	SHL or <<	Binary Shift left
	SHR or >>	Binary Shift right
	SAL and SAR	Arithmetic shifts
	NOT or ~	One's complement
	+ (unary)	Ignored
	- (unary)	Negation
	:	Segment and offset composition
Highest	.	Member selection (see chapter 12)

Table 7.1: Operators and their precedence

Argument 1	Argument 2	Result	Example
Integer	Integer	Integer	3 + 2 = 5
Integer	Real	Real	3 + 2.0 = 5.0
Real	Integer	Real	3.0 + 2 = 5.0
Real	Real	Real	3.0 + 2.0 = 5.0

Table 7.2: Argument types combinations

	Decimal	Binary
Argument 1	-75	$1\cdots110101$
Argument 2	3	$0\cdots000011$
AND Result	1	$0\cdots000001$
OR Result	-9	$1\cdots110111$
XOR Result	-10	$1\cdots110110$

Table 7.3: Example of bitwise boolean operations on signed numbers

	Positive	Negative
Positive	Positive	Positive
Negative	Positive	Negative

Table 7.4: Relation between AND operator and the sign of its arguments

	Positive	Negative
Positive	Positive	Negative
Negative	Negative	Negative

Table 7.5: Relation between OR operator and the sign of its arguments

	Positive	Negative
Positive	Positive	Negative
Negative	Negative	Positive

Table 7.6: Relation between XOR operator and the sign of its arguments

Chapter 8

Registers

PopAsm **supports all registers present in each x86 CPU, except the ones added in IA-64 architecture (which will be added in a later version):**

- **8-bit general-purpose registers: AL, BL, CL, DL, AH, BH, CH and DH**
- **16-bit general-purpose registers: AX, BX, CX, DX, SP, BP, SI and DI**
- **32-bit general-purpose registers: EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI**
- **Segment registers: CS, DS, ES, FS, GS and SS**
- **Control registers: CR0, CR2, CR3 and CR4**
- **Debug registers: DR0 thru DR7**
- **Test registers: TR3 thru TR7**
- **FPU registers: ST(0) thru ST(7). ST0 thru ST7 are also accepted for compatibility with *NASM*. ST is an alias for ST(0).**
- **MMX registers: MM0 thru MM7. MM is an alias for MM0.**
- **XMM registers: XMM0 thru XMM7. XMM is an alias for XMM0.**

Register names are case insensitive (i.e. AX, ax, aX and Ax are all the same thing).

Chapter 9

Constants

Constants can be used to make code easier to understand and maintain. There are two kinds of constants supported by *PopAsm*, which are discussed in the next sections.

9.1 Defining constant constants

Constant constants are assembly-time symbols that hold values that cannot be changed. This may sound redundant at first, because no constant could change its value anyway, so there would be no reason to say "constant constant".

This kind of constant can be defined by the EQU command. Its syntax is:

```
label           EQU     expression
```

where:

- *label* is the name of the constant to be defined. It *must not* be followed by a colon. Rules for specifying labels can be found in section 4.1.
- *expression* is *any* sort of expression. Expressions are explained in chapter 10.

For example:

```
LINE_FEED       EQU     0Ah
BIOS_SIGNATURE  EQU     0AA55h
MAX_RETRIES     EQU     3
```

The lines of code above define three constants: LINE_FEED (equal to 0Ah), BIOS_SIGNATURE (equal to 0AA55h) and MAX_RETRIES (equal to 3).

Besides the usual restrictions about defining labels, a constant constant is not allowed to be redefined. For example:

```
MAX_RETRIES     EQU     3                       ; 1st definition
MAX_RETRIES     EQU     4                       ; error! redefined constant
MAX_RETRIES     EQU     3                       ; error! redefined constant
```

As the third line of the example above shows, constant constants cannot be redefined even if the value of the new definition is the same as the previous one.

9.2 Defining mutable constants

Mutable constants are the ones that can be changed (redefined) in assembly-time. They can be defined (and redefined) by the = operator. Despite sounding paradoxal, this makes sense, as the next example shows:

```
MAX_RETRIES     =       3
; Normal code comes here

MAX_RETRIES     =       10                      ; Redefinition ok.
; Critical code! Must be allowed to retry more times!
```

That is, the same constant is being used in different parts of the code. Such freedom may lead to bugs if misused, though. If all constants are made mutable, one may accidentally redefine it thinking he would be defining a new one. Because constant constants cannot be redefined, you will get an error message if you forget about an existing constant and attempt to redefine it. In short: always use constant constants, unless you need to redefine it later.

9.3 Using constants

Once a constant is defined, it can be used the way anyone would expect, as shown below:

```
FOO             EQU     5

                MOV     AL,5                    ; AL = 5
                MOV     AL,FOO                  ; Same as above
```

Note that constants may hold expressions, not just numbers. Let's check the next example:

```
; Stack image of SI register after a PUSHA instruction
STACK_SI        EQU     WORD [BP+2]

                PUSHA                           ;
                MOV     STACK_SI,17h            ; SI = 17h
                POPA                            ;
```

As can be seen, the MOV instruction altered the contents of the SI register image in the stack, using the STACK_SI constant. When *PopAsm* finds the MOV instruction, it replaces the constant's contents and assemble the instruction normally.

Another key point is that, because *PopAsm* is a multi-pass assembler, users can reference constants that are defined later on. However, when dealing with variable constants, special care should be taken. For example:

```
                MOV     AL,FOO                  ; AL = 5 or 6?

FOO             =       5
                MOV     BL,FOO                  ; BL = 5

FOO             =       6
                MOV     CL,FOO                  ; CL = 6
```

That is, when referencing a variable constant that has not been defined yet, *PopAsm* uses the value of the first definition (in this case 5). So, in the first line above FOO equals to 5.

Chapter 10

Expressions

An expression may be either a single term or a sequence of terms connected by operators. A term, in turn, can be a number or a symbol. This section explains how *PopAsm* performs assembly-time arithmetics and works with expressions.

10.1 Numeric expressions

As mentioned in chapter 5, *PopAsm* internally stores numbers using as many bits as necessary to hold them. As a consequence, the expression evaluation in *PopAsm* benefits from this facility. *PopAsm* also delays number rounding and truncation as much as possible, avoiding precision losses and overflows.

For example:

```
MOV     AL,17014018740175480164581h MOD 100h
```

is a valid *PopAsm* statement, because the expression on the second argument of the MOV instruction evaluates to 81h, which lies in the 00h-FFh range, and thus can be written into AL. Note that the first term of the expression is clearly larger than the 32-bit maximum value, and could cause an overflow in other assemblers.

Numeric terms may be either numbers or numeric constants (see chapter 9). Also, operators execute in decreasing order of precedence. If two or more operators have the same precedence, they are executed from left to right. If necessary, parenthesis may be used to alter the evaluation sequence. There is no limit for parenthesis nesting. See table 7.1 for a list of operators and their precedences.

The code below shows some examples of expression evaluation.

```
FOO             EQU     -37
BAR             EQU      7
MINUS_ONE       EQU     -1

; MINUS_ONE is always equal to -1, but -1 is encoded as 0FFh, 0FFFFh, etc.
; depending on the operand size expected. That is, -1 = 0FFh for the DB
; command, -1 = 0FFFFh for the DW command and so on.
                DB      MINUS_ONE >> 4          ; 0Fh
                DW      MINUS_ONE >> 4          ; 0FFFh
                DD      MINUS_ONE >> 4          ; 0FFF_FFFFh
                DQ      MINUS_ONE >> 4          ; 0FFF_FFFF_FFFF_FFFFh

; Arithmetic shift examples. Note that only the final results are
; converted to two's complement.
                DB      FOO SAR 2               ; -10 = 0F6h
                DW      FOO SAR 2               ; -10 = 0FFF6h
                DD      FOO SAR 2               ; -10 = 0FFFF_FFF6h
                DQ      FOO SAR 2               ; -10 = 0FFFF_FFFF_FFFF_FFF6h

; Shows how parenthesis can be used to alter the evaluation sequence.
; Because AL must be in [-128, +127] or [0, 255], we have an error in
; the second line.
                MOV     AL,-FOO + 5 * BAR       ; AL = 6
                MOV     AL,(FOO + 5) * BAR      ; AL = -182 <- Error!

; Signed and unsigned division. -37 = 219 because BL is 8 bits wide.
                MOV     BL,FOO / 5              ; BL = -7 = 0F9h
                MOV     BL,FOO // 5             ; BL = 43

; Same as above, but -37 = 65499 because BX is 16 bits wide.
                MOV     BX,FOO / 5              ; BX = -7 = 0FFF9h
                MOV     BX,FOO // 5             ; BX = 13099

; Signed and unsigned remainer. -37 = 219 because BL is 8 bits wide.
; In the fourth line, -5 = 251
                MOV     BL,FOO % 5              ; BL = -2 = 0FEh
                MOV     BL,FOO %% 5             ; BL = 4
                MOV     BL,-FOO % -5            ; BL = 2
                MOV     BL,-FOO %% -5           ; BL = 219
```

10.1.1 String expressions

10.2 Memory references

Memory references can be specified using square brackets [ ] around an expression. For instance:

```
MOV     AX,[1234]
```

writes the word pointed to by 1234 into AX. In this case, DS is used as the default segment register. If you wanted to use another segment register (say, ES), you would write:

```
MOV     AX,ES:[1234]
MOV     AX,[ES:1234]            ; same as above
```

This latter form was added for compatibility with ***NASM*** and ***TASM*** ideal mode. This document uses the former syntax throughoutly, though.

10.2.1 Weak and strong memory references

Programmers usually need to store data in variables. Those variables can then be referenced by their names. For example, if VAR is a word variable, its contens can be copied into CX by the following command:

```
MOV     CX,VAR                  ; CX = VAR's contents
```

Unfortunately, different assemblers may give different meanings to the line of code above. ***TASM*** behaves as described here, but ***NASM*** would place the variable's offset into CX instead. Because the name of a variable may be treated both as its contents and its offset (depending on the assembler being used), such memory references will be called weak memory references in ***PopAsm*** documentation.

PopAsm behaves as described above (a variable's name means its contens, not its offset.) with respect to weak memory references. If you wish to refer to a variable's offset, you should use the ***OFFSET*** keyword. The example above would be rewritten as:

```
MOV     CX,OFFSET VAR           ; CX = VAR's offset
```

On the other hand, a strong memory reference is always enclosed within a pair of matching square brackets. Example:

```
MOV     CX,[VAR]                ; CX = VAR's contents
```

The reader is advised to avoid using weak memory references, for their meaning is assembler-dependant. Instead, whenever you refer to a variable's offset, you should use the *OFFSET* keyword, and whenever you want to reference the variable's contents, a strong memory reference is the best choice. Doing that way, your code will rely neither on *PopAsm*'s default behavior, nor the presence/absence of command-line options.

10.2.2 Additional memory references

PopAsm **has full support to all syntaxes for referencing memory in both 16- and 32-bit modes. For the next example,** `TABLE` **is an array of words:**

```
MOV     AX,[ES:TABLE+BX+SI+10]  ; like NASM
MOV     AX,ES:TABLE[BX+SI+10]   ; same as above
MOV     AX,ES:TABLE[BX][SI][10] ; same as above

MOV     AX,TABLE[4]             ; AX = 3rd word of the array
MOV     AX,[TABLE+4]            ; same as above
```

The first three lines perform the same action. The fourth line is accepted due to compatibility with *TASM*, despite looking confusing at first glance. The fifth line does the same as the fourth one, but looks better.

PopAsm **always encode memory displacements the most efficient way. Use size qualifiers if you need to change this behavior. For example:**

```
MOV     AX,[BX+5]               ; Efficient encoding
MOV     AX,[BX+BYTE 5]          ; same as above
MOV     AX,[BYTE BX+5]          ; same as above

MOV     AX,[BX+WORD 5]          ; Forces 16-bit displacement
MOV     AX,[WORD BX+5]          ; same as above
```

The first line encodes the displacement (5) as a byte by default. The next line does the same thing, as third one, supported because of compatibility with *NASM*. The next two lines force the displacement being encoded as a word.

It is also possible to force a null displacement being encoded as a byte or word. See the next example:

```
MOV     AX,[BX]                 ; No displacement

MOV     AX,[BX+BYTE 0]          ; Forces a null 8-bit displacement
MOV     AX,[BYTE BX]            ; same as above

MOV     AX,[BX+WORD 0]          ; Forces a null 16-bit displacement
MOV     AX,[WORD BX]            ; same as above

INC     BYTE [WORD BX]          ; Increments the byte pointed to by BX
INC     BYTE [BX+WORD 0]        ; same as above
INC     WORD [BYTE BX]          ; Increments the word pointed to by BX
INC     WORD [BX+BYTE 0]        ; same as above
```

The first line has no displacement; the second one forces a zero dummy displacement being encoded as a byte. The third line does the same, despite looking weird (supported due to compatibility with *NASM*). The fourth and fifth lines use a null 16-bit dummy displacement. In 32-bit mode, the displacement may be encoded as either a byte or a dword.

The last four lines are there as a warning: one must be careful when using *NASM* syntax for dummy displacements. The size qualifier inside the square brackets specify the size of the displacement, not the size of the variable being referenced.

The next example shows some memory access in 32-bit mode:

```
MOV     AX,TABLE[ECX*2]         ; Encoded as TABLE[ECX+ECX]
MOV     AX,TABLE[NOSPLIT ECX*2] ; Encoded as TABLE

MOV     AX,TABLE[ECX+EDX]       ; ECX is base
MOV     AX,TABLE[EDX+ECX]       ; EDX is base

MOV     AX,DS:TABLE[EBP+ECX]    ; Optimizeable
MOV     AX,SS:TABLE[ECX+EBP]    ;
```

```
DS:             MOV     AX,TABLE[EBP+ECX]       ; Non-optimizeable

                DS:                             ; Same as above
                MOV     AX,TABLE[EBP+ECX]       ;
```

The memory reference of the first line is encoded as `TABLE[ECX+ECX]`. Such behavior can be avoided using the *NOSPLIT* keyword, borrowed from *NASM*, as in the second line.

The third and fourth lines look equal, but there is a slight difference betwwen them: in the third line, ECX is the base register and EDX is the index one. In the fourth line, the role of ECX and EDX are exchanged. Whenever possible, the first register will always be encoded as a base register and the second one as the index register within the SIB.

The next two lines illustrate an exception to that rule. If EBP is used as the base register, the default segment register for that memory access would be SS. Conversely, if ECX is used as the base register, DS will be the default segment register, allowing the segment prefix in the fifth line to be skipped. Likewise, in the sixth line, exchanging the order of the registers inside the brackets will make SS the default segment register, turning the explicit segment override prefix redundant. By default, *PopAsm* performs such optimizations automatically, but they can be disabled as shown in Chapter 3.

It is more common, however, to disable such optimization for a single critical line of code, as in the seventh line, where the segment prefix appears to be a label[1]. Such syntax forces *PopAsm* to write the segment prefix and to leave the registers inside the brackets in their given order. The eigth and ninth lines show an alternative way of doing so.

[1] As with labels, the colon is optionsl.

Chapter 11

Defining data

This chapter shows how to allocate static storage for variables and, if desirable, give them an initial value. The general syntax for definig data is:

```
label:          DEF_CMD expression_list
```

where:

- `label` **is the name of the variable to be defined. It is optional, as well as the colon following it.**
- `DEF_CMD` **is a definition command (see below).**
- `expression_list` **is a comma-separated list of numeric expressions[1]. Such expressions determine the initial value of the variable. If the developer wants to define uninitialized data, the special value "?" may be used instead.**

The definition command may be any of the following:

- **DB — Define Byte: defines 8-bits integer variables.**
- **DW — Define Word: defines 16-bits[2] integer variables.**
- **DD — Define Double word: defines 32-bits integer or floating point variables.**

[1]Recall that a quoted string is ultimatelly treated as a number (based on the ASCII code of its characters), as mentioned in chapter 6. Note also that such a string must fit into the desired variable size. For example, 32-bit string constants must be up to 4 characters.

[2]Regardless of whether the assembler is in 16- or 32-bits mode

- **DP — Define triPle word: defines 48-bit integer variables, often used to store a 48-bits far pointer (16 bits due to segment; 32 bits due offset).**
- **DF — Define Far word: same as DP. Included in *PopAsm* for compatibility with *TASM*.**
- **DQ — Define Quadruple word: defines 64-bits integer or floating point variables.**
- **DT — Define Ten bytes: defines a 80-bits floating point variable or an 80-bits integer packed BCD variable.**
- **DO — Define Octuple word: defines 128-bits XMM data.**

For example:

```
FOO             DB      1, 2, 3
                DW      123 + 456, 1234h, ?, -175
NEGATIVE_ZERO   DD      -0.0
ENTRY_POINT     DD      07C0h:0000h
FAR_ADDRESS     DP      1234h:56789ABCh
HUGE_VALUE      DQ      INFINITY
```

The first line defines FOO as a byte variable. It allocates three bytes in a row and assign them the 1, 2 and 3 initial values. The second line defines no variable at all, but allocates four words just after the last byte defined in the first line. Note that the third word is undefined and that any numeric expression can be accepted as initial value. The next line defines a double word (32 bits) variable named NEGATIVE_ZERO, initialized to the IEEE representation for -0.0. The fourth and fifth lines define a double and a triple word , but they store a full pointer instead of a sinple integer. The last line encodes the IEEE representation for $+\infty$ (see Section 5.2.1).

11.1 Defining strings

Most applications display messages to their users. Those messages are usually defined by means of *DB* command, as in the code fragment below[3]:

[3]The null byte after the string is *not* placed by *PopAsm* automatically; the developer must do it himself if desired.

```
        DB      'Hello, world!', 0
```

As mentioned in chapter 6, a string may be interpreted differently depending on whether *PopAsm* behaves like *NASM* or *TASM*. However, the *DB* command is free of such influence. Hence the two lines of code below have the same effect regardless of any other factors.

```
        DB      'ab'
        DB      'a', 'b'                ; same as above
```

11.2 DUP expressions

Sometimes it is necessary to allocate large amounts of memory. There is a special operator for that purpose: the *DUP* operator. It can only be used within definition statements. Its role is to duplicate a sequence of expressions. For example:

```
FOO             DB      0, 0, 0, 0, 0           ; 5 null bytes
BAR             DB      5 DUP (0)               ; same as above
```

The parenthesis surrounding the expression to be duplicated are optional. There is a pitfall regarding their use, shown below:

```
FOO             DB      4 DUP (0, 1)            ; defines 8 bytes
FOO2            DB      0, 1, 0, 1, 0, 1, 0, 1  ; same as above

BAR             DB      4 DUP 0, 1              ; defines 5 bytes
BAR2            DB      0, 0, 0, 0, 1           ; same as above
```

Note that only the expressions inside parenthesis are duplicated. If they are not used, the DUP operator will act only on the first one, as showm in defining BAR variable.

DUP expressions can also be nested:

```
        DB      2 DUP (1, 2, 2 DUP 3)   ; defines 8 bytes
        DB      1, 2, 3, 3, 1, 2, 3, 3  ; same as above
```

11.3 Label variables

It is not necessary to allocate any storage to define a variable. In case one needs to define a variable wothout reserving any memory space, the keyword `LABEL` may be used. Its syntax is:

```
VARIABLE_NAME   LABEL   type
```

where

- `VARIABLE_NAME` **is the name of the variable to be defined.**
- `type` **is one of the size specifiers discussed in Section 7.1.**

For example:

```
INFO_START      LABEL   BYTE                    ; Marks beginning of INFO
LENGTH          DD      35.8                    ;
WIDTH           DD      12.5                    ; Some stuff here...
HEIGTH          DD      80.1                    ;
INFO_END        LABEL   BYTE                    ; Marks end of INFO

INFO_LENGTH     EQU     INFO_END - INFO_START   ; Total storage for INFO
```

11.4 Reserving storage

In order to achieve compatibility with *NASM*, *PopAsm* also supports the RESB, RESW, ..., etc. directives. They are used in the BSS section of a module to allocate uninitialized memory. Their syntax is:

```
LABEL           RES_CMD qty
```

where

- `LABEL` **is the variable to be defined (optional). It may be followed by an optional colon.**
- `RES_CMD` **is any of the RESB, RESW, ..., etc. commands.**
- `qty` **is the number of bytes, words, etc. to reserve.**

For example:

```
STACK1          RESD    1024                    ; reserve 1024 dwords
BUFFER          RESB    32768                   ; reserve 32768 bytes
```

11.5 Notes on labels

One of the greatest features of ***PopAsm*** is its type checking mechanism. When one defines a variable as a word, the assembler will keep track of that and will issue error messages when it finds type mismatchs, unless the developer uses a type specifier (see Section 7.1) to explicitly override a variable's type.

Note, however, that in order to achieve compability with ***TASM***, all labels followed by the optional colon will be treated as code labels (that is, a label one can JMP to). For example:

```
ENTRY_POINT:    DW      OFFSET FUNCTION         ; Watch out!
ENTRY_POINT2    DW      OFFSET FUNCTION         ; Ok.

                .
                .
                .

                JMP     ENTRY_POINT             ; Disaster!
                JMP     [ENTRY_POINT]           ; Ok.
                JMP     ENTRY_POINT2            ; Better.
```

The first label, ENTRY_POINT, is defined as a code label. the JMP instruction will transfer the CPU execution to its offset, NOT to its contens, unless the square brackets are used. Note that code labels have no type at all, so if ENTRY_POINT were a far pointer instead of a near one the wrong instruction would be generated.

ENTRY_POINT2 is defined using the recommended syntax. Because no colon is used, ***PopAsm*** assigns the proper typing info to that variable, so the last line will generate the correct instruction.

In order to avoid this pitfall, the developer may follow a simple rule: use the colon *only* if the symbol is a code label. Because defining such labels does not require a colon, one may choose not to use it at all.

The reader might wonder why ***PopAsm*** does not keep track of the typing info even when the colon is used. The reason for that is that some assemblers did not recognize recent instructions (such as earlier versions of ***TASM***.). The developer, then, was forced to encode some instructions manually. For example, if an assembler does not recognize the EMMS instruction then someone could encode it this way:

```
SOME_LABEL:     DW      77FFh                   ; EMMS

                .
                .
                .

                JMP     SOME_LABEL              ; Ok.
```

If SOME_LABEL were treated as an ordinary variable the JMP instruction would jump to address 0FF77h instead of the desired target.

Part III

PopAsm Advanced Syntax

This part presents advanced *PopAsm* features, including commands and macros that one might never use, but are here anyway due to the compatibiliy guideline that drove *PopAsm* so far.

Chapter 12

Aggregates

This chapter discusses the three ways a developer may aggregate several data fields into a single data unit, which can be manipulated easier than disjoint variables spread everywhere. Another good reason to pack data this way is that your code will become easier to read and maintain without any execution performance loss.

12.1 STRUCT

Structures are the simplest aggregates that can be defined. They are a sequence of data members that are placed in adjacent memory addresses. The syntax for defining structures is:

```
struct_name     STRUCT
FIELD_NAME      DEF_CMD EXPRESSION
FIELD_NAME      DEF_CMD EXPRESSION
.
.
.
FIELD_NAME      DEF_CMD EXPRESSION
ENDS
```

where FIELD_NAME are the names of each field (member), DEF_CMD is any of the data defining commands (DB, DW, etc., see Chapter 11) and EXPRESSION is the default value for the field if it is not specified when the structure is used (see details further in this section). The ENDS command stands for "END Structure". Also, STRUCT and STRUC are both accepted indistinctly.

For example, let's suppose we want to define a simple structure to hold dates. Its fields will be month, day and year values:

```
DATE_STRUCT     STRUCT
MONTH           DB      6
DAY             DB      5
YEAR            DW      1979
ENDS
```

Note that the code above defines no data at all; it only defines a structure that may now be used (instantiated). The syntax for instantiating structure is:

```
INSTANCE_NAME   STRUCT_NAME <EXPR1, EXPR2, ..., EXPRN>
```

The syntax above define a variable named INSTANCE_NAME, and instantiate the structure to the expressions given (EXPR1, EXPR2, ..., EXPRN). For example, if someone wants to use the above structure to define a variable that holds the date "Oct 22nd 1977" then the next line will do it:

```
IMPORTANT_DATE  DATE_STRUCT <10, 20 + 2, 1977>
```

As can be seen, the first expression (10) will be placed in the first field of the structure (MONTH), the second expression, which evaluates to 22 (just to remind that any numeric expression can be used, not just numbers), will be copied to the next field and so on. The original default values of the structure definition remain unchanged. The next example shows how to use those default values:

```
JOHN            DATE_STRUCT <2, 7, 1977>
MARY            DATE_STRUCT <5, 1, 1975>
LINDA           DATE_STRUCT < ,  , 1981>
```

Here, we define three dates. The last one looks incomplete, but the blanks will be filled by their default values (MONTH = 6 and DAY = 5). The blank spaces between the commas and the left angle bracket are optional, but were included to give the code a better look.

Once a structure is instantiated, its fields can be accessed as any other variable using the field selection operator (.). Using the structures used in the last example, one could write `JOHN.MONTH` to refer to the byte related to MONTH in JOHN variable. Other examples follow:

```
MOV     AL,[JOHN.DAY]            ; AL = 7

MOV     BX,OFFSET JOHN.DAY       ; Same as above
MOV     AL,[BX]                  ;

MOV     BX,OFFSET JOHN           ;
MOV     SI,OFFSET DATE_STRUCT.DAY; Same as above
MOV     AL,[BX+SI]               ;

MOV     [DATE_STRUCT.DAY],19     ; Wrong!
```

The first line copies the contents of the DAY field of the JOHN variable (declared as a DATE_STRUCT in the previous example) into AL. The same thing could have been achieved by pointing BX to that field (as in the second line) and using that register as a pointer to the field. Another variant of this approach is to use two pointers: one pointing to the variable being referenced and the other to the desired field address (relative to the beginning of the structure).

Such a relative address can be got by specifying the name of the structure, followed by the period, followed by the field name. In the fifth line of the last example, `DATE_STRUCT.DAY` returns the offset of the `DAY` field relative to the beginning of the structure. Note that that offset cannot be used to access memory, given it does not point to any structure in particular. That's why the last line of the example above is wrong.

As a final remark, because the fields of structures occupy memory addresses individually, the total size of a structure is the sum of the sizes of all its fields. This value can be returned by the `SIZE` keyword. For example, `SIZE DATE_STRUCT` returns 4 (DAY and MONTH occupy one byte each, plus two bytes due to the YEAR field).

12.2 UNION

Unions are like structures, except that all of its fields are placed in the same memory address. Writing to one of its fields thus modifies all of them, given the shared memory nature of the union.

The syntax for declaring a union is similar to the one for structures, except that the `STRUCT` keyword is replaced for `UNION`. Note,

however, that because unions fields occupy the same memory location it is possible to specify a default value for *at most* one of their fields. For example:

```
EXAMPLE         UNION
BYTE_FIELD      DB      ?
WORD_FIELD      DW      ?
DWORD_FIELD     DD      1234_5678h
ENDS            ; Yes, unions are ended with ENDS, not ENDU
```

Unions fields are accessed the same way as the fields of a structure (via the period operator). The offsets of their fields are referred by the same means as with structures, but keep in mind that the offset of all fields are the same as the offset of the union variable itself. Considering the union defined above:

```
INSTANCE        EXAMPLE <, 4321h,>
INSTANCE2       EXAMPLE <, 4321h, 123>          ; Error!

                MOV     AL,INSTANCE.BYTE_FIELD  ; AL = 21h
                MOV     AL,BYTE INSTANCE        ; same as above
                MOV     EAX,INSTANCE.DWORD_FIELD; EAX = 1234_4321h
```

That it, the default value for the `DWORD_FIELD` is used and then it's least significant word is overriden by the specification of the second field. The second line is not correct because it attempts to initialize two fields at the same time, which is not allowed.

Some developers may claim that unions are useless. In fact, one can define a single variable as the largest field of the union. Then, using size specifiers it would be possible to access the variable as one with a different size. For example, let's suppose someone wishes to create a variable that emulates the EAX register. That could be accomplished this way:

```
EMULATED_EAX    DD      0

                MOV     [EMULATED_EAX],EAX
                MOV     WORD [EMULATED_EAX],AX
                MOV     BYTE [EMULATED_EAX],AL
                MOV     BYTE [EMULATED_EAX+1],AH
```

That is, a single dword emulates EAX and when the AL, AH or AX emulated counterparts need to be accessed, a size specifier overrides the original size of the variable. In the last line, the person must also remember to add 1 to the base offset because AH lies in the second least significant byte in EAX. Needless to say, this is too error prone. An example of how structures and unions can be used to provide a safer solution is shown in Section 12.4.

12.3 RECORD

To be written.

12.4 Nesting Aggregates

Aggregates can be nested as well. They behave as a field of the aggregate that contains them. For example, let's suppose we wish to rewrite the EAX simulation code (shown in Section 12.2) using nested aggregates. The EAX register can be seen as the union of a dword (EAX itself), a word (AX) and two bytes (AH and AL).

The problem is that AH and AL do not share the same storage. Instead, they should be combined into a 16-bits data unit, which will be one of the fields of the union. The code below does exactely that.

```
; AL and AH must be combined in a 16-bits data unit
AX_STRUCT       STRUCT
SIMULATED_AL    DB      ?
SIMULATED_AH    DB      ?
ENDS

EAX_UNION       UNION
SIMULATED_EAX   DD      ?
SIMULATED_AX    DW      ?
SIMULATED_AH_AL AX_STRUCT       <,>
ENDS

EAX_CLONE       EAX_UNION <0,,>

                MOV     AL,EAX_CLONE.SIMULATED_AH_AL.SIMULATED_AL
                MOV     AH,EAX_CLONE.SIMULATED_AH_AL.SIMULATED_AH
```

```
                MOV     AX,EAX_CLONE.SIMULATED_AX
                MOV     EAX,EAX_CLONE.SIMULATED_EAX
```

The code above first defines a structure to mimic AX lower and upper bytes. This structure is then used as a field of the EAX_UNION, occupying the same memory space as SIMULATED_EAX itself. Note that a major drawback of this approach is that the user cannot simply write EAX_CLONE.SIMULATED_AL because EAX_UNION does not have such a field. Instead, it should be written EAX_CLONE.AX_STRUCT.SIMULATED_AL, which is too awkward.

There is a way to remedy that situation, though. If the structure is defined "on the fly" its namespace will merge with the union's one (for details about namespaces, see Section 12.5). The example above can be rewritten as:

```
EAX_UNION       UNION
SIMULATED_EAX   DD      ?
SIMULATED_AX    DW      ?

                STRUCT                          ;
SIMULATED_AL    DB      ?                       ; Structure defined
SIMULATED_AH    DB      ?                       ; on the fly
ENDS                                            ;

ENDS

EAX_CLONE       EAX_UNION <,,>

                MOV     AL,EAX_UNION.SIMULATED_AL
                MOV     AH,EAX_UNION.SIMULATED_AH
                MOV     AX,EAX_UNION.SIMULATED_AX
                MOV     EAX,EAX_UNION.SIMULATED_EAX
```

12.5 Namespace considerations

As a project grows in size, there are likely to be many variables, structures and unions defined throughout the code. This implies that there is a greater probability of name clashes occur (that is, an attempt to define a symbol whose name is already being used by another symbol definition).

The reason why many compilers complain about name clashes is to avoid ambiguity when such names are referenced. Unfortunatelly some assemblers are just too rigid in this constraint. For example, consider the code below:

```
NAME            DB      'John Adams', 0

STUDENT         STRUCT
NAME            DB      50 DUP (0)
AGE             DB      ?
ENDS

EMPLOYEE        STRUCT
NAME            DB      50 DUP (0)
SALARY          DD      ?
ENDS

CHARLIE         STUDENT <'Charlie Smith', 12>
DAVE            EMPLOYEE <'Dave Thompson', 4000.00>

                MOV     DX,OFFSET NAME
                MOV     DX,OFFSET CHARLIE.NAME
                MOV     DX,OFFSET DAVE.NAME
```

As can be seen, a variable `NAME` appears three times there. It has a stand-alone version (defined as John Adams), and it is also a field in both structures.

The first MOV instruction copies the offset of `NAME` to DX, but which `NAME`? In *PopAsm*, because nothing more has being said so far, the stand-alone version is used. The second MOV instruction makes it clear it refers to the field of the `STUDENT` structure (given `CHARLIE` is a student in this code). The third MOV instruction has no ambiguity either, because it specifies the `NAME` in question is a field on EMPLOYEE structure.

The code above is accepted by *PopAsm* but some other assemblers claim there is a name clash in that code. The reason why *PopAsm* sees no ambiguity in the example above is that it defines a separate namespace for each existing segment, structure, union and record [1].

[1]Note, however, that aggregates defined on the fly have their namespace intentionally merged with the one of the aggregate that contains it. See Section 12.4

Because *PopAsm* must also be compatible with existing code, there are means to force the use of a global namespace for all symbols. See section 3.1.15 for details. The code below illustrates this problem:

```
STUDENT         STRUCT
NAME            DB      50 DUP (0)
AGE             DB      ?
ENDS

                MOV     AX,OFFSET AGE           ; Not good
                MOV     AX,OFFSET STUDENT.AGE   ; Better
```

The first MOV instruction relies on the fact that `AGE` lies in a global namespace. Such a statement would be treated as an error, unless *PopAsm* is configured to use the global namespace. The second MOV instruction shows the prefered way of doing what the line before it does in other assemblers.

Chapter 13

Macros

This chapter shows how macros can be defined and used. A macro is like a procedure, but there is a main difference: macros are expanded ("interpreted") at assembly time; procedures are executed at run-time.

Macros can be seen as an implicit way of performing "cut-and-paste" of portions of code or data. Some of them are built-in and ready for use. Others can be defined by the developer as necessary.

13.1 Built-in macros

The suported built-in macros are described in the next subsections.

13.1.1 REPT — REPeaT

This macro allows one to repeat a block of statements several times. It's syntax is:

```
REPT n
                Block of statements
ENDM
```

where n is the number of times the block of statements must be repeated. It can be any numeric expression that evaluates to an integer number greater than or equal to zero. Note that if n equals to zero the block will be ignored. Example:

```
REPT 10
                DB      1
```

```
                DB      2, 3
ENDM

                DB      10 DUP (1, 2, 3)        ; Same as above
```

This macro also supports nesting, that is, it is allowed to use REPT inside another REPT block.

13.1.2 TIMES — repeat n TIMES

Use this macro to repeat *a single statement* several times. If you need to repeat several statements use the REPT macro instead. Its syntax is:

```
TIMES n         statement
```

where n is the number of times the statement must be repeated. It can be any numeric expression that evaluates to an integer number greater or equal to zero. Note that if n equals to zero the statement will be ignored. Example:

```
TIMES 10        DB      1, 2, 3
                DB      10 DUP (1, 2, 3)        ; Same as above
```

13.2 Defining your own macros

Because *PopAsm* is compatible with *TASM* and *NASM*, both syntaxes for defining macros are supported.

13.2.1 *TASM* syntax

13.2.2 *NASM* syntax

Appendix A

PopAsm commands

This chapter is a list of all commands supported by *PopAsm* (except machine instructions, summarized in appendix B) in alphabetic order. If a command has already been mentioned before in this manual, it will be quickly summarized here, along with a reference to where it is detailed.

A.1 .RADIX

This command allows the developer to change the default radix of numbers from the point the command is issued until the next .RADIX command or end-of-file is found. *PopAsm* sets the default radix to 10 at the start of each pass.

A.1.1 Syntax

```
.RADIX  expression
```

where *expression* is a constant integer numeric expression that evaluates to 2, 8, 10 or 16. *expression* cannot be affected by .RADIX commands, that is, its default radix is always decimal regardless of previous commands.

A.1.2 Examples

Our first example writes the same number in binary, using the "B" suffix and the .RADIX command.

```
BINARY_NUMBER   DB      11000111B             ; 'B' means binary

                .RADIX  2
                ; Numbers default to be binary from now on

BINARY_NUMBER2  DB      11000111              ; same as above

                .RADIX  10
                ; Numbers default to be decimal again

DECIMAL_NUMBER  DD      11000111              ; decimal!
```

It is still possible to use other bases when the default radix is not decimal, but special care should be taken when numbers default to be hex. In the next example, the developer changes the default radix to be 16 and attempts to define a variable whose value is 12 in decimal.

```
                .RADIX  16
                ; Numbers default to be hex from now on

AMBIGUOUS       DD      12D                   ; What does 'D' mean?
DECIMAL_NUMBER  DD      12T                   ; 'T' means decimal
HEX_NUMBER      DD      12                    ; 12h
```

Note that 'D' is a valid digit in hex notation. This means that "12D" is interpreted as "12Dh", not 12 in decimal notation. In order to avoid this ambiguity, *PopAsm* also accepts "T" as a suffix for decimal notation. It can be used no matter what the current radix is. The same remarks hold for the binary suffixes "B" and "Y": "11B" is an hex number when the current radix is 16, but "11Y" is always binary.

A.2 INCLUDE

A.2.1 INCLUDE path

Appendix B

Instruction Set Summary

The purpose of this chapter is to summarize all machine instructions of all CPU's supported by *PopAsm* and their valid syntaxes. The reader should refer to Intel and AMD documentation for further details.

The conventions adopted here are:

- **immed8 — an 8-bit immediate value**
- **signed8 — a signed 8-bit immediate value**
- **unsigned8 — an unsigned 8-bit immediate value**
- **mem - A memory reference**
- **reg - a general purpose register**
- **xmmreg - an XMM register**

B.1 AAA — Ascii Adjust after Addition

Valid Syntaxes	Example
AAA (No arguments)	AAA

B.2 AAD — Ascii Adjust before Division

Valid Syntaxes	Example
AAD (No arguments)	AAD
AAD unsigned8	AAD 16

B.2.1 Notes

***PopAsm* does not check whether or not the immediate argument for this instruction is zero.**

B.3 AAM — Ascii Adjust after Multiply

Valid Syntaxes	Example
AAD (No arguments)	AAM
AAD unsigned8	AAM 10h

B.4 AAS — Ascii Adjust after Subtraction

Valid Syntaxes	Example
AAS (No arguments)	AAS

B.5 ADC — ADd with Carry

Valid Syntaxes	Example
ADC reg,reg/mem	ADC CL,DH
ADC mem,reg	ADC [80h],EDX
ADC reg/mem,immed	ADC [VAR],18
ADC reg/mem,signed8	ADC EAX,-6

B.6 ADD — ADDition

Valid Syntaxes	Example
ADD reg,reg/mem	ADD CL,DH
ADD mem,reg	ADD [80h],EDX
ADD reg/mem,immed	ADD [VAR],18
ADD reg/mem,signed8	ADD EAX,-6

B.7 ADDPD — ADD Packed Double-precision floating-point values

Valid Syntaxes	Example
ADDPD xmmreg,xmmreg/mem128	ADDPD XMM,XMM1

B.8 ADDPS — ADD Packed Single-precision floating-point values

Valid Syntaxes	Example
ADDPS xmmreg,xmmreg/mem128	ADDPS XMM5,TABLE[ESI]

B.9 ADDSD — ADD Scalar Double-precision floating-point values

Valid Syntaxes	Example
ADDSD xmmreg,xmmreg/mem64	ADDSD XMM5,[EDI+ARRAY]

B.10 ADDSS — ADD Scalar Single-precision floating-point values

Valid Syntaxes	Example
ADDSS xmmreg,xmmreg/mem32	ADDSS XMM3,ES:[BX]

B.11 AND — logical AND

Valid Syntaxes	Example
AND reg,reg/mem	AND AX,BX
AND mem,reg	AND [80h],DL
AND reg/mem,immed	AND BYTE [GS:WORD_VAR],18
AND reg/mem,signed8	AND ESI,-6

B.12 ANDPD — bitwise AND Packed Double-precision floating-point values

Valid Syntaxes	Example
ANDPD xmmreg,xmmreg/mem128	ANDPD XMM7,[EAX]

B.13 ANDPS — bitwise AND Packed Single-precision floating-point values

Valid Syntaxes	Example
ANDPS xmmreg,xmmreg/mem128	ANDPS XMM0,[ESI*4]

B.14 ANDNPD — bitwise AND Not Packed Double-precision floating-point values

Valid Syntaxes	Example
ANDNPD xmmreg,xmmreg/mem128	ANDNPD XMM7,[EAX]

B.15 ANDNPS — bitwise AND Not Packed Single-precision floating-point values

Valid Syntaxes	Example
ANDNPS xmmreg,xmmreg/mem128	ANDNPS XMM0,[ESI*4]

B.16 ARPL — Adjust RPL field of segment selector

Valid Syntaxes	Example
ARPL reg16/mem16,reg16	ARPL AX,[BX]

B.17 BOUND — check array index against BOUNDs

Valid Syntaxes	Example
BOUND reg16,mem32	BOUND AX,[BX]
BOUND reg32,mem64	BOUND ECX,[FULL_POINTER]

B.18 BSF — Bit Scan Forward

Valid Syntaxes	Example
`BSF reg16,reg16/mem16`	`BSF CX,SI`
`BSF reg32,reg32/mem32`	`BSF EBX,SS:[SUM]`

B.19 BSR — Bit Scan Reverse

Valid Syntaxes	Example
`BSR reg16,reg16/mem16`	`BSR CX,SI`
`BSR reg32,reg32/mem32`	`BSR EBX,SS:[SUM]`

B.20 BSWAP — Byte SWAP

Valid Syntaxes	Example
`BSWAP reg32`	`BSWAP EDX`

B.21 BT — Bit Test

Valid Syntaxes	Example
`BT reg16/mem16,reg16`	`BT AX,BX`
`BT reg32/mem32,reg32`	`BT [DI],ECX`
`BT reg16/mem16,unsigned8`	`BT DX,15`
`BT reg32/mem32,unsigned8`	`BT EAX,25`

B.21.1 Notes

PopAsm **always checks whether the immediate argument is valid. The bit number must be lesser than the size in bits of the first argument. For example, if the first argument is 16-bits wide the immediate argument must be in the 0–15 interval.**

B.22 BTC — Bit Test and Complement

Valid Syntaxes	Example
`BTC reg16/mem16,reg16`	`BTC AX,BX`
`BTC reg32/mem32,reg32`	`BTC [DI],ECX`
`BTC reg16/mem16,unsigned8`	`BTC DX,15`
`BTC reg32/mem32,unsigned8`	`BTC EAX,25`

B.22.1 Notes

PopAsm **always checks whether the immediate argument is valid. The bit number must be lesser than the size in bits of the first argument. For example, if the first argument is 16-bits wide the immediate argument must be in the 0–15 interval.**

B.23 BTR — Bit Test and Reset

Valid Syntaxes	Example
`BTR reg16/mem16,reg16`	`BTR AX,BX`
`BTR reg32/mem32,reg32`	`BTR [DI],ECX`
`BTR reg16/mem16,unsigned8`	`BTR DX,15`
`BTR reg32/mem32,unsigned8`	`BTR EAX,25`

B.23.1 Notes

PopAsm **always checks whether the immediate argument is valid. The bit number must be lesser than the size in bits of the first argument. For example, if the first argument is 16-bits wide the immediate argument must be in the 0–15 interval.**

B.24 BTS — Bit Test and Set

Valid Syntaxes	Example
`BTS reg16/mem16,reg16`	`BTS AX,BX`
`BTS reg32/mem32,reg32`	`BTS [DI],ECX`
`BTS reg16/mem16,unsigned8`	`BTS DX,15`
`BTS reg32/mem32,unsigned8`	`BTS EAX,25`

B.24.1 Notes

PopAsm **always checks whether the immediate argument is valid. The bit number must be lesser than the size in bits of the first argument. For example, if the first argument is 16-bits wide the immediate argument must be in the 0–15 interval.**

B.25 CALL — CALL procedure

Valid Syntaxes	Example
CALL immed	CALL PRINT_NUMBER
CALL reg16/mem16	CALL BX
CALL reg32/mem32/mem48	CALL PWORD [SI]
CALL farptr32	CALL 0FFFFh:0000h
CALL farpre48	CALL 1234h:DWORD 5Ch

B.25.1 Notes

When in 16-bits mode *PopAsm* expects 16-bits displacements. Using 32-bits displacements in 16-bits mode requires an explicit cast. Accordingly, in 32-bits mode all displacements are encoded in 32-bits, unless they are preceeded by a WORD size specifier. For example:

```
                BITS    16
                CALL    1234h:0FFFFh            ; Ok
                CALL    1234h:10000h            ; Error!
                CALL    1234h:DWORD 10000h      ; Ok!

                BITS    32
                CALL    1234h:5Ch               ; 32-bits offset
                CALL    1234h:WORD 5Ch          ; 16-bits offset
```

Also, for convenience reasons, as well as for compatibility with *TASM* DWORD memory variables are treated as FAR arguments, in 16-bits mode, unless such an assumption is overriden by a NEAR distance specifier. DWORD variables hold near address in 32-bits mode, though. For example:

```
                BITS    16
DWORD_VAR       DD      ?

                CALL    [DWORD_VAR]             ; 16-bits far call
                CALL    NEAR [DWORD_VAR]        ; 32-bits near call
                CALL    WORD NEAR [DWORD_VAR]   ; 16-bits near call

                BITS    32
DWORD_VAR2      DD      ?
```

```
CALL    [DWORD_VAR2]            ; 32-bits near call
CALL    FAR [DWORD_VAR2]        ; 16-bits far call
```

B.26 CBW — Convert Byte to Word

Valid Syntaxes	Example
CBW (no arguments)	CBW

B.27 CDQ — Convert Doubleword to Quadword

Valid Syntaxes	Example
CDQ (no arguments)	CDQ

B.28 CLC — CLear Carry flag

Valid Syntaxes	Example
CLC (no arguments)	CLC

B.29 CLD — CLear Direction flag

Valid Syntaxes	Example
CLD (no arguments)	CLD

B.30 CLFLUSH — Cache Line FLUSH

Valid Syntaxes	Example
CLFLUSH mem8	CLFLUSH [VAR]

B.30.1 Notes

If the argument size is not specified *PopAsm* assumes it is a byte variable, so both statements below are ok:

```
CLFLUSH BYTE [BX]
CLFLUSH [BX]
```

B.31 CLI — CLear Interrupt flag

Valid Syntaxes	Example
CLD (no arguments)	CLD

B.32 CLTS — CLear Task Switched flag

Valid Syntaxes	Example
CLTS (no arguments)	CLTS

B.33 CMC — CoMplement Carry flag

Valid Syntaxes	Example
CMC (no arguments)	CMC

B.34 CMOVcc — Contidional MOVe

Valid Syntaxes	Example
CMOVcc reg16,reg16/mem16	CMOVcc AX,[BX]
CMOVcc reg32,reg32/mem32	CMOVcc EAX,[BX]

B.34.1 Notes

CMOVcc is a generic name for a set of instructions. The supported variants are:

- **CMOVA — Move if above**
- **CMOVAE — Move if above or equal to**
- **CMOVB — Move if below**
- **CMOVBE — Move if below or equal to**
- **CMOVC — Move if carry**
- **CMOVE — Move if equal**
- **CMOVG — Move if greater**

- **CMOVGE — Move if greater or equal**
- **CMOVL — Move if less**
- **CMOVLE — Move if less or equal**
- **CMOVNA — Move if not above**
- **CMOVNAE — Move if not above or equal**
- **CMOVNB — Move if not below**
- **CMOVNBE — Move if not below or equal**
- **CMOVNC — Move if not carry**
- **CMOVNE — Move if not equal**
- **CMOVNG — Move if not greater**
- **CMOVNGE — Move if not greater or equal**
- **CMOVNL — Move if not less**
- **CMOVNLE — Move if not less or equal**
- **CMOVNO — Move if not overflow**
- **CMOVNP — Move if not parity**
- **CMOVNS — Move if not sign**
- **CMOVNZ — Move if not zero**
- **CMOVO — Move if overflow**
- **CMOVP — Move if parity**
- **CMOVPE — Move if parity even**
- **CMOVPO — Move if parity odd**
- **CMOVS — Move if sign**
- **CMOVZ — Move if zero**

Note that some of them have the same machine opcode (eg. CMOVA is the same as CMOVNBE).

B.35 CMP — CoMPare two operands

Valid Syntaxes	Example
`CMP reg,reg/mem`	`CMP AX,BX`
`CMP mem,reg`	`CMP [80h],DL`
`CMP reg/mem,immed`	`CMP BYTE [GS:WORD_VAR],18`
`CMP reg/mem,signed8`	`CMP ESI,-6`

B.36 CMPPD — CoMPare Packed Double-precision floating point values

Valid Syntaxes	Example
`CMPPD xmmreg,xmmreg,unsigned8`	`CMPPD XMM,XMM1,2`
`CMPPD xmmreg,mem128f,unsigned8`	`CMPPD XMM3,OWORD [VAR32],7`

B.36.1 Notes

The third argument for this instruction is an unsigned immediate value ranging from 0 thru 7. *PopAsm* will check this and emit an error message if the third argument lies outside this boundary.

Additionally, *PopAsm* accepts the following two-argument alias for this instruction:

- **CMPEQPD — EQual. Implied third argument is 0.**
- **CMPLTPD — Less Than. Implied third argument is 1.**
- **CMPLEPD — Less Than or Equal. Implied third argument is 2.**
- **CMPUNORDPD — UNORDered. Implied third argument is 3.**
- **CMPNEQPD — Not EQual. Implied third argument is 4.**
- **CMPNLTPD — Not Less Than. Implied third argument is 5.**
- **CMPNLEPD — Not Less than Equal. Implied third argument is 6.**
- **CMPORDPD — ORDered. Implied third argument is 7.**

For example, the next two lines of code are assembled the same way:

```
CMPPD   XMM3,XMM,0
CMPEQPD XMM3,XMM
```

B.37 CMPPS — CoMPare Packed Single-precision floating point values

Valid Syntaxes	Example
CMPPS xmmreg,xmmreg,unsigned8	CMPPS XMM,XMM1,2
CMPPS xmmreg,mem128f,unsigned8	CMPPS XMM3,OWORD [VAR32],7

B.37.1 Notes

The third argument for this instruction is an unsigned immediate value ranging from 0 thru 7. *PopAsm* will check this and emit an error message if the third argument lies outside this boundary.

Additionally, *PopAsm* accepts the following two-argument alias for this instruction:

- **CMPEQPS — EQual. Implied third argument is 0.**
- **CMPLTPS — Less Than. Implied third argument is 1.**
- **CMPLEPS — Less Than or Equal. Implied third argument is 2.**
- **CMPUNORDPS — UNORDered. Implied third argument is 3.**
- **CMPNEQPS — Not EQual. Implied third argument is 4.**
- **CMPNLTPS — Not Less Than. Implied third argument is 5.**
- **CMPNLEPS — Not Less than Equal. Implied third argument is 6.**
- **CMPORDPS — ORDered. Implied third argument is 7.**

For example, the next two lines of code are assembled the same way:

```
CMPPS   XMM3,XMM,0
CMPEQPS XMM3,XMM
```

B.38 CMPS / CMPSB / CMPSW / CMPSD — CoMPare Strings

Valid Syntaxes	Example
CMPS mem8,mem8	CMPS BYTE [SI],[DI]
CMPS mem16,mem16	CMPS WORD GS:[SI],[BX]
CMPS mem32,mem32	CMPS DW_ARRAY,[DI]
CMPSB (no arguments)	CMPSB
CMPSW (no arguments)	CMPSW
CMPSD (no arguments)	CMPSD

B.38.1 Notes

PopAsm **accepts a segment override prefix only for the first argument of CMPS instruction. Besides that restriction, any memory references are accepted as arguments, given their operand sizes match as well as their memory access modes. For example:**

```
CMPS    BYTE [SI],WORD [DI]     ; Type mismatch
CMPS    BYTE [SI],BYTE [EDI]    ; Mode mismatch
CMPS    BYTE [SI],BYTE GS:[EDI] ; Segment override not ok.
```

B.39 CMPSD — CoMPare Scalar Double-precision floating point values

Valid Syntaxes	Example
CMPPD xmmreg,xmmreg,unsigned8	CMPSD XMM,XMM1,2
CMPPD xmmreg,mem64f,unsigned8	CMPSD XMM3,QWORD [VAR32],7

B.39.1 Notes

The third argument for this instruction is an unsigned immediate value ranging from 0 thru 7. ***PopAsm*** **will check this and emit an error message if the third argument lies outside this boundary.**

Additionally, ***PopAsm*** **accepts the following two-argument alias for this instruction:**

- **CMPEQSD — EQual. Implied third argument is 0.**

- **CMPLTSD — Less Than. Implied third argument is 1.**
- **CMPLESD — Less Than or Equal. Implied third argument is 2.**
- **CMPUNORDSD — UNORDered. Implied third argument is 3.**
- **CMPNEQSD — Not EQual. Implied third argument is 4.**
- **CMPNLTSD — Not Less Than. Implied third argument is 5.**
- **CMPNLESD — Not Less than Equal. Implied third argument is 6.**
- **CMPORDSD — ORDered. Implied third argument is 7.**

For example, the next two lines of code are assembled the same way:

```
CMPSD   XMM3,XMM,0
CMPEQSD XMM3,XMM
```

B.40 CMPSS — CoMPare Scalar Single-precision floating point values

Valid Syntaxes	Example
`CMPSS xmmreg,xmmreg,unsigned8`	`CMPPS XMM,XMM1,2`
`CMPSS xmmreg,mem32f,unsigned8`	`CMPPS XMM3,DWORD [VAR],7`

B.40.1 Notes

The third argument for this instruction is an unsigned immediate value ranging from 0 thru 7. *PopAsm* will check this and emit an error message if the third argument lies outside this boundary.

Additionally, *PopAsm* accepts the following two-argument alias for this instruction:

- **CMPEQSS — EQual. Implied third argument is 0.**
- **CMPLTSS — Less Than. Implied third argument is 1.**

- **CMPLESS — Less Than or Equal. Implied third argument is 2.**
- **CMPUNORDSS — UNORDered. Implied third argument is 3.**
- **CMPNEQSS — Not EQual. Implied third argument is 4.**
- **CMPNLTSS — Not Less Than. Implied third argument is 5.**
- **CMPNLESS — Not Less than Equal. Implied third argument is 6.**
- **CMPORDSS — ORDered. Implied third argument is 7.**

For example, the next two lines of code are assembled the same way:

```
CMPSS   XMM3,XMM,0
CMPEQSS XMM3,XMM
```

B.41 CMPXCHG — CoMPare and eXCHanGe

Valid Syntaxes	Example
`CMPXCHG reg8/mem8,reg8`	`CMPXCHG FS:[EBP],AH`
`CMPXCHG reg16/mem16,reg16`	`CMPXCHG AX,BX`
`CMPXCHG reg32/mem32,reg32`	`CMPXCHG [DI],ECX`

B.42 CMPXCHG8B — CoMPare and eXCHanGe 8 Bytes

Valid Syntaxes	Example
`CMPXCHG8B mem64i`	`CMPXCHG8B FS:[ECX+EBP*4]`

B.42.1 Notes

If the argument size is undefined (as in the example above), *PopAsm* assumes it is a 64-bits variable.

B.43 COMISD — COMpare Scalar ordered Double-precision floating point values

Valid Syntaxes	Example
COMISD xmm1,xmm2/mem64	COMISD XMM7,QWORD [MEM32]

B.44 COMISS — COMpare Scalar ordered Single-precision floating point values

Valid Syntaxes	Example
COMISS xmm1,xmm2/mem64	COMISS XMM7,QWORD [MEM32]

B.45 CPUID — CPU IDentitication

Valid Syntaxes	Example
CPUID (no arguments)	CPUID

B.46 CVTDQ2PD — ConVerT Packed Doubleword integers to Packed Double precision floating-point values

Valid Syntaxes	Example
CVTDQ2PD xmmreg,xmmreg/mem64	CVTDQ2PD XMM,QWORD [BX]

B.47 CVTDQ2PS — ConVerT Packed Doubleword integers to Packed Single precision floating-point values

Valid Syntaxes	Example
CVTDQ2PD xmmreg,xmmreg/mem128	CVTDQ2PD XMM,OWORD [BX]

B.48 CVTPD2DQ — ConVerT Packed Double precision floating-point values to Packed doubleword integers

Valid Syntaxes	Example
CVTPD2DQ xmmreg,xmmreg/mem128	CVTPD2DQ XMM,OWORD [BX]

B.49 CVTPD2PI — ConVerT Packed Double precision floating-point values to Packed Doubleword Integers

Valid Syntaxes	Example
CVTPD2PI xmmreg,xmmreg/mem128	CVTPD2PI XMM,OWORD [BX]

B.50 CVTPD2PS — ConVerT Packed Double precision floating-point values to Packed Single-precision floating-point

Valid Syntaxes	Example
CVTPD2PS xmmreg,xmmreg/mem128	CVTPD2PS XMM,OWORD [BX]

B.51 CVTPI2PD — ConVerT Packed doubleword Integers to Packed Double precision floating-point values

Valid Syntaxes	Example
CVTPI2PD xmmreg,xmmreg/mem64	CVTPI2PD XMM,QWORD [BX]

B.52 CVTPI2PS — ConVerT Packed doubleword Integers to Packed Single precision floating-point values

Valid Syntaxes	Example
CVTPI2PS xmmreg,xmmreg/mem64	CVTPI2PS XMM,QWORD [BX]

B.53 CVTPS2DQ — ConVerT Packed Single precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTPS2DQ xmmreg,xmmreg/mem128	CVTPS2DQ XMM,OWORD [BX]

B.54 CVTPS2PD — ConVerT Packed Single precision floating-point values to Packed Double-precision floating-point values

Valid Syntaxes	Example
CVTPS2PD xmmreg,xmmreg/mem64	CVTPS2PD XMM,QWORD [BX]

B.55 CVTPS2PI — ConVerT Packed Single precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTPS2PI mmxreg,xmmreg/mem64	CVTPS2PI MM,QWORD [BX]

B.56 CVTSD2SI — ConVerT Scalar Double-precision floating-point values to doubleword Integers

Valid Syntaxes	Example
CVTSD2SI reg32,xmmreg/mem64	CVTSD2SI ESI,QWORD [BX]

B.57 CVTSD2SS — ConVerT Scalar Double precision floating-point values to Scalar Single-precision floating-point value

Valid Syntaxes	Example
CVTSD2SS mmxreg,xmmreg/mem64	CVTSD2SS XMM,QWORD [BX]

B.58 CVTSI2SD — ConVerT doubleword Integer to Scalar Double-precision floating-point value

Valid Syntaxes	Example
CVTSI2SD mmxreg,reg32/mem32	CVTSI2SD XMM,DWORD [BX]

B.59 CVTSI2SS — ConVerT doubleword Integer to Scalar Single-precision floating-point value

Valid Syntaxes	Example
CVTSI2SS mmxreg,reg32/mem32	CVTSI2SS XMM,DWORD [BX]

B.60 CVTSS2SD — ConVerT Scalar Single precision floating-point value to Scalar Double-precision floating-point value

Valid Syntaxes	Example
CVTSS2SD xmmreg,xmmreg/mem32	CVTSS2SD XMM,DWORD [BX]

B.61 CVTSS2SI — ConVerT Scalar Single precision floating-point value to doubleword Integer

Valid Syntaxes	Example
CVTSS2SI xmmreg,xmmreg/mem32	CVTSS2SI EBP,DWORD [BX]

B.62 CVTTPD2PI — ConVerT with Truncation Packed Double-precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTTPD2PI mmxreg,xmmreg/mem128	CVTPD2PS MM,OWORD [BX]

B.63 CVTTPD2DQ — ConVerT with Truncation Packed Double-precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTTPD2DQ mmxreg,xmmreg/mem128	CVTPD2PS XMM,OWORD [BX]

B.64 CVTTPS2DQ — ConVerT with Truncation Packed Single-precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTTPS2DQ xmmreg,xmmreg/mem128	CVTPD2PS XMM,OWORD [BX]

B.65 CVTTPS2PI — ConVerT with Truncation Packed Single-precision floating-point values to Packed doubleword Integers

Valid Syntaxes	Example
CVTTPS2DQ mmxreg,xmmreg/mem64	CVTPD2PS MM,QWORD [BX]

B.66 CVTTSD2SI — ConVerT with Truncation Scalar Double-precision floating-point value to Signed doubleword Integer

Valid Syntaxes	Example
CVTTPS2DQ reg32,xmmreg/mem64	CVTPD2PS EDX,QWORD [BX]

B.67 CVTTSS2SI — ConVerT with Truncation Scalar Single-precision floating-point value to doubleword Integer

Valid Syntaxes	Example
CVTTPS2DQ reg32,xmmreg/mem32	CVTPD2PS EDX,DWORD [BX]

B.68 CWD — Convert Word to Doubleword

Valid Syntaxes	Example
CWD (no arguments)	CWD

B.69 CWDE — Convert Word to Dword in Eax

Valid Syntaxes	Example
CWDE (no arguments)	CWDE

B.70 DAA — Decimal Adjust after Addition

Valid Syntaxes	Example
DAA (no arguments)	DAA

B.71 DAS — Decimal Adjust after Subtraction

Valid Syntaxes	Example
DAS (no arguments)	DAS

B.72 DEC — DECrement by 1

Valid Syntaxes	Example
DEC reg8/mem8	DEC CH
DEC reg16/mem16	DEC WORD [BX]
DEC reg32/mem32	DEC EBP

B.72.1 Notes

As with most single argument instructions, the size of the argument must be defined, otherwise *PopAsm* will emit an error message.

B.73 DIV — unsigned DIVide

Valid Syntaxes	Example
DIV reg8/mem8	DIV CH
DIV reg16/mem16	DIV WORD [BX]
DIV reg32/mem32	DIV EBP

B.73.1 Notes

As with most single argument instructions, the size of the argument must be defined, otherwise *PopAsm* will emit an error message.

B.74 DIVPD — DIVide Packed Double-precision floating-point values

Valid Syntaxes	Example
DIVPD xmmreg,xmmreg/mem128	DIVPD XMM1,OWORD ES:[100h]

B.75 DIVPS — DIVide Packed Single-precision floating-point values

Valid Syntaxes	Example
DIVPS xmmreg,xmmreg/mem128	DIVPS XMM1,OWORD ES:[100h]

B.76 DIVSD — DIVide Scalar Double-precision floating-point values

Valid Syntaxes	Example
DIVSD xmmreg,xmmreg/mem64	DIVSD XMM1,QWORD ES:[100h]

B.77 DIVSS — DIVide Scalar Single-precision floating-point values

Valid Syntaxes	Example
DIVSS xmmreg,xmmreg/mem64	DIVSS XMM1,QWORD ES:[100h]

B.78 EMMS — Empty MMx State

Valid Syntaxes	Example
EMMS (no arguments)	EMMS

B.79 ENTER — make stack frame for procedure parameters

Valid Syntaxes	Example
ENTER unsigned16,unsigned8	ENTER 64,0

B.79.1 Notes

The second argument must be in 0–31 interval. *PopAsm* will issue an error message if the argument is out of that interval.

B.80 F2XM1 — Compute $2^x - 1$

Valid Syntaxes	Example
F2XM1 (no arguments)	F2XM1

B.81 FABS — ABSolute value

Valid Syntaxes	Example
FABS (no arguments)	FABS

B.82 FADD — ADD

Valid Syntaxes	Example
FADD mem32f	FADD [BRIGHTNESS]
FADD mem64f	FADD ES:[LIGHT_SPEED]
FADD ST,fpureg	FADD ST,ST(1)
FADD fpureg,ST	FADD ST6,ST0

B.83 FADDP — ADD and Pop

Valid Syntaxes	Example
FADDP (no arguments)	FADDP
FADDP fpureg,ST	FADDP ST(5),ST0

B.84 FIADD — Integer ADD

Valid Syntaxes	Example
FIADD mem16i	FIADD [QTY_APPLES]
FIADD mem32i	FIADD FS:[POPULATION]

B.85 FBLD — Bcd LoaD

Valid Syntaxes	Example
FIADD mem80bcd	FBLD [HUGE_VAL]

B.86 FBSTP — Bcd STore and Pop

Valid Syntaxes	Example
`FBSTP mem80bcd`	`FBSTP [HUGE_VAL]`

B.87 FCHS — CHange Sign

Valid Syntaxes	Example
`FCHS (no arguments)`	`FCHS`

B.88 FCLEX / FNCLEX — CLEar eXceptions

Valid Syntaxes	Example
`FCLEX (no arguments)`	`FCLEX`
`FNCLEX (no arguments)`	`FNCLEX`

B.89 FCMOVcc — Floating-point Conditional MOVe

Valid Syntaxes	Example
`FCMOVcc ST,fpureg`	`FCMOVcc ST0,ST2`

B.89.1 Notes

FCMOVcc actually refers to a set of instructions that check for different conditions. Suportted variants are:

- **FCMOVB — Move if below**
- **FCMOVE — Move if equal**
- **FCMOVBE — Move if below or equal**
- **FCMOVU — Move if unordered**
- **FCMOVNB — Move if not below**
- **FCMOVNE — Move if not equal**
- **FCMOVNBE — Move if not below or equal**
- **FCMOVNU — Move if not unordered**

B.90 FCOM — COMpare

Valid Syntaxes	Example
FCOM (no arguments)	FCOM
FCOM fpureg	FCOM ST(4)
FCOM mem32f	FCOM [SALARY]
FCOM mem64f	FCOM [PARSEC]

B.91 FCOMP — COMpare and Pop

Valid Syntaxes	Example
FCOMP (no arguments)	FCOMP
FCOMP fpureg	FCOMP ST(4)
FCOMP mem32f	FCOMP [SALARY]
FCOMP mem64f	FCOMP [PARSEC]

B.92 FCOMPP — COMpare and Pop twice

Valid Syntaxes	Example
FCOMPP (no arguments)	FCOMPP

B.93 FCOMI — COMpare and set eflags

Valid Syntaxes	Example
FCOMI ST,fpureg	FCOMI ST(0),ST(4)

B.94 FCOMIP — COMpare, set eflags and Pop

Valid Syntaxes	Example
FCOMIP ST,fpureg	FCOMIP ST(0),ST(4)

B.95 FUCOMI — COMpare, check for ordered and set eflags

Valid Syntaxes	Example
FUCOMI ST,fpureg	FUCOMI ST(0),ST(4)

B.96 FUCOMIP — COMpare, check for ordered, set eflags and Pop

Valid Syntaxes	Example
FUCOMIP ST,fpureg	FUCOMIP ST(0),ST(4)

B.97 FCOS — COSine

Valid Syntaxes	Example
FCOS (no arguments)	FCOS

B.98 FDECSTP — DECrement Stack-Top Pointer

Valid Syntaxes	Example
FDECSTP (no arguments)	FDECSTP

B.99 FDIV — DIVide

Valid Syntaxes	Example
FDIV ST,fpureg	FDIV ST0,ST(4)
FDIV fpureg,ST	FDIV ST0,ST(4)
FDIV mem32f	FDIV [SALARY]
FDIV mem64f	FDIV [PARSEC]

B.100 FDIVP — DIVide and Pop

Valid Syntaxes	Example
FDIVP (no arguments)	FDIVP
FDIVP fpureg,ST	FDIVP ST(4),ST

B.101 FIDIV — Integer DIVide

Valid Syntaxes	Example
FIDIV mem16i	FIDIV [QTY_APPLES]
FIDIV mem32i	FIDIV [POPULATION]

B.102 FDIVR — Reverse DIVide

Valid Syntaxes	Example
FDIVR ST,fpureg	FDIVR ST0,ST(4)
FDIVR fpureg,ST	FDIVR ST0,ST(4)
FDIVR mem32f	FDIVR [SALARY]
FDIVR mem64f	FDIVR [PARSEC]

B.103 FDIVP — Reverse DIVide and Pop

Valid Syntaxes	Example
FDIVRP (no arguments)	FDIVRP
FDIVRP fpureg,ST	FDIVRP ST(4),ST

B.104 FIDIV — Reverse Integer DIVide

Valid Syntaxes	Example
FIDIVR mem32i	FIDIVR [QTY_APPLES]
FIDIVR mem64i	FIDIVR [POPULATION]

B.105 FEMMS — Faster Enter/Exit of the MMx or floating-point State

Valid Syntaxes	Example
FEMMS (no arguments)	FEMMS

B.106 FFREE — FREE floating-point register

Valid Syntaxes	Example
FFREE fpureg	FFREE ST6

B.107 FICOM — Integer COMpare

Valid Syntaxes	Example
FICOM mem16i	FICOM [QTY_APPLES]
FICOM mem32i	FICOM [POPULATION]

B.108 FICOMP — Integer COMpare and Pop

Valid Syntaxes	Example
FICOMP mem16i	FICOMP [QTY_APPLES]
FICOMP mem32i	FICOMP [POPULATION]

B.109 FILD — Integer LoaD

Valid Syntaxes	Example
FILD mem16i	FILD [QTY_APPLES]
FILD mem32i	FILD [POPULATION]
FILD mem64i	FILD [QTY_STARS]

B.110 FINCSTP — INCrement Stack-Top Pointer

Valid Syntaxes	Example
FINCSTP (no arguments)	FINCSTP

B.111 FINIT / FNINIT — INITialize fpu

Valid Syntaxes	Example
FINIT (no arguments)	FINIT
FNINIT (no arguments)	FNINIT

B.112 FIST — STore Integer

Valid Syntaxes	Example
`FIST mem16i`	`FIST [QTY_APPLES]`
`FIST mem32i`	`FIST [POPULATION]`

B.113 FISTP — STore Integer and Pop

Valid Syntaxes	Example
`FISTP mem16i`	`FISTP [QTY_APPLES]`
`FISTP mem32i`	`FISTP [POPULATION]`
`FISTP mem64i`	`FISTP [QTY_STARS]`

B.114 FLD — LoaD floating-point value

Valid Syntaxes	Example
`FLD mem32f`	`FLD [SALARY]`
`FLD mem64f`	`FLD [PARSEC]`
`FLD mem80f`	`FLD TBYTE [BX]`

B.115 FLD1 — LoaD 1

Valid Syntaxes	Example
`FLD1 (no arguments)`	`FLD1`

B.116 FLDL2T — LoaD $\log_2 10$

Valid Syntaxes	Example
`FLDL2T (no arguments)`	`FLDL2T`

B.117 FLDL2E — LoaD $\log_2 e$

Valid Syntaxes	Example
`FLDL2E (no arguments)`	`FLDL2E`

B.118 FLDPI — LoaD π

Valid Syntaxes	Example
`FLDPI (no arguments)`	`FLDPI`

B.119 FLDLG2 — LoaD $\log_{10} 2$

Valid Syntaxes	Example
`FLDLG2 (no arguments)`	`FLDLG2`

B.120 FLDLN2 — LoaD $\ln 2$

Valid Syntaxes	Example
`FLDLN2 (no arguments)`	`FLDLN2`

B.121 FLDZ — LoaD Zero

Valid Syntaxes	Example
`FLDZ (no arguments)`	`FLDZ`

B.122 FLDCW — LoaD Control Word

Valid Syntaxes	Example
`FLDCW mem16`	`FLDCW  WORD [BX]`

B.123 FLDENV — LoaD fpu ENVironment

Valid Syntaxes	Example
`FLDENV  mem112/mem224`	`FLDENV  [BUFFER]`

B.123.1 Notes

Because *PopAsm* has no direct support for 112- and 224-bits data types, *any* memory reference will be accepted as argument for this command.

B.124 FMUL — MULtiply

Valid Syntaxes	Example
FMUL ST,fpureg	FMUL ST0,ST(4)
FMUL fpureg,ST	FMUL ST0,ST(4)
FMUL mem32f	FMUL [SALARY]
FMUL mem64f	FMUL [PARSEC]

B.125 FMULP — MULtiply and Pop

Valid Syntaxes	Example
FMULP (no arguments)	FMULP
FMULP fpureg,ST	FMULP ST(4),ST

B.126 FIMUL — Integer MULtiply

Valid Syntaxes	Example
FIMUL mem16i	FIMUL [QTY_APPLES]
FIMUL mem32i	FIMUL [POPULATION]

B.127 FNOP — No OPeration

Valid Syntaxes	Example
FNOP (no arguments)	FNOP

B.128 FPATAN — Partial ArcTANgent

Valid Syntaxes	Example
FPATAN (no arguments)	FPATAN

B.129 FPREM — Partial REMainer

Valid Syntaxes	Example
FPREM (no arguments)	FPREM

B.130 FPREM1 — Partial REMainer

Valid Syntaxes	Example
FPREM1 (no arguments)	FPREM1

B.131 FPTAN — Partial TANgent

Valid Syntaxes	Example
FPTAN (no arguments)	FPTAN

B.132 FRNDINT — RouND to INTeger

Valid Syntaxes	Example
FRNDINT (no arguments)	FRNDINT

B.133 FRSTOR — ReSToRe fpu state

Valid Syntaxes	Example
FRSTOR mem752/mem864	FRSTOR [EBP]

B.133.1 Notes

Because *PopAsm* has no direct support for 752- and 864-bits data types, *any* memory reference will be accepted as argument for this command.

B.134 FSAVE / FNSAVE — SAVE fpu state

Valid Syntaxes	Example
FSAVE mem752/mem864	FSAVE [EBP]
FNSAVE mem752/mem864	FNSAVE [EBP]

B.134.1 Notes

Because *PopAsm* has no direct support for 752- and 864-bits data types, *any* memory reference will be accepted as argument for this command.

B.135 FSCALE — Scale

Valid Syntaxes	Example
FSCALE (no arguments)	FSCALE

B.136 FSIN — SINe

Valid Syntaxes	Example
FSIN (no arguments)	FSIN

B.137 FSINCOS — SINe and COSine

Valid Syntaxes	Example
FSINCOS (no arguments)	FSINCOS

B.138 FSQRT — SQuare RooT

Valid Syntaxes	Example
FSQRT (no arguments)	FSQRT

B.139 FST — STore floating point value

Valid Syntaxes	Example	
FST fpureg	FST	ST(2)
FST mem32f	FST	[SALARY]
FST mem64f	FST	FS:[STAR_MASS]

B.140 FSTP — STore floating point value and Pop

Valid Syntaxes	Example	
FSTP fpureg	FSTP	ST(2)
FSTP mem32f	FSTP	[SALARY]
FSTP mem64f	FSTP	FS:[STAR_MASS]
FSTP mem80f	FSTP	GS:[HUGE_VALUE]

B.141 FSTCW / FNSTCW — STore Control Word

Valid Syntaxes	Example
FSTCW mem16	FSTCW WORD [BX]
FNSTCW mem16	FNSTCW WORD [BX]

B.142 FSTENV / FNSTENV — STore fpu ENVironment

Valid Syntaxes	Example
FSTENV mem112/mem224	FSTENV [BUFFER]
FNSTENV mem112/mem224	FNSTENV [BUFFER]

B.142.1 Notes

Because *PopAsm* has no direct support for 112- and 224-bits data types, *any* memory reference will be accepted as argument for this command.

B.143 FSTSW / FNSTSW — STore fpu Status Word

Valid Syntaxes	Example
FSTSW AX	FSTSW AX
FSTSW mem16	FSTSW [BUFFER]
FNSTSW AX	FNSTSW AX
FNSTSW mem16	FNSTSW [BUFFER]

B.144 FSUB — SUBtract

Valid Syntaxes	Example
FSUB mem32f	FSUB [BRIGHTNESS]
FSUB mem64f	FSUB ES:[LIGHT_SPEED]
FSUB ST,fpureg	FSUB ST,ST(1)
FSUB fpureg,ST	FSUB ST6,ST0

B.145 FSUBP — SUBtract and Pop

Valid Syntaxes	Example
FSUBP (no arguments)	FSUBP
FSUBP fpu_reg,ST	FSUBP ST(3),ST

B.146 FISUB — Integer SUBtract

Valid Syntaxes	Example
FISUB mem16i	FISUB [QTY_APPLES]
FISUB mem32i	FISUB FS:[POPULATION]

B.147 FSUBR— Reverse SUBtract

Valid Syntaxes	Example
FSUBR mem32f	FSUBR [BRIGHTNESS]
FSUBR mem64f	FSUBR ES:[LIGHT_SPEED]
FSUBR ST,fpureg	FSUBR ST,ST(1)
FSUBR fpureg,ST	FSUBR ST6,ST0

B.148 FSUBRP — Reverse SUBtract and Pop

Valid Syntaxes	Example
FSUBRP (no arguments)	FSUBRP
FSUBRP fpu_reg,ST	FSUBRP ST(3),ST

B.149 FISUBR — Reverse Integer SUBtract

Valid Syntaxes	Example
FISUBR mem16i	FISUBR [QTY_APPLES]
FISUBR mem32i	FISUBR FS:[POPULATION]

B.150 FTST — TeST

Valid Syntaxes	Example
FTST (no arguments)	FTST

B.151 FUCOM — Unordered COMpare

Valid Syntaxes	Example
FUCOM (no arguments)	FUCOM
FUCOM fpureg	FUCOM ST4

B.152 FUCOMP — Unordered COMpare and Pop

Valid Syntaxes	Example
FUCOMP (no arguments)	FUCOMP
FUCOMP fpureg	FUCOMP ST4

B.153 FUCOMPP — Unordered COMpare and Pop twice

Valid Syntaxes	Example
FUCOMPP (no arguments)	FUCOMPP

B.154 FWAIT — WAIT

Valid Syntaxes	Example
FWAIT (no arguments)	FWAIT

B.155 FXAM — eXAMine

Valid Syntaxes	Example
FXCH (no arguments)	FXCH

B.156 FXCH — eXCHange register contents

Valid Syntaxes	Example
FXCH (no arguments)	FXCH
FXCH flureg	FXCH ST4

B.157 FXRSTOR — ResTORe fpu, mmX, sse and ss2 state

Valid Syntaxes	Example
FXRSTOR mem4096	FXRSTOR ES:[SI]

B.157.1 Notes

Because *PopAsm* has no direct support for 4096-bits data types, *any* memory reference will be accepted as argument for this command.

B.158 FXSAVE — SAVE fpu, mmX, sse and ss2 state

Valid Syntaxes	Example
FXSAVE mem4096	FXSAVE ES:[SI]

B.158.1 Notes

Because *PopAsm* has no direct support for 4096-bits data types, *any* memory reference will be accepted as argument for this command.

B.159 FXTRACT — eXTRACT exponent and significand

Valid Syntaxes	Example
FXTRACT (no arguments)	FXTRACT

B.160 FYL2X — computes $y \times \log_2 x$

Valid Syntaxes	Example
FYL2X (no arguments)	FYL2X

B.161 FYL2XP1 — computes $y \times \log_2(x+1)$

Valid Syntaxes	Example
FYL2XP1 (no arguments)	FYL2XP1

B.162 HLT — HaLT

Valid Syntaxes	Example
HLT (no arguments)	HLT

B.163 IDIV — signed DIVide

Valid Syntaxes	Example
IDIV reg8/mem8	IDIV CL
IDIV reg16/mem16	IDIV WORD FS:[COUNTER]
IDIV reg32/mem32	IDIV ESI

B.164 IMUL — signed MULtiply

Valid Syntaxes	Example
IMUL reg8/mem8	IMUL CL
IMUL reg16/mem16	IMUL WORD FS:[COUNTER]
IMUL reg32/mem32	IMUL ESI
IMUL reg16,reg16/mem16	IMUL BX,GS:[500h]
IMUL reg32,reg32/mem32	IMUL ESI,EDI
IMUL reg16,reg16/mem16,signed8	IMUL BX,GS:[500h],-78
IMUL reg32,reg32/mem32,signed8	IMUL ESI,EDI,13
IMUL reg16,signed8	IMUL BX,7
IMUL reg32,signed8	IMUL ESI,-90
IMUL reg16,reg16/mem16,signed16	IMUL BX,GS:[500h],-78
IMUL reg32,reg32/mem32,signed32	IMUL ESI,EDI,13
IMUL reg16,signed16	IMUL BX,200h
IMUL reg32,signed32	IMUL ESI,12345h

B.164.1 Notes

Because this instruction deals with signed arithmetics, all immediate values are treated as signed numbers. Thus, the next statements will cause an error if included in a source file:

```
IMUL    BX,0FFFFh
IMUL    DX, 8000h
```

because the processor would treat `0FFFFh` as -1 and `8000h` as `-8000h`.

B.165 IN — INput from port

Valid Syntaxes	Example
`IN accum,unsigned8`	`IN AL,60h`
`IN accum,DX`	`IN EAX,DX`

B.166 INC — INCrement by 1

Valid Syntaxes	Example
`INC reg8/mem8`	`INC CH`
`INC reg16/mem16`	`INC WORD [BX]`
`INC reg32/mem32`	`INC EBP`

B.167 INS / INSB / INSW / INSD — INput from port to String

Valid Syntaxes	Example
`INS mem8,DX`	`INS BYTE DS:[ESI],DX`
`INS mem16,DX`	`INS WORD [BX],DX`
`INS mem32,DX`	`INS DWORD [BUFFER],DX`
`INSB (no arguments)`	`INSB`
`INSW (no arguments)`	`INSW`
`INSD (no arguments)`	`INSD`

B.168 INT — call to INTerrupt procedure

Valid Syntaxes	Example
INT unsigned8	INT 80h

B.168.1 Notes

There is a special single-byte encoding for the `INT 3` command. *PopAsm* uses that optimized encoding as default, unless the developer explictly uses a BYTE specifier. Also, for compatibility with *NASM*, *PopAsm* also suports its "INT3" command (which uses the optimized encoding). The next example shows all assembly possibilities for the INT command when its argument's value is 3.

```
INT     3                       ; 1 byte
INT3                            ; 1 byte
INT     BYTE 3                  ; 2 bytes
```

B.169 INTO — INTerrupt of Overflow

Valid Syntaxes	Example
INTO (no arguments)	INTO

B.170 INVD — INValiDate internal caches

Valid Syntaxes	Example
INVD (no arguments)	INVD

B.171 INVLPG — INValidate tlb entry

Valid Syntaxes	Example
INVLPG mem	INVLPG GS:[BP+SI]

B.172 IRET / IRETD — Interrupt RETurn

Valid Syntaxes	Example
IRET (no arguments)	IRET
IRETD (no arguments)	IRETD

B.173 Jcc — Jump if condition is met

Valid Syntaxes	Example
JCC rel8/rel16/rel32	JCC NEXT_STEP

B.173.1 Notes

Jcc is actually a reference to a set of instructions where *cc* is replaced by the condition to test for. Suported variants are:

- **JA — Jump if Above**
- **JAE — Jump if Above or Equal**
- **JB — Jump if Below**
- **JBE — Jump if Below or Equal**
- **JC — Jump if Carry**
- **JE — Jump if Equal**
- **JG — Jump if Greater**
- **JGE — Jump if Greater or Equal**
- **JL — Jump if Less**
- **JLE — Jump if Less or Equal**
- **JNA — Jump if Not Above**
- **JNAE — Jump if Not Above or Equal**
- **JNB — Jump if Not Below**
- **JNBE — Jump if Not Below or Equal**

- **JNC — Jump if Not Carry**
- **JNE — Jump if Not Equal**
- **JNG — Jump if Not Greater**
- **JNGE — Jump if Not Greater or Equal**
- **JNL — Jump if Not Less**
- **JNLE — Jump if Not Less or Equal**
- **JNO — Jump if Not Overflow**
- **JNP — Jump if Not Parity**
- **JNS — Jump if Not Sign**
- **JNZ — Jump if Not Zero**
- **JO — Jump if Overflow**
- **JP — Jump if Parity**
- **JPE — Jump if Parity Even**
- **JPO — Jump if Parity Odd**
- **JS — Jump if Sign**
- **JZ — Jump if Zero**

The short version of these instructions is used by default. Whenever the target is too far for a short jump the near version for the current assembly mode is then used. Note that this behavior can be changed by size and distance specifiers, as shown in the next example:

```
JA      $+10                    ; 8-bits displacement

BITS    16
JA      NEAR $+10               ; 16-bits displacement
JA      DWORD $+10              ; 32-bits displacement

BITS    32
JA      NEAR $+10               ; 32-bits displacement
JA      WORD $+10               ; 16-bits displacement
```

B.174 JCXZ / JECXZ — Jump if CX / ECX is zero

Valid Syntaxes	Example
JCXZ rel8	JCXZ LOOP_EXIT
JECXZ rel8	JECXZ LOOP_EXIT

B.175 JMP — JuMP

Valid Syntaxes	Example
JMP rel8/reg16/rel32	JMP PROC_EXIT
JMP reg16/mem16	JMP BX
JMP reg32/mem32/mem48	JMP PWORD [SI]
JMP farptr32	JMP 0FFFFh:0000h
JMP farptr48	JMP 1234h:DWORD 5Ch

B.175.1 Notes

When in 16-bits mode *PopAsm* expects 16-bits displacements. Using 32-bits displacements in 16-bits mode requires an explicit cast. Accordingly, in 32-bits mode all displacements are encoded in 32-bits, unless they are preceeded by a WORD size specifier. For example:

```
BITS    16
JMP     1234h:0FFFFh            ; Ok
JMP     1234h:10000h            ; Error!
JMP     1234h:DWORD 10000h      ; Ok!

BITS    32
JMP     1234h:5Ch               ; 32-bits offset
JMP     1234h:WORD 5Ch          ; 16-bits offset
```

Also, for convenience reasons, as well as for compatibility with *TASM* DWORD memory variables are treated as FAR arguments, in 16-bits mode, unless such an assumption is overriden by a NEAR distance specifier. DWORD variables hold near address in 32-bits mode, though. For example:

```
                BITS    16
DWORD_VAR       DD      ?

                JMP     [DWORD_VAR]             ; 16-bits far jmp
                JMP     NEAR [DWORD_VAR]        ; 32-bits near jmp
                JMP     WORD NEAR [DWORD_VAR]   ; 16-bits near jmp

                BITS    32
DWORD_VAR2      DD      ?

                JMP     [DWORD_VAR2]            ; 32-bits near jmp
                JMP     FAR [DWORD_VAR2]        ; 16-bits far jmp
```

B.176 LAHF — Load AH with Flags

Valid Syntaxes	Example
LAHF (no arguments)	LAHF

B.177 LAR — Load Access Rights

Valid Syntaxes	Example
LAR reg16,reg16/mem16	LAR CX,SI
LAR reg32,reg32/mem32	LAR EBX,SS:[DWORD_VAR]

B.178 LDMXCSR — Load MXCSR register

Valid Syntaxes	Example
LDMXCSR mem32	LDMXCSR [DWORD_VAR]

B.179 Lxx — Load far pointer

Valid Syntaxes	Example
LXX reg16,mem32	LXX CX,[SI]
LXX reg32,mem48	LXX EBX,ES:[ENTRY_POINT]

B.179.1 Notes

Lxx actually refer to a set of instructions that load far pointers. Suported variants are:

- **LDS — Load DS:dest**
- **LES — Load ES:dest**
- **LFS — Load FS:dest**
- **LGS — Load GS:dest**
- **LSS — Load SS:dest**

B.180 LEA — Load Effective Address

Valid Syntaxes	Example
`LEA reg16/reg32,mem16/mem32`	`LEA ECX,[EAX + EBX*4 + 200h]`

B.181 LEAVE — LEAVE

Valid Syntaxes	Example
`LEAVE (no arguments)`	`LEAVE`

B.182 LFENCE — Load FENCE

Valid Syntaxes	Example
`LFENCE (no arguments)`	`LFENCE`

B.183 LGDT — Load GDTr

Valid Syntaxes	Example
`LGDT mem48`	`LGDT [INITIAL_GDTR]`

B.184 LIDT — Load IDTr

Valid Syntaxes	Example
`LIDT mem48`	`LGDT [INITIAL_IDTR]`

B.185 LLDT — Load LDTr

Valid Syntaxes	Example
LLDT reg16/mem16	LLDT AX

B.186 LMSW — Load Machine Status Word

Valid Syntaxes	Example
LMSW reg16/mem16	LMSW AX

B.187 LOCK — assert LOCK signal prefix

Valid Syntaxes	Example
LOCK (no arguments)	LOCK
LOCK statement	LOCK XCHG AX,[BX]

B.188 LODS / LODSB / LODSW / LODSD — LOaD String

Valid Syntaxes	Example
LODS mem8/mem16/mem32	LODS BYTE ES:[ESI]
LODSB (no arguments)	LODSB
LODSW (no arguments)	LODSW
LODSD (no arguments)	LODSD

B.189 LOOP / LOOPCC — LOOP according to cx / ecx

Valid Syntaxes	Example
LOOP rel8	LOOP PRINT_STRING
LOOPCC rel8	LOOPCC NEXT_STEP

B.189.1 Notes

LOOPCC refers to a set of instructions that test for particular conditions. Suported variants are:

- **LOOPE — LOOP if Equal**
- **LOOPNE — LOOP if Not Equal**
- **LOOPNZ — LOOP if Not Zero**
- **LOOPZ — LOOP if Zero**

B.190 LSL — Load Segment Limits

Valid Syntaxes	Example
`LSL reg16,reg16/mem16`	`LSL CX,SI`
`LSL reg32,reg32/mem32`	`LSL EBX,SS:[DWORD_VAR]`

B.191 LTR — Load Task Register

Valid Syntaxes	Example
`LTR reg16/mem16`	`LTR CX`

B.192 MASKMOVDQU — store selected bytes of Double Quadword

Valid Syntaxes	Example
`MASKMOVDQU xmmreg,xmmreg`	`MASKMOVDQU XMM4,XMM1`

B.193 MASKMOVQ — store selected bytes of Quadword

Valid Syntaxes	Example
`MASKMOVQ mmreg,mmreg`	`MASKMOVQ MM4,MM1`

B.194 MAXPD — return MAXimum Packed Double-precision floating-point values

Valid Syntaxes	Example
MAXPD xmmreg,xmmreg/mem128	MAXPD XMM,[OWORD_VAR]

B.195 MAXPS — return MAXimum Packed Single-precision floating-point values

Valid Syntaxes	Example
MAXPS xmmreg,xmmreg/mem128	MAXPS XMM,[OWORD_VAR]

B.196 MAXSD — return MAXimum Scalar Double-precision floating-point values

Valid Syntaxes	Example
MAXSD xmmreg,xmmreg/mem64	MAXSD XMM,[QWORD_VAR]

B.197 MAXSS — return MAXimum Scalar Single-precision floating-point values

Valid Syntaxes	Example
MAXSS xmmreg,xmmreg/mem64	MAXSS XMM,[DWORD_VAR]

B.198 MFENCE — Memory FENCE

Valid Syntaxes	Example
MFENCE (no arguments)	MFENCE

B.199 MINPD — return MINimum Packed Double-precision floating-point values

Valid Syntaxes	Example
MINPD xmmreg,xmmreg/mem128	MINPD XMM,[OWORD_VAR]

B.200 MINPS — return MINimum Packed Single-precision floating-point values

Valid Syntaxes	Example
MINPS xmmreg,xmmreg/mem128	MINPS XMM,[OWORD_VAR]

B.201 MINSD — return MINimum Scalar Double-precision floating-point values

Valid Syntaxes	Example
MINSD xmmreg,xmmreg/mem64	MINSD XMM,[QWORD_VAR]

B.202 MINSS — return MINimum Scalar Single-precision floating-point values

Valid Syntaxes	Example
MINSS xmmreg,xmmreg/mem64	MINSS XMM,[DWORD_VAR]

B.203 MOV — MOVe

Valid Syntaxes	Example
MOV reg,reg/mem	MOV AX,BX
MOV mem,reg	MOV [80h],DL
MOV reg/mem,immed	MOV BYTE [GS:WORD_VAR],18
MOV segreg,reg16/mem16	MOV ES,[2Ch]
MOV reg16/mem16,segreg	MOV AX,DS
MOV creg/treg/dreg,reg32	MOV CR0,EAX
MOV reg32,creg/treg/dreg	MOV ECX,DR5

B.203.1 Notes

It is not possible to MOV to CS.

B.204 MOVAPD — MOVe Aligned Packed Double-precision floating-point values

Valid Syntaxes	Example
MOVAPD xmmreg,xmmreg/mem128	MOVAPD XMM3,FS:[BP+SI-40]
MOVAPD xmmreg/mem128,xmmreg	MOVAPD XMM6,XMM2

B.205 MOVAPS — MOVe Aligned Packed Single-precision floating-point values

Valid Syntaxes	Example
MOVAPS xmmreg,xmmreg/mem128	MOVAPS XMM3,FS:[BP+SI-40]
MOVAPS xmmreg/mem128,xmmreg	MOVAPS XMM6,XMM2

B.206 MOVD — MOVe Doubleword

Valid Syntaxes	Example
MOVD mmreg,reg32/mem32	MOVD MM3,DWORD_VAR[ECX*4]
MOVD reg32/mem32,mmreg	MOVD MM3,MM
MOVD xmmreg,reg32/mem32	MOVD XMM3,DWORD_VAR[ECX*4]
MOVD reg32/mem32,xmmreg	MOVD XMM3,XMM

B.207 MOVDQA — MOVe Aligned Double Quadword

Valid Syntaxes	Example
MOVDQA xmmreg,xmmreg/mem128	MOVDQA XMM3,OWORD_VAR[ECX*4]
MOVDQA xmmreg/mem128,xmmreg	MOVDQA XMM3,XMM

B.208 MOVDQU — MOVe Unaligned Double Quadword

Valid Syntaxes	Example
MOVDQU xmmreg,xmmreg/mem128	MOVDQU XMM3,OWORD_VAR[ECX*4]
MOVDQU xmmreg/mem128,xmmreg	MOVDQU XMM3,XMM

B.209 MOVDQ2Q — MOVe Quadword from xmm to mmx register

Valid Syntaxes	Example
MOVQ2Q mmreg,xmmreg	MOVQ2Q MM3,XMM4

B.210 MOVHLPS — MOVe Packed Single-precision floating-point values High to Low

Valid Syntaxes	Example
MOVHLPS xmmreg,xmmreg	MOVHLPS XMM3,XMM4

B.211 MOVHPD — MOVe High Packed Double-precision floating-point value

Valid Syntaxes	Example
MOVHPD xmmreg,mem64	MOVHPD XMM0,[QWORD_VAR]
MOVHPD mem64,xmmreg	MOVHPD [QWORD_VAR],XMM7

B.212 MOVHPS — MOVe High Packed Single-precision floating-point values

Valid Syntaxes	Example
MOVHPS xmmreg,mem64	MOVHPS XMM0,[QWORD_VAR]
MOVHPS mem64,xmmreg	MOVHPS [QWORD_VAR],XMM7

B.213 MOVLHPS — MOVe Packed Single-precision floating-point values Low to High

Valid Syntaxes	Example
MOVLHPS xmmreg,xmmreg	MOVLHPS XMM3,XMM4

B.214 MOVLPD — MOVe Low Packed Double-precision floating-point value

Valid Syntaxes	Example
MOVLPD xmmreg,mem64	MOVLPD XMM0,[QWORD_VAR]
MOVLPD mem64,xmmreg	MOVLPD [QWORD_VAR],XMM7

B.215 MOVLPS — MOVe Low Packed Single-precision floating-point value

Valid Syntaxes	Example
MOVLPS xmmreg,mem64	MOVLPS XMM0,[QWORD_VAR]
MOVLPS mem64,xmmreg	MOVLPS [QWORD_VAR],XMM7

B.216 MOVMSKPD — extract Packed Double-precision floating-point sign mask

Valid Syntaxes	Example
MOVMSKPD reg32,xmmreg	MOVMSKPD ESI,XMM5

B.217 MOVMSKPS — extract Packed Single-precision floating-point sign mask

Valid Syntaxes	Example
MOVMSKPS reg32,xmmreg	MOVMSKPS ESI,XMM5

B.218 MOVNTDQ — store Double Quadword using Non-Temporal hint

Valid Syntaxes	Example
MOVLHPS mem128,xmmreg	MOVLHPS [OWORD_VAR],XMM4

B.219 MOVNTI — store Doubleword using Non-Temporal hint

Valid Syntaxes	Example
MOVNTI mem32,reg32	MOVNTI [DWORD_VAR],ESP

B.220 MOVNTPD — store Packed Double-precision floating-point values using Non-Temporal hint

Valid Syntaxes	Example
MOVNTPD mem128,xmmreg	MOVNTPD XMM_DATA[16],XMM3

B.221 MOVNTPS — store Packed Single-precision floating-point values using Non-Temporal hint

Valid Syntaxes	Example
MOVNTPS mem128,xmmreg	MOVNTPS XMM_DATA[16],XMM3

B.222 MOVNTQ — store Quadword using Non-Temporal hint

Valid Syntaxes	Example
MOVNTQ mem128,xmmreg	MOVNTQ ES:MM_DATA[16],MM3

B.223 MOVQ — MOVe Quadword

Valid Syntaxes	Example
MOVQ mmreg,reg32/mem32	MOVQ MM3,DWORD_VAR[ECX*4]
MOVQ reg32/mem32,mmreg	MOVQ MM3,MM
MOVQ xmmreg,reg32/mem32	MOVQ XMM3,DWORD_VAR[ECX*4]
MOVQ reg32/mem32,xmmreg	MOVQ XMM3,XMM

B.224 MOVQ2DQ — MOVe Quadword from mmx to xmm Register

Valid Syntaxes	Example
MOVQ2DQ xmmreg,mmxreg	MOVQ2DQ XMM3,MM6

B.225 MOVS / MOVSB / MOVSW / MOVSD — MOVe Strings

Valid Syntaxes	Example
MOVS mem8,mem8	MOVS BYTE [DI],[SI]
MOVS mem16,mem16	MOVS WORD [DI],FS:[BX]
MOVS mem32,mem32	MOVS DW_ARRAY,[SI]
MOVSB (no arguments)	MOVSB
MOVSW (no arguments)	MOVSW
MOVSD (no arguments)	MOVSD

B.225.1 Notes

PopAsm **accepts a segment override prefix only for the second argument of MOVS instruction. Besides that restriction, any memory references are accepted as arguments, given their operand sizes match as well as their memory access modes. For example:**

```
MOVS    BYTE [DI],WORD [SI]     ; Type mismatch
MOVS    BYTE [DI],BYTE [ESI]    ; Mode mismatch
MOVS    BYTE GS:[DI],BYTE [ESI] ; Segment override not ok.
```

B.226 MOVSD — MOVe Scalar Double-precision floating-point value

Valid Syntaxes	Example
MOVSD xmmreg,xmmreg/mem64	MOVSD XMM5,[LIGHT_SPEED]
MOVSD xmmreg/mem64,xmmreg	MOVSD [QWORD_VAR],XMM4

B.227 MOVSS — MOVe Scalar Single-precision floating-point value

Valid Syntaxes	Example
MOVSS xmmreg,xmmreg/mem32	MOVSD XMM5,[SALARY]
MOVSS xmmreg/mem32,xmmreg	MOVSD [DWORD_VAR],XMM4

B.228 MOVSX — MOVe with Sign-eXtension

Valid Syntaxes	Example
MOVSX reg16/reg32,reg8/mem8	MOVSX CX,BYTE [80h]
MOVSX reg32,reg16/mem16	MOVSX EAX,DX

B.229 MOVUPD — MOVe Unaligned Packed Double-precision floating-point values

Valid Syntaxes	Example
MOVUPD xmmreg,xmmreg/mem128	MOVUPD XMM3,OWORD_VAR[ECX*4]
MOVUPD xmmreg/mem128,xmmreg	MOVUPD XMM3,XMM

B.230 MOVZX — MOVe with Zero-eXtension

Valid Syntaxes	Example
MOVZX reg16/reg32,reg8/mem8	MOVZX CX,BYTE [80h]
MOVZX reg32,reg16/mem16	MOVZX EAX,DX

B.231 MOVUPS — MOVe Unaligned Packed Single-precision floating-point values

Valid Syntaxes	Example
MOVUPS xmmreg,xmmreg/mem128	MOVUPS XMM3,OWORD_VAR[ECX*4]
MOVUPS xmmreg/mem128,xmmreg	MOVUPS XMM3,XMM

B.232 MUL — unsigned MULtiply

Valid Syntaxes	Example
MUL reg8/mem8	MUL CH
MUL reg16/mem16	MUL WORD [BX]
MUL reg32/mem32	MUL EBP

B.233 MULPD — MULtiply Packed Double-precision floating-point values

Valid Syntaxes	Example
MULPD xmmreg,xmmreg/mem128	MULPD XMM2,XMM7

B.234 MULPS — MULtiply Packed Single-precision floating-point values

Valid Syntaxes	Example
MULPS xmmreg,xmmreg/mem128	MULPS XMM2,[EBX]

B.235 MULSD — MULtiply Scalar Double-precision floating-point values

Valid Syntaxes	Example
MULSD xmmreg,xmmreg/mem64	MULSD XMM2,[EBX]

B.236 MULSS — MULtiply Scalar Single-precision floating-point values

Valid Syntaxes	Example
MULSS xmmreg,xmmreg/mem32	MULSS XMM2,[EBX]

B.237 NEG — two's complement NEGation

Valid Syntaxes	Example
NEG reg8/mem8	NEG CH
NEG reg16/mem16	NEG WORD [BX]
NEG reg32/mem32	NEG EBP

B.238 NOP — No OPeration

Valid Syntaxes	Example
NOP (no arguments)	NOP

B.239 NOT — one's complement negation

Valid Syntaxes	Example
NOT reg8/mem8	NOT CH
NOT reg16/mem16	NOT WORD [BX]
NOT reg32/mem32	NOT EBP

B.240 OR — logical inclusive OR

Valid Syntaxes	Example
OR reg,reg/mem	OR AX,BX
OR mem,reg	OR [80h],DL
OR reg/mem,immed	OR BYTE [GS:WORD_VAR],18
OR reg/mem,signed8	OR ESI,-6

B.241 ORPD — bitwise logical OR of Double-precision floating-point values

Valid Syntaxes	Example
ORPD xmmreg,xmmreg/mem128	ORPD XMM2,[EBX]

B.242 ORPS — bitwise logical OR of Single-precision floating-point values

Valid Syntaxes	Example
ORPS xmmreg,xmmreg/mem128	ORPS XMM2,[EBX]

B.243 OUT — OUTput to port

Valid Syntaxes	Example
OUT unsigned8,accum	OUT 42h,AL
OUT DX,accum	OUT DX,EAX

B.244 OUTS / OUTSB / OUTSW / OUTSD — OUTput String

Valid Syntaxes	Example
OUTS DX,mem8/mem16/mem32	OUTS DX,[MY_NAME]
OUTSB (no arguments)	OUTSB
OUTSW (no arguments)	OUTSW
OUTSD (no arguments)	OUTSD

B.245 PACKSSWB / PACKSSDW — PACK with Sign Saturation

Valid Syntaxes	Example
PACKSSWB mmreg,mmreg/mem64	PACKSSWB MM5,[BP+DI+500h]
PACKSSWB xmmreg,xmmreg/mem128	PACKSSWB XMM4,XMM7
PACKSSDW mmreg,mmreg/mem64	PACKSSDW MM5,[BP+DI+500h]
PACKSSDW xmmreg,xmmreg/mem128	PACKSSDW XMM4,XMM7

B.246 PACKUSWB — PACK with Unsigned Saturation

Valid Syntaxes	Example
PACKUSWB mmreg,mmreg/mem64	PACKUSWB MM5,[BP+DI+500h]
PACKUSWB xmmreg,xmmreg/mem128	PACKUSWB XMM4,XMM7

B.247 PADDB / PADDW / PADDD / PADDQ — ADD Packed integers

Valid Syntaxes	Example
PADDB mmreg,mmreg/mem64	PADDB MM6,MM1
PADDB xmmreg,xmmreg/mem128	PADDB XMM,ES:[BX+DI-32]
PADDW mmreg,mmreg/mem64	PADDW MM6,MM1
PADDW xmmreg,xmmreg/mem128	PADDW XMM,ES:[BX+DI-32]
PADDD mmreg,mmreg/mem64	PADDD MM6,MM1
PADDD xmmreg,xmmreg/mem128	PADDD XMM,ES:[BX+DI-32]
PADDQ mmreg,mmreg/mem64	PADDQ MM6,MM1
PADDQ xmmreg,xmmreg/mem128	PADDQ XMM,ES:[BX+DI-32]

B.248 PADDSB / PADDSW — ADD Packed Signed integers with signed saturation

Valid Syntaxes	Example
PADDSB mmreg,mmreg/mem64	PADDSB MM6,MM1
PADDSB xmmreg,xmmreg/mem128	PADDSB XMM,ES:[BX+DI-32]
PADDSW mmreg,mmreg/mem64	PADDSW MM6,MM1
PADDSW xmmreg,xmmreg/mem128	PADDSW XMM,ES:[BX+DI-32]

B.249 PADDUSB / PADDUSW — ADD Packed Unsigned integers with unsigned saturation

Valid Syntaxes	Example
PADDUSB mmreg,mmreg/mem64	PADDUSB MM6,MM1
PADDUSB xmmreg,xmmreg/mem128	PADDUSB XMM,ES:[BX+DI-32]
PADDUSW mmreg,mmreg/mem64	PADDUSW MM6,MM1
PADDUSW xmmreg,xmmreg/mem128	PADDUSW XMM,ES:[BX+DI-32]

B.250 PAND — Logical AND

Valid Syntaxes	Example
PAND mmreg,mmreg/mem64	PAND MM6,MM1
PAND xmmreg,xmmreg/mem128	PAND XMM,ES:[BX+DI-32]

B.251 PANDN — Logical AND NOT

Valid Syntaxes	Example
PANDN mmreg,mmreg/mem64	PANDN MM6,MM1
PANDN xmmreg,xmmreg/mem128	PANDN XMM,ES:[BX+DI-32]

B.252 PAUSE — spin loop hint

Valid Syntaxes	Example
PAUSE (no argumenta)	PAUSE

B.253 PAVGB / PAVGW — AVeraGe Packed integers

Valid Syntaxes	Example
PAVGB mmreg,mmreg/mem64	PAVGB MM6,MM1
PAVGB xmmreg,xmmreg/mem128	PAVGB XMM,ES:[BX+DI-32]
PAVGW mmreg,mmreg/mem64	PAVGW MM6,MM1
PAVGW xmmreg,xmmreg/mem128	PAVGW XMM,ES:[BX+DI-32]

B.254 PAVGUSB — AVeraGe of unsigned Packed 8-bit values

Valid Syntaxes	Example
PAVGUSB mmreg,mmreg/mem64	PAVGUSB MM6,MM1
PAVGUSB xmmreg,xmmreg/mem128	PAVGUSB XMM,ES:[BX+DI-32]

B.255 PCMPEQB / PCMPEQW / PCMPEQD — CoMPare Packed data for EQual

Valid Syntaxes	Example
PCMPEQB mmreg,mmreg/mem64	PCMPEQB MM6,MM1
PCMPEQB xmmreg,xmmreg/mem128	PCMPEQB XMM,ES:[BX+DI-32]
PCMPEQW mmreg,mmreg/mem64	PCMPEQW MM6,MM1
PCMPEQW xmmreg,xmmreg/mem128	PCMPEQW XMM,ES:[BX+DI-32]
PCMPEQD mmreg,mmreg/mem64	PCMPEQD MM6,MM1
PCMPEQD xmmreg,xmmreg/mem128	PCMPEQD XMM,ES:[BX+DI-32]

B.256 PCMPGTB / PCMPGTW / PCMPGTD — CoMPare Packed signed integers for Greater Than

Valid Syntaxes	Example
PCMPGTB mmreg,mmreg/mem64	PCMPGTB MM6,MM1
PCMPGTB xmmreg,xmmreg/mem128	PCMPGTB XMM,ES:[BX+DI-32]
PCMPGTW mmreg,mmreg/mem64	PCMPGTW MM6,MM1
PCMPGTW xmmreg,xmmreg/mem128	PCMPGTW XMM,ES:[BX+DI-32]
PCMPGTD mmreg,mmreg/mem64	PCMPGTD MM6,MM1
PCMPGTD xmmreg,xmmreg/mem128	PCMPGTD XMM,ES:[BX+DI-32]

B.257 PEXTRW — EXTRact Word

Valid Syntaxes	Example
PEXTRW reg32,mmreg,unsigned8	PEXTRW ECX,MM,3
PEXTRW reg32,xmmreg,unsigned8	PEXTRW ECX,XMM0,5

B.257.1 Notes

PopAsm **checks whether or not the immediate argument is valid. If the second argument is a MMX register, the third argument must be in 0–3 interval. Likewise, if the second argument is a XMM register, the third argument should be in 0–7 interval.**

B.258 PF2ID — convert Packed Floating-point operand to packed 32-bit Integer

Valid Syntaxes	Example
PF2ID mmreg,mmreg/mem64	PF2ID MM6,MM1

B.259 PF2IW — convert Packed Floating-point operand to Integer Word with sign-extend

Valid Syntaxes	Example
PF2IW mmreg,mmreg/mem64	PF2IW MM6,MM1

B.260 PFACC — Packed Floating-point ACCumulate

Valid Syntaxes	Example
PFACC mmreg,mmreg/mem64	PFACC MM6,MM1

B.261 PFADD — Packed Floating-point ADDition

Valid Syntaxes	Example
PFADD mmreg,mmreg/mem64	PFADD MM6,MM1

B.262 PFCMPEQ — Packed Floating-point CoMParison, EQual

Valid Syntaxes	Example
PFCMPEQ mmreg,mmreg/mem64	PFCMPEQ MM6,MM1

B.263 PFCMPGE — Packed Floating-point CoMParison, Greater or Equal

Valid Syntaxes	Example
PFCMPGE mmreg,mmreg/mem64	PFCMPGE MM6,MM1

B.264 PFCMPGT — Packed Floating-point CoMParison, Greater Than

Valid Syntaxes	Example
PFCMPGT mmreg,mmreg/mem64	PFCMPGT MM6,MM1

B.265 PFMAX — Packed Floating-point MAXimum

Valid Syntaxes	Example
PFMAX mmreg,mmreg/mem64	PFMAX MM6,MM1

B.266 PFMIN — Packed Floating-point MINimum

Valid Syntaxes	Example
PFMIN mmreg,mmreg/mem64	PFMIN MM6,MM1

B.267 PFMUL — Packed Floating-point MULtiplication

Valid Syntaxes	Example
PFMUL mmreg,mmreg/mem64	PFMUL MM6,MM1

B.268 PFNACC — Packed Floating-point Negative ACCumulate

Valid Syntaxes	Example
PFACC mmreg,mmreg/mem64	PFACC MM6,MM1

B.269 PFPNACC — Packed Floating-point mixed Positive-Negative ACCumulate

Valid Syntaxes	Example
PFNACC mmreg,mmreg/mem64	PFNACC MM6,MM1

B.270 PFRCP — Packed Floating-point ReCiProcal approximation

Valid Syntaxes	Example
PFRCP mmreg,mmreg/mem64	PFRCP MM6,MM1

B.271 PFRCPIT1 — Packed Floating-point ReCiProcal approximation, firat ITeration step

Valid Syntaxes	Example
PFRCPIT1 mmreg,mmreg/mem64	PFRCPIT1 MM6,MM1

B.272 PFRCPIT2 — Packed Floating-point ReCiProcal approximation, second ITeration step

Valid Syntaxes	Example
PFRCPIT2 mmreg,mmreg/mem64	PFRCPIT2 MM6,[BP+DI+32]

B.273 PFRSQIT1 — Packed Floating-point Reciprocal SQuare root, firat ITeration step

Valid Syntaxes	Example
PFRSQIT1 mmreg,mmreg/mem64	PFRSQIT1 MM6,MM1

B.274 PFRSQRT — Packed Floating-point Reciprocal SQuare RooT approximation

Valid Syntaxes	Example
PFRSQRT mmreg,mmreg/mem64	PFRSQRT MM3,MM

B.275 PFSUB — Packed Floating-point SUBtraction

Valid Syntaxes	Example
PFSUB mmreg,mmreg/mem64	PFSUB MM7,MM1

B.276 PFSUBR — Packed Floating-point Reverse SUBtraction

Valid Syntaxes	Example
PFSUBR mmreg,mmreg/mem64	PFSUBR MM7,MM1

B.277 PI2FD — Packed 32-bit Integer to Floating-point conversion

Valid Syntaxes	Example
PI2FD mmreg,mmreg/mem64	PI2FD MM7,MM1

B.278 PI2FW — Packed 16-bit Integer to Floating-point conversion

Valid Syntaxes	Example
PI2FW mmreg,mmreg/mem64	PI2FW MM6,MM1

B.279 PINSRW — INSeRt Word

Valid Syntaxes	Example
PINSRW mmreg,reg32/mem16,unsigned8	PINSRW MM,ECX,3
PINSRW xmmreg,reg32/mem16,unsigned8	PINSRW XMM0,EDX,5

B.279.1 Notes

PopAsm **checks whether or not the immediate argument is valid. If the first argument is a MMX register, the third argument must be in 0–3 interval. Likewise, if the first argument is a XMM register, the third argument should be in 0–7 interval.**

B.280 PMADDWD — Multiply and ADD Packed integers

Valid Syntaxes	Example
PMADDWD mmreg,mmreg/mem64	PMADDWD MM0,QWORD [VAR]
PMADDWW xmmreg,xmmreg/mem128	PMADDWD XMM3,XMM1

B.281 PMAXSW — MAXimum of Packed Signed Word integers

Valid Syntaxes	Example
PMAXSW mmreg,mmreg/mem64	PMAXSW MM0,QWORD [VAR]
PMAXSW xmmreg,xmmreg/mem128	PMAXSW XMM3,XMM1

B.282 PMAXUB — MAXimum of Packed Unsigned Byte integers

Valid Syntaxes	Example
PMAXUB mmreg,mmreg/mem64	PMAXUB MM0,QWORD [VAR]
PMAXUB xmmreg,xmmreg/mem128	PMAXUB XMM3,XMM1

B.283 PMINSW — MINimum of Packed Signed Word integers

Valid Syntaxes	Example
PMINSW mmreg,mmreg/mem64	PMINSW MM0,QWORD [VAR]
PMINSW xmmreg,xmmreg/mem128	PMINSW XMM3,XMM1

B.284 PMINUB — MINimum of Packed Unsigned Byte integers

Valid Syntaxes	Example
PMINUB mmreg,mmreg/mem64	PMINUB MM0,QWORD [VAR]
PMINUB xmmreg,xmmreg/mem128	PMINUB XMM3,XMM1

B.285 PMOVMSKB — MOV Byte MaSK

Valid Syntaxes	Example
PMOVMSKB reg32,mmreg	PMOVMSKB EDX,MM6
PMOVMSKB reg32,xmmreg	PMOVMSKB EBP,XMM4

B.286 PMULHRW — MULtiply signed Packed 16-bit values with Rounding and store the High 16 bits

Valid Syntaxes	Example
PMULHRW mmreg,mmreg/mem64	PMULHRW MM7,MM1

B.287 PMULHUW — MULtiply Packed Unsigned integers and store High result

Valid Syntaxes	Example
PMULHUW mmreg,mmreg/mem64	PMULHUW MM4,[VAR]
PMULHUW xmmreg,xmmreg/mem64	PMULHUW XMM5,XMM1

B.288 PMULHW — MULtiply Packed signed integers and store High result

Valid Syntaxes	Example
PMULHW mmreg,mmreg/mem64	PMULHW MM4,[VAR]
PMULHW xmmreg,xmmreg/mem64	PMULHW XMM5,XMM1

B.289 PMULLW — MULtiply Packed signed integers and store Low result

Valid Syntaxes	Example
PMULLW mmreg,mmreg/mem64	PMULLW MM4,[VAR]
PMULLW xmmreg,xmmreg/mem64	PMULLW XMM5,XMM1

B.290 PMULUDQ — MULtiply Packed Unsigned Doubleword integers

Valid Syntaxes	Example
PMULUDQ mmreg,mmreg/mem64	PMULUDQ MM4,[VAR]
PMULUDQ xmmreg,xmmreg/mem64	PMULUDQ XMM5,XMM1

B.291 POP — POP a value from the stack

Valid Syntaxes	Example
POP reg16/reg32/mem16/mem32	POP AX
POP segreg	POP GS

B.291.1 Notes

It is not possible to POP CS.

B.292 POPA / POPAD — POP All general-purpose registers

Valid Syntaxes	Example
POPA (no arguments)	POPA
POPAD (no arguments)	POPAD

B.293 POPF / POPFD — POP stack into eFlags register

Valid Syntaxes	Example
POPF (no arguments)	POPF
POPFD (no arguments)	POPFD

B.294 POR — Logical OR

Valid Syntaxes	Example
POR mmreg,mmreg/mem64	POR MM6,MM1
POR xmmreg,xmmreg/mem128	POR XMM,ES:[BX+DI-32]

B.295 PREFETCHh — PREFETCH data into caches

Valid Syntaxes	Example
PREFETCH mem8	PREFETCH [EBX+ECX]
PREFETCH0 mem8	PREFETCH0 [BX+12]
PREFETCH1 mem8	PREFETCH1 [CLIENTS]
PREFETCH2 mem8	PREFETCH2 ES:[BP-64]
PREFETCHNTA mem8	PREFETCHNTA [EAX]
PREFETCHW mem8	PREFETCHW [EBX+ECX]

B.295.1 Notes

If the argument size is not specified *PopAsm* assumes it is a byte variable, so both statements below are ok:

```
PREFETCH0 BYTE [BX]
PREFETCH0 [BX]
```

B.296 PSADDBW — compute Sum of Absolute Differences

Valid Syntaxes	Example
PSADBW mmreg,mmreg/mem64	PSADBW MM6,MM1
PSADBW xmmreg,xmmreg/mem128	PSADBW XMM,ES:[BX+DI-32]

B.297 PSHUFD — SHUFfle Packed Doublewords

Valid Syntaxes	Example
PSHUFD xmmreg,xmmreg/mem128,unsigned8	PSHUFD XMM,XMM1,00_01_10_11B

B.298 PSHUFHW — SHUFfle Packed High Words

Valid Syntaxes	Example
PSHUFHW xmmreg,xmmreg/mem128,unsigned8	PSHUFHW XMM,XMM1,00_01_10_11B

B.299 PSHUFLW — SHUFfle Packed Low Words

Valid Syntaxes	Example
PSHUFLW xmmreg,xmmreg/mem128,unsigned8	PSHUFLW XMM,XMM1,00_01_10_11B

B.300 PSHUFW — SHUFfle Packed Words

Valid Syntaxes	Example
PSHUFW mmreg,mmreg/mem64,unsigned8	PSHUFW MM,MM1,00_01_10_11B

B.301 PSLLDQ — Shift Double Quadword Left Logical

Valid Syntaxes	Example
PSLLDQ xmmreg,unsigned8	PSLLDQ XMM,00_01_10_11B

B.302 PSLLW / PSLLD / PSLLQ — Shift Packed Data Left Logical

Valid Syntaxes	Example
PSLLW mmreg,mmreg/mem64	PSLLW MM5,[AMOUNTS]
PSLLW xmmreg,xmmreg/mem128	PSLLW XMM5,XMM2
PSLLW mmreg,mmreg/mem64	PSLLW MM5,3
PSLLW xmmreg,xmmreg/mem128	PSLLW XMM3,4
PSLLD mmreg,mmreg/mem64	PSLLD MM5,[AMOUNTS]
PSLLD xmmreg,xmmreg/mem128	PSLLD XMM5,XMM2
PSLLD mmreg,mmreg/mem64	PSLLD MM5,3
PSLLD xmmreg,xmmreg/mem128	PSLLD XMM3,4
PSLLQ mmreg,mmreg/mem64	PSLLQ MM5,[AMOUNTS]
PSLLQ xmmreg,xmmreg/mem128	PSLLQ XMM5,XMM2
PSLLQ mmreg,mmreg/mem64	PSLLQ MM5,3
PSLLQ xmmreg,xmmreg/mem128	PSLLQ XMM3,4

B.302.1 Notes

When those instructions are used with immediate arguments, *PopAsm* checks if they are in 0–15, 0–31 and 0–63 intervals for PSLLW, PSLLD and PSLLQ, respectively.

B.303 PSRAW / PSRAD — Shift Packed Data Left Logical

Valid Syntaxes	Example
`PSRAW mmreg,mmreg/mem64`	`PSRAW MM5,[AMOUNTS]`
`PSRAW xmmreg,xmmreg/mem128`	`PSRAW XMM5,XMM2`
`PSRAW mmreg,mmreg/mem64`	`PSRAW MM5,3`
`PSRAW xmmreg,xmmreg/mem128`	`PSRAW XMM3,4`
`PSRAD mmreg,mmreg/mem64`	`PSRAD MM5,[AMOUNTS]`
`PSRAD xmmreg,xmmreg/mem128`	`PSRAD XMM5,XMM2`
`PSRAD mmreg,mmreg/mem64`	`PSRAD MM5,3`
`PSRAD xmmreg,xmmreg/mem128`	`PSRAD XMM3,4`

B.303.1 Notes

When those instructions are used with immediate arguments, *PopAsm* checks if they are in 0–15 and 0–31 intervals for PSRAW and PSRAD, respectively.

B.304 PSRLDQ — Shift Double Quadword Right Logical

Valid Syntaxes	Example
`PSRLDQ xmmreg,unsigned8`	`PSRLDQ XMM5,1`

B.304.1 Notes

***PopAsm* checks if the second argument is greater than 15. If it is an error message is issued.**

B.305 PSRLW / PSRLD / PSRLQ — Shift Packed Data Right Logical

Valid Syntaxes	Example	
PSRLW mmreg,mmreg/mem64	PSRLW	MM5,[AMOUNTS]
PSRLW xmmreg,xmmreg/mem128	PSRLW	XMM5,XMM2
PSRLW mmreg,mmreg/mem64	PSRLW	MM5,3
PSRLW xmmreg,xmmreg/mem128	PSRLW	XMM3,4
PSRLD mmreg,mmreg/mem64	PSRLD	MM5,[AMOUNTS]
PSRLD xmmreg,xmmreg/mem128	PSRLD	XMM5,XMM2
PSRLD mmreg,mmreg/mem64	PSRLD	MM5,3
PSRLD xmmreg,xmmreg/mem128	PSRLD	XMM3,4
PSRLQ mmreg,mmreg/mem64	PSRLQ	MM5,[AMOUNTS]
PSRLQ xmmreg,xmmreg/mem128	PSRLQ	XMM5,XMM2
PSRLQ mmreg,mmreg/mem64	PSRLQ	MM5,3
PSRLQ xmmreg,xmmreg/mem128	PSRLQ	XMM3,4

B.305.1 Notes

When those instructions are used with immediate arguments, *PopAsm* checks if they are in 0–15, 0–31 and 0–63 intervals for PSRLW, PSRLD and PSRLQ, respectively.

B.306 PSUBB / PSUBW / PSUBD / PSUBQ — ADD Packed integers

Valid Syntaxes	Example	
PSUBB mmreg,mmreg/mem64	PSUBB	MM6,MM1
PSUBB xmmreg,xmmreg/mem128	PSUBB	XMM,ES:[BX+DI-32]
PSUBW mmreg,mmreg/mem64	PSUBW	MM6,MM1
PSUBW xmmreg,xmmreg/mem128	PSUBW	XMM,ES:[BX+DI-32]
PSUBD mmreg,mmreg/mem64	PSUBD	MM6,MM1
PSUBD xmmreg,xmmreg/mem128	PSUBD	XMM,ES:[BX+DI-32]
PSUBQ mmreg,mmreg/mem64	PSUBQ	MM6,MM1
PSUBQ xmmreg,xmmreg/mem128	PSUBQ	XMM,ES:[BX+DI-32]

B.307 PSUBSB / PSUBSW — SUBtract Packed Signed integers with signed saturation

Valid Syntaxes	Example
PSUBSB mmreg,mmreg/mem64	PSUBSB MM6,MM1
PSUBSB xmmreg,xmmreg/mem128	PSUBSB XMM,ES:[BX+DI-32]
PSUBSW mmreg,mmreg/mem64	PSUBSW MM6,MM1
PSUBSW xmmreg,xmmreg/mem128	PSUBSW XMM,ES:[BX+DI-32]

B.308 PSUBUSB / PSUBUSW — ADD Packed Unsigned integers with unsigned saturation

Valid Syntaxes	Example
PSUBUSB mmreg,mmreg/mem64	PSUBUSB MM6,MM1
PSUBUSB xmmreg,xmmreg/mem128	PSUBUSB XMM,ES:[BX+DI-32]
PSUBUSW mmreg,mmreg/mem64	PSUBUSW MM6,MM1
PSUBUSW xmmreg,xmmreg/mem128	PSUBUSW XMM,ES:[BX+DI-32]

B.309 PSWAPD — Packed SWAP Doubleword

Valid Syntaxes	Example
PSWAPD mmreg,mmreg/mem64	PSWAPD MM6,MM1

B.310 PUNPCKHBW / PUNPCKHWD / PUNPCKHDQ / PUNPCKHQDQ — UNPaCK High data

Valid Syntaxes		Example	
PUNPCKHBW	mmreg,mmreg/mem64	PUNPCKHBW	MM2,[SAMPLES]
PUNPCKHBW	xmmreg,xmmreg/mem128	PUNPCKHBW	XMM0,XMM6
PUNPCKHWD	mmreg,mmreg/mem64	PUNPCKHWD	MM2,[SAMPLES]
PUNPCKHWD	xmmreg,xmmreg/mem128	PUNPCKHWD	XMM0,XMM6
PUNPCKHDQ	mmreg,mmreg/mem64	PUNPCKHDQ	MM2,[SAMPLES]
PUNPCKHDQ	xmmreg,xmmreg/mem128	PUNPCKHDQ	XMM0,XMM6
PUNPCKHQDQ	xmmreg,xmmreg/mem128	PUNPCKHQDQ	XMM0,XMM6

B.311 PUNPCKLBW / PUNPCKLWD / PUNPCKLDQ / PUNPCKLQDQ — UNPaCK Low data

Valid Syntaxes		Example	
PUNPCKLBW	mmreg,mmreg/mem64	PUNPCKLBW	MM2,[SAMPLES]
PUNPCKLBW	xmmreg,xmmreg/mem128	PUNPCKLBW	XMM0,XMM6
PUNPCKLWD	mmreg,mmreg/mem64	PUNPCKLWD	MM2,[SAMPLES]
PUNPCKLWD	xmmreg,xmmreg/mem128	PUNPCKLWD	XMM0,XMM6
PUNPCKLDQ	mmreg,mmreg/mem64	PUNPCKLDQ	MM2,[SAMPLES]
PUNPCKLDQ	xmmreg,xmmreg/mem128	PUNPCKLDQ	XMM0,XMM6
PUNPCKLQDQ	xmmreg,xmmreg/mem128	PUNPCKLQDQ	XMM0,XMM6

B.312 PUSH — PUSH word or doubleword onto the stack

Valid Syntaxes	Example
PUSH reg16/reg32/mem16/mem32	PUSH EBP
PUSH signed8	PUSH -5
PUSH immed16/immed32	PUSH OFFSET BUFFER
PUSH segreg	PUSH DS

B.312.1 Notes

The short form for this instruction is used whenever possible. However, the longer form can be forced by specifying the argument size. Likewise, a 32-bits PUSH can be issued in 16-bits mode and vice-versa by means of size specifiers. Finally, immediate values larger that 0FFFFh must be preceeded by a DWORD size specifier when in 16-bits mode. All of this is shown in the mext example:

```
PUSH    5                       ; 8-bits
PUSH    BYTE 5                  ; same as above

BITS    16

PUSH    1234h                   ; 16-bits
PUSH    DWORD 1234h             ; 32-bits
```

```
PUSH    10000h                  ; Error!
PUSH    DWORD 10000h            ; Ok, 32-bits

BITS    32

PUSH    1234h                   ; 32-bits
PUSH    WORD 1234h              ; 16-bits
```

B.313 PUSHA / PUSHAD — PUSH All general-purpose registers

Valid Syntaxes	Example
PUSHA (no arguments)	PUSHA
PUSHAD (no arguments)	PUSHAD

B.314 PUSHF / PUSHFD — PUSH stack into eFlags register

Valid Syntaxes	Example
PUSHF (no arguments)	PUSHF
PUSHFD (no arguments)	PUSHFD

B.315 PXOR — Logical XOR

Valid Syntaxes	Example
PXOR mmreg,mmreg/mem64	PXOR MM6,MM1
PXOR xmmreg,xmmreg/mem128	PXOR XMM,ES:[BX+DI-32]

B.316 RCL / RCR / ROL / ROR — Rotate

Valid Syntaxes	Example
RCL reg8/mem8,unsigned8	RCL AL,1
RCL reg16/mem16,unsigned8	RCL BX,3
RCL reg32/mem32,unsigned8	RCL EDX,3
RCL reg8/mem8,CL	RCL AL,CL

```
RCL reg16/mem16,CL          RCL     BX,CL
RCL reg32/mem32,CL          RCL     EDX,CL
RCR reg8/mem8,unsigned8     RCR     AL,1
RCR reg16/mem16,unsigned8   RCR     BX,3
RCR reg32/mem32,unsigned8   RCR     EDX,3
RCR reg8/mem8,CL            RCR     AL,CL
RCR reg16/mem16,CL          RCR     BX,CL
RCR reg32/mem32,CL          RCR     EDX,CL
ROL reg8/mem8,unsigned8     ROL     AL,1
ROL reg16/mem16,unsigned8   ROL     BX,3
ROL reg32/mem32,unsigned8   ROL     EDX,3
ROL reg8/mem8,CL            ROL     AL,CL
ROL reg16/mem16,CL          ROL     BX,CL
ROL reg32/mem32,CL          ROL     EDX,CL
ROR reg8/mem8,unsigned8     ROR     AL,1
ROR reg16/mem16,unsigned8   ROR     BX,3
ROR reg32/mem32,unsigned8   ROR     EDX,3
ROR reg8/mem8,CL            ROR     AL,CL
ROR reg16/mem16,CL          ROR     BX,CL
ROR reg32/mem32,CL          ROR     EDX,CL
```

B.316.1 Notes

When the immediate argument is 1, there is an optimized encoding for the instruction, which is used by default. Such behavior may be overriden by a size specifier. For example:

```
RCL     AX,1
RCL     AX,BYTE 1
```

Also, the value of the argument is checked against the size of the first argument. If it is greater than or equal, an error message is issued.

B.317 RCPPS — compute ReCiProcals of Packed Single-precision floating-point values

Valid Syntaxes	Example
RCPPS xmmreg,xmmreg/mem128	RCPPS XMM3,[ES:VALUES]

B.318 RCPSS — compute ReCiProcals of Scalar Single-precision floating-point values

Valid Syntaxes	Example
RCPSS xmmreg,xmmreg/mem32	RCPSS XMM3,[DISTANCE]

B.319 RDMSR — ReaD from Model Specific Register

Valid Syntaxes	Example
RDMSR (no arguments)	RDMSR

B.320 RDPMC — ReaD from Performance Monitoring Counters

Valid Syntaxes	Example
RDPMC (no arguments)	RDPMC

B.321 RDTSC — ReaD from Time Stamp Counter

Valid Syntaxes	Example
RDTSC (no arguments)	RDTSC

B.322 REP / REPE / REPZ / REPNE / REPNZ — REPeat string operation prefix

Valid Syntaxes	Example

REP (no arguments)	REP	
REP instruction	REP	STOSB
REPE (no arguments)	REPE	
REPE instruction	REPE	CMPSB
REPZ (no arguments)	REPZ	
REPZ instruction	REPZ	SCASB
REPNE (no arguments)	REPNE	
REPNE instruction	REPNE	SCASD
REPNZ (no arguments)	REPNZ	
REPNZ instruction	REPNZ	SCASW

B.323 RET / RETN / RETF — RETurn from procedure

Valid Syntaxes	Example	
RET (no arguments)	RET	
RET unsigned16	RET	16
RETN (no arguments)	RETN	
RETN unsigned16	RETN	256
RETF (no arguments)	RETF	
RETF unsigned16	RETF	4

B.323.1 Notes

PopAsm **replaces RET instructions for either RETN or RETF depending on local context (i.e. the procedure being exitted is NEAR or FAR). If the instruction is used outside a procedure, its NEAR form is used. The developer may also use the explicit forms RETN inside far procedures and RETF inside near procedures as necessary.**

B.324 RSM — Resume from System Management mode

Valid Syntaxes	Example
RSM (no arguments)	RSM

B.325 RSQRTPS — compute Reciprocals of SQuare RooTs of Packed Single-precision floating-point values

Valid Syntaxes	Example
RSQRTPS xmmreg,xmmreg/mem128	RSQRTPS XMM3,[ES:VALUES]

B.326 RSQRTSS — compute ReCiProcals of SQuare RooTs of Scalar Single-precision floating-point values

Valid Syntaxes	Example
RSQRTSS xmmreg,xmmreg/mem32	RSQRTSS XMM3,[DISTANCES]

B.327 SAHF — Store AH into Flags

Valid Syntaxes	Example
SAHF (no arguments)	SAHF

B.328 SHL / SHR / SAL / SAR — Shift

Valid Syntaxes	Example
SHL reg8/mem8,unsigned8	SHL AL,1
SHL reg16/mem16,unsigned8	SHL BX,3
SHL reg32/mem32,unsigned8	SHL EDX,3
SHL reg8/mem8,CL	SHL AL,CL
SHL reg16/mem16,CL	SHL BX,CL
SHL reg32/mem32,CL	SHL EDX,CL
SHR reg8/mem8,unsigned8	SHR AL,1
SHR reg16/mem16,unsigned8	SHR BX,3
SHR reg32/mem32,unsigned8	SHR EDX,3
SHR reg8/mem8,CL	SHR AL,CL
SHR reg16/mem16,CL	SHR BX,CL
SHR reg32/mem32,CL	SHR EDX,CL
SAL reg8/mem8,unsigned8	SAL AL,1

```
SAL reg16/mem16,unsigned8    SAL     BX,3
SAL reg32/mem32,unsigned8    SAL     EDX,3
SAL reg8/mem8,CL             SAL     AL,CL
SAL reg16/mem16,CL           SAL     BX,CL
SAL reg32/mem32,CL           SAL     EDX,CL
SAR reg8/mem8,unsigned8      SAR     AL,1
SAR reg16/mem16,unsigned8    SAR     BX,3
SAR reg32/mem32,unsigned8    SAR     EDX,3
SAR reg8/mem8,CL             SAR     AL,CL
SAR reg16/mem16,CL           SAR     BX,CL
SAR reg32/mem32,CL           SAR     EDX,CL
```

B.328.1 Notes

When the immediate argument is 1, there is an optimized encoding for the instruction, which is used by default. Such behavior may be overriden by a size specifier. For example:

```
SHL     AX,1
SHL     AX,BYTE 1
```

Also, the value of the argument is checked against the size of the first argument. If it is greater than or equal, an error message is issued.

B.329 SBB — integer SuBtraction with Borrow

Valid Syntaxes	Example	
SBB reg,reg/mem	SBB	CL,DH
SBB mem,reg	SBB	[80h],EDX
SBB reg/mem,immed	SBB	[VAR],18
SBB reg/mem,signed8	SBB	EAX,-6

B.330 SCAS / SCASB / SCASW / SCASD — SCAn String

Valid Syntaxes	Example

```
SCAS mem8/mem16/mem32        SCAS    BYTE ES:[ESI]
SCASB (no arguments)         SCASB
SCASW (no arguments)         SCASW
SCASD (no arguments)         SCASD
```

B.331 SETcc — SET byte on condition

Valid Syntaxes	Example
`SETCC reg8/mem8`	`SETCC DH`

B.331.1 Notes

SETcc refers to a set of instructions that check different conditions. Suported variants are:

- **SETA — SET if Above**
- **SETAE — SET if Above or Equal**
- **SETB — SET if Below**
- **SETBE — SET if Below or Equal**
- **SETC — SET if Carry**
- **SETE — SET if Equal**
- **SETG — SET if Greater**
- **SETGE — SET if Greater or Equal**
- **SETL — SET if Less**
- **SETLE — SET if Less or Equal**
- **SETNA — SET if Not Above**
- **SETNAE — SET if Not Above or Equal**
- **SETNB — SET if Not Below**
- **SETNBE — SET if Not Below or Equal**
- **SETNC — SET if Not Carry**

- **SETNE — SET if Not Equal**
- **SETNG — SET if Not Greater**
- **SETNGE — SET if Not Greater or Equal**
- **SETNL — SET if Not Less**
- **SETNLE — SET if Not Less or Equal**
- **SETNO — SET if Not Overflow**
- **SETNP — SET if Not Parity**
- **SETNS — SET if Not Sign**
- **SETNZ — SET if Not Zero**

B.332 SFENCE — Store FENCE

Valid Syntaxes	Example
SFENCE (no arguments)	SFENCE

B.333 SGDT — Store Global Descriptor Table register

Valid Syntaxes	Example
SGDT mem48	SGDT [INITIAL_GDTR]

B.334 SIDT — Store Interrupt Descriptor Table register

Valid Syntaxes	Example
SIDT mem48	SGDT [INITIAL_IDTR]

B.335 SHLD / SHRD — Double-precision SHift

Valid Syntaxes	Example
SHLD reg16/mem16,reg16,unsigned8	SHLD AX,BX,1
SHLD reg16/mem32,reg32,unsigned8	SHLD EAX,EBX,1
SHLD reg16/mem16,reg16,CL	SHLD AX,BX,CL
SHLD reg16/mem32,reg32,CL	SHLD EAX,EBX,CL
SHRD reg16/mem16,reg16,unsigned8	SHRD AX,BX,1
SHRD reg16/mem32,reg32,unsigned8	SHRD EAX,EBX,1
SHRD reg16/mem16,reg16,CL	SHRD AX,BX,CL
SHRD reg16/mem32,reg32,CL	SHRD EAX,EBX,CL

B.335.1 Notes

If the last argument is an immediate value, *PopAsm* checks whether it is less than the other arguments' sizes. If not, an error message is issued.

B.336 SHUFPD — SHUFfle Packed Double-word floating-point values

Valid Syntaxes	Example
SHUFPD xmmreg,xmmreg/mem128,unsigned8	SHUFPD XMM2,XMM6,1

B.336.1 Notes

***PopAsm* checks whether the last argument is in 0–3 interval. If it is not an error message is displayed.**

B.337 SHUFPS — SHUFfle Packed Single-precision floating-point values

Valid Syntaxes	Example
SHUFPS xmmreg,xmmreg/mem128,unsigned8	SHUFPD XMM2,XMM6,10_01_11_00B

B.338 SLDT — Store Local Descriptor Table register

Valid Syntaxes	Example
SLDT reg16/mem16/reg32/mem32	SLDT DX

B.339 SMSW — Store Machine Status Word

Valid Syntaxes	Example
SMSW reg16/mem16/reg32	SMSW DX

B.340 SQRTPD — compute SQuare RooTs of Packed Double-precision floating-point values

Valid Syntaxes	Example
SQRTPD xmmreg,xmmreg/mem128	SQRTPD XMM2,XMM6

B.341 SQRTPS — compute SQuare RooTs of Packed Single-precision floating-point values

Valid Syntaxes	Example
SQRTPS xmmreg,xmmreg/mem128	SQRTPS XMM2,XMM6

B.342 SQRTSD — compute SQuare RooTs of Scalar Double-precision floating-point values

Valid Syntaxes	Example
SQRTSD xmmreg,xmmreg/mem64	SQRTSD XMM2,XMM6

B.343 SQRTSS — compute SQuare RooTs of Scalar Single-precision floating-point values

Valid Syntaxes	Example
SQRTSS xmmreg,xmmreg/mem32	SQRTSS XMM2,XMM6

B.344 STC — SeT Carry flag

Valid Syntaxes	Example
STC (no arguments)	STC

B.345 STD — SeT Direction flag

Valid Syntaxes	Example
STD (no arguments)	STD

B.346 STI — SeT Interrupt flag

Valid Syntaxes	Example
STI (no arguments)	STI

B.347 STMXCSR — STore MXCSR state

Valid Syntaxes	Example
STMXCSR (no arguments)	STMXCSR

B.348 STOS / STOSB / STOSW / STOSD — STOre String

Valid Syntaxes	Example
STOS mem8/mem16/mem32	STOS BYTE ES:[ESI]
STOSB (no arguments)	STOSB
STOSW (no arguments)	STOSW
STOSD (no arguments)	STOSD

B.349 STR — Store Task Register

Valid Syntaxes	Example
STR reg16/mem16	STR AX

B.350 SUB — SUBtract

Valid Syntaxes	Example
SUB reg,reg/mem	SUB CL,DH
SUB mem,reg	SUB [80h],EDX
SUB reg/mem,immed	SUB [VAR],18
SUB reg/mem,signed8	SUB EAX,-6

B.351 SUBPD — SUBtract Packed Double-precision floating-point values

Valid Syntaxes	Example
SUBPD xmmreg,xmmreg/mem128	SUBPD XMM,XMM1

B.352 SUBPS — SUBtract Packed Single-precision floating-point values

Valid Syntaxes	Example
SUBPS xmmreg,xmmreg/mem128	SUBPS XMM5,TABLE[ESI]

B.353 SUBSD — SUBtract Scalar Double-precision floating-point values

Valid Syntaxes	Example
SUBSD xmmreg,xmmreg/mem64	SUBSD XMM5,[EDI+ARRAY]

B.354 SUBSS — SUBtract Scalar Single-precision floating-point values

Valid Syntaxes	Example
SUBSS xmmreg,xmmreg/mem32	SUBSS XMM3,ES:[BX]

B.355 SYSENTER — fast SYStem call

Valid Syntaxes	Example
SYSENTER (no arguments)	SYSENTER

B.356 SYSEXIT — fast return from fast SYStem call

Valid Syntaxes	Example
SYSEXIT (no arguments)	SYSENTER

B.357 TEST — logical compare

Valid Syntaxes	Example
TEST reg,reg/mem	TEST AX,BX
TEST reg/mem,reg	TEST [80h],DL
TEST reg/mem,immed	TEST BYTE [GS:WORD_VAR],18
TEST reg/mem,signed8	TEST ESI,-6

B.357.1 Notes

Conversely to what has been stated above, this instruction has no suport for the "reg,mem" argument combination. However, because the order of the arguments does not affect the result, *PopAsm* inverts that argument combination so that the instruction can be assembled correctly.

B.358 UCOMISD — Unordered COMpare Scalar Double-precision floating-point values and set eflags

Valid Syntaxes	Example
UCOMISD xmmreg,xmmreg/mem64	UCOMISD XMM5,[EDI+ARRAY]

B.359 UCOMISS — Unordered COMpare Scalar Single-precision floating-point values

Valid Syntaxes	Example
UCOMISS xmmreg,xmmreg/mem32	UCOMISS XMM3,ES:[BX]

B.360 UD2 — UnDefined instruction

Valid Syntaxes	Example
UD2 (no arguments)	UD2

B.361 UNPCKHPD — UNPaCK and interleave High Packed Double-precision floating-point values

Valid Syntaxes	Example
UNPCKHPD xmmreg,xmmreg/mem128	UNPCKHPD XMM3,ES:[BX]

B.362 UNPCKHPS — UNPaCK and interleave High Packed Single-precision floating-point values

Valid Syntaxes	Example
UNPCKHPS xmmreg,xmmreg/mem128	UNPCKHPS XMM3,XMM1

B.363 UNPCKLPD — UNPaCK and interleave Low Packed Double-precision floating-point values

Valid Syntaxes	Example
UNPCKLPD xmmreg,xmmreg/mem128	UNPCKLPD XMM3,ES:[BX]

B.364 UNPCKLPS — UNPaCK and interleave Low Packed Single-precision floating-point values

Valid Syntaxes	Example
UNPCKLPS xmmreg,xmmreg/mem128	UNPCKLPS XMM3,XMM1

B.365 VERR / VERW — VERify a segment for Reading / Writing

Valid Syntaxes	Example
VERR reg16/mem16	VERR AX
VERW reg16/mem16	VERR CX

B.366 WAIT — WAIT

Valid Syntaxes	Example
WAIT (no arguments)	WAIT

B.367 WBINVD — Write Back and INVali-Date cache

Valid Syntaxes	Example
WBINVD (no arguments)	WBINVD

B.368 WRMSR — WRite to Model Specific Register

Valid Syntaxes	Example
WRMSR (no arguments)	WRMSR

B.369 XADD — eXchange and ADD

Valid Syntaxes	Example
XADD reg8/mem8,reg8	XADD AL,BL
XADD reg16/mem16,reg16	XADD AX,BX
XADD reg32/mem32,reg32	XADD EAX,EBX

B.370 XCHG — eXCHanGe

Valid Syntaxes	Example
XCHG reg8/mem8,reg8	XCHG AL,BL
XCHG reg8,reg8/mem8	XCHG AL,[BX]
XCHG reg16/mem16,reg16	XCHG AX,BX
XCHG reg16/mem16,reg16	XCHG AL,[BX]
XCHG reg32/mem32,reg32	XCHG EAX,EBX
XCHG reg32,reg32/mem32	XCHG EAX,[EBX]

B.371 XLAT / XLATB — table look-up translation

Valid Syntaxes	Example
XLAT mem8	XLAT BYTE ES:[EBX]
XLATB (no arguments)	XLATB

B.372 XOR — logical eXclusive OR

Valid Syntaxes	Example
XOR reg,reg/mem	XOR AX,BX
XOR mem,reg	XOR [80h],DL
XOR reg/mem,immed	XOR BYTE [GS:WORD_VAR],18
XOR reg/mem,signed8	XOR ESI,-6

B.373 XORPD — bitwise logical eXclusive OR for Packed Double-precision floating-point values

Valid Syntaxes	Example
`XORPD xmmreg,xmmreg/mem128`	`XORPD XMM7,[EAX]`

B.374 XORPS — bitwise logical eXclusive OR for Packed Single-precision floating-point values

Valid Syntaxes	Example
`XORPS xmmreg,xmmreg/mem128`	`XORPS XMM0,[ESI*4]`

Appendix C

Contacting info

General considerations

I am glad to hear from anyone about any subject related to this project. This includes suggestions, bug reports, doubts, feedback, feature requests and even criticisms (provided they are not offensive and follow the basic ettiquetes guidelines).

Note, however, that depending on the mail volume I may currently be receiving, it may me difficult to reply your message quickly. I can only say that I'll do my best.

If you want to ask a question...

Please, check if your question is already listed in the online FAQ. If it is, both of us save time.

Please do not ask questions about assembly language itself, unless they are related to *PopAsm* (eg. why does *PopAsm* encodes ADD instructions this way, not that way?) and are not covered in any of the manuals (see Download Center).

If you want to make a feature request...

Feel free to request implementation of any feature you believe to be useful (assembly directives, new instructions, command-line options, etc), specially if it increases compatibility with existing code and/or other assemblers.

If you want to make a comment...

Comments are always welcome, both praises and criticisms. My favorite comments are those that can provide some sort of feedback, but other comments are appreciated as well.

If you want to submit a bug report...

Please check for the latest version of *PopAsm* at the Download Center. The bug you have found may have already been fixed. If not, please:

- Be as clear as possible in your description, including the behavior you expected and the one you got.
- If possible, attach a piece of code (the smaller the better) where the bug occurs.
- Information such as *PopAsm* version, operating system, etc.

If you are not sure about whether or not to contact me

In that case you should do it. I'd rather get a few more mails than running the risk of not hearing a possibly useful comment.

Contact info

Great, if you did what I ask you above please go ahead. You can mail me at helcio@users.sourceforge.net. Thanks!

Bibliography

[1] **Federal University of Espírito Santo – ES – Brazil. http://www.ufes.br**

[2] **Pontificial Catholic University of Rio de Janeiro – RJ – Brazil. http://www.puc-rio.br**

[3] **SourceForge. http://sorceforge.net**

[4] **Advogato. http://www.advogato.com**

[5] **Hélcio Mello's personal page at Advogato. http://www.advogato.com/person/helcio**

[6] **FreshMeat. http://www.freahmeat.net**

[7] **Professor PhD Sérgio A. A. Freitas. http://www.inf.ufes.br/~sergio.**

[8] **GNU Autoconf. http://www.gnu.org/software/autoconf/autoconf.html**

[9] **GNU Automake. http://www.gnu.org/software/automake/automake.html**

[10] **GNU Make. http://www.gnu.org/software/make/make.html**

[11] **FPU Programming on x86 FPU's**