NASM: The Netwide Assembler

This file documents NASM, the Netwide Assembler: an assembler targetting the Intel x86 series of processors, with portable source.

Chapter 1: Introduction

1.1. What Is NASM?

The Netwide Assembler, NASM, is an 80x86 assembler designed for portability and modularity. It supports a range of object file formats, including Linux `a.out` and ELF, NetBSD/FreeBSD, COFF, Microsoft 16-bit OBJ and Win32. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex. It supports Pentium, P6 and MMX opcodes, and has macro capability.

Section 1.1.1: Why Yet Another Assembler?
Section 1.1.2: Licence Conditions

1.1.1. Why Yet Another Assembler?

The Netwide Assembler grew out of an idea on `comp.lang.asm.x86` (or possibly `alt.lang.asm` – I forget which), which was essentially that there didn't seem to be a good free x86-series assembler around, and that maybe someone ought to write one.

- `a86` is good, but not free, and in particular you don't get any 32-bit capability until you pay. It's DOS only, too.
- `gas` is free, and ports over DOS and Unix, but it's not very good, since it's designed to be a back end to `gcc`, which always feeds it correct code. So its error checking is minimal. Also, its syntax is horrible, from the point of view of anyone trying to actually *write* anything in it. Plus you can't write 16-bit code in it (properly).
- `as86` is Linux-specific, and (my version at least) doesn't seem to have much (or any) documentation.
- MASM isn't very good, and it's expensive, and it runs only under DOS.
- TASM is better, but still strives for MASM compatibility, which means millions of directives and tons of red tape. And its syntax is essentially MASM's, with the contradictions and quirks that entails (although it sorts out some of those by means of Ideal mode). It's expensive too. And it's DOS-only.

So here, for your coding pleasure, is NASM. At present it's still in prototype stage – we don't promise that it can outperform any of these assemblers. But please, *please* send us bug reports, fixes, helpful information, and anything else you can get your hands on (and thanks to the many people who've done this already! You all know who you are), and we'll improve it out of all recognition. Again.

1.1.2. Licence Conditions

Please see the file `Licence`, supplied as part of any NASM distribution archive, for the licence conditions under which you may use NASM.

1.2. Contact Information

NASM has a WWW page at `http://www.cryogen.com/Nasm`. The authors are emailable as `jules@earthcorp.com` and `anakin@pobox.com`. If you want to report a bug to us, please read section 10.2 first.

New releases of NASM are uploaded to `sunsite.unc.edu`, `ftp.simtel.net` and `ftp.coast.net`. Announcements are posted to `comp.lang.asm.x86`, `alt.lang.asm`, `comp.os.linux.announce` and `comp.archives.msdos.announce` (the last one is done automagically by uploading to `ftp.simtel.net`).

If you don't have Usenet access, or would rather be informed by email when new releases come out, email `anakin@pobox.com` and ask.

1.3. Installation

1.3.1. Installing NASM under MS-DOS or Windows

Once you've obtained the DOS archive for NASM, `nasmXXX.zip` (where `XXX` denotes the version number of NASM contained in the archive), unpack it into its own directory (for example `c:\nasm`).

The archive will contain four executable files: the NASM executable files `nasm.exe` and `nasmw.exe`, and the NDISASM executable files `ndisasm.exe` and `ndisasmw.exe`. In each case, the file whose name ends in `w` is a Win32 executable, designed to run under Windows 95 or Windows NT Intel, and the other one is a 16-bit DOS executable.

The only file NASM needs to run is its own executable, so copy (at least) one of `nasm.exe` and `nasmw.exe` to a directory on your PATH, or alternatively edit `autoexec.bat` to add the `nasm` directory to your `PATH`. (If you're only installing the Win32 version, you may wish to rename it to `nasm.exe`.)

That's it – NASM is installed. You don't need the `nasm` directory to be present to run NASM (unless you've added it to your `PATH`), so you can delete it if you need to save space; however, you may want to keep the documentation or test programs.

If you've downloaded the DOS source archive, `nasmXXXs.zip`, the `nasm` directory will also contain the full NASM source code, and a selection of Makefiles you can (hopefully) use to rebuild your copy of NASM from scratch. The file `Readme` lists the various Makefiles and which compilers they work with. Note that the source files `insnsa.c` and `insnsd.c` are automatically generated from the master instruction table `insns.dat` by a Perl script; a QBasic version of the program is provided, but it is recommended that you use the Perl version. A DOS port of Perl is available from www.perl.org.

1.3.2. Installing NASM under Unix

Once you've obtained the Unix source archive for NASM, `nasmX.XX.tar.gz` (where `X.XX` denotes the version number of NASM contained in the archive), unpack it into a directory such as `/usr/local/src`. The archive, when unpacked, will create its own subdirectory `nasm-X.XX`.

NASM is an auto-configuring package: once you've unpacked it, `cd` to the directory it's been unpacked into and type `./configure`. This shell script will find the best C compiler to use for building NASM and set up Makefiles accordingly.

Once NASM has auto-configured, you can type `make` to build the `nasm` and `ndisasm` binaries, and then `make install` to install them in `/usr/local/bin` and install the man pages `nasm.1` and `ndisasm.1` in `/usr/local/man/man1`. Alternatively, you can give options such as `prefix` to the `configure` script (see the file `INSTALL` for more details), or install the programs yourself.

NASM also comes with a set of utilities for handling the RDOFF custom object-file format, which are in the `rdoff` subdirectory of the NASM archive. You can build these with `make rdf` and install them with `make rdf_install`, if you want them.

If NASM fails to auto-configure, you may still be able to make it compile by using the fall-back Unix makefile `Makefile.unx`. Copy or rename that file to `Makefile` and try typing `make`. There is also a `Makefile.unx` file in the `rdoff` subdirectory.

Chapter 2: Running NASM

2.1. NASM CommandLine Syntax

To assemble a file, you issue a command of the form

```
nasm -f <format> <filename> [-o <output>]
```

For example,

```
nasm -f elf myfile.asm
```

will assemble `myfile.asm` into an ELF object file `myfile.o`. And

```
nasm -f bin myfile.asm -o myfile.com
```

will assemble `myfile.asm` into a raw binary file `myfile.com`.

To produce a listing file, with the hex codes output from NASM displayed on the left of the original sources, use the `l` option to give a listing file name, for example:

```
nasm -f coff myfile.asm -l myfile.lst
```

To get further usage instructions from NASM, try typing

```
nasm -h
```

This will also list the available output file formats, and what they are.

If you use Linux but aren't sure whether your system is `a.out` or ELF, type

```
file nasm
```

(in the directory in which you put the NASM binary when you installed it). If it says something like

```
nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1
```

then your system is ELF, and you should use the option `f elf` when you want NASM to produce Linux object files. If it says

```
nasm: Linux/i386 demand-paged executable (QMAGIC)
```

or something similar, your system is `a.out`, and you should use `f aout` instead.

Like Unix compilers and assemblers, NASM is silent unless it goes wrong: you won't see any output at all, unless it gives error messages.

2.1.1. The o Option: Specifying the Output File Name

NASM will normally choose the name of your output file for you; precisely how it does this is dependent on the object file format. For Microsoft object file formats (`obj` and `win32`), it will remove the `.asm` extension (or whatever extension you like to use – NASM doesn't care) from your source file name and substitute `.obj`. For Unix object file formats (`aout`, `coff`, `elf` and `as86`) it will substitute `.o`. For `rdf`, it will use `.rdf`, and for the `bin` format it will simply remove the extension, so that `myfile.asm` produces the output file `myfile`.

If the output file already exists, NASM will overwrite it, unless it has the same name as the input file, in which case it will give a warning and use `nasm.out` as the output file name instead.

For situations in which this behaviour is unacceptable, NASM provides the `o` command-line option, which allows you to specify your desired output file name. You invoke `o` by following it with the name you wish for the output file, either with or without an intervening space. For example:

```
nasm -f bin program.asm -o program.com
nasm -f bin driver.asm -odriver.sys
```

2.1.2. The `f` Option: Specifying the Output File Format

If you do not supply the `f` option to NASM, it will choose an output file format for you itself. In the distribution versions of NASM, the default is always `bin`; if you've compiled your own copy of NASM, you can redefine `OF_DEFAULT` at compile time and choose what you want the default to be.

Like `o`, the intervening space between `f` and the output file format is optional; so `f elf` and `-felf` are both valid.

A complete list of the available output file formats can be given by issuing the command `nasm h`.

2.1.3. The `l` Option: Generating a Listing File

If you supply the `l` option to NASM, followed (with the usual optional space) by a file name, NASM will generate a sourcelisting file for you, in which addresses and generated code are listed on the left, and the actual source code, with expansions of multi-line macros (except those which specifically request no expansion in source listings: see section 4.2.9) on the right. For example:

```
nasm -f elf myfile.asm -l myfile.lst
```

2.1.4. The `s` Option: Send Errors to `stdout`

Under MS-DOS it can be difficult (though there are ways) to redirect the standard-error output of a program to a file. Since NASM usually produces its warning and error messages on `stderr`, this can make it hard to capture the errors if (for example) you want to load them into an editor.

NASM therefore provides the `s` option, requiring no argument, which causes errors to be sent to standard output rather than standard error. Therefore you can redirect the errors into a file by typing

```
nasm -s -f obj myfile.asm > myfile.err
```

2.1.5. The `i` Option: Include File Search Directories

When NASM sees the `%include` directive in a source file (see section 4.5), it will search for the given file not only in the current directory, but also in any directories specified on the command line by the use of the `i` option. Therefore you can include files from a macro library, for example, by typing

```
nasm -ic:\macrolib\ -f obj myfile.asm
```

(As usual, a space between `i` and the path name is allowed, and optional).

NASM, in the interests of complete source-code portability, does not understand the file naming conventions of the OS it is running on; the string you provide as an argument to the `i` option will be prepended exactly as written to the name of the include file. Therefore the trailing backslash in the above example is necessary. Under Unix, a trailing forward slash is similarly necessary.

(You can use this to your advantage, if you're really perverse, by noting that the option `ifoo` will cause `%include "bar.i"` to search for the file `foobar.i`...)

If you want to define a *standard* include search path, similar to `/usr/include` on Unix systems, you should place one or more `i` directives in the `NASM` environment variable (see section 2.1.11).

2.1.6. The `p` Option: Pre-Include a File

NASM allows you to specify files to be *preincluded* into your source file, by the use of the `p` option. So running

```
nasm myfile.asm -p myinc.inc
```

is equivalent to running `nasm myfile.asm` and placing the directive `%include "myinc.inc"` at the start of the file.

2.1.7. The d Option: Pre-Define a Macro

Just as the p option gives an alternative to placing %include directives at the start of a source file, the d option gives an alternative to placing a %define directive. You could code

```
nasm myfile.asm -dFOO=100
```

as an alternative to placing the directive

```
%define FOO 100
```

at the start of the file. You can miss off the macro value, as well: the option dFOO is equivalent to coding %define FOO. This form of the directive may be useful for selecting assemblytime options which are then tested using %ifdef, for example dDEBUG.

2.1.8. The `e` Option: Preprocess Only

NASM allows the preprocessor to be run on its own, up to a point. Using the `e` option (which requires no arguments) will cause NASM to preprocess its input file, expand all the macro references, remove all the comments and preprocessor directives, and print the resulting file on standard output (or save it to a file, if the `o` option is also used).

This option cannot be applied to programs which require the preprocessor to evaluate expressions which depend on the values of symbols: so code such as

```
%assign tablesize ($-tablestart)
```

will cause an error in preprocessonly mode.

2.1.9. The `a` Option: Don't Preprocess At All

If NASM is being used as the back end to a compiler, it might be desirable to suppress preprocessing completely and assume the compiler has already done it, to save time and increase compilation speeds. The `a` option, requiring no argument, instructs NASM to replace its powerful preprocessor with a stub preprocessor which does nothing.

2.1.10. The `w` Option: Enable or Disable Assembly Warnings

NASM can observe many conditions during the course of assembly which are worth mentioning to the user, but not a sufficiently severe error to justify NASM refusing to generate an output file. These conditions are reported like errors, but come up with the word `warning' before the message. Warnings do not prevent NASM from generating an output file and returning a success status to the operating system.

Some conditions are even less severe than that: they are only sometimes worth mentioning to the user. Therefore NASM supports the `w` command-line option, which enables or disables certain classes of assembly warning. Such warning classes are described by a name, for example `orphanlabels`; you can enable warnings of this class by the command-line option `-w+orphanlabels` and disable it by `worphanlabels`.

The suppressible warning classes are:

- `macroparams` covers warnings about multiline macros being invoked with the wrong number of parameters. This warning class is enabled by default; see section 4.2.1 for an example of why you might want to disable it.
- `orphanlabels` covers warnings about source lines which contain no instruction but define a label without a trailing colon. NASM does not warn about this somewhat obscure condition by default; see section 3.1 for an example of why you might want it to.
- `numberoverflow` covers warnings about numeric constants which don't fit in 32 bits (for example, it's easy to type one too many Fs and produce `0x7fffffff` by mistake). This warning class is enabled by default.

2.1.11. The NASM Environment Variable

If you define an environment variable called `NASM`, the program will interpret it as a list of extra command-line options, which are processed before the real command line. You can use this to define standard search directories for include files, by putting `i` options in the `NASM` variable.

The value of the variable is split up at white space, so that the value `s ic:\nasmlib` will be treated as two separate options. However, that means that the value `dNAME="my name"` won't do what you might want, because it will be split at the space and the NASM command-line processing will get confused by the two nonsensical words `dNAME="my` and `name"`.

To get round this, NASM provides a feature whereby, if you begin the `NASM` environment variable with some character that isn't a minus sign, then NASM will treat this character as the separator character for options. So setting the `NASM` variable to the value `!s!ic:\nasmlib` is equivalent to setting it to `s ic:\nasmlib`, but `!dNAME="my name"` will work.

2.2. Quick Start for MASM Users

If you're used to writing programs with MASM, or with TASM in MASM-compatible (non-Ideal) mode, or with `a86`, this section attempts to outline the major differences between MASM's syntax and NASM's. If you're not already used to MASM, it's probably worth skipping this section.

2.2.1. NASM Is Case-Sensitive

One simple difference is that NASM is case-sensitive. It makes a difference whether you call your label `foo`, `Foo` or `FOO`. If you're assembling to DOS or OS/2 `.OBJ` files, you can invoke the `UPPERCASE` directive (documented in section 6.2) to ensure that all symbols exported to other code modules are forced to be upper case; but even then, *within* a single module, NASM will distinguish between labels differing only in case.

2.2.2. NASM Requires Square Brackets For Memory References

NASM was designed with simplicity of syntax in mind. One of the design goals of NASM is that it should be possible, as far as is practical, for the user to look at a single line of NASM code and tell what opcode is generated by it. You can't do this in MASM: if you declare, for example,

```
foo        equ 1
bar        dw 2
```

then the two lines of code

```
           mov ax,foo
           mov ax,bar
```

generate completely different opcodes, despite having identical-looking syntaxes.

NASM avoids this undesirable situation by having a much simpler syntax for memory references. The rule is simply that any access to the *contents* of a memory location requires square brackets around the address, and any access to the *address* of a variable doesn't. So an instruction of the form `mov ax,foo` will *always* refer to a compile-time constant, whether it's an `EQU` or the address of a variable; and to access the *contents* of the variable `bar`, you must code `mov ax,[bar]`.

This also means that NASM has no need for MASM's `OFFSET` keyword, since the MASM code `mov ax,offset bar` means exactly the same thing as NASM's `mov ax,bar`. If you're trying to get large amounts of MASM code to assemble sensibly under NASM, you can always code `%idefine offset` to make the preprocessor treat the `OFFSET` keyword as a no-op.

This issue is even more confusing in `a86`, where declaring a label with a trailing colon defines it to be a `label' as opposed to a `variable' and causes `a86` to adopt NASM-style semantics; so in `a86`, `mov ax,var` has different behaviour depending on whether `var` was declared as `var: dw 0` (a label) or `var dw 0` (a word-size variable). NASM is very simple by comparison: *everything* is a label.

NASM, in the interests of simplicity, also does not support the hybrid syntaxes supported by MASM and its clones, such as `mov ax,table[bx]`, where a memory reference is denoted by one portion outside square brackets and another portion inside. The correct syntax for the above is `mov ax,[table+bx]`. Likewise, `mov ax,es:[di]` is wrong and `mov ax,[es:di]` is right.

2.2.3. NASM Doesn't Store Variable Types

NASM, by design, chooses not to remember the types of variables you declare. Whereas MASM will remember, on seeing `var dw 0`, that you declared `var` as a word-size variable, and will then be able to fill in the ambiguity in the size of the instruction `mov var,2`, NASM will deliberately remember nothing about the symbol `var` except where it begins, and so you must explicitly code `mov word [var],2`.

For this reason, NASM doesn't support the `LODS`, `MOVS`, `STOS`, `SCAS`, `CMPS`, `INS`, or `OUTS` instructions, but only supports the forms such as `LODSB`, `MOVSW`, and `SCASD`, which explicitly specify the size of the components of the strings being manipulated.

2.2.4. NASM Doesn't `ASSUME`

As part of NASM's drive for simplicity, it also does not support the `ASSUME` directive. NASM will not keep track of what values you choose to put in your segment registers, and will never *automatically* generate a segment override prefix.

2.2.5. NASM Doesn't Support Memory Models

NASM also does not have any directives to support different 16-bit memory models. The programmer has to keep track of which functions are supposed to be called with a far call and which with a near call, and is responsible for putting the correct form of `RET` instruction (`RETN` or `RETF`; NASM accepts `RET` itself as an alternate form for `RETN`); in addition, the programmer is responsible for coding CALL FAR instructions where necessary when calling *external* functions, and must also keep track of which external variable definitions are far and which are near.

2.2.6. FloatingPoint Differences

NASM uses different names to refer to floating-point registers from MASM: where MASM would call them `ST(0)`, `ST(1)` and so on, and `a86` would call them simply `0`, `1` and so on, NASM chooses to call them `st0`, `st1` etc.

As of version 0.96, NASM now treats the instructions with `nowait' forms in the same way as MASM-compatible assemblers. The idiosyncratic treatment employed by 0.95 and earlier was based on a misunderstanding by the authors.

2.2.7. Other Differences

For historical reasons, NASM uses the keyword `TWORD` where MASM and compatible assemblers use `TBYTE`.

NASM does not declare uninitialised storage in the same way as MASM: where a MASM programmer might use `stack db 64 dup (?)`, NASM requires `stack resb 64`, intended to be read as `reserve 64 bytes'. For a limited amount of compatibility, since NASM treats `?` as a valid character in symbol names, you can code `? equ 0` and then writing `dw ?` will at least do something vaguely useful. `DUP` is still not a supported syntax, however.

In addition to all of this, macros and directives work completely differently to MASM. See chapter 4 and chapter 5 for further details.

Chapter 3: The NASM Language

3.1. Layout of a NASM Source Line

Like most assemblers, each NASM source line contains (unless it is a macro, a preprocessor directive or an assembler directive: see chapter 4 and chapter 5) some combination of the four fields

```
label:    instruction operands        ; comment
```

As usual, most of these fields are optional; the presence or absence of any combination of a label, an instruction and a comment is allowed. Of course, the operand field is either required or forbidden by the presence and nature of the instruction field.

NASM places no restrictions on white space within a line: labels may have white space before them, or instructions may have no space before them, or anything. The colon after a label is also optional. (Note that this means that if you intend to code `lodsb` alone on a line, and type `lodab` by accident, then that's still a valid source line which does nothing but define a label. Running NASM with the command-line option `w+orphanlabels` will cause it to warn you if you define a label alone on a line without a trailing colon.)

Valid characters in labels are letters, numbers, `_`, `$`, `#`, `@`, `~`, `.`, and `?`. The only characters which may be used as the *first* character of an identifier are letters, `.` (with special meaning: see section 3.8), `_` and `?`. An identifier may also be prefixed with a `$` to indicate that it is intended to be read as an identifier and not a reserved word; thus, if some other module you are linking with defines a symbol called `eax`, you can refer to `$eax` in NASM code to distinguish the symbol from the register.

The instruction field may contain any machine instruction: Pentium and P6 instructions, FPU instructions, MMX instructions and even undocumented instructions are all supported. The instruction may be prefixed by `LOCK`, `REP`, `REPE/REPZ` or `REPNE/REPNZ`, in the usual way. Explicit address-size and operandsize prefixes `A16`, `A32`, `O16` and `O32` are provided – one example of their use is given in chapter 9. You can also use the name of a segment register as an instruction prefix: coding `es mov [bx],ax` is equivalent to coding `mov [es:bx],ax`. We recommend the latter syntax, since it is consistent with other syntactic features of the language, but for instructions such as `LODSB`, which has no operands and yet can require a segment override, there is no clean syntactic way to proceed apart from `es lodsb`.

An instruction is not required to use a prefix: prefixes such as `CS`, `A32`, `LOCK` or `REPE` can appear on a line by themselves, and NASM will just generate the prefix bytes.

In addition to actual machine instructions, NASM also supports a number of pseudo-instructions, described in section 3.2.

Instruction operands may take a number of forms: they can be registers, described simply by the register name (e.g. `ax`, `bp`, `ebx`, `cr0`: NASM does not use the `gas`–style syntax in which register names must be prefixed by a `%` sign), or they can be effective addresses (see section 3.3), constants (section 3.4) or expressions (section 3.5).

For floatingpoint instructions, NASM accepts a wide range of syntaxes: you can use two-operand forms like MASM supports, or you can use NASM's native single-operand forms in most cases. Details of all forms of each supported instruction are given in appendix A. For example, you can code:

```
fadd st1                ; this sets st0 := st0 + st1
fadd st0,st1            ; so does this

fadd st1,st0            ; this sets st1 := st1 + st0
fadd to st1             ; so does this
```

Almost any floating-point instruction that references memory must use one of the prefixes `DWORD`, `QWORD` or `TWORD` to indicate what size of memory operand it refers to.

3.2. PseudoInstructions

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudo-instructions are `DB`, `DW`, `DD`, `DQ` and `DT`, their uninitialised counterparts `RESB`, `RESW`, `RESD`, `RESQ` and `REST`, the `INCBIN` command, the `EQU` command, and the `TIMES` prefix.

3.2.1. DB and friends: Declaring Initialised Data

DB, DW, DD, DQ and DT are used, much as in MASM, to declare initialised data in the output file. They can be invoked in a wide range of ways:

```
db 0x55                ; just the byte 0x55
db 0x55,0x56,0x57      ; three bytes in succession
db 'a',0x55            ; character constants are OK
db 'hello',13,10,'$'   ; so are string constants
dw 0x1234              ; 0x34 0x12
dw 'a'                 ; 0x41 0x00 (it's just a number)
dw 'ab'                ; 0x41 0x42 (character constant)
dw 'abc'               ; 0x41 0x42 0x43 0x00 (string)
dd 0x12345678          ; 0x78 0x56 0x34 0x12
dd 1.234567e20         ; floating-point constant
dq 1.234567e20         ; double-precision float
dt 1.234567e20         ; extended-precision float
```

DQ and DT do not accept numeric constants or string constants as operands.

3.2.2. `RESB` and friends: Declaring Uninitialised Data

`RESB`, `RESW`, `RESD`, `RESQ` and `REST` are designed to be used in the BSS section of a module: they declare *uninitialised* storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. As stated in section 2.2.7, NASM does not support the MASM/TASM syntax of reserving uninitialised space by writing `DW ?` or similar things: this is what it does instead. The operand to a `RESB`–type pseudo-instruction is a *critical expression*: see section 3.7.

For example:

```
buffer:   resb 64               ; reserve 64 bytes
wordvar:  resw 1                ; reserve a word
realarray resq 10               ; array of ten reals
```

3.2.3. `INCBIN`: Including External Binary Files

`INCBIN` is borrowed from the old Amiga assembler DevPac: it includes a binary file verbatim into the output file. This can be handy for (for example) including graphics and sound data directly into a game executable file. It can be called in one of these three ways:

```
incbin "file.dat"      ; include the whole file
incbin "file.dat",1024 ; skip the first 1024 bytes
incbin "file.dat",1024,512 ; skip the first 1024, and
                           ; actually include at most 512
```

3.2.4. `EQU`: Defining Constants

`EQU` defines a symbol to a given constant value: when `EQU` is used, the source line must contain a label. The action of `EQU` is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. So, for example,

```
message   db 'hello, world'
msglen    equ $-message
```

defines `msglen` to be the constant 12. `msglen` may not then be redefined later. This is not a preprocessor definition either: the value of `msglen` is evaluated *once*, using the value of `$` (see section 3.5 for an explanation of `$`) at the point of definition, rather than being evaluated wherever it is referenced and using the value of `$` at the point of reference. Note that the operand to an `EQU` is also a critical expression (section 3.7).

3.2.5. `TIMES`: Repeating Instructions or Data

The `TIMES` prefix causes the instruction to be assembled multiple times. This is partly present as NASM's equivalent of the `DUP` syntax supported by MASM–compatible assemblers, in that you can code

```
zerobuf:  times 64 db 0
```

or similar things; but `TIMES` is more versatile than that. The argument to `TIMES` is not just a numeric constant, but a numeric *expression*, so you can do things like

```
buffer:   db 'hello, world'
          times 64-$+buffer db ' '
```

which will store exactly enough spaces to make the total length of `buffer` up to 64. Finally, `TIMES` can be applied to ordinary instructions, so you can code trivial unrolled loops in it:

```
          times 100 movsb
```

Note that there is no effective difference between `times 100 resb 1` and `resb 100`, except that the latter will be assembled about 100 times faster due to the internal structure of the assembler.

The operand to `TIMES`, like that of `EQU` and those of `RESB` and friends, is a critical expression (section 3.7).

Note also that `TIMES` can't be applied to macros: the reason for this is that `TIMES` is processed after the macro phase, which allows the argument to `TIMES` to contain expressions such as `64$ +buffer` as above. To repeat more than one line of code, or a complex macro, use the preprocessor `%rep` directive.

3.3. Effective Addresses

An effective address is any operand to an instruction which references memory. Effective addresses, in NASM, have a very simple syntax: they consist of an expression evaluating to the desired address, enclosed in square brackets. For example:

```
wordvar   dw 123
          mov ax,[wordvar]
          mov ax,[wordvar+1]
          mov ax,[es:wordvar+bx]
```

Anything not conforming to this simple system is not a valid memory reference in NASM, for example `es:wordvar[bx]`.

More complicated effective addresses, such as those involving more than one register, work in exactly the same way:

```
          mov eax,[ebx*2+ecx+offset]
          mov ax,[bp+di+8]
```

NASM is capable of doing algebra on these effective addresses, so that things which don't necessarily *look* legal are perfectly all right:

```
          mov eax,[ebx*5]        ; assembles as [ebx*4+ebx]
          mov eax,[label1*2-label2] ; ie [label1+(label1-label2)]
```

Some forms of effective address have more than one assembled form; in most such cases NASM will generate the smallest form it can. For example, there are distinct assembled forms for the 32-bit effective addresses `[eax*2+0]` and `[eax+eax]`, and NASM will generally generate the latter on the grounds that the former requires four bytes to store a zero offset.

NASM has a hinting mechanism which will cause `[eax+ebx]` and `[ebx+eax]` to generate different opcodes; this is occasionally useful because `[esi+ebp]` and `[ebp+esi]` have different default segment registers.

However, you can force NASM to generate an effective address in a particular form by the use of the keywords `BYTE`, `WORD`, `DWORD` and `NOSPLIT`. If you need `[eax+3]` to be assembled using a double-word offset field instead of the one byte NASM will normally generate, you can code `[dword eax+3]`. Similarly, you can force NASM to use a byte offset for a small value which it hasn't seen on the first pass (see section 3.7 for an example of such a code fragment) by using `[byte eax+offset]`. As special cases, `[byte eax]` will code `[eax+0]` with a byte offset of zero, and `[dword eax]` will code it with a double-word offset of zero. The normal form, `[eax]`, will be coded with no offset field.

Similarly, NASM will split `[eax*2]` into `[eax+eax]` because that allows the offset field to be absent and space to be saved; in fact, it will also split `[eax*2+offset]` into `[eax+eax+offset]`. You can combat this behaviour by the use of the `NOSPLIT` keyword: `[nosplit eax*2]` will force `[eax*2+0]` to be generated literally.

3.4. Constants

NASM understands four different types of constant: numeric, character, string and floating-point.

3.4.1. Numeric Constants

A numeric constant is simply a number. NASM allows you to specify numbers in a variety of number bases, in a variety of ways: you can suffix `H`, `Q` and `B` for hex, octal and binary, or you can prefix `0x` for hex in the style of C, or you can prefix `$` for hex in the style of Borland Pascal. Note, though, that the `$` prefix does double duty as a prefix on identifiers (see section 3.1), so a hex number prefixed with a `$` sign must have a digit after the `$` rather than a letter.

Some examples:

```
mov ax,100              ; decimal
mov ax,0a2h             ; hex
mov ax,$0a2             ; hex again: the 0 is required
mov ax,0xa2             ; hex yet again
mov ax,777q             ; octal
mov ax,10010011b        ; binary
```

3.4.2. Character Constants

A character constant consists of up to four characters enclosed in either single or double quotes. The type of quote makes no difference to NASM, except of course that surrounding the constant with single quotes allows double quotes to appear within it and vice versa.

A character constant with more than one character will be arranged with littleendian order in mind: if you code

```
mov eax,'abcd'
```

then the constant generated is not `0x61626364`, but `0x64636261`, so that if you were then to store the value into memory, it would read `abcd` rather than `dcba`. This is also the sense of character constants understood by the Pentium's `CPUID` instruction (see section A.22).

3.4.3. String Constants

String constants are only acceptable to some pseudo-instructions, namely the `DB` family and `INCBIN`.

A string constant looks like a character constant, only longer. It is treated as a concatenation of maximum-size character constants for the conditions. So the following are equivalent:

```
db 'hello'              ; string constant
db 'h','e','l','l','o' ; equivalent character constants
```

And the following are also equivalent:

```
dd 'ninechars'          ; doubleword string constant
dd 'nine','char','s'    ; becomes three doublewords
db 'ninechars',0,0,0    ; and really looks like this
```

Note that when used as an operand to `db`, a constant like `'ab'` is treated as a string constant despite being short enough to be a character constant, because otherwise `db 'ab'` would have the same effect as `db 'a'`, which would be silly. Similarly, three-character or four-character constants are treated as strings when they are operands to `dw`.

3.4.4. Floating-Point Constants

Floatingpoint constants are acceptable only as arguments to `DD`, `DQ` and `DT`. They are expressed in the traditional form: digits, then a period, then optionally more digits, then optionally an `E` followed by an exponent. The period is mandatory, so that NASM can distinguish between `dd 1`, which declares an integer constant, and `dd 1.0` which declares a floating-point constant.

Some examples:

```
dd 1.2                    ; an easy one
dq 1.e10                  ; 10,000,000,000
dq 1.e+10                 ; synonymous with 1.e10
dq 1.e-10                 ; 0.000 000 000 1
dt 3.14159265358979323846 ; pi
```

NASM cannot do compile-time arithmetic on floating-point constants. This is because NASM is designed to be portable – although it always generates code to run on x86 processors, the assembler itself can run on any system with an ANSI C compiler. Therefore, the assembler cannot guarantee the presence of a floating-point unit capable of handling the Intel number formats, and so for NASM to be able to do floating arithmetic it would have to include its own complete set of floating-point routines, which would significantly increase the size of the assembler for very little benefit.

3.5. Expressions

Expressions in NASM are similar in syntax to those in C.

NASM does not guarantee the size of the integers used to evaluate expressions at compile time: since NASM can compile and run on 64-bit systems quite happily, don't assume that expressions are evaluated in 32-bit registers and so try to make deliberate use of integer overflow. It might not always work. The only thing NASM will guarantee is what's guaranteed by ANSI C: you always have *at least* 32 bits to work in.

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the `$` and `$$` tokens. `$` evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using `JMP $`. `$$` evaluates to the beginning of the current section; so you can tell how far into the section you are by using `($$$)`.

The arithmetic operators provided by NASM are listed here, in increasing order of precedence.

3.5.1. |: Bitwise OR Operator

The | operator gives a bitwise OR, exactly as performed by the OR machine instruction. Bitwise OR is the lowest-priority arithmetic operator supported by NASM.

3.5.2. ^: Bitwise XOR Operator

^ provides the bitwise XOR operation.

3.5.3. &: Bitwise AND Operator

& provides the bitwise AND operation.

3.5.4. << and >>: Bit Shift Operators

<< gives a bit-shift to the left, just as it does in C. So `5<<3` evaluates to 5 times 8, or 40. >> gives a bit-shift to the right; in NASM, such a shift is *always* unsigned, so that the bits shifted in from the left-hand end are filled with zero rather than a sign-extension of the previous highest bit.

3.5.5. + and −: Addition and Subtraction Operators

The + and − operators do perfectly ordinary addition and subtraction.

3.5.6. `*`, `/`, `//`, `%` and `%%`: Multiplication and Division

`*` is the multiplication operator. `/` and `//` are both division operators: `/` is unsigned division and `//` is signed division. Similarly, `%` and `%%` provide unsigned and signed modulo operators respectively.

NASM, like ANSI C, provides no guarantees about the sensible operation of the signed modulo operator.

Since the `%` character is used extensively by the macro preprocessor, you should ensure that both the signed and unsigned modulo operators are followed by white space wherever they appear.

3.5.7. Unary Operators: +, -, ~ and SEG

The highest-priority operators in NASM's expression grammar are those which only apply to one argument. - negates its operand, + does nothing (it's provided for symmetry with -), ~ computes the one's complement of its operand, and SEG provides the segment address of its operand (explained in more detail in section 3.6).

3.6. SEG and WRT

When writing large 16-bit programs, which must be split into multiple segments, it is often necessary to be able to refer to the segment part of the address of a symbol. NASM supports the SEG operator to perform this function.

The SEG operator returns the *preferred* segment base of a symbol, defined as the segment base relative to which the offset of the symbol makes sense. So the code

```
mov ax,seg symbol
mov es,ax
mov bx,symbol
```

will load ES:BX with a valid pointer to the symbol symbol.

Things can be more complex than this: since 16-bit segments and groups may overlap, you might occasionally want to refer to some symbol using a different segment base from the preferred one. NASM lets you do this, by the use of the WRT (With Reference To) keyword. So you can do things like

```
mov ax,weird_seg       ; weird_seg is a segment base
mov es,ax
mov bx,symbol wrt weird_seg
```

to load ES:BX with a different, but functionally equivalent, pointer to the symbol symbol.

NASM supports far (inter-segment) calls and jumps by means of the syntax call segment:offset, where segment and offset both represent immediate values. So to call a far procedure, you could code either of

```
call (seg procedure):procedure
call weird_seg:(procedure wrt weird_seg)
```

(The parentheses are included for clarity, to show the intended parsing of the above instructions. They are not necessary in practice.)

NASM supports the syntax call far procedure as a synonym for the first of the above usages. JMP works identically to CALL in these examples.

To declare a far pointer to a data item in a data segment, you must code

```
dw symbol, seg symbol
```

NASM supports no convenient synonym for this, though you can always invent one using the macro processor.

3.7. Critical Expressions

A limitation of NASM is that it is a twopass assembler; unlike TASM and others, it will always do exactly two assembly passes. Therefore it is unable to cope with source files that are complex enough to require three or more passes.

The first pass is used to determine the size of all the assembled code and data, so that the second pass, when generating all the code, knows all the symbol addresses the code refers to. So one thing NASM can't handle is code whose size depends on the value of a symbol declared after the code in question. For example,

```
          times (label-$) db 0
label:    db 'Where am I?'
```

The argument to `TIMES` in this case could equally legally evaluate to anything at all; NASM will reject this example because it cannot tell the size of the `TIMES` line when it first sees it. It will just as firmly reject the slightly paradoxical code

```
          times (label-$+1) db 0
label:    db 'NOW where am I?'
```

in which *any* value for the `TIMES` argument is by definition wrong!

NASM rejects these examples by means of a concept called a *critical expression*, which is defined to be an expression whose value is required to be computable in the first pass, and which must therefore depend only on symbols defined before it. The argument to the `TIMES` prefix is a critical expression; for the same reason, the arguments to the `RESB` family of pseudo-instructions are also critical expressions.

Critical expressions can crop up in other contexts as well: consider the following code.

```
          mov ax,symbol1
symbol1   equ symbol2
symbol2:
```

On the first pass, NASM cannot determine the value of `symbol1`, because `symbol1` is defined to be equal to `symbol2` which NASM hasn't seen yet. On the second pass, therefore, when it encounters the line `mov ax,symbol1`, it is unable to generate the code for it because it still doesn't know the value of `symbol1`. On the next line, it would see the `EQU` again and be able to determine the value of `symbol1`, but by then it would be too late.

NASM avoids this problem by defining the right-hand side of an `EQU` statement to be a critical expression, so the definition of `symbol1` would be rejected in the first pass.

There is a related issue involving forward references: consider this code fragment.

```
          mov eax,[ebx+offset]
offset    equ 10
```

NASM, on pass one, must calculate the size of the instruction `mov eax,[ebx+offset]` without knowing the value of `offset`. It has no way of knowing that `offset` is small enough to fit into a one-byte offset field and that it could therefore get away with generating a shorter form of the effectiveaddress encoding; for all it knows, in pass one, `offset` could be a symbol

in the code segment, and it might need the full four-byte form. So it is forced to compute the size of the instruction to accommodate a four-byte address part. In pass two, having made this decision, it is now forced to honour it and keep the instruction large, so the code generated in this case is not as small as it could have been. This problem can be solved by defining `offset` before using it, or by forcing byte size in the effective address by coding `[byte ebx+offset]`.

3.8. Local Labels

NASM gives special treatment to symbols beginning with a period. A label beginning with a single period is treated as a *local* label, which means that it is associated with the previous non-local label. So, for example:

```
label1    ; some code
.loop     ; some more code
          jne .loop
          ret
label2    ; some code
.loop     ; some more code
          jne .loop
          ret
```

In the above code fragment, each `JNE` instruction jumps to the line immediately before it, because the two definitions of `.loop` are kept separate by virtue of each being associated with the previous non-local label.

This form of local label handling is borrowed from the old Amiga assembler DevPac; however, NASM goes one step further, in allowing access to local labels from other parts of the code. This is achieved by means of *defining* a local label in terms of the previous non-local label: the first definition of `.loop` above is really defining a symbol called `label1.loop`, and the second defines a symbol called `label2.loop`. So, if you really needed to, you could write

```
label3    ; some more code
          ; and some more
          jmp label1.loop
```

Sometimes it is useful – in a macro, for instance – to be able to define a label which can be referenced from anywhere but which doesn't interfere with the normal local-label mechanism. Such a label can't be non-local because it would interfere with subsequent definitions of, and references to, local labels; and it can't be local because the macro that defined it wouldn't know the label's full name. NASM therefore introduces a third type of label, which is probably only useful in macro definitions: if a label begins with the special prefix `..@`, then it does nothing to the local label mechanism. So you could code

```
label1:   ; a non-local label
.local:   ; this is really label1.local
..@foo:    ; this is a special symbol
label2:   ; another non-local label
.local:   ; this is really label2.local
          jmp ..@foo              ; this will jump three lines up
```

NASM has the capacity to define other special symbols beginning with a double period: for example, `..start` is used to specify the entry point in the `obj` output format (see section 6.2.6).

Chapter 4: The NASM Preprocessor

NASM contains a powerful macro processor, which supports conditional assembly, multi-level file inclusion, two forms of macro (single-line and multi-line), and a `context stack' mechanism for extra macro power. Preprocessor directives all begin with a % sign.

4.1. SingleLine Macros

4.1.1. The Normal Way: `%define`

Single-line macros are defined using the `%define` preprocessor directive. The definitions work in a similar way to C; so you can do things like

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
          mov byte [param(2,ebx)], ctrl 'D'
```

which will expand to

```
          mov byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

When the expansion of a single-line macro contains tokens which invoke another macro, the expansion is performed at invocation time, not at definition time. Thus the code

```
%define a(x) 1+b(x)
%define b(x) 2*x
          mov ax,a(8)
```

will evaluate in the expected way to `mov ax,1+2*8`, even though the macro `b` wasn't defined at the time of definition of `a`.

Macros defined with `%define` are case sensitive: after `%define foo bar`, only `foo` will expand to `bar`: `Foo` or `FOO` will not. By using `%idefine` instead of `%define` (the `i' stands for `insensitive') you can define all the case variants of a macro at once, so that `%idefine foo bar` would cause `foo`, `Foo`, `FOO`, `fOO` and so on all to expand to `bar`.

There is a mechanism which detects when a macro call has occurred as a result of a previous expansion of the same macro, to guard against circular references and infinite loops. If this happens, the preprocessor will only expand the first occurrence of the macro. Hence, if you code

```
%define a(x) 1+a(x)
          mov ax,a(3)
```

the macro `a(3)` will expand once, becoming `1+a(3)`, and will then expand no further. This behaviour can be useful: see section 8.1 for an example of its use.

You can overload single-line macros: if you write

```
%define foo(x) 1+x
%define foo(x,y) 1+x*y
```

the preprocessor will be able to handle both types of macro call, by counting the parameters you pass; so `foo(3)` will become `1+3` whereas `foo(ebx,2)` will become `1+ebx*2`. However, if you define

```
%define foo bar
```

then no other definition of `foo` will be accepted: a macro with no parameters prohibits the definition of the same name as a macro *with* parameters, and vice versa.

This doesn't prevent single-line macros being *redefined*: you can perfectly well define a macro with

```
%define foo bar
```

and then re-define it later in the same source file with

```
%define foo baz
```

Then everywhere the macro `foo` is invoked, it will be expanded according to the most recent definition. This is particularly useful when defining single-line macros with `%assign` (see section 4.1.2).

You can predefine single-line macros using the `-d' option on the NASM command line: see section 2.1.7.

4.1.2. Preprocessor Variables: `%assign`

An alternative way to define single-line macros is by means of the `%assign` command (and its case sensitivecase-insensitive counterpart `%iassign`, which differs from `%assign` in exactly the same way that `%idefine` differs from `%define`).

`%assign` is used to define single-line macros which take no parameters and have a numeric value. This value can be specified in the form of an expression, and it will be evaluated once, when the `%assign` directive is processed.

Like `%define`, macros defined using `%assign` can be re-defined later, so you can do things like

```
%assign i i+1
```

to increment the numeric value of a macro.

`%assign` is useful for controlling the termination of `%rep` preprocessor loops: see section 4.4 for an example of this. Another use for `%assign` is given in section 7.4 and section 8.1.

The expression passed to `%assign` is a critical expression (see section 3.7), and must also evaluate to a pure number (rather than a relocatable reference such as a code or data address, or anything involving a register).

4.2. MultiLine Macros: `%macro`

Multi-line macros are much more like the type of macro seen in MASM and TASM: a multi-line macro definition in NASM looks something like this.

```
%macro prologue 1
          push ebp
          mov ebp,esp
          sub esp,%1
%endmacro
```

This defines a C-like function prologue as a macro: so you would invoke the macro with a call such as

```
myfunc:   prologue 12
```

which would expand to the three lines of code

```
myfunc:   push ebp
          mov ebp,esp
          sub esp,12
```

The number `1` after the macro name in the `%macro` line defines the number of parameters the macro `prologue` expects to receive. The use of `%1` inside the macro definition refers to the first parameter to the macro call. With a macro taking more than one parameter, subsequent parameters would be referred to as `%2`, `%3` and so on.

Multi-line macros, like single-line macros, are casesensitive, unless you define them using the alternative directive `%imacro`.

If you need to pass a comma as *part* of a parameter to a multi-line macro, you can do that by enclosing the entire parameter in braces. So you could code things like

```
%macro silly 2
%2:       db %1
%endmacro
          silly 'a', letter_a    ; letter_a:  db 'a'
          silly 'ab', string_ab  ; string_ab: db 'ab'
          silly @{13,10@}, crlf    ; crlf:      db 13,10
```

4.2.1. Overloading MultiLine Macros

As with single-line macros, multi-line macros can be overloaded by defining the same macro name several times with different numbers of parameters. This time, no exception is made for macros with no parameters at all. So you could define

```
%macro prologue 0
          push ebp
          mov ebp,esp
%endmacro
```

to define an alternative form of the function prologue which allocates no local stack space.

Sometimes, however, you might want to `overload' a machine instruction; for example, you might want to define

```
%macro push 2
          push %1
          push %2
%endmacro
```

so that you could code

```
          push ebx                ; this line is not a macro call
          push eax,ecx            ; but this one is
```

Ordinarily, NASM will give a warning for the first of the above two lines, since `push` is now defined to be a macro, and is being invoked with a number of parameters for which no definition has been given. The correct code will still be generated, but the assembler will give a warning. This warning can be disabled by the use of the `wmacroparams` command-line option (see section 2.1.10).

4.2.2. MacroLocal Labels

NASM allows you to define labels within a multi-line macro definition in such a way as to make them local to the macro call: so calling the same macro multiple times will use a different label each time. You do this by prefixing `%%` to the label name. So you can invent an instruction which executes a `RET` if the `Z` flag is set by doing this:

```
%macro retz 0
          jnz %%skip
          ret
%%skip:
%endmacro
```

You can call this macro as many times as you want, and every time you call it NASM will make up a different `real' name to substitute for the label `%%skip`. The names NASM invents are of the form `..@2345.skip`, where the number 2345 changes with every macro call. The `..@` prefix prevents macro-local labels from interfering with the local label mechanism, as described in section 3.8. You should avoid defining your own labels in this form (the `..@` prefix, then a number, then another period) in case they interfere with macro-local labels.

4.2.3. Greedy Macro Parameters

Occasionally it is useful to define a macro which lumps its entire command line into one parameter definition, possibly after extracting one or two smaller parameters from the front. An example might be a macro to write a text string to a file in MS-DOS, where you might want to be able to write

```
writefile [filehandle],"hello, world",13,10
```

NASM allows you to define the last parameter of a macro to be *greedy*, meaning that if you invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one along with the separating commas. So if you code:

```
%macro writefile 2+
        jmp %%endstr
%%str:  db %2
%%endstr: mov dx,%%str
        mov cx,%%endstr-%%str
        mov bx,%1
        mov ah,0x40
        int 0x21
%endmacro
```

then the example call to `writefile` above will work as expected: the text before the first comma, `[filehandle]`, is used as the first macro parameter and expanded when `%1` is referred to, and all the subsequent text is lumped into `%2` and placed after the `db`.

The greedy nature of the macro is indicated to NASM by the use of the + sign after the parameter count on the `%macro` line.

If you define a greedy macro, you are effectively telling NASM how it should expand the macro given *any* number of parameters from the actual number specified up to infinity; in this case, for example, NASM now knows what to do when it sees a call to `writefile` with 2, 3, 4 or more parameters. NASM will take this into account when overloading macros, and will not allow you to define another form of `writefile` taking 4 parameters (for example).

Of course, the above macro could have been implemented as a non-greedy macro, in which case the call to it would have had to look like

```
writefile [filehandle], @{"hello, world",13,10@}
```

NASM provides both mechanisms for putting commas in macro parameters, and you choose which one you prefer for each macro definition.

See section 5.2.1 for a better way to write the above macro.

4.2.4. Default Macro Parameters

NASM also allows you to define a multi-line macro with a *range* of allowable parameter counts. If you do this, you can specify defaults for omitted parameters. So, for example:

```
%macro die 0-1 "Painful program death has occurred."
          writefile 2,%1
          mov ax,0x4c01
          int 0x21
%endmacro
```

This macro (which makes use of the `writefile` macro defined in section 4.2.3) can be called with an explicit error message, which it will display on the error output stream before exiting, or it can be called with no parameters, in which case it will use the default error message supplied in the macro definition.

In general, you supply a minimum and maximum number of parameters for a macro of this type; the minimum number of parameters are then required in the macro call, and then you provide defaults for the optional ones. So if a macro definition began with the line

```
%macro foobar 1-3 eax,[ebx+2]
```

then it could be called with between one and three parameters, and `%1` would always be taken from the macro call. `%2`, if not specified by the macro call, would default to `eax`, and `%3` if not specified would default to `[ebx+2]`.

You may omit parameter defaults from the macro definition, in which case the parameter default is taken to be blank. This can be useful for macros which can take a variable number of parameters, since the `%0` token (see section 4.2.5) allows you to determine how many parameters were really passed to the macro call.

This defaulting mechanism can be combined with the greedy-parameter mechanism; so the `die` macro above could be made more powerful, and more useful, by changing the first line of the definition to

```
%macro die 0-1+ "Painful program death has occurred.",13,10
```

The maximum parameter count can be infinite, denoted by `*`. In this case, of course, it is impossible to provide a *full* set of default parameters. Examples of this usage are shown in section 4.2.6.

4.2.5. `%0`: Macro Parameter Counter

For a macro which can take a variable number of parameters, the parameter reference `%0` will return a numeric constant giving the number of parameters passed to the macro. This can be used as an argument to `%rep` (see section 4.4) in order to iterate through all the parameters of a macro. Examples are given in section 4.2.6.

4.2.6. `%rotate`: Rotating Macro Parameters

Unix shell programmers will be familiar with the `shift` shell command, which allows the arguments passed to a shell script (referenced as `$1`, `$2` and so on) to be moved left by one place, so that the argument previously referenced as `$2` becomes available as `$1`, and the argument previously referenced as `$1` is no longer available at all.

NASM provides a similar mechanism, in the form of `%rotate`. As its name suggests, it differs from the Unix `shift` in that no parameters are lost: parameters rotated off the left end of the argument list reappear on the right, and vice versa.

`%rotate` is invoked with a single numeric argument (which may be an expression). The macro parameters are rotated to the left by that many places. If the argument to `%rotate` is negative, the macro parameters are rotated to the right.

So a pair of macros to save and restore a set of registers might work as follows:

```
%macro multipush 1-*
%rep %0
          push %1
%rotate 1
%endrep
%endmacro
```

This macro invokes the `PUSH` instruction on each of its arguments in turn, from left to right. It begins by pushing its first argument, `%1`, then invokes `%rotate` to move all the arguments one place to the left, so that the original second argument is now available as `%1`. Repeating this procedure as many times as there were arguments (achieved by supplying `%0` as the argument to `%rep`) causes each argument in turn to be pushed.

Note also the use of `*` as the maximum parameter count, indicating that there is no upper limit on the number of parameters you may supply to the `multipush` macro.

It would be convenient, when using this macro, to have a `POP` equivalent, which *didn't* require the arguments to be given in reverse order. Ideally, you would write the `multipush` macro call, then cut-and-paste the line to where the pop needed to be done, and change the name of the called macro to `multipop`, and the macro would take care of popping the registers in the opposite order from the one in which they were pushed.

This can be done by the following definition:

```
%macro multipop 1-*
%rep %0
%rotate -1
          pop %1
%endrep
%endmacro
```

This macro begins by rotating its arguments one place to the *right*, so that the original *last* argument appears as `%1`. This is then popped, and the arguments are rotated right again, so the second-to-last argument becomes `%1`. Thus the arguments are iterated through in reverse order.

4.2.7. Concatenating Macro Parameters

NASM can concatenate macro parameters on to other text surrounding them. This allows you to declare a family of symbols, for example, in a macro definition. If, for example, you wanted to generate a table of key codes along with offsets into the table, you could code something like

```
%macro keytab_entry 2
keypos%1 equ $-keytab
         db %2
%endmacro
keytab:
         keytab_entry F1,128+1
         keytab_entry F2,128+2
         keytab_entry Return,13
```

which would expand to

```
keytab:
keyposF1 equ $-keytab
         db 128+1
keyposF2 equ $-keytab
         db 128+2
keyposReturn equ $-keytab
         db 13
```

You can just as easily concatenate text on to the other end of a macro parameter, by writing `%1foo`.

If you need to append a *digit* to a macro parameter, for example defining labels `foo1` and `foo2` when passed the parameter `foo`, you can't code `%11` because that would be taken as the eleventh macro parameter. Instead, you must code `%{1}1`, which will separate the first `1` (giving the number of the macro parameter) from the second (literal text to be concatenated to the parameter).

This concatenation can also be applied to other preprocessor in-line objects, such as macro-local labels (section 4.2.2) and context-local labels (section 4.6.2). In all cases, ambiguities in syntax can be resolved by enclosing everything after the `%` sign and before the literal text in braces: so `%{%foo}bar` concatenates the text `bar` to the end of the real name of the macro-local label `%%foo`. (This is unnecessary, since the form NASM uses for the real names of macro-local labels means that the two usages `%{%foo}bar` and `%%foobar` would both expand to the same thing anyway; nevertheless, the capability is there.)

4.2.8. Condition Codes as Macro Parameters

NASM can give special treatment to a macro parameter which contains a condition code. For a start, you can refer to the macro parameter `%1` by means of the alternative syntax `%+1`, which informs NASM that this macro parameter is supposed to contain a condition code, and will cause the preprocessor to report an error message if the macro is called with a parameter which is *not* a valid condition code.

Far more usefully, though, you can refer to the macro parameter by means of `%1`, which NASM will expand as the *inverse* condition code. So the `retz` macro defined in section 4.2.2 can be replaced by a general conditionalreturn macro like this:

```
%macro retc 1
          j%-1 %%skip
          ret
%%skip:
%endmacro
```

This macro can now be invoked using calls like `retc ne`, which will cause the conditional-jump instruction in the macro expansion to come out as `JE`, or `retc po` which will make the jump a `JPE`.

The `%+1` macro-parameter reference is quite happy to interpret the arguments `CXZ` and `ECXZ` as valid condition codes; however, `%1` will report an error if passed either of these, because no inverse condition code exists.

4.2.9. Disabling Listing Expansion

When NASM is generating a listing file from your program, it will generally expand multi-line macros by means of writing the macro call and then listing each line of the expansion. This allows you to see which instructions in the macro expansion are generating what code; however, for some macros this clutters the listing up unnecessarily.

NASM therefore provides the `.nolist` qualifier, which you can include in a macro definition to inhibit the expansion of the macro in the listing file. The `.nolist` qualifier comes directly after the number of parameters, like this:

```
%macro foo 1.nolist
```

Or like this:

```
%macro bar 1-5+.nolist a,b,c,d,e,f,g,h
```

4.3. Conditional Assembly

Similarly to the C preprocessor, NASM allows sections of a source file to be assembled only if certain conditions are met. The general syntax of this feature looks like this:

```
%if<condition>
; some code which only appears if <condition> is met
%elif<condition2>
; only appears if <condition> is not met but <condition2> is
%else
; this appears if neither <condition> nor <condition2> was met
%endif
```

The `%else` clause is optional, as is the `%elif` clause. You can have more than one `%elif` clause as well.

4.3.1. `%ifdef`: Testing SingleLine Macro Existence

Beginning a conditional-assembly block with the line `%ifdef MACRO` will assemble the subsequent code if, and only if, a single-line macro called `MACRO` is defined. If not, then the `%elif` and `%else` blocks (if any) will be processed instead.

For example, when debugging a program, you might want to write code such as

```
          ; perform some function
%ifdef DEBUG
          writefile 2,"Function performed successfully",13,10
%endif
          ; go and do something else
```

Then you could use the command-line option `dDEBUG` to create a version of the program which produced debugging messages, and remove the option to generate the final release version of the program.

You can test for a macro *not* being defined by using `%ifndef` instead of `%ifdef`. You can also test for macro definitions in `%elif` blocks by using `%elifdef` and `%elifndef`.

4.3.2. `%ifctx`: Testing the Context Stack

The conditional-assembly construct `%ifctx ctxname` will cause the subsequent code to be assembled if and only if the top context on the preprocessor's context stack has the name `ctxname`. As with `%ifdef`, the inverse and `%elif` forms `%ifnctx`, `%elifctx` and `%elifnctx` are also supported.

For more details of the context stack, see section 4.6. For a sample use of `%ifctx`, see section 4.6.5.

4.3.3. `%if`: Testing Arbitrary Numeric Expressions

The conditional-assembly construct `%if expr` will cause the subsequent code to be assembled if and only if the value of the numeric expression `expr` is non-zero. An example of the use of this feature is in deciding when to break out of a `%rep` preprocessor loop: see section 4.4 for a detailed example.

The expression given to `%if`, and its counterpart `%elif`, is a critical expression (see section 3.7).

`%if` extends the normal NASM expression syntax, by providing a set of relational operators which are not normally available in expressions. The operators =, <, >, <=, >= and <> test equality, less-than, greater-than, less-or-equal, greater-or-equal and not-equal respectively. The C-like forms == and != are supported as alternative forms of = and <>. In addition, low-priority logical operators `&&`, `^^` and `||` are provided, supplying logical AND, logical XOR and logical OR. These work like the C logical operators (although C has no logical XOR), in that they always return either 0 or 1, and treat any non-zero input as 1 (so that `^^`, for example, returns 1 if exactly one of its inputs is zero, and 0 otherwise). The relational operators also return 1 for true and 0 for false.

4.3.4. `%ifidn` and `%ifidni`: Testing Exact Text Identity

The construct `%ifidn text1,text2` will cause the subsequent code to be assembled if and only if `text1` and `text2`, after expanding single-line macros, are identical pieces of text. Differences in white space are not counted.

`%ifidni` is similar to `%ifidn`, but is caseinsensitive.

For example, the following macro pushes a register or number on the stack, and allows you to treat `IP` as a real register:

```
%macro pushparam 1
%ifidni %1,ip
          call %%label
%%label:
%else
          push %1
%endif
%endmacro
```

Like most other `%if` constructs, `%ifidn` has a counterpart `%elifidn`, and negative forms `%ifnidn` and `%elifnidn`. Similarly, `%ifidni` has counterparts `%elifidni`, `%ifnidni` and `%elifnidni`.

4.3.5. `%ifid`, `%ifnum`, `%ifstr`: Testing Token Types

Some macros will want to perform different tasks depending on whether they are passed a number, a string, or an identifier. For example, a string output macro might want to be able to cope with being passed either a string constant or a pointer to an existing string.

The conditional assembly construct `%ifid`, taking one parameter (which may be blank), assembles the subsequent code if and only if the first token in the parameter exists and is an identifier. `%ifnum` works similarly, but tests for the token being a numeric constant; `%ifstr` tests for it being a string.

For example, the `writefile` macro defined in section 4.2.3 can be extended to take advantage of `%ifstr` in the following fashion:

```
%macro writefile 2-3+
%ifstr %2
          jmp %%endstr
%if %0 = 3
%%str:      db %2,%3
%else
%%str:      db %2
%endif
%%endstr: mov dx,%%str
          mov cx,%%endstr-%%str
%else
       mov dx,%2
       mov cx,%3
%endif
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

Then the `writefile` macro can cope with being called in either of the following two ways:

```
          writefile [file], strpointer, length
          writefile [file], "hello", 13, 10
```

In the first, `strpointer` is used as the address of an already-declared string, and `length` is used as its length; in the second, a string is given to the macro, which therefore declares it itself and works out the address and length for itself.

Note the use of `%if` inside the `%ifstr`: this is to detect whether the macro was passed two arguments (so the string would be a single string constant, and `db %2` would be adequate) or more (in which case, all but the first two would be lumped together into `%3`, and `db %2,%3` would be required).

The usual `%elifXXX`, `%ifnXXX` and `%elifnXXX` versions exist for each of `%ifid`, `%ifnum` and `%ifstr`.

4.3.6. `%error`: Reporting UserDefined Errors

The preprocessor directive `%error` will cause NASM to report an error if it occurs in assembled code. So if other users are going to try to assemble your source files, you can ensure that they define the right macros by means of code like this:

```
%ifdef SOME_MACRO
; do some setup
%elifdef SOME_OTHER_MACRO
; do some different setup
%else
%error Neither SOME_MACRO nor SOME_OTHER_MACRO was defined.
%endif
```

Then any user who fails to understand the way your code is supposed to be assembled will be quickly warned of their mistake, rather than having to wait until the program crashes on being run and then not knowing what went wrong.

4.4. Preprocessor Loops: `%rep`

NASM's `TIMES` prefix, though useful, cannot be used to invoke a multi-line macro multiple times, because it is processed by NASM after macros have already been expanded. Therefore NASM provides another form of loop, this time at the preprocessor level: `%rep`.

The directives `%rep` and `%endrep` (`%rep` takes a numeric argument, which can be an expression; `%endrep` takes no arguments) can be used to enclose a chunk of code, which is then replicated as many times as specified by the preprocessor:

```
%assign i 0
%rep 64
          inc word [table+2*i]
%assign i i+1
%endrep
```

This will generate a sequence of 64 `INC` instructions, incrementing every word of memory from `[table]` to `[table+126]`.

For more complex termination conditions, or to break out of a repeat loop part way along, you can use the `%exitrep` directive to terminate the loop, like this:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
%exitrep
%endif
          dw j
%assign k j+i
%assign i j
%assign j k
%endrep
fib_number equ ($-fibonacci)/2
```

This produces a list of all the Fibonacci numbers that will fit in 16 bits. Note that a maximum repeat count must still be given to `%rep`. This is to prevent the possibility of NASM getting into an infinite loop in the preprocessor, which (on multitasking or multi-user systems) would typically cause all the system memory to be gradually used up and other applications to start crashing.

4.5. Including Other Files

Using, once again, a very similar syntax to the C preprocessor, NASM's preprocessor lets you include other source files into your code. This is done by the use of the `%include` directive:

```
%include "macros.mac"
```

will include the contents of the file `macros.mac` into the source file containing the `%include` directive.

Include files are searched for in the current directory (the directory you're in when you run NASM, as opposed to the location of the NASM executable or the location of the source file), plus any directories specified on the NASM command line using the `i` option.

The standard C idiom for preventing a file being included more than once is just as applicable in NASM: if the file `macros.mac` has the form

```
%ifndef MACROS_MAC
%define MACROS_MAC
; now define some macros
%endif
```

then including the file more than once will not cause errors, because the second time the file is included nothing will happen because the macro `MACROS_MAC` will already be defined.

You can force a file to be included even if there is no `%include` directive that explicitly includes it, by using the `p` option on the NASM command line (see section 2.1.6).

4.6. The Context Stack

Having labels that are local to a macro definition is sometimes not quite powerful enough: sometimes you want to be able to share labels between several macro calls. An example might be a `REPEAT` ... `UNTIL` loop, in which the expansion of the `REPEAT` macro would need to be able to refer to a label which the `UNTIL` macro had defined. However, for such a macro you would also want to be able to nest these loops.

NASM provides this level of power by means of a *context stack*. The preprocessor maintains a stack of *contexts*, each of which is characterised by a name. You add a new context to the stack using the `%push` directive, and remove one using `%pop`. You can define labels that are local to a particular context on the stack.

4.6.1. `%push` and `%pop`: Creating and Removing Contexts

The `%push` directive is used to create a new context and place it on the top of the context stack. `%push` requires one argument, which is the name of the context. For example:

```
%push foobar
```

This pushes a new context called `foobar` on the stack. You can have several contexts on the stack with the same name: they can still be distinguished.

The directive `%pop`, requiring no arguments, removes the top context from the context stack and destroys it, along with any labels associated with it.

4.6.2. ContextLocal Labels

Just as the usage `%%foo` defines a label which is local to the particular macro call in which it is used, the usage `%$foo` is used to define a label which is local to the context on the top of the context stack. So the `REPEAT` and `UNTIL` example given above could be implemented by means of:

```
%macro repeat 0
%push repeat
%$begin:
%endmacro

%macro until 1
          j%-1 %$begin
%pop
%endmacro
```

and invoked by means of, for example,

```
          mov cx,string
          repeat
          add cx,3
          scasb
          until e
```

which would scan every fourth byte of a string in search of the byte in `AL`.

If you need to define, or access, labels local to the context *below* the top one on the stack, you can use `%$$foo`, or `%$$$foo` for the context below that, and so on.

4.6.3. ContextLocal SingleLine Macros

NASM also allows you to define single-line macros which are local to a particular context, in just the same way:

```
%define %$localmac 3
```

will define the single-line macro `%$localmac` to be local to the top context on the stack. Of course, after a subsequent `%push`, it can then still be accessed by the name `%$$localmac`.

4.6.4. `%repl`: Renaming a Context

If you need to change the name of the top context on the stack (in order, for example, to have it respond differently to `%ifctx`), you can execute a `%pop` followed by a `%push`; but this will have the side effect of destroying all context-local labels and macros associated with the context that was just popped.

NASM provides the directive `%repl`, which *replaces* a context with a different name, without touching the associated macros and labels. So you could replace the destructive code

```
%pop
%push newname
```

with the non-destructive version `%repl newname`.

4.6.5. Example Use of the Context Stack: Block IFs

This example makes use of almost all the context-stack features, including the conditional-assembly construct `%ifctx`, to implement a block IF statement as a set of macros.

```
%macro if 1
    %push if
    j%-1 %$ifnot
%endmacro

%macro else 0
    %ifctx if
        %repl else
        jmp %$ifend
        %$ifnot:
    %else
        %error "expected `if' before `else'"
    %endif
%endmacro

%macro endif 0
    %ifctx if
        %$ifnot:
        %pop
    %elifctx else
        %$ifend:
        %pop
    %else
        %error "expected `if' or `else' before `endif'"
    %endif
%endmacro
```

This code is more robust than the `REPEAT` and `UNTIL` macros given in section 4.6.2, because it uses conditional assembly to check that the macros are issued in the right order (for example, not calling `endif` before `if`) and issues a `%error` if they're not.

In addition, the `endif` macro has to be able to cope with the two distinct cases of either directly following an `if`, or following an `else`. It achieves this, again, by using conditional assembly to do different things depending on whether the context on top of the stack is `if` or `else`.

The `else` macro has to preserve the context on the stack, in order to have the `%$ifnot` referred to by the `if` macro be the same as the one defined by the `endif` macro, but has to change the context's name so that `endif` will know there was an intervening `else`. It does this by the use of `%repl`.

A sample usage of these macros might look like:

```
        cmp ax,bx
        if ae
          cmp bx,cx
          if ae
```

```
    mov ax,cx
  else
    mov ax,bx
  endif
else
  cmp ax,cx
  if ae
    mov ax,cx
  endif
endif
```

The block-`IF` macros handle nesting quite happily, by means of pushing another context, describing the inner `if`, on top of the one describing the outer `if`; thus `else` and `endif` always refer to the last unmatched `if` or `else`.

4.7. Standard Macros

NASM defines a set of standard macros, which are already defined when it starts to process any source file. If you really need a program to be assembled with no pre-defined macros, you can use the `%clear` directive to empty the preprocessor of everything.

Most userlevel assembler directives (see chapter 5) are implemented as macros which invoke primitive directives; these are described in chapter 5. The rest of the standard macro set is described here.

4.7.1. `__NASM_MAJOR__` and `__NASM_MINOR__`: NASM Version

The single-line macros `__NASM_MAJOR__` and `__NASM_MINOR__` expand to the major and minor parts of the version number of NASM being used. So, under NASM 0.96 for example, `__NASM_MAJOR__` would be defined to be 0 and `__NASM_MINOR__` would be defined as 96.

4.7.2. `__FILE__` and `__LINE__`: File Name and Line Number

Like the C preprocessor, NASM allows the user to find out the file name and line number containing the current instruction. The macro `__FILE__` expands to a string constant giving the name of the current input file (which may change through the course of assembly if `%include` directives are used), and `__LINE__` expands to a numeric constant giving the current line number in the input file.

These macros could be used, for example, to communicate debugging information to a macro, since invoking `__LINE__` inside a macro definition (either single-line or multi-line) will return the line number of the macro *call*, rather than *definition*. So to determine where in a piece of code a crash is occurring, for example, one could write a routine `stillhere`, which is passed a line number in `EAX` and outputs something like `line 155: still here'. You could then write a macro

```
%macro notdeadyet 0
          push eax
          mov eax,__LINE__
          call stillhere
          pop eax
%endmacro
```

and then pepper your code with calls to `notdeadyet` until you find the crash point.

4.7.3. `STRUC` and `ENDSTRUC`: Declaring Structure Data Types

The core of NASM contains no intrinsic means of defining data structures; instead, the preprocessor is sufficiently powerful that data structures can be implemented as a set of macros. The macros `STRUC` and `ENDSTRUC` are used to define a structure data type.

`STRUC` takes one parameter, which is the name of the data type. This name is defined as a symbol with the value zero, and also has the suffix `_size` appended to it and is then defined as an `EQU` giving the size of the structure. Once `STRUC` has been issued, you are defining the structure, and should define fields using the `RESB` family of pseudo-instructions, and then invoke `ENDSTRUC` to finish the definition.

For example, to define a structure called `mytype` containing a longword, a word, a byte and a string of bytes, you might code

```
            struc mytype
mt_long:    resd 1
mt_word:    resw 1
mt_byte:    resb 1
mt_str:     resb 32
            endstruc
```

The above code defines six symbols: `mt_long` as 0 (the offset from the beginning of a `mytype` structure to the longword field), `mt_word` as 4, `mt_byte` as 6, `mt_str` as 7, `mytype_size` as 39, and `mytype` itself as zero.

The reason why the structure type name is defined at zero is a side effect of allowing structures to work with the local label mechanism: if your structure members tend to have the same names in more than one structure, you can define the above structure like this:

```
            struc mytype
.long:      resd 1
.word:      resw 1
.byte:      resb 1
.str:       resb 32
            endstruc
```

This defines the offsets to the structure fields as `mytype.long`, `mytype.word`, `mytype.byte` and `mytype.str`.

NASM, since it has no *intrinsic* structure support, does not support any form of period notation to refer to the elements of a structure once you have one (except the above local-label notation), so code such as `mov ax,[mystruc.mt_word]` is not valid. `mt_word` is a constant just like any other constant, so the correct syntax is `mov ax,[mystruc+mt_word]` or `mov ax,[mystruc+mytype.word]`.

4.7.4. `ISTRUC`, `AT` and `IEND`: Declaring Instances of Structures

Having defined a structure type, the next thing you typically want to do is to declare instances of that structure in your data segment. NASM provides an easy way to do this in the `ISTRUC` mechanism. To declare a structure of type `mytype` in a program, you code something like this:

```
mystruc:  istruc mytype
          at mt_long, dd 123456
          at mt_word, dw 1024
          at mt_byte, db 'x'
          at mt_str, db 'hello, world', 13, 10, 0
          iend
```

The function of the `AT` macro is to make use of the `TIMES` prefix to advance the assembly position to the correct point for the specified structure field, and then to declare the specified data. Therefore the structure fields must be declared in the same order as they were specified in the structure definition.

If the data to go in a structure field requires more than one source line to specify, the remaining source lines can easily come after the `AT` line. For example:

```
          at mt_str, db 123,134,145,156,167,178,189
          db 190,100,0
```

Depending on personal taste, you can also omit the code part of the `AT` line completely, and start the structure field on the next line:

```
          at mt_str
          db 'hello, world'
          db 13,10,0
```

4.7.5. `ALIGN` and `ALIGNB`: Data Alignment

The `ALIGN` and `ALIGNB` macros provides a convenient way to align code or data on a word, longword, paragraph or other boundary. (Some assemblers call this directive `EVEN`.) The syntax of the `ALIGN` and `ALIGNB` macros is

```
          align 4                ; align on 4-byte boundary
          align 16               ; align on 16-byte boundary
          align 8,db 0           ; pad with 0s rather than NOPs
          align 4,resb 1         ; align to 4 in the BSS
          alignb 4               ; equivalent to previous line
```

Both macros require their first argument to be a power of two; they both compute the number of additional bytes required to bring the length of the current section up to a multiple of that power of two, and then apply the `TIMES` prefix to their second argument to perform the alignment.

If the second argument is not specified, the default for `ALIGN` is `NOP`, and the default for `ALIGNB` is `RESB 1`. So if the second argument is specified, the two macros are equivalent. Normally, you can just use `ALIGN` in code and data sections and `ALIGNB` in BSS sections, and never need the second argument except for special purposes.

`ALIGN` and `ALIGNB`, being simple macros, perform no error checking: they cannot warn you if their first argument fails to be a power of two, or if their second argument generates more than one byte of code. In each of these cases they will silently do the wrong thing.

`ALIGNB` (or `ALIGN` with a second argument of `RESB 1`) can be used within structure definitions:

```
          struc mytype2
mt_byte:  resb 1
          alignb 2
mt_word:  resw 1
          alignb 4
mt_long:  resd 1
mt_str:   resb 32
          endstruc
```

This will ensure that the structure members are sensibly aligned relative to the base of the structure.

A final caveat: `ALIGN` and `ALIGNB` work relative to the beginning of the *section*, not the beginning of the address space in the final executable. Aligning to a 16-byte boundary when the section you're in is only guaranteed to be aligned to a 4-byte boundary, for example, is a waste of effort. Again, NASM does not check that the section's alignment characteristics are sensible for the use of `ALIGN` or `ALIGNB`.

Chapter 5: Assembler Directives

NASM, though it attempts to avoid the bureaucracy of assemblers like MASM and TASM, is nevertheless forced to support a *few* directives. These are described in this chapter.

NASM's directives come in two types: userlevel directives*userlevel* directives and primitive directives*primitive* directives. Typically, each directive has a user-level form and a primitive form. In almost all cases, we recommend that users use the user-level forms of the directives, which are implemented as macros which call the primitive forms.

Primitive directives are enclosed in square brackets; user-level directives are not.

In addition to the universal directives described in this chapter, each object file format can optionally supply extra directives in order to control particular features of that file format. These formatspecific directives*formatspecific* directives are documented along with the formats that implement them, in chapter 6.

5.1. `BITS`: Specifying Target Processor Mode

The `BITS` directive specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, or code designed to run on a processor operating in 32-bit mode. The syntax is `BITS 16` or `BITS 32`.

In most cases, you should not need to use `BITS` explicitly. The `aout`, `coff`, `elf` and `win32` object formats, which are designed for use in 32-bit operating systems, all cause NASM to select 32-bit mode by default. The `obj` object format allows you to specify each segment you define as either `USE16` or `USE32`, and NASM will set its operating mode accordingly, so the use of the `BITS` directive is once again unnecessary.

The most likely reason for using the `BITS` directive is to write 32-bit code in a flat binary file; this is because the `bin` output format defaults to 16-bit mode in anticipation of it being used most frequently to write DOS `.COM` programs, DOS `.SYS` device drivers and boot loader software.

You do *not* need to specify `BITS 32` merely in order to use 32-bit instructions in a 16-bit DOS program; if you do, the assembler will generate incorrect code because it will be writing code targeted at a 32-bit platform, to be run on a 16-bit one.

When NASM is in `BITS 16` state, instructions which use 32-bit data are prefixed with an 0x66 byte, and those referring to 32-bit addresses have an 0x67 prefix. In `BITS 32` state, the reverse is true: 32-bit instructions require no prefixes, whereas instructions using 16-bit data need an 0x66 and those working in 16-bit addresses need an 0x67.

The `BITS` directive has an exactly equivalent primitive form, `[BITS 16]` and `[BITS 32]`. The user-level form is a macro which has no function other than to call the primitive form.

5.2. `SECTION` or `SEGMENT`: Changing and Defining Sections

The `SECTION` directive (`SEGMENT` is an exactly equivalent synonym) changes which section of the output file the code you write will be assembled into. In some object file formats, the number and names of sections are fixed; in others, the user may make up as many as they wish. Hence `SECTION` may sometimes give an error message, or may define a new section, if you try to switch to a section that does not (yet) exist.

The Unix object formats, and the `bin` object format, all support the standardised section names `.text`, `.data` and `.bss` for the code, data and uninitialised-data sections. The `obj` format, by contrast, does not recognise these section names as being special, and indeed will strip off the leading period of any section name that has one.

Section 5.2.1: The __SECT__ Macro

5.2.1. The __SECT__ Macro

The SECTION directive is unusual in that its user-level form functions differently from its primitive form. The primitive form, [SECTION xyz], simply switches the current target section to the one given. The user-level form, SECTION xyz, however, first defines the single-line macro __SECT__ to be the primitive [SECTION] directive which it is about to issue, and then issues it. So the user-level directive

```
          SECTION .text
```

expands to the two lines

```
%define __SECT__ [SECTION .text]
          [SECTION .text]
```

Users may find it useful to make use of this in their own macros. For example, the writefile macro defined in section 4.2.3 can be usefully rewritten in the following more sophisticated form:

```
%macro writefile 2+
          [section .data]
%%str:    db %2
%%endstr:
          __SECT__
          mov dx,%%str
          mov cx,%%endstr-%%str
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

This form of the macro, once passed a string to output, first switches temporarily to the data section of the file, using the primitive form of the SECTION directive so as not to modify __SECT__. It then declares its string in the data section, and then invokes __SECT__ to switch back to *whichever* section the user was previously working in. It thus avoids the need, in the previous version of the macro, to include a JMP instruction to jump over the data, and also does not fail if, in a complicated OBJ format module, the user could potentially be assembling the code in any of several separate code sections.

5.3. ABSOLUTE: Defining Absolute Labels

The ABSOLUTE directive can be thought of as an alternative form of SECTION: it causes the subsequent code to be directed at no physical section, but at the hypothetical section starting at the given absolute address. The only instructions you can use in this mode are the RESB family.

ABSOLUTE is used as follows:

```
          absolute 0x1A
kbuf_chr  resw 1
kbuf_free resw 1
kbuf      resw 16
```

This example describes a section of the PC BIOS data area, at segment address 0x40: the above code defines kbuf_chr to be 0x1A, kbuf_free to be 0x1C, and kbuf to be 0x1E.

The user-level form of ABSOLUTE, like that of SECTION, redefines the __SECT__ macro when it is invoked.

STRUC and ENDSTRUC are defined as macros which use ABSOLUTE (and also __SECT__).

ABSOLUTE doesn't have to take an absolute constant as an argument: it can take an expression (actually, a critical expression: see section 3.7) and it can be a value in a segment. For example, a TSR can re-use its setup code as run-time BSS like this:

```
          org 100h              ; it's a .COM program
          jmp setup             ; setup code comes last
          ; the resident part of the TSR goes here
setup:    ; now write the code that installs the TSR here
          absolute setup
runtimevar1 resw 1
runtimevar2 resd 20
tsr_end:
```

This defines some variables `on top of' the setup code, so that after the setup has finished running, the space it took up can be re-used as data storage for the running TSR. The symbol `tsr_end' can be used to calculate the total size of the part of the TSR that needs to be made resident.

5.4. `EXTERN`: Importing Symbols from Other Modules

`EXTERN` is similar to the MASM directive `EXTRN` and the C keyword `extern`: it is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one. Not every object-file format can support external variables: the `bin` format cannot.

The `EXTERN` directive takes as many arguments as you like. Each argument is the name of a symbol:

```
extern _printf
extern _sscanf,_fscanf
```

Some object-file formats provide extra features to the `EXTERN` directive. In all cases, the extra features are used by suffixing a colon to the symbol name followed by object-format specific text. For example, the `obj` format allows you to declare that the default segment base of an external should be the group `dgroup` by means of the directive

```
extern _variable:wrt dgroup
```

The primitive form of `EXTERN` differs from the user-level form only in that it can take only one argument at a time: the support for multiple arguments is implemented at the preprocessor level.

You can declare the same variable as `EXTERN` more than once: NASM will quietly ignore the second and later redeclarations. You can't declare a variable as `EXTERN` as well as something else, though.

5.5. GLOBAL: Exporting Symbols to Other Modules

GLOBAL is the other end of EXTERN: if one module declares a symbol as EXTERN and refers to it, then in order to prevent linker errors, some other module must actually *define* the symbol and declare it as GLOBAL. Some assemblers use the name PUBLIC for this purpose.

The GLOBAL directive applying to a symbol must appear *before* the definition of the symbol.

GLOBAL uses the same syntax as EXTERN, except that it must refer to symbols which *are* defined in the same module as the GLOBAL directive. For example:

```
          global _main
_main:    ; some code
```

GLOBAL, like EXTERN, allows object formats to define private extensions by means of a colon. The elf object format, for example, lets you specify whether global data items are functions or data:

```
          global hashlookup:function, hashtable:data
```

Like EXTERN, the primitive form of GLOBAL differs from the user-level form only in that it can take only one argument at a time.

5.6. COMMON: Defining Common Data Areas

The COMMON directive is used to declare *common variables*. A common variable is much like a global variable declared in the uninitialised data section, so that

```
          common intvar 4
```

is similar in function to

```
          global intvar
          section .bss
intvar    resd 1
```

The difference is that if more than one module defines the same common variable, then at link time those variables will be *merged*, and references to `intvar` in all modules will point at the same piece of memory.

Like GLOBAL and EXTERN, COMMON supports object-format specific extensions. For example, the `obj` format allows common variables to be NEAR or FAR, and the `elf` format allows you to specify the alignment requirements of a common variable:

```
          common commvar 4:near  ; works in OBJ
          common intarray 100:4  ; works in ELF: 4 byte aligned
```

Once again, like EXTERN and GLOBAL, the primitive form of COMMON differs from the user-level form only in that it can take only one argument at a time.

Chapter 6: Output Formats

NASM is a portable assembler, designed to be able to compile on any ANSI C-supporting platform and produce output to run on a variety of Intel x86 operating systems. For this reason, it has a large number of available output formats, selected using the `f` option on the NASM command line. Each of these formats, along with its extensions to the base NASM syntax, is detailed in this chapter.

As stated in section 2.1.1, NASM chooses a default name for your output file based on the input file name and the chosen output format. This will be generated by removing the extension (`.asm`, `.s`, or whatever you like to use) from the input file name, and substituting an extension defined by the output format. The extensions are given with each format below.

6.1. `bin`: FlatForm Binary Output

The `bin` format does not produce object files: it generates nothing in the output file except the code you wrote. Such `pure binary' files are used by MSDOS: `.COM` executables and `.SYS` device drivers are pure binary files. Pure binary output is also useful for operatingsystem and boot loader development.

`bin` supports the three standardised section names `.text`, `.data` and `.bss` only. The file NASM outputs will contain the contents of the `.text` section first, followed by the contents of the `.data` section, aligned on a four-byte boundary. The `.bss` section is not stored in the output file at all, but is assumed to appear directly after the end of the `.data` section, again aligned on a four-byte boundary.

If you specify no explicit `SECTION` directive, the code you write will be directed by default into the `.text` section.

Using the `bin` format puts NASM by default into 16-bit mode (see section 5.1). In order to use `bin` to write 32-bit code such as an OS kernel, you need to explicitly issue the `BITS 32` directive.

`bin` has no default output file name extension: instead, it leaves your file name as it is once the original extension has been removed. Thus, the default is for NASM to assemble `binprog.asm` into a binary file called `binprog`.

6.1.1. ORG: Binary File Program Origin

The `bin` format provides an additional directive to the list given in chapter 5: `ORG`. The function of the `ORG` directive is to specify the origin address which NASM will assume the program begins at when it is loaded into memory.

For example, the following code will generate the longword `0x00000104`:

```
          org 0x100
          dd label
label:
```

Unlike the `ORG` directive provided by MASM-compatible assemblers, which allows you to jump around in the object file and overwrite code you have already generated, NASM's `ORG` does exactly what the directive says: *origin*. Its sole function is to specify one offset which is added to all internal address references within the file; it does not permit any of the trickery that MASM's version does. See section 10.1.3 for further comments.

6.1.2. `bin` Extensions to the `SECTION` Directive

The `bin` output format extends the `SECTION` (or `SEGMENT`) directive to allow you to specify the alignment requirements of segments. This is done by appending the `ALIGN` qualifier to the end of the section-definition line. For example,

```
section .data align=16
```

switches to the section `.data` and also specifies that it must be aligned on a 16-byte boundary.

The parameter to `ALIGN` specifies how many low bits of the section start address must be forced to zero. The alignment value given may be any power of two.

6.2. `obj`: Microsoft OMF Object Files

The `obj` file format (NASM calls it `obj` rather than `omf` for historical reasons) is the one produced by MASM and TASM, which is typically fed to 16-bit DOS linkers to produce `.EXE` files. It is also the format used by OS/2.

`obj` provides a default output file-name extension of `.obj`.

`obj` is not exclusively a 16-bit format, though: NASM has full support for the 32-bit extensions to the format. In particular, 32-bit `obj` format files are used by Borland's Win32 compilers, instead of using Microsoft's newer `win32` object file format.

The `obj` format does not define any special segment names: you can call your segments anything you like. Typical names for segments in `obj` format files are `CODE`, `DATA` and `BSS`.

If your source file contains code before specifying an explicit `SEGMENT` directive, then NASM will invent its own segment called `__NASMDEFSEG` for you.

When you define a segment in an `obj` file, NASM defines the segment name as a symbol as well, so that you can access the segment address of the segment. So, for example:

```
          segment data
dvar:     dw 1234
          segment code
function: mov ax,data           ; get segment address of data
          mov ds,ax             ; and move it into DS
          inc word [dvar]       ; now this reference will work
          ret
```

The `obj` format also enables the use of the `SEG` and `WRT` operators, so that you can write code which does things like

```
          extern foo
          mov ax,seg foo        ; get preferred segment of foo
          mov ds,ax
          mov ax,data           ; a different segment
          mov es,ax
          mov ax,[ds:foo]       ; this accesses `foo'
          mov [es:foo wrt data],bx  ; so does this
```

6.2.1. `obj` Extensions to the `SEGMENT` Directive

The `obj` output format extends the `SEGMENT` (or `SECTION`) directive to allow you to specify various properties of the segment you are defining. This is done by appending extra qualifiers to the end of the segment-definition line. For example,

```
segment code private align=16
```

defines the segment `code`, but also declares it to be a private segment, and requires that the portion of it described in this code module must be aligned on a 16-byte boundary.

The available qualifiers are:

- `PRIVATE`, `PUBLIC`, `COMMON` and `STACK` specify the combination characteristics of the segment. `PRIVATE` segments do not get combined with any others by the linker; `PUBLIC` and `STACK` segments get concatenated together at link time; and `COMMON` segments all get overlaid on top of each other rather than stuck end-to-end.
- `ALIGN` is used, as shown above, to specify how many low bits of the segment start address must be forced to zero. The alignment value given may be any power of two from 1 to 4096; in reality, the only values supported are 1, 2, 4, 16, 256 and 4096, so if 8 is specified it will be rounded up to 16, and 32, 64 and 128 will all be rounded up to 256, and so on. Note that alignment to 4096-byte boundaries is a PharLap extension to the format and may not be supported by all linkers.
- `CLASS` can be used to specify the segment class; this feature indicates to the linker that segments of the same class should be placed near each other in the output file. The class name can be any word, e.g. `CLASS=CODE`.
- `OVERLAY`, like `CLASS`, is specified with an arbitrary word as an argument, and provides overlay information to an overlay-capable linker.
- Segments can be declared as `USE16` or `USE32`, which has the effect of recording the choice in the object file and also ensuring that NASM's default assembly mode when assembling in that segment is 16-bit or 32-bit respectively.
- When writing OS/2 object files, you should declare 32-bit segments as `FLAT`, which causes the default segment base for anything in the segment to be the special group `FLAT`, and also defines the group if it is not already defined.
- The `obj` file format also allows segments to be declared as having a pre-defined absolute segment address, although no linkers are currently known to make sensible use of this feature; nevertheless, NASM allows you to declare a segment such as `SEGMENT SCREEN ABSOLUTE=0xB800` if you need to. The `ABSOLUTE` and `ALIGN` keywords are mutually exclusive.

NASM's default segment attributes are `PUBLIC`, `ALIGN=1`, no class, no overlay, and `USE16`.

6.2.2. GROUP: Defining Groups of Segments

The obj format also allows segments to be grouped, so that a single segment register can be used to refer to all the segments in a group. NASM therefore supplies the GROUP directive, whereby you can code

```
segment data
; some data
segment bss
; some uninitialised data
group dgroup data bss
```

which will define a group called dgroup to contain the segments data and bss. Like SEGMENT, GROUP causes the group name to be defined as a symbol, so that you can refer to a variable var in the data segment as var wrt data or as var wrt dgroup, depending on which segment value is currently in your segment register.

If you just refer to var, however, and var is declared in a segment which is part of a group, then NASM will default to giving you the offset of var from the beginning of the *group*, not the *segment*. Therefore SEG var, also, will return the group base rather than the segment base.

NASM will allow a segment to be part of more than one group, but will generate a warning if you do this. Variables declared in a segment which is part of more than one group will default to being relative to the first group that was defined to contain the segment.

A group does not have to contain any segments; you can still make WRT references to a group which does not contain the variable you are referring to. OS/2, for example, defines the special group FLAT with no segments in it.

6.2.3. `UPPERCASE`: Disabling Case Sensitivity in Output

Although NASM itself is case sensitive, some OMF linkers are not; therefore it can be useful for NASM to output single-case object files. The `UPPERCASE` format-specific directive causes all segment, group and symbol names that are written to the object file to be forced to upper case just before being written. Within a source file, NASM is still case-sensitive; but the object file can be written entirely in upper case if desired.

`UPPERCASE` is used alone on a line; it requires no parameters.

6.2.4. `IMPORT`: Importing DLL Symbols

The `IMPORT` format-specific directive defines a symbol to be imported from a DLL, for use if you are writing a DLL's import library in NASM. You still need to declare the symbol as `EXTERN` as well as using the `IMPORT` directive.

The `IMPORT` directive takes two required parameters, separated by white space, which are (respectively) the name of the symbol you wish to import and the name of the library you wish to import it from. For example:

```
import WSAStartup wsock32.dll
```

A third optional parameter gives the name by which the symbol is known in the library you are importing it from, in case this is not the same as the name you wish the symbol to be known by to your code once you have imported it. For example:

```
import asyncsel wsock32.dll WSAAsyncSelect
```

6.2.5. `EXPORT`: Exporting DLL Symbols

The `EXPORT` format-specific directive defines a global symbol to be exported as a DLL symbol, for use if you are writing a DLL in NASM. You still need to declare the symbol as `GLOBAL` as well as using the `EXPORT` directive.

`EXPORT` takes one required parameter, which is the name of the symbol you wish to export, as it was defined in your source file. An optional second parameter (separated by white space from the first) gives the *external* name of the symbol: the name by which you wish the symbol to be known to programs using the DLL. If this name is the same as the internal name, you may leave the second parameter off.

Further parameters can be given to define attributes of the exported symbol. These parameters, like the second, are separated by white space. If further parameters are given, the external name must also be specified, even if it is the same as the internal name. The available attributes are:

- `resident` indicates that the exported name is to be kept resident by the system loader. This is an optimisation for frequently used symbols imported by name.
- `nodata` indicates that the exported symbol is a function which does not make use of any initialised data.
- `parm=NNN`, where `NNN` is an integer, sets the number of parameter words for the case in which the symbol is a call gate between 32-bit and 16-bit segments.
- An attribute which is just a number indicates that the symbol should be exported with an identifying number (ordinal), and gives the desired number.

For example:

```
export myfunc
export myfunc TheRealMoreFormalLookingFunctionName
export myfunc myfunc 1234  ; export by ordinal
export myfunc myfunc resident parm=23 nodata
```

6.2.6. `..start`: Defining the Program Entry Point

OMF linkers require exactly one of the object files being linked to define the program entry point, where execution will begin when the program is run. If the object file that defines the entry point is assembled using NASM, you specify the entry point by declaring the special symbol `..start` at the point where you wish execution to begin.

6.2.7. `obj` Extensions to the `EXTERN` Directive

If you declare an external symbol with the directive

```
extern foo
```

then references such as `mov ax,foo` will give you the offset of `foo` from its preferred segment base (as specified in whichever module `foo` is actually defined in). So to access the contents of `foo` you will usually need to do something like

```
mov ax,seg foo          ; get preferred segment base
mov es,ax               ; move it into ES
mov ax,[es:foo]         ; and use offset `foo' from it
```

This is a little unwieldy, particularly if you know that an external is going to be accessible from a given segment or group, say `dgroup`. So if `DS` already contained `dgroup`, you could simply code

```
mov ax,[foo wrt dgroup]
```

However, having to type this every time you want to access `foo` can be a pain; so NASM allows you to declare `foo` in the alternative form

```
extern foo:wrt dgroup
```

This form causes NASM to pretend that the preferred segment base of `foo` is in fact `dgroup`; so the expression `seg foo` will now return `dgroup`, and the expression `foo` is equivalent to `foo wrt dgroup`.

This default-`WRT` mechanism can be used to make externals appear to be relative to any group or segment in your program. It can also be applied to common variables: see section 6.2.8.

6.2.8. `obj` Extensions to the `COMMON` Directive

The `obj` format allows common variables to be either near or far; NASM allows you to specify which your variables should be by the use of the syntax

```
common nearvar 2:near  ; `nearvar' is a near common
common farvar 10:far   ; and `farvar' is far
```

Far common variables may be greater in size than 64Kb, and so the OMF specification says that they are declared as a number of *elements* of a given size. So a 10-byte far common variable could be declared as ten one-byte elements, five two-byte elements, two five-byte elements or one ten-byte element.

Some OMF linkers require the element size, as well as the variable size, to match when resolving common variables declared in more than one module. Therefore NASM must allow you to specify the element size on your far common variables. This is done by the following syntax:

```
common c_5by2 10:far 5 ; two five-byte elements
common c_2by5 10:far 2 ; five two-byte elements
```

If no element size is specified, the default is 1. Also, the `FAR` keyword is not required when an element size is specified, since only far commons may have element sizes at all. So the above declarations could equivalently be

```
common c_5by2 10:5     ; two five-byte elements
common c_2by5 10:2     ; five two-byte elements
```

In addition to these extensions, the `COMMON` directive in `obj` also supports default-`WRT` specification like `EXTERN` does (explained in section 6.2.7). So you can also declare things like

```
common foo 10:wrt dgroup
common bar 16:far 2:wrt data
common baz 24:wrt data:6
```

6.3. `win32`: Microsoft Win32 Object Files

The `win32` output format generates Microsoft Win32 object files, suitable for passing to Microsoft linkers such as Visual C++. Note that Borland Win32 compilers do not use this format, but use `obj` instead (see section 6.2).

`win32` provides a default output file-name extension of `.obj`.

Note that although Microsoft say that Win32 object files follow the COFF (Common Object File Format) standard, the object files produced by Microsoft Win32 compilers are not compatible with COFF linkers such as DJGPP's, and vice versa. This is due to a difference of opinion over the precise semantics of PC-relative relocations. To produce COFF files suitable for DJGPP, use NASM's `coff` output format; conversely, the `coff` format does not produce object files that Win32 linkers can generate correct output from.

Section 6.3.1: win32 Extensions to the SECTION Directive

6.3.1. `win32` Extensions to the `SECTION` Directive

Like the `obj` format, `win32` allows you to specify additional information on the `SECTION` directive line, to control the type and properties of sections you declare. Section types and properties are generated automatically by NASM for the standard section names `.text`, `.data` and `.bss`, but may still be overridden by these qualifiers.

The available qualifiers are:

- `code`, or equivalently `text`, defines the section to be a code section. This marks the section as readable and executable, but not writable, and also indicates to the linker that the type of the section is code.
- `data` and `bss` define the section to be a data section, analogously to `code`. Data sections are marked as readable and writable, but not executable. `data` declares an initialised data section, whereas `bss` declares an uninitialised data section.
- `info` defines the section to be an informational section, which is not included in the executable file by the linker, but may (for example) pass information *to* the linker. For example, declaring an `info`–type section called `.drectve` causes the linker to interpret the contents of the section as command-line options.
- `align=`, used with a trailing number as in `obj`, gives the alignment requirements of the section. The maximum you may specify is 64: the Win32 object file format contains no means to request a greater section alignment than this. If alignment is not explicitly specified, the defaults are 16-byte alignment for code sections, and 4-byte alignment for data (and BSS) sections. Informational sections get a default alignment of 1 byte (no alignment), though the value does not matter.

The defaults assumed by NASM if you do not specify the above qualifiers are:

```
section .text code align=16
section .data data align=4
section .bss bss align=4
```

Any other section name is treated by default like `.text`.

6.4. `coff`: Common Object File Format

The `coff` output type produces COFF object files suitable for linking with the DJGPP linker.

`coff` provides a default output file-name extension of `.o`.

The `coff` format supports the same extensions to the `SECTION` directive as `win32` does, except that the `align` qualifier and the `info` section type are not supported.

6.5. `elf`: Linux ELFObject Files

The `elf` output format generates ELF32 (Executable and Linkable Format) object files, as used by Linux. `elf` provides a default output file-name extension of `.o`.

6.5.1. `elf` Extensions to the `SECTION` Directive

Like the `obj` format, `elf` allows you to specify additional information on the `SECTION` directive line, to control the type and properties of sections you declare. Section types and properties are generated automatically by NASM for the standard section names `.text`, `.data` and `.bss`, but may still be overridden by these qualifiers.

The available qualifiers are:

- `alloc` defines the section to be one which is loaded into memory when the program is run. `noalloc` defines it to be one which is not, such as an informational or comment section.
- `exec` defines the section to be one which should have execute permission when the program is run. `noexec` defines it as one which should not.
- `write` defines the section to be one which should be writable when the program is run. `nowrite` defines it as one which should not.
- `progbits` defines the section to be one with explicit contents stored in the object file: an ordinary code or data section, for example, `nobits` defines the section to be one with no explicit contents given, such as a BSS section.
- `align=`, used with a trailing number as in `obj`, gives the alignment requirements of the section.

The defaults assumed by NASM if you do not specify the above qualifiers are:

```
section .text progbits alloc   exec nowrite align=16
section .data progbits alloc noexec   write align=4
section .bss    nobits alloc noexec   write align=4
section other progbits alloc noexec nowrite align=1
```

(Any section name other than `.text`, `.data` and `.bss` is treated by default like `other` in the above code.)

6.5.2. PositionIndependent Code: `elf` Special Symbols and `WRT`

The ELF specification contains enough features to allow position-independent code (PIC) to be written, which makes ELF shared libraries very flexible. However, it also means NASM has to be able to generate a variety of strange relocation types in ELF object files, if it is to be an assembler which can write PIC.

Since ELF does not support segment-base references, the `WRT` operator is not used for its normal purpose; therefore NASM's `elf` output format makes use of `WRT` for a different purpose, namely the PIC-specific relocation types.

`elf` defines five special symbols which you can use as the right-hand side of the `WRT` operator to obtain PIC relocation types. They are `..gotpc`, `..gotoff`, `..got`, `..plt` and `..sym`. Their functions are summarised here:

- Referring to the symbol marking the global offset table base using `wrt ..gotpc` will end up giving the distance from the beginning of the current section to the global offset table. (`_GLOBAL_OFFSET_TABLE_` is the standard symbol name used to refer to the GOT.) So you would then need to add `$$` to the result to get the real address of the GOT.
- Referring to a location in one of your own sections using `wrt ..gotoff` will give the distance from the beginning of the GOT to the specified location, so that adding on the address of the GOT would give the real address of the location you wanted.
- Referring to an external or global symbol using `wrt ..got` causes the linker to build an entry *in* the GOT containing the address of the symbol, and the reference gives the distance from the beginning of the GOT to the entry; so you can add on the address of the GOT, load from the resulting address, and end up with the address of the symbol.
- Referring to a procedure name using `wrt ..plt` causes the linker to build a procedure linkage table entry for the symbol, and the reference gives the address of the PLT entry. You can only use this in contexts which would generate a PC-relative relocation normally (i.e. as the destination for `CALL` or `JMP`), since ELF contains no relocation type to refer to PLT entries absolutely.
- Referring to a symbol name using `wrt ..sym` causes NASM to write an ordinary relocation, but instead of making the relocation relative to the start of the section and then adding on the offset to the symbol, it will write a relocation record aimed directly at the symbol in question. The distinction is a necessary one due to a peculiarity of the dynamic linker.

A fuller explanation of how to use these relocation types to write shared libraries entirely in NASM is given in section 8.2.

6.5.3. `elf` Extensions to the `GLOBAL` Directive

ELF object files can contain more information about a global symbol than just its address: they can contain the size of the symbol and its type as well. These are not merely debugger conveniences, but are actually necessary when the program being written is a shared library. NASM therefore supports some extensions to the `GLOBAL` directive, allowing you to specify these features.

You can specify whether a global variable is a function or a data object by suffixing the name with a colon and the word `function` or `data`. (`object` is a synonym for `data`.) For example:

```
global hashlookup:function, hashtable:data
```

exports the global symbol `hashlookup` as a function and `hashtable` as a data object.

You can also specify the size of the data associated with the symbol, as a numeric expression (which may involve labels, and even forward references) after the type specifier. Like this:

```
global hashtable:data (hashtable.end - hashtable)
hashtable:
db this,that,theother  ; some data here
.end:
```

This makes NASM automatically calculate the length of the table and place that information into the ELF symbol table.

Declaring the type and size of global symbols is necessary when writing shared library code. For more information, see section 8.2.4.

6.5.4. `elf` Extensions to the `COMMON` Directive

ELF also allows you to specify alignment requirements on common variables. This is done by putting a number (which must be a power of two) after the name and size of the common variable, separated (as usual) by a colon. For example, an array of doublewords would benefit from 4-byte alignment:

```
common dwordarray 128:4
```

This declares the total size of the array to be 128 bytes, and requires that it be aligned on a 4-byte boundary.

6.6. `aout`: Linux `a.out` Object Files

The `aout` format generates `a.out` object files, in the form used by early Linux systems. (These differ from other `a.out` object files in that the magic number in the first four bytes of the file is different. Also, some implementations of `a.out`, for example NetBSD's, support position-independent code, which Linux's implementation doesn't.)

`a.out` provides a default output file-name extension of `.o`.

`a.out` is a very simple object format. It supports no special directives, no special symbols, no use of `SEG` or `WRT`, and no extensions to any standard directives. It supports only the three standard section names `.text`, `.data` and `.bss`.

6.7. `aoutb`: NetBSD/FreeBSD/OpenBSD `a.out` Object Files

The `aoutb` format generates `a.out` object files, in the form used by the various free BSD Unix clones, NetBSD, FreeBSD and OpenBSD. For simple object files, this object format is exactly the same as `aout` except for the magic number in the first four bytes of the file. However, the `aoutb` format supports positionindependent code in the same way as the `elf` format, so you can use it to write BSD shared libraries.

`aoutb` provides a default output file-name extension of `.o`.

`aoutb` supports no special directives, no special symbols, and only the three standard section names `.text`, `.data` and `.bss`. However, it also supports the same use of `WRT` as `elf` does, to provide position-independent code relocation types. See section 6.5.2 for full documentation of this feature.

`aoutb` also supports the same extensions to the `GLOBAL` directive as `elf` does: see section 6.5.3 for documentation of this.

6.8. `as86`: Linux `as86` Object Files

The Linux 16-bit assembler `as86` has its own non-standard object file format. Although its companion linker `ld86` produces something close to ordinary `a.out` binaries as output, the object file format used to communicate between `as86` and `ld86` is not itself `a.out`.

NASM supports this format, just in case it is useful, as `as86`. `as86` provides a default output file-name extension of `.o`.

`as86` is a very simple object format (from the NASM user's point of view). It supports no special directives, no special symbols, no use of `SEG` or `WRT`, and no extensions to any standard directives. It supports only the three standard section names `.text`, `.data` and `.bss`.

6.9. `rdf`: Relocatable Dynamic Object File Format

The `rdf` output format produces RDOFF object files. RDOFF (Relocatable Dynamic Object File Format) is a home-grown object-file format, designed alongside NASM itself and reflecting in its file format the internal structure of the assembler.

RDOFF is not used by any well-known operating systems. Those writing their own systems, however, may well wish to use RDOFF as their object format, on the grounds that it is designed primarily for simplicity and contains very little file-header bureaucracy.

The Unix NASM archive, and the DOS archive which includes sources, both contain an `rdoff` subdirectory holding a set of RDOFF utilities: an RDF linker, an RDF static-library manager, an RDF file dump utility, and a program which will load and execute an RDF executable under Linux.

`rdf` supports only the standard section names `.text`, `.data` and `.bss`.

Section 6.9.1: Requiring a Library: The LIBRARY Directive

6.9.1. Requiring a Library: The `LIBRARY` Directive

RDOFF contains a mechanism for an object file to demand a given library to be linked to the module, either at load time or run time. This is done by the `LIBRARY` directive, which takes one argument which is the name of the module:

```
library mylib.rdl
```

6.10. dbg: Debugging Format

The dbg output format is not built into NASM in the default configuration. If you are building your own NASM executable from the sources, you can define OF_DBG in outform.h or on the compiler command line, and obtain the dbg output format.

The dbg format does not output an object file as such; instead, it outputs a text file which contains a complete list of all the transactions between the main body of NASM and the output-format back end module. It is primarily intended to aid people who want to write their own output drivers, so that they can get a clearer idea of the various requests the main program makes of the output driver, and in what order they happen.

For simple files, one can easily use the dbg format like this:

```
nasm -f dbg filename.asm
```

which will generate a diagnostic file called filename.dbg. However, this will not work well on files which were designed for a different object format, because each object format defines its own macros (usually user-level forms of directives), and those macros will not be defined in the dbg format. Therefore it can be useful to run NASM twice, in order to do the preprocessing with the native object format selected:

```
nasm -e -f rdf -o rdfprog.i rdfprog.asm
nasm -a -f dbg rdfprog.i
```

This preprocesses rdfprog.asm into rdfprog.i, keeping the rdf object format selected in order to make sure RDF special directives are converted into primitive form correctly. Then the preprocessed source is fed through the dbg format to generate the final diagnostic output.

This workaround will still typically not work for programs intended for obj format, because the obj SEGMENT and GROUP directives have side effects of defining the segment and group names as symbols; dbg will not do this, so the program will not assemble. You will have to work around that by defining the symbols yourself (using EXTERN, for example) if you really need to get a dbg trace of an obj–specific source file.

dbg accepts any section name and any directives at all, and logs them all to its output file.

Chapter 7: Writing 16-bit Code (DOS, Windows 3/3.1)

This chapter attempts to cover some of the common issues encountered when writing 16-bit code to run under MS-DOS or Windows 3.x. It covers how to link programs to produce `.EXE` or `.COM` files, how to write `.SYS` device drivers, and how to interface assembly language code with 16-bit C compilers and with Borland Pascal.

7.1. Producing `.EXE` Files

Any large program written under DOS needs to be built as a `.EXE` file: only `.EXE` files have the necessary internal structure required to span more than one 64K segment. Windows programs, also, have to be built as `.EXE` files, since Windows does not support the `.COM` format.

In general, you generate `.EXE` files by using the `obj` output format to produce one or more `.OBJ` files, and then linking them together using a linker. However, NASM also supports the direct generation of simple DOS `.EXE` files using the `bin` output format (by using `DB` and `DW` to construct the `.EXE` file header), and a macro package is supplied to do this. Thanks to Yann Guidon for contributing the code for this.

NASM may also support `.EXE` natively as another output format in future releases.

7.1.1. Using the `obj` Format To Generate `.EXE` Files

This section describes the usual method of generating `.EXE` files by linking `.OBJ` files together.

Most 16-bit programming language packages come with a suitable linker; if you have none of these, there is a free linker called VAL, available in `LZH` archive format from `x2ftp.oulu.fi`. An LZH archiver can be found at `ftp.simtel.net`. There is another `free' linker (though this one doesn't come with sources) called FREELINK, available from `www.pcorner.com`. A third, `djlink`, written by DJ Delorie, is available at `www.delorie.com`.

When linking several `.OBJ` files into a `.EXE` file, you should ensure that exactly one of them has a start point defined (using the `..start` special symbol defined by the `obj` format: see section 6.2.6). If no module defines a start point, the linker will not know what value to give the entry-point field in the output file header; if more than one defines a start point, the linker will not know *which* value to use.

An example of a NASM source file which can be assembled to a `.OBJ` file and linked on its own to a `.EXE` is given here. It demonstrates the basic principles of defining a stack, initialising the segment registers, and declaring a start point. This file is also provided in the `test` subdirectory of the NASM archives, under the name `objexe.asm`.

```
           segment code

..start:   mov ax,data
           mov ds,ax
           mov ax,stack
           mov ss,ax
           mov sp,stacktop
```

This initial piece of code sets up `DS` to point to the data segment, and initialises `SS` and `SP` to point to the top of the provided stack. Notice that interrupts are implicitly disabled for one instruction after a move into `SS`, precisely for this situation, so that there's no chance of an interrupt occurring between the loads of `SS` and `SP` and not having a stack to execute on.

Note also that the special symbol `..start` is defined at the beginning of this code, which means that will be the entry point into the resulting executable file.

```
           mov dx,hello
           mov ah,9
           int 0x21
```

The above is the main program: load `DS:DX` with a pointer to the greeting message (`hello` is implicitly relative to the segment `data`, which was loaded into `DS` in the setup code, so the full pointer is valid), and call the DOS print-string function.

```
           mov ax,0x4c00
           int 0x21
```

This terminates the program using another DOS system call.

```
           segment data
```

```
hello:    db 'hello, world', 13, 10, '$'
```

The data segment contains the string we want to display.

```
          segment stack stack
          resb 64
stacktop:
```

The above code declares a stack segment containing 64 bytes of uninitialised stack space, and points `stacktop` at the top of it. The directive `segment stack stack` defines a segment *called* `stack`, and also of *type* `STACK`. The latter is not necessary to the correct running of the program, but linkers are likely to issue warnings or errors if your program has no segment of type `STACK`.

The above file, when assembled into a `.OBJ` file, will link on its own to a valid `.EXE` file, which when run will print `hello, world' and then exit.

7.1.2. Using the `bin` Format To Generate `.EXE` Files

The `.EXE` file format is simple enough that it's possible to build a `.EXE` file by writing a pure-binary program and sticking a 32-byte header on the front. This header is simple enough that it can be generated using `DB` and `DW` commands by NASM itself, so that you can use the `bin` output format to directly generate `.EXE` files.

Included in the NASM archives, in the `misc` subdirectory, is a file `exebin.mac` of macros. It defines three macros: `EXE_begin`, `EXE_stack` and `EXE_end`.

To produce a `.EXE` file using this method, you should start by using `%include` to load the `exebin.mac` macro package into your source file. You should then issue the `EXE_begin` macro call (which takes no arguments) to generate the file header data. Then write code as normal for the `bin` format – you can use all three standard sections `.text`, `.data` and `.bss`. At the end of the file you should call the `EXE_end` macro (again, no arguments), which defines some symbols to mark section sizes, and these symbols are referred to in the header code generated by `EXE_begin`.

In this model, the code you end up writing starts at `0x100`, just like a `.COM` file – in fact, if you strip off the 32-byte header from the resulting `.EXE` file, you will have a valid `.COM` program. All the segment bases are the same, so you are limited to a 64K program, again just like a `.COM` file. Note that an `ORG` directive is issued by the `EXE_begin` macro, so you should not explicitly issue one of your own.

You can't directly refer to your segment base value, unfortunately, since this would require a relocation in the header, and things would get a lot more complicated. So you should get your segment base by copying it out of `CS` instead.

On entry to your `.EXE` file, `SS:SP` are already set up to point to the top of a 2Kb stack. You can adjust the default stack size of 2Kb by calling the `EXE_stack` macro. For example, to change the stack size of your program to 64 bytes, you would call `EXE_stack 64`.

A sample program which generates a `.EXE` file in this way is given in the `test` subdirectory of the NASM archive, as `binexe.asm`.

7.2. Producing `.COM` Files

While large DOS programs must be written as `.EXE` files, small ones are often better written as `.COM` files. `.COM` files are pure binary, and therefore most easily produced using the `bin` output format.

7.2.1. Using the `bin` Format To Generate `.COM` Files

`.COM` files expect to be loaded at offset `100h` into their segment (though the segment may change). Execution then begins at `100h`, i.e. right at the start of the program. So to write a `.COM` program, you would create a source file looking like

```
          org 100h
          section .text
start:    ; put your code here
          section .data
          ; put data items here
          section .bss
          ; put uninitialised data here
```

The `bin` format puts the `.text` section first in the file, so you can declare data or BSS items before beginning to write code if you want to and the code will still end up at the front of the file where it belongs.

The BSS (uninitialised data) section does not take up space in the `.COM` file itself: instead, addresses of BSS items are resolved to point at space beyond the end of the file, on the grounds that this will be free memory when the program is run. Therefore you should not rely on your BSS being initialised to all zeros when you run.

To assemble the above program, you should use a command line like

```
nasm myprog.asm -fbin -o myprog.com
```

The `bin` format would produce a file called `myprog` if no explicit output file name were specified, so you have to override it and give the desired file name.

7.2.2. Using the `obj` Format To Generate `.COM` Files

If you are writing a `.COM` program as more than one module, you may wish to assemble several `.OBJ` files and link them together into a `.COM` program. You can do this, provided you have a linker capable of outputting `.COM` files directly (TLINK does this), or alternatively a converter program such as `EXE2BIN` to transform the `.EXE` file output from the linker into a `.COM` file.

If you do this, you need to take care of several things:

- The first object file containing code should start its code segment with a line like `RESB 100h`. This is to ensure that the code begins at offset `100h` relative to the beginning of the code segment, so that the linker or converter program does not have to adjust address references within the file when generating the `.COM` file. Other assemblers use an `ORG` directive for this purpose, but `ORG` in NASM is a format-specific directive to the `bin` output format, and does not mean the same thing as it does in MASM-compatible assemblers.
- You don't need to define a stack segment.
- All your segments should be in the same group, so that every time your code or data references a symbol offset, all offsets are relative to the same segment base. This is because, when a `.COM` file is loaded, all the segment registers contain the same value.

7.3. Producing `.SYS` Files

MSDOS device drivers – `.SYS` files – are pure binary files, similar to `.COM` files, except that they start at origin zero rather than `100h`. Therefore, if you are writing a device driver using the `bin` format, you do not need the `ORG` directive, since the default origin for `bin` is zero. Similarly, if you are using `obj`, you do not need the `RESB 100h` at the start of your code segment.

`.SYS` files start with a header structure, containing pointers to the various routines inside the driver which do the work. This structure should be defined at the start of the code segment, even though it is not actually code.

For more information on the format of `.SYS` files, and the data which has to go in the header structure, a list of books is given in the Frequently Asked Questions list for the newsgroup `comp.os.msdos.programmer`.

7.4. Interfacing to 16-bit C Programs

This section covers the basics of writing assembly routines that call, or are called from, C programs. To do this, you would typically write an assembly module as a `.OBJ` file, and link it with your C modules to produce a mixedlanguage program.

7.4.1. External Symbol Names

C compilers have the convention that the names of all global symbols (functions or data) they define are formed by prefixing an underscore to the name as it appears in the C program. So, for example, the function a C programmer thinks of as `printf` appears to an assembly language programmer as `_printf`. This means that in your assembly programs, you can define symbols without a leading underscore, and not have to worry about name clashes with C symbols.

If you find the underscores inconvenient, you can define macros to replace the `GLOBAL` and `EXTERN` directives as follows:

```
%macro cglobal 1
          global _%1
%define %1 _%1
%endmacro

%macro cextern 1
          extern _%1
%define %1 _%1
%endmacro
```

(These forms of the macros only take one argument at a time; a `%rep` construct could solve this.)

If you then declare an external like this:

```
          cextern printf
```

then the macro will expand it as

```
          extern _printf
%define printf _printf
```

Thereafter, you can reference `printf` as if it was a symbol, and the preprocessor will put the leading underscore on where necessary.

The `cglobal` macro works similarly. You must use `cglobal` before defining the symbol in question, but you would have had to do that anyway if you used `GLOBAL`.

7.4.2. Memory Models

NASM contains no mechanism to support the various C memory models directly; you have to keep track yourself of which one you are writing for. This means you have to keep track of the following things:

- In models using a single code segment (tiny, small and compact), functions are near. This means that function pointers, when stored in data segments or pushed on the stack as function arguments, are 16 bits long and contain only an offset field (the `CS` register never changes its value, and always gives the segment part of the full function address), and that functions are called using ordinary near `CALL` instructions and return using `RETN` (which, in NASM, is synonymous with `RET` anyway). This means both that you should write your own routines to return with `RETN`, and that you should call external C routines with near `CALL` instructions.
- In models using more than one code segment (medium, large and huge), functions are far. This means that function pointers are 32 bits long (consisting of a 16-bit offset followed by a 16-bit segment), and that functions are called using `CALL FAR` (or `CALL seg:offset`) and return using `RETF`. Again, you should therefore write your own routines to return with `RETF` and use `CALL FAR` to call external routines.
- In models using a single data segment (tiny, small and medium), data pointers are 16 bits long, containing only an offset field (the `DS` register doesn't change its value, and always gives the segment part of the full data item address).
- In models using more than one data segment (compact, large and huge), data pointers are 32 bits long, consisting of a 16-bit offset followed by a 16-bit segment. You should still be careful not to modify `DS` in your routines without restoring it afterwards, but `ES` is free for you to use to access the contents of 32-bit data pointers you are passed.
- The huge memory model allows single data items to exceed 64K in size. In all other memory models, you can access the whole of a data item just by doing arithmetic on the offset field of the pointer you are given, whether a segment field is present or not; in huge model, you have to be more careful of your pointer arithmetic.
- In most memory models, there is a *default* data segment, whose segment address is kept in `DS` throughout the program. This data segment is typically the same segment as the stack, kept in `SS`, so that functions' local variables (which are stored on the stack) and global data items can both be accessed easily without changing `DS`. Particularly large data items are typically stored in other segments. However, some memory models (though not the standard ones, usually) allow the assumption that `SS` and `DS` hold the same value to be removed. Be careful about functions' local variables in this latter case.

In models with a single code segment, the segment is called `_TEXT`, so your code segment must also go by this name in order to be linked into the same place as the main code segment. In models with a single data segment, or with a default data segment, it is called `_DATA`.

7.4.3. Function Definitions and Function Calls

The C calling convention in 16-bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function's parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
- The caller then executes a `CALL` instruction to pass control to the callee. This `CALL` is either near or far depending on the memory model.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `SP` in `BP` so as to be able to use `BP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `BP` must be preserved by any C function. Hence the callee, if it is going to set up `BP` as a *frame pointer*, must push the previous value first.
- The callee may then access its parameters relative to `BP`. The word at `[BP]` holds the previous value of `BP` as it was pushed; the next word, at `[BP+2]`, holds the offset part of the return address, pushed implicitly by `CALL`. In a small-model (near) function, the parameters start after that, at `[BP+4]`; in a large-model (far) function, the segment part of the return address lives at `[BP+4]`, and the parameters begin at `[BP+6]`. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from `BP`; the others follow, at successively greater offsets. Thus, in a function such as `printf` which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
- The callee may also wish to decrease `SP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `BP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `DX:AX` depending on the size of the value. Floating-point results are sometimes (depending on the compiler) returned in `ST0`.
- Once the callee has finished processing, it restores `SP` from `BP` if it had allocated local stack space, then pops the previous value of `BP`, and returns via `RETN` or `RETF` depending on memory model.
- When the caller regains control from the callee, the function parameters are still on the stack, so it typically adds an immediate constant to `SP` to remove them (instead of executing a number of slow `POP` instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the caller, which *knows* how many parameters it pushed, does the removing.

It is instructive to compare this calling convention with that for Pascal programs (described in section 7.5.1). Pascal has a simpler convention, since no functions have variable numbers of

parameters. Therefore the callee knows how many parameters it should have been passed, and is able to deallocate them from the stack itself by passing an immediate argument to the `RET` or `RETF` instruction, so the caller does not have to do it. Also, the parameters are pushed in left-to-right order, not right-to-left, which means that a compiler can give better guarantees about sequence points without performance suffering.

Thus, you would define a function in C style in the following way. The following example is for small model:

```
          global _myfunc
_myfunc:  push bp
          mov bp,sp
          sub sp,0x40            ; 64 bytes of local stack space
          mov bx,[bp+4]          ; first parameter to function
          ; some more code
          mov sp,bp              ; undo "sub sp,0x40" above
          pop bp
          ret
```

For a large-model function, you would replace `RET` by `RETF`, and look for the first parameter at `[BP+6]` instead of `[BP+4]`. Of course, if one of the parameters is a pointer, then the offsets of *subsequent* parameters will change depending on the memory model as well: far pointers take up four bytes on the stack when passed as a parameter, whereas near pointers take up two.

At the other end of the process, to call a C function from your assembly code, you would do something like this:

```
          extern _printf
          ; and then, further down...
          push word [myint]      ; one of my integer variables
          push word mystring     ; pointer into my data segment
          call _printf
          add sp,byte 4          ; `byte' saves space
          ; then those data items...
          segment _DATA
myint     dw 1234
mystring  db 'This number -> %d <- should be 1234',10,0
```

This piece of code is the small-model assembly equivalent of the C code

```
    int myint = 1234;
    printf("This number -> %d <- should be 1234\n", myint);
```

In large model, the function-call code might look more like this. In this example, it is assumed that `DS` already holds the segment base of the segment `_DATA`. If not, you would have to initialise it first.

```
          push word [myint]
          push word seg mystring ; Now push the segment, and...
          push word mystring     ; ... offset of "mystring"
          call far _printf
          add sp,byte 6
```

The integer value still takes up one word on the stack, since large model does not affect the size of the `int` data type. The first argument (pushed last) to `printf`, however, is a data pointer, and therefore has to contain a segment and offset part. The segment should be stored second in memory, and therefore must be pushed first. (Of course, `PUSH DS` would have been a shorter instruction than `PUSH WORD SEG mystring`, if `DS` was set up as the above example assumed.) Then the actual call becomes a far call, since functions expect far calls in large model; and `SP` has to be increased by 6 rather than 4 afterwards to make up for the extra word of parameters.

7.4.4. Accessing Data Items

To get at the contents of C variables, or to declare variables which C can access, you need only declare the names as `GLOBAL` or `EXTERN`. (Again, the names require leading underscores, as stated in section 7.4.1.) Thus, a C variable declared as `int i` can be accessed from assembler as

```
          extern _i
          mov ax,[_i]
```

And to declare your own integer variable which C programs can access as `extern int j`, you do this (making sure you are assembling in the `_DATA` segment, if necessary):

```
          global _j
_j        dw 0
```

To access a C array, you need to know the size of the components of the array. For example, `int` variables are two bytes long, so if a C program declares an array as `int a[10]`, you can access `a[3]` by coding `mov ax,[_a+6]`. (The byte offset 6 is obtained by multiplying the desired array index, 3, by the size of the array element, 2.) The sizes of the C base types in 16-bit compilers are: 1 for `char`, 2 for `short` and `int`, 4 for `long` and `float`, and 8 for `double`.

To access a C data structure, you need to know the offset from the base of the structure to the field you are interested in. You can either do this by converting the C structure definition into a NASM structure definition (using `STRUC`), or by calculating the one offset and using just that.

To do either of these, you should read your C compiler's manual to find out how it organises data structures. NASM gives no special alignment to structure members in its own `STRUC` macro, so you have to specify alignment yourself if the C compiler generates it. Typically, you might find that a structure like

```
struct @{
    char c;
    int i;
@} foo;
```

might be four bytes long rather than three, since the `int` field would be aligned to a two-byte boundary. However, this sort of feature tends to be a configurable option in the C compiler, either using command-line options or `#pragma` lines, so you have to find out how your own compiler does it.

7.4.5. `c16.mac`: Helper Macros for the 16-bit C Interface

Included in the NASM archives, in the `misc` directory, is a file `c16.mac` of macros. It defines three macros: `proc`, `arg` and `endproc`. These are intended to be used for C-style procedure definitions, and they automate a lot of the work involved in keeping track of the calling convention.

An example of an assembly function using the macro set is given here:

```
          proc _nearproc
%$i       arg
%$j       arg
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          add ax,[bx]
          endproc
```

This defines `_nearproc` to be a procedure taking two arguments, the first (`i`) an integer and the second (`j`) a pointer to an integer. It returns `i + *j`.

Note that the `arg` macro has an `EQU` as the first line of its expansion, and since the label before the macro call gets prepended to the first line of the expanded macro, the `EQU` works, defining `%$i` to be an offset from `BP`. A context-local variable is used, local to the context pushed by the `proc` macro and popped by the `endproc` macro, so that the same argument name can be used in later procedures. Of course, you don't *have* to do that.

The macro set produces code for near functions (tiny, small and compact-model code) by default. You can have it generate far functions (medium, large and huge-model code) by means of coding `%define FARCODE`. This changes the kind of return instruction generated by `endproc`, and also changes the starting point for the argument offsets. The macro set contains no intrinsic dependency on whether data pointers are far or not.

`arg` can take an optional parameter, giving the size of the argument. If no size is given, 2 is assumed, since it is likely that many function parameters will be of type `int`.

The large-model equivalent of the above function would look like this:

```
%define FARCODE
          proc _farproc
%$i       arg
%$j       arg 4
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          mov es,[bp + %$j + 2]
          add ax,[bx]
          endproc
```

This makes use of the argument to the `arg` macro to define a parameter of size 4, because `j` is now a far pointer. When we load from `j`, we must load a segment and an offset.

7.5. Interfacing to Borland Pascal Programs

Interfacing to Borland Pascal programs is similar in concept to interfacing to 16-bit C programs. The differences are:

- The leading underscore required for interfacing to C programs is not required for Pascal.
- The memory model is always large: functions are far, data pointers are far, and no data item can be more than 64K long. (Actually, some functions are near, but only those functions that are local to a Pascal unit and never called from outside it. All assembly functions that Pascal calls, and all Pascal functions that assembly routines are able to call, are far.) However, all static data declared in a Pascal program goes into the default data segment, which is the one whose segment address will be in `DS` when control is passed to your assembly code. The only things that do not live in the default data segment are local variables (they live in the stack segment) and dynamically allocated variables. All data *pointers*, however, are far.
- The function calling convention is different – described below.
- Some data types, such as strings, are stored differently.
- There are restrictions on the segment names you are allowed to use – Borland Pascal will ignore code or data declared in a segment it doesn't like the name of. The restrictions are described below.

Section 7.5.1: The Pascal Calling Convention
Section 7.5.2: Borland Pascal Segment Name Restrictions
Section 7.5.3: Using c16.mac With Pascal Programs

7.5.1. The Pascal Calling Convention

The 16-bit Pascal calling convention is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function's parameters on the stack, one after another, in normal order (left to right, so that the first argument specified to the function is pushed first).
- The caller then executes a far `CALL` instruction to pass control to the callee.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `SP` in `BP` so as to be able to use `BP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `BP` must be preserved by any function. Hence the callee, if it is going to set up `BP` as a frame pointer, must push the previous value first.
- The callee may then access its parameters relative to `BP`. The word at `[BP]` holds the previous value of `BP` as it was pushed. The next word, at `[BP+2]`, holds the offset part of the return address, and the next one at `[BP+4]` the segment part. The parameters begin at `[BP+6]`. The rightmost parameter of the function, since it was pushed last, is accessible at this offset from `BP`; the others follow, at successively greater offsets.
- The callee may also wish to decrease `SP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `BP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `DX:AX` depending on the size of the value. Floating-point results are returned in `ST0`. Results of type `Real` (Borland's own custom floating-point data type, not handled directly by the FPU) are returned in `DX:BX:AX`. To return a result of type `String`, the caller pushes a pointer to a temporary string before pushing the parameters, and the callee places the returned string value at that location. The pointer is not a parameter, and should not be removed from the stack by the `RETF` instruction.
- Once the callee has finished processing, it restores `SP` from `BP` if it had allocated local stack space, then pops the previous value of `BP`, and returns via `RETF`. It uses the form of `RETF` with an immediate parameter, giving the number of bytes taken up by the parameters on the stack. This causes the parameters to be removed from the stack as a side effect of the return instruction.
- When the caller regains control from the callee, the function parameters have already been removed from the stack, so it needs to do nothing further.

Thus, you would define a function in Pascal style, taking two `Integer`–type parameters, in the following way:

```
          global myfunc
myfunc:   push bp
          mov bp,sp
          sub sp,0x40             ; 64 bytes of local stack space
          mov bx,[bp+8]           ; first parameter to function
```

```
mov bx,[bp+6]          ; second parameter to function
; some more code
mov sp,bp              ; undo "sub sp,0x40" above
pop bp
retf 4                 ; total size of params is 4
```

At the other end of the process, to call a Pascal function from your assembly code, you would do something like this:

```
extern SomeFunc
; and then, further down...
push word seg mystring ; Now push the segment, and...
push word mystring     ; ... offset of "mystring"
push word [myint]      ; one of my variables
call far SomeFunc
```

This is equivalent to the Pascal code

```
procedure SomeFunc(String: PChar; Int: Integer);
    SomeFunc(@@mystring, myint);
```

7.5.2. Borland Pascal Segment Name Restrictions

Since Borland Pascal's internal unit file format is completely different from `OBJ`, it only makes a very sketchy job of actually reading and understanding the various information contained in a real `OBJ` file when it links that in. Therefore an object file intended to be linked to a Pascal program must obey a number of restrictions:

- Procedures and functions must be in a segment whose name is either `CODE`, `CSEG`, or something ending in `_TEXT`.
- Initialised data must be in a segment whose name is either `CONST` or something ending in `_DATA`.
- Uninitialised data must be in a segment whose name is either `DATA`, `DSEG`, or something ending in `_BSS`.
- Any other segments in the object file are completely ignored. `GROUP` directives and segment attributes are also ignored.

7.5.3. Using `c16.mac` With Pascal Programs

The `c16.mac` macro package, described in section 7.4.5, can also be used to simplify writing functions to be called from Pascal programs, if you code `%define PASCAL`. This definition ensures that functions are far (it implies `FARCODE`), and also causes procedure return instructions to be generated with an operand.

Defining `PASCAL` does not change the code which calculates the argument offsets; you must declare your function's arguments in reverse order. For example:

```
%define PASCAL
          proc _pascalproc
%$j       arg 4
%$i       arg
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          mov es,[bp + %$j + 2]
          add ax,[bx]
          endproc
```

This defines the same routine, conceptually, as the example in section 7.4.5: it defines a function taking two arguments, an integer and a pointer to an integer, which returns the sum of the integer and the contents of the pointer. The only difference between this code and the large-model C version is that `PASCAL` is defined instead of `FARCODE`, and that the arguments are declared in reverse order.

Chapter 8: Writing 32-bit Code (Unix, Win32, DJGPP)

This chapter attempts to cover some of the common issues involved when writing 32-bit code, to run under Win32 or Unix, or to be linked with C code generated by a Unix-style C compiler such as DJGPP. It covers how to write assembly code to interface with 32-bit C routines, and how to write position-independent code for shared libraries.

Almost all 32-bit code, and in particular all code running under Win32, DJGPP or any of the PC Unix variants, runs in *flat* memory model. This means that the segment registers and paging have already been set up to give you the same 32-bit 4Gb address space no matter what segment you work relative to, and that you should ignore all segment registers completely. When writing flat-model application code, you never need to use a segment override or modify any segment register, and the code-section addresses you pass to `CALL` and `JMP` live in the same address space as the data-section addresses you access your variables by and the stack-section addresses you access local variables and procedure parameters by. Every address is 32 bits long and contains only an offset part.

8.1. Interfacing to 32-bit C Programs

A lot of the discussion in section 7.4, about interfacing to 16-bit C programs, still applies when working in 32 bits. The absence of memory models or segmentation worries simplifies things a lot.

8.1.1. External Symbol Names

Most 32-bit C compilers share the convention used by 16-bit compilers, that the names of all global symbols (functions or data) they define are formed by prefixing an underscore to the name as it appears in the C program. However, not all of them do: the ELF specification states that C symbols do *not* have a leading underscore on their assembly-language names.

The older Linux `a.out` C compiler, all Win32 compilers, DJGPP, and NetBSD and FreeBSD, all use the leading underscore; for these compilers, the macros `cextern` and `cglobal`, as given in section 7.4.1, will still work. For ELF, though, the leading underscore should not be used.

8.1.2. Function Definitions and Function Calls

The C calling conventionThe C calling convention in 32-bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function's parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
- The caller then executes a near `CALL` instruction to pass control to the callee.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `ESP` in `EBP` so as to be able to use `EBP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `EBP` must be preserved by any C function. Hence the callee, if it is going to set up `EBP` as a frame pointer, must push the previous value first.
- The callee may then access its parameters relative to `EBP`. The doubleword at `[EBP]` holds the previous value of `EBP` as it was pushed; the next doubleword, at `[EBP+4]`, holds the return address, pushed implicitly by `CALL`. The parameters start after that, at `[EBP+8]`. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from `EBP`; the others follow, at successively greater offsets. Thus, in a function such as `printf` which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
- The callee may also wish to decrease `ESP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `EBP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `EAX` depending on the size of the value. Floating-point results are typically returned in `ST0`.
- Once the callee has finished processing, it restores `ESP` from `EBP` if it had allocated local stack space, then pops the previous value of `EBP`, and returns via `RET` (equivalently, `RETN`).
- When the caller regains control from the callee, the function parameters are still on the stack, so it typically adds an immediate constant to `ESP` to remove them (instead of executing a number of slow `POP` instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the caller, which *knows* how many parameters it pushed, does the removing.

There is an alternative calling convention used by Win32 programs for Windows API calls, and also for functions called *by* the Windows API such as window procedures: they follow what Microsoft calls the `__stdcall` convention. This is slightly closer to the Pascal convention, in that the callee clears the stack by passing a parameter to the `RET` instruction. However, the parameters are still pushed in right-to-left order.

Thus, you would define a function in C style in the following way:

```
          global _myfunc
_myfunc:  push ebp
          mov ebp,esp
          sub esp,0x40            ; 64 bytes of local stack space
          mov ebx,[ebp+8]         ; first parameter to function
          ; some more code
          leave                   ; mov esp,ebp / pop ebp
          ret
```

At the other end of the process, to call a C function from your assembly code, you would do something like this:

```
          extern _printf
          ; and then, further down...
          push dword [myint]      ; one of my integer variables
          push dword mystring     ; pointer into my data segment
          call _printf
          add esp,byte 8          ; `byte' saves space
          ; then those data items...
          segment _DATA
myint     dd 1234
mystring  db 'This number -> %d <- should be 1234',10,0
```

This piece of code is the assembly equivalent of the C code

```
int myint = 1234;
printf("This number -> %d <- should be 1234\n", myint);
```

8.1.3. Accessing Data Items

To get at the contents of C variables, or to declare variables which C can access, you need only declare the names as `GLOBAL` or `EXTERN`. (Again, the names require leading underscores, as stated in section 8.1.1.) Thus, a C variable declared as `int i` can be accessed from assembler as

```
          extern _i
          mov eax,[_i]
```

And to declare your own integer variable which C programs can access as `extern int j`, you do this (making sure you are assembling in the `_DATA` segment, if necessary):

```
          global _j
_j        dd 0
```

To access a C array, you need to know the size of the components of the array. For example, `int` variables are four bytes long, so if a C program declares an array as `int a[10]`, you can access `a[3]` by coding `mov ax,[_a+12]`. (The byte offset 12 is obtained by multiplying the desired array index, 3, by the size of the array element, 4.) The sizes of the C base types in 32-bit compilers are: 1 for `char`, 2 for `short`, 4 for `int`, `long` and `float`, and 8 for `double`. Pointers, being 32-bit addresses, are also 4 bytes long.

To access a C data structure, you need to know the offset from the base of the structure to the field you are interested in. You can either do this by converting the C structure definition into a NASM structure definition (using `STRUC`), or by calculating the one offset and using just that.

To do either of these, you should read your C compiler's manual to find out how it organises data structures. NASM gives no special alignment to structure members in its own `STRUC` macro, so you have to specify alignment yourself if the C compiler generates it. Typically, you might find that a structure like

```
struct @{
    char c;
    int i;
@} foo;
```

might be eight bytes long rather than five, since the `int` field would be aligned to a four-byte boundary. However, this sort of feature is sometimes a configurable option in the C compiler, either using command-line options or `#pragma` lines, so you have to find out how your own compiler does it.

8.1.4. `c32.mac`: Helper Macros for the 32-bit C Interface

Included in the NASM archives, in the `misc` directory, is a file `c32.mac` of macros. It defines three macros: `proc`, `arg` and `endproc`. These are intended to be used for C-style procedure definitions, and they automate a lot of the work involved in keeping track of the calling convention.

An example of an assembly function using the macro set is given here:

```
          proc _proc32
%$i       arg
%$j       arg
          mov eax,[ebp + %$i]
          mov ebx,[ebp + %$j]
          add eax,[ebx]
          endproc
```

This defines `_proc32` to be a procedure taking two arguments, the first (`i`) an integer and the second (`j`) a pointer to an integer. It returns `i + *j`.

Note that the `arg` macro has an `EQU` as the first line of its expansion, and since the label before the macro call gets prepended to the first line of the expanded macro, the `EQU` works, defining `%$i` to be an offset from `BP`. A context-local variable is used, local to the context pushed by the `proc` macro and popped by the `endproc` macro, so that the same argument name can be used in later procedures. Of course, you don't *have* to do that.

`arg` can take an optional parameter, giving the size of the argument. If no size is given, 4 is assumed, since it is likely that many function parameters will be of type `int` or pointers.

8.2. Writing NetBSD/FreeBSD/OpenBSD and Linux/ELF Shared Libraries

ELF replaced the older `a.out` object file format under Linux because it contains support for positionindependent code (PIC), which makes writing shared libraries much easier. NASM supports the ELF position-independent code features, so you can write Linux ELF shared libraries in NASM.

NetBSD, and its close cousins FreeBSD and OpenBSD, take a different approach by hacking PIC support into the `a.out` format. NASM supports this as the `aoutb` output format, so you can write BSD shared libraries in NASM too.

The operating system loads a PIC shared library by memory-mapping the library file at an arbitrarily chosen point in the address space of the running process. The contents of the library's code section must therefore not depend on where it is loaded in memory.

Therefore, you cannot get at your variables by writing code like this:

```
mov eax,[myvar]         ; WRONG
```

Instead, the linker provides an area of memory called the *global offset table*, or GOT; the GOT is situated at a constant distance from your library's code, so if you can find out where your library is loaded (which is typically done using a `CALL` and `POP` combination), you can obtain the address of the GOT, and you can then load the addresses of your variables out of linker-generated entries in the GOT.

The *data* section of a PIC shared library does not have these restrictions: since the data section is writable, it has to be copied into memory anyway rather than just paged in from the library file, so as long as it's being copied it can be relocated too. So you can put ordinary types of relocation in the data section without too much worry (but see section 8.2.4 for a caveat).

8.2.1. Obtaining the Address of the GOT

Each code module in your shared library should define the GOT as an external symbol:

```
          extern _GLOBAL_OFFSET_TABLE_    ; in ELF
          extern __GLOBAL_OFFSET_TABLE_   ; in BSD a.out
```

At the beginning of any function in your shared library which plans to access your data or BSS sections, you must first calculate the address of the GOT. This is typically done by writing the function in this form:

```
func:     push ebp
          mov ebp,esp
          push ebx
          call .get_GOT
.get_GOT: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc
          ; the function body comes here
          mov ebx,[ebp-4]
          mov esp,ebp
          pop ebp
          ret
```

(For BSD, again, the symbol `_GLOBAL_OFFSET_TABLE` requires a second leading underscore.)

The first two lines of this function are simply the standard C prologue to set up a stack frame, and the last three lines are standard C function epilogue. The third line, and the fourth to last line, save and restore the `EBX` register, because PIC shared libraries use this register to store the address of the GOT.

The interesting bit is the `CALL` instruction and the following two lines. The `CALL` and `POP` combination obtains the address of the label `.get_GOT`, without having to know in advance where the program was loaded (since the `CALL` instruction is encoded relative to the current position). The `ADD` instruction makes use of one of the special PIC relocation types: GOTPC relocation. With the `WRT ..gotpc` qualifier specified, the symbol referenced (here `_GLOBAL_OFFSET_TABLE_`, the special symbol assigned to the GOT) is given as an offset from the beginning of the section. (Actually, ELF encodes it as the offset from the operand field of the `ADD` instruction, but NASM simplifies this deliberately, so you do things the same way for both ELF and BSD.) So the instruction then *adds* the beginning of the section, to get the real address of the GOT, and subtracts the value of `.get_GOT` which it knows is in `EBX`. Therefore, by the time that instruction has finished, `EBX` contains the address of the GOT.

If you didn't follow that, don't worry: it's never necessary to obtain the address of the GOT by any other means, so you can put those three instructions into a macro and safely ignore them:

```
%macro get_GOT 0
          call %%getgot
%%getgot: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-%%getgot wrt ..gotpc
%endmacro
```

8.2.2. Finding Your Local Data Items

Having got the GOT, you can then use it to obtain the addresses of your data items. Most variables will reside in the sections you have declared; they can be accessed using the `..gotoff` special `WRT` type. The way this works is like this:

```
lea eax,[ebx+myvar wrt ..gotoff]
```

The expression `myvar wrt ..gotoff` is calculated, when the shared library is linked, to be the offset to the local variable `myvar` from the beginning of the GOT. Therefore, adding it to `EBX` as above will place the real address of `myvar` in `EAX`.

If you declare variables as `GLOBAL` without specifying a size for them, they are shared between code modules in the library, but do not get exported from the library to the program that loaded it. They will still be in your ordinary data and BSS sections, so you can access them in the same way as local variables, using the above `..gotoff` mechanism.

Note that due to a peculiarity of the way BSD `a.out` format handles this relocation type, there must be at least one non-local symbol in the same section as the address you're trying to access.

8.2.3. Finding External and Common Data Items

If your library needs to get at an external variable (external to the *library*, not just to one of the modules within it), you must use the `..got` type to get at it. The `..got` type, instead of giving you the offset from the GOT base to the variable, gives you the offset from the GOT base to a GOT *entry* containing the address of the variable. The linker will set up this GOT entry when it builds the library, and the dynamic linker will place the correct address in it at load time. So to obtain the address of an external variable `extvar` in `EAX`, you would code

```
mov eax,[ebx+extvar wrt ..got]
```

This loads the address of `extvar` out of an entry in the GOT. The linker, when it builds the shared library, collects together every relocation of type `..got`, and builds the GOT so as to ensure it has every necessary entry present.

Common variables must also be accessed in this way.

8.2.4. Exporting Symbols to the Library User

If you want to export symbols to the user of the library, you have to declare whether they are functions or data, and if they are data, you have to give the size of the data item. This is because the dynamic linker has to build procedure linkage table entries for any exported functions, and also moves exported data items away from the library's data section in which they were declared.

So to export a function to users of the library, you must use

```
          global func:function   ; declare it as a function
func:     push ebp
          ; etc.
```

And to export a data item such as an array, you would have to code

```
          global array:data array.end-array ; give the size too
array:    resd 128
.end:
```

Be careful: If you export a variable to the library user, by declaring it as `GLOBAL` and supplying a size, the variable will end up living in the data section of the main program, rather than in your library's data section, where you declared it. So you will have to access your own global variable with the `..got` mechanism rather than `..gotoff`, as if it were external (which, effectively, it has become).

Equally, if you need to store the address of an exported global in one of your data sections, you can't do it by means of the standard sort of code:

```
dataptr:  dd global_data_item    ; WRONG
```

NASM will interpret this code as an ordinary relocation, in which `global_data_item` is merely an offset from the beginning of the `.data` section (or whatever); so this reference will end up pointing at your data section instead of at the exported global which resides elsewhere.

Instead of the above code, then, you must write

```
dataptr:  dd global_data_item wrt ..sym
```

which makes use of the special `WRT` type `..sym` to instruct NASM to search the symbol table for a particular symbol at that address, rather than just relocating by section base.

Either method will work for functions: referring to one of your functions by means of

```
funcptr:  dd my_function
```

will give the user the address of the code you wrote, whereas

```
funcptr:  dd my_function wrt ..sym
```

will give the address of the procedure linkage table for the function, which is where the calling program will *believe* the function lives. Either address is a valid way to call the function.

8.2.5. Calling Procedures Outside the Library

Calling procedures outside your shared library has to be done by means of a *procedure linkage table*, or PLT. The PLT is placed at a known offset from where the library is loaded, so the library code can make calls to the PLT in a position-independent way. Within the PLT there is code to jump to offsets contained in the GOT, so function calls to other shared libraries or to routines in the main program can be transparently passed off to their real destinations.

To call an external routine, you must use another special PIC relocation type, `WRT ..plt`. This is much easier than the GOT-based ones: you simply replace calls such as `CALL printf` with the PLT-relative version `CALL printf WRT ..plt`.

8.2.6. Generating the Library File

Having written some code modules and assembled them to `.o` files, you then generate your shared library with a command such as

```
ld -shared -o library.so module1.o module2.o        # for ELF
ld -Bshareable -o library.so module1.o module2.o    # for BSD
```

For ELF, if your shared library is going to reside in system directories such as `/usr/lib` or `/lib`, it is usually worth using the `soname` flag to the linker, to store the final library file name, with a version number, into the library:

```
ld -shared -soname library.so.1 -o library.so.1.2 *.o
```

You would then copy `library.so.1.2` into the library directory, and create `library.so.1` as a symbolic link to it.

Chapter 9: Mixing 16 and 32 Bit Code

This chapter tries to cover some of the issues, largely related to unusual forms of addressing and jump instructions, encountered when writing operating system code such as protected-mode initialisation routines, which require code that operates in mixed segment sizes, such as code in a 16-bit segment trying to modify data in a 32-bit one, or jumps between different-size segments.

9.1. Mixed-Size Jumps

The most common form of mixedsize instruction is the one used when writing a 32-bit OS: having done your setup in 16-bit mode, such as loading the kernel, you then have to boot it by switching into protected mode and jumping to the 32-bit kernel start address. In a fully 32-bit OS, this tends to be the *only* mixed-size instruction you need, since everything before it can be done in pure 16-bit code, and everything after it can be pure 32-bit.

This jump must specify a 48-bit far address, since the target segment is a 32-bit one. However, it must be assembled in a 16-bit segment, so just coding, for example,

```
jmp 0x1234:0x56789ABC  ; wrong!
```

will not work, since the offset part of the address will be truncated to `0x9ABC` and the jump will be an ordinary 16-bit far one.

The Linux kernel setup code gets round the inability of `as86` to generate the required instruction by coding it manually, using `DB` instructions. NASM can go one better than that, by actually generating the right instruction itself. Here's how to do it right:

```
jmp dword 0x1234:0x56789ABC  ; right
```

The `DWORD` prefix (strictly speaking, it should come *after* the colon, since it is declaring the *offset* field to be a doubleword; but NASM will accept either form, since both are unambiguous) forces the offset part to be treated as far, in the assumption that you are deliberately writing a jump from a 16-bit segment to a 32-bit one.

You can do the reverse operation, jumping from a 32-bit segment to a 16-bit one, by means of the `WORD` prefix:

```
jmp word 0x8765:0x4321 ; 32 to 16 bit
```

If the `WORD` prefix is specified in 16-bit mode, or the `DWORD` prefix in 32-bit mode, they will be ignored, since each is explicitly forcing NASM into a mode it was in anyway.

9.2. Addressing Between Different-Size Segments

If your OS is mixed 16 and 32-bit, or if you are writing a DOS extender, you are likely to have to deal with some 16-bit segments and some 32-bit ones. At some point, you will probably end up writing code in a 16-bit segment which has to access data in a 32-bit segment, or vice versa.

If the data you are trying to access in a 32-bit segment lies within the first 64K of the segment, you may be able to get away with using an ordinary 16-bit addressing operation for the purpose; but sooner or later, you will want to do 32-bit addressing from 16-bit mode.

The easiest way to do this is to make sure you use a register for the address, since any effective address containing a 32-bit register is forced to be a 32-bit address. So you can do

```
mov eax,offset_into_32_bit_segment_specified_by_fs
mov dword [fs:eax],0x11223344
```

This is fine, but slightly cumbersome (since it wastes an instruction and a register) if you already know the precise offset you are aiming at. The x86 architecture does allow 32-bit effective addresses to specify nothing but a 4-byte offset, so why shouldn't NASM be able to generate the best instruction for the purpose?

It can. As in section 9.1, you need only prefix the address with the `DWORD` keyword, and it will be forced to be a 32-bit address:

```
mov dword [fs:dword my_offset],0x11223344
```

Also as in section 9.1, NASM is not fussy about whether the `DWORD` prefix comes before or after the segment override, so arguably a nicer-looking way to code the above instruction is

```
mov dword [dword fs:my_offset],0x11223344
```

Don't confuse the `DWORD` prefix *outside* the square brackets, which controls the size of the data stored at the address, with the one `inside` the square brackets which controls the length of the address itself. The two can quite easily be different:

```
mov word [dword 0x12345678],0x9ABC
```

This moves 16 bits of data to an address specified by a 32-bit offset.

You can also specify `WORD` or `DWORD` prefixes along with the `FAR` prefix to indirect far jumps or calls. For example:

```
call dword far [fs:word 0x4321]
```

This instruction contains an address specified by a 16-bit offset; it loads a 48-bit far pointer from that (16-bit segment and 32-bit offset), and calls that address.

9.3. Other Mixed-Size Instructions

The other way you might want to access data might be using the string instructions (`LODSx`, `STOSx` and so on) or the `XLATB` instruction. These instructions, since they take no parameters, might seem to have no easy way to make them perform 32-bit addressing when assembled in a 16-bit segment.

This is the purpose of NASM's `a16` and `a32` prefixes. If you are coding `LODSB` in a 16-bit segment but it is supposed to be accessing a string in a 32-bit segment, you should load the desired address into `ESI` and then code

```
a32 lodsb
```

The prefix forces the addressing size to 32 bits, meaning that `LODSB` loads from `[DS:ESI]` instead of `[DS:SI]`. To access a string in a 16-bit segment when coding in a 32-bit one, the corresponding `a16` prefix can be used.

The `a16` and `a32` prefixes can be applied to any instruction in NASM's instruction table, but most of them can generate all the useful forms without them. The prefixes are necessary only for instructions with implicit addressing: `CMPSx` (section A.19), `SCASx` (section A.149), `LODSx` (section A.98), `STOSx` (section A.157), `MOVSx` (section A.105), `INSx` (section A.80), `OUTSx` (section A.112), and `XLATB` (section A.169). Also, the various push and pop instructions (`PUSHA` and `POPF` as well as the more usual `PUSH` and `POP`) can accept `a16` or `a32` prefixes to force a particular one of `SP` or `ESP` to be used as a stack pointer, in case the stack segment in use is a different size from the code segment.

`PUSH` and `POP`, when applied to segment registers in 32-bit mode, also have the slightly odd behaviour that they push and pop 4 bytes at a time, of which the top two are ignored and the bottom two give the value of the segment register being manipulated. To force the 16-bit behaviour of segment-register push and pop instructions, you can use the operand-size prefix `o16`:

```
o16 push ss
o16 push ds
```

This code saves a doubleword of stack space by fitting two segment registers into the space which would normally be consumed by pushing one.

(You can also use the `o32` prefix to force the 32-bit behaviour when in 16-bit mode, but this seems less useful.)

Chapter 10: Troubleshooting

This chapter describes some of the common problems that users have been known to encounter with NASM, and answers them. It also gives instructions for reporting bugs in NASM if you find a difficulty that isn't listed here.

10.1. Common Problems

10.1.1. NASM Generates Inefficient Code

I get a lot of `bug' reports about NASM generating inefficient, or even `wrong', code on instructions such as `ADD ESP,8`. This is a deliberate design feature, connected to predictability of output: NASM, on seeing `ADD ESP,8`, will generate the form of the instruction which leaves room for a 32-bit offset. You need to code `ADD ESP,BYTE 8` if you want the space-efficient form of the instruction. This isn't a bug: at worst it's a misfeature, and that's a matter of opinion only.

10.1.2. My Jumps are Out of Range

Similarly, people complain that when they issue conditional jumps (which are `SHORT` by default) that try to jump too far, NASM reports `short jump out of range' instead of making the jumps longer.

This, again, is partly a predictability issue, but in fact has a more practical reason as well. NASM has no means of being told what type of processor the code it is generating will be run on; so it cannot decide for itself that it should generate `Jcc NEAR` type instructions, because it doesn't know that it's working for a 386 or above. Alternatively, it could replace the out-of-range short `JNE` instruction with a very short `JE` instruction that jumps over a `JMP NEAR`; this is a sensible solution for processors below a 386, but hardly efficient on processors which have good branch prediction *and* could have used `JNE NEAR` instead. So, once again, it's up to the user, not the assembler, to decide what instructions should be generated.

10.1.3. `ORG` Doesn't Work

People writing boot sector programs in the `bin` format often complain that `ORG` doesn't work the way they'd like: in order to place the `0xAA55` signature word at the end of a 512-byte boot sector, people who are used to MASM tend to code

```
ORG 0
; some boot sector code
ORG 510
DW 0xAA55
```

This is not the intended use of the `ORG` directive in NASM, and will not work. The correct way to solve this problem in NASM is to use the `TIMES` directive, like this:

```
ORG 0
; some boot sector code
TIMES 510-($-$$) DB 0
DW 0xAA55
```

The `TIMES` directive will insert exactly enough zero bytes into the output to move the assembly point up to 510. This method also has the advantage that if you accidentally fill your boot sector too full, NASM will catch the problem at assembly time and report it, so you won't end up with a boot sector that you have to disassemble to find out what's wrong with it.

10.1.4. `TIMES` Doesn't Work

The other common problem with the above code is people who write the `TIMES` line as

```
TIMES 510-$ DB 0
```

by reasoning that `$` should be a pure number, just like 510, so the difference between them is also a pure number and can happily be fed to `TIMES`.

NASM is a *modular* assembler: the various component parts are designed to be easily separable for re-use, so they don't exchange information unnecessarily. In consequence, the `bin` output format, even though it has been told by the `ORG` directive that the `.text` section should start at 0, does not pass that information back to the expression evaluator. So from the evaluator's point of view, `$` isn't a pure number: it's an offset from a section base. Therefore the difference between `$` and 510 is also not a pure number, but involves a section base. Values involving section bases cannot be passed as arguments to `TIMES`.

The solution, as in the previous section, is to code the `TIMES` line in the form

```
TIMES 510-($-$$) DB 0
```

in which `$` and `$$` are offsets from the same section base, and so their difference is a pure number. This will solve the problem and generate sensible code.

10.2. Bugs

We have never yet released a version of NASM with any *known* bugs. That doesn't usually stop there being plenty we didn't know about, though. Any that you find should be reported to `anakin@pobox.com`.

Please read section 2.2 first, and don't report the bug if it's listed in there as a deliberate feature. (If you think the feature is badly thought out, feel free to send us reasons why you think it should be changed, but don't just send us mail saying `This is a bug' if the documentation says we did it on purpose.) Then read section 10.1, and don't bother reporting the bug if it's listed there.

If you do report a bug, *please* give us all of the following information:

- What operating system you're running NASM under. DOS, Linux, NetBSD, Win16, Win32, VMS (I'd be impressed), whatever.
- If you're running NASM under DOS or Win32, tell us whether you've compiled your own executable from the DOS source archive, or whether you were using the standard distribution binaries out of the archive. If you were using a locally built executable, try to reproduce the problem using one of the standard binaries, as this will make it easier for us to reproduce your problem prior to fixing it.
- Which version of NASM you're using, and exactly how you invoked it. Give us the precise command line, and the contents of the `NASM` environment variable if any.
- Which versions of any supplementary programs you're using, and how you invoked them. If the problem only becomes visible at link time, tell us what linker you're using, what version of it you've got, and the exact linker command line. If the problem involves linking against object files generated by a compiler, tell us what compiler, what version, and what command line or options you used. (If you're compiling in an IDE, please try to reproduce the problem with the command-line version of the compiler.)
- If at all possible, send us a NASM source file which exhibits the problem. If this causes copyright problems (e.g. you can only reproduce the bug in restricted-distribution code) then bear in mind the following two points: firstly, we guarantee that any source code sent to us for the purposes of debugging NASM will be used *only* for the purposes of debugging NASM, and that we will delete all our copies of it as soon as we have found and fixed the bug or bugs in question; and secondly, we would prefer *not* to be mailed large chunks of code anyway. The smaller the file, the better. A three-line sample file that does nothing useful *except* demonstrate the problem is much easier to work with than a fully fledged ten-thousand-line program. (Of course, some errors *do* only crop up in large files, so this may not be possible.)
- A description of what the problem actually *is*. `It doesn't work' is *not* a helpful description! Please describe exactly what is happening that shouldn't be, or what isn't happening that should. Examples might be: `NASM generates an error message saying Line 3 for an error that's actually on Line 5'; `NASM generates an error message that I believe it shouldn't be generating at all'; `NASM fails to generate an error message that I believe it *should* be generating'; `the object file produced from this source code crashes my linker'; `the ninth byte of the output file is 66 and I think it should be 77 instead'.

- If you believe the output file from NASM to be faulty, send it to us. That allows us to determine whether our own copy of NASM generates the same file, or whether the problem is related to portability issues between our development platforms and yours. We can handle binary files mailed to us as MIME attachments, uuencoded, and even BinHex. Alternatively, we may be able to provide an FTP site you can upload the suspect files to; but mailing them is easier for us.
- Any other information or data files that might be helpful. If, for example, the problem involves NASM failing to generate an object file while TASM can generate an equivalent file without trouble, then send us *both* object files, so we can see what TASM is doing differently from us.

Appendix A: Intel x86 Instruction Reference

This appendix provides a complete list of the machine instructions which NASM will assemble, and a short description of the function of each one.

It is not intended to be exhaustive documentation on the fine details of the instructions' function, such as which exceptions they can trigger: for such documentation, you should go to Intel's Web site, `http://www.intel.com`.

Instead, this appendix is intended primarily to provide documentation on the way the instructions may be used within NASM. For example, looking up `LOOP` will tell you that NASM allows `CX` or `ECX` to be specified as an optional second argument to the `LOOP` instruction, to enforce which of the two possible counter registers should be used if the default is not the one desired.

The instructions are not quite listed in alphabetical order, since groups of instructions with similar functions are lumped together in the same entry. Most of them don't move very far from their alphabetic position because of this.

Section A.170: XOR: Bitwise Exclusive OR

A.1. Key to Operand Specifications

The instruction descriptions in this appendix specify their operands using the following notation:

- Registers: `reg8` denotes an 8-bit general purpose register, `reg16` denotes a 16-bit general purpose register, and `reg32` a 32-bit one. `fpureg` denotes one of the eight FPU stack registers, `mmxreg` denotes one of the eight 64-bit MMX registers, and `segreg` denotes a segment register. In addition, some registers (such as `AL`, `DX` or `ECX`) may be specified explicitly.
- Immediate operands: `imm` denotes a generic immediate operand. `imm8`, `imm16` and `imm32` are used when the operand is intended to be a specific size. For some of these instructions, NASM needs an explicit specifier: for example, `ADD ESP,16` could be interpreted as either `ADD r/m32,imm32` or `ADD r/m32,imm8`. NASM chooses the former by default, and so you must specify `ADD ESP,BYTE 16` for the latter.
- Memory references: `mem` denotes a generic memory reference; `mem8`, `mem16`, `mem32`, `mem64` and `mem80` are used when the operand needs to be a specific size. Again, a specifier is needed in some cases: `DEC [address]` is ambiguous and will be rejected by NASM. You must specify `DEC BYTE [address]`, `DEC WORD [address]` or `DEC DWORD [address]` instead.
- Restricted memory references: one form of the `MOV` instruction allows a memory address to be specified *without* allowing the normal range of register combinations and effective address processing. This is denoted by `memoffs8`, `memoffs16` and `memoffs32`.
- Register or memory choices: many instructions can accept either a register *or* a memory reference as an operand. `r/m8` is a shorthand for `reg8/mem8`; similarly `r/m16` and `r/m32`. `r/m64` is MMX-related, and is a shorthand for `mmxreg/mem64`.

A.2. Key to Opcode Descriptions

This appendix also provides the opcodes which NASM will generate for each form of each instruction. The opcodes are listed in the following way:

- A hex number, such as `3F`, indicates a fixed byte containing that number.
- A hex number followed by `+r`, such as `C8+r`, indicates that one of the operands to the instruction is a register, and the `register value' of that register should be added to the hex number to produce the generated byte. For example, EDX has register value 2, so the code `C8+r`, when the register operand is EDX, generates the hex byte `CA`. Register values for specific registers are given in section A.2.1.
- A hex number followed by `+cc`, such as `40+cc`, indicates that the instruction name has a condition code suffix, and the numeric representation of the condition code should be added to the hex number to produce the generated byte. For example, the code `40+cc`, when the instruction contains the `NE` condition, generates the hex byte `45`. Condition codes and their numeric representations are given in section A.2.2.
- A slash followed by a digit, such as `/2`, indicates that one of the operands to the instruction is a memory address or register (denoted `mem` or `r/m`, with an optional size). This is to be encoded as an effective address, with a ModR/M byte, an optional SIB byte, and an optional displacement, and the spare (register) field of the ModR/M byte should be the digit given (which will be from 0 to 7, so it fits in three bits). The encoding of effective addresses is given in section A.2.3.
- The code `/r` combines the above two: it indicates that one of the operands is a memory address or `r/m`, and another is a register, and that an effective address should be generated with the spare (register) field in the ModR/M byte being equal to the `register value' of the register operand. The encoding of effective addresses is given in section A.2.3; register values are given in section A.2.1.
- The codes `ib`, `iw` and `id` indicate that one of the operands to the instruction is an immediate value, and that this is to be encoded as a byte, little-endian word or little-endian doubleword respectively.
- The codes `rb`, `rw` and `rd` indicate that one of the operands to the instruction is an immediate value, and that the *difference* between this value and the address of the end of the instruction is to be encoded as a byte, word or doubleword respectively. Where the form `rw/rd` appears, it indicates that either `rw` or `rd` should be used according to whether assembly is being performed in `BITS 16` or `BITS 32` state respectively.
- The codes `ow` and `od` indicate that one of the operands to the instruction is a reference to the contents of a memory address specified as an immediate value: this encoding is used in some forms of the `MOV` instruction in place of the standard effective-address mechanism. The displacement is encoded as a word or doubleword. Again, `ow/od` denotes that `ow` or `od` should be chosen according to the `BITS` setting.
- The codes `o16` and `o32` indicate that the given form of the instruction should be assembled with operand size 16 or 32 bits. In other words, `o16` indicates a `66` prefix in `BITS 32` state,

but generates no code in `BITS 16` state; and `o32` indicates a `66` prefix in `BITS 16` state but generates nothing in `BITS 32`.

- The codes `a16` and `a32`, similarly to `o16` and `o32`, indicate the address size of the given form of the instruction. Where this does not match the `BITS` setting, a `67` prefix is required.

Section A.2.1: Register Values
Section A.2.2: Condition Codes
Section A.2.3: Effective Address Encoding: ModR/M and SIB

A.2.1. Register Values

Where an instruction requires a register value, it is already implicit in the encoding of the rest of the instruction what type of register is intended: an 8-bit general-purpose register, a segment register, a debug register, an MMX register, or whatever. Therefore there is no problem with registers of different types sharing an encoding value.

The encodings for the various classes of register are:

- 8-bit general registers: `AL` is 0, `CL` is 1, `DL` is 2, `BL` is 3, `AH` is 4, `CH` is 5, `DH` is 6, and `BH` is 7.
- 16-bit general registers: `AX` is 0, `CX` is 1, `DX` is 2, `BX` is 3, `SP` is 4, `BP` is 5, `SI` is 6, and `DI` is 7.
- 32-bit general registers: `EAX` is 0, `ECX` is 1, `EDX` is 2, `EBX` is 3, `ESP` is 4, `EBP` is 5, `ESI` is 6, and `EDI` is 7.
- Segment registers: `ES` is 0, `CS` is 1, `SS` is 2, `DS` is 3, `FS` is 4, and `GS` is 5.
- {Floating-point registers}: `ST0` is 0, `ST1` is 1, `ST2` is 2, `ST3` is 3, `ST4` is 4, `ST5` is 5, `ST6` is 6, and `ST7` is 7.
- 64-bit MMX registers: `MM0` is 0, `MM1` is 1, `MM2` is 2, `MM3` is 3, `MM4` is 4, `MM5` is 5, `MM6` is 6, and `MM7` is 7.
- Control registers: `CR0` is 0, `CR2` is 2, `CR3` is 3, and `CR4` is 4.
- Debug registers: `DR0` is 0, `DR1` is 1, `DR2` is 2, `DR3` is 3, `DR6` is 6, and `DR7` is 7.
- Test registers: `TR3` is 3, `TR4` is 4, `TR5` is 5, `TR6` is 6, and `TR7` is 7.

(Note that wherever a register name contains a number, that number is also the register value for that register.)

A.2.2. Condition Codes

The available condition codes are given here, along with their numeric representations as part of opcodes. Many of these condition codes have synonyms, so several will be listed at a time.

In the following descriptions, the word `either', when applied to two possible trigger conditions, is used to mean `either or both'. If `either but not both' is meant, the phrase `exactly one of' is used.

- `O` is 0 (trigger if the overflow flag is set); `NO` is 1.
- `B`, `C` and `NAE` are 2 (trigger if the carry flag is set); `AE`, `NB` and `NC` are 3.
- `E` and `Z` are 4 (trigger if the zero flag is set); `NE` and `NZ` are 5.
- `BE` and `NA` are 6 (trigger if either of the carry or zero flags is set); `A` and `NBE` are 7.
- `S` is 8 (trigger if the sign flag is set); `NS` is 9.
- `P` and `PE` are 10 (trigger if the parity flag is set); `NP` and `PO` are 11.
- `L` and `NGE` are 12 (trigger if exactly one of the sign and overflow flags is set); `GE` and `NL` are 13.
- `LE` and `NG` are 14 (trigger if either the zero flag is set, or exactly one of the sign and overflow flags is set); `G` and `NLE` are 15.

Note that in all cases, the sense of a condition code may be reversed by changing the low bit of the numeric representation.

A.2.3. Effective Address Encoding: ModR/M and SIB

An effective address is encoded in up to three parts: a ModR/M byte, an optional SIB byte, and an optional byte, word or doubleword displacement field.

The ModR/M byte consists of three fields: the `mod` field, ranging from 0 to 3, in the upper two bits of the byte, the `r/m` field, ranging from 0 to 7, in the lower three bits, and the spare (register) field in the middle (bit 3 to bit 5). The spare field is not relevant to the effective address being encoded, and either contains an extension to the instruction opcode or the register value of another operand.

The ModR/M system can be used to encode a direct register reference rather than a memory access. This is always done by setting the `mod` field to 3 and the `r/m` field to the register value of the register in question (it must be a general-purpose register, and the size of the register must already be implicit in the encoding of the rest of the instruction). In this case, the SIB byte and displacement field are both absent.

In 16-bit addressing mode (either `BITS 16` with no `67` prefix, or `BITS 32` with a `67` prefix), the SIB byte is never used. The general rules for `mod` and `r/m` (there is an exception, given below) are:

- The `mod` field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means two bytes.
- The `r/m` field encodes the combination of registers to be added to the displacement to give the accessed address: 0 means `BX+SI`, 1 means `BX+DI`, 2 means `BP+SI`, 3 means `BP+DI`, 4 means `SI` only, 5 means `DI` only, 6 means `BP` only, and 7 means `BX` only.

However, there is a special case:

- If `mod` is 0 and `r/m` is 6, the effective address encoded is not `[BP]` as the above rules would suggest, but instead `[disp16]`: the displacement field is present and is two bytes long, and no registers are added to the displacement.

Therefore the effective address `[BP]` cannot be encoded as efficiently as `[BX]`; so if you code `[BP]` in a program, NASM adds a notional 8-bit zero displacement, and sets `mod` to 1, `r/m` to 6, and the one-byte displacement field to 0.

In 32-bit addressing mode (either `BITS 16` with a `67` prefix, or `BITS 32` with no `67` prefix) the general rules (again, there are exceptions) for `mod` and `r/m` are:

- The `mod` field gives the length of the displacement field: 0 means no displacement, 1 means one byte, and 2 means four bytes.
- If only one register is to be added to the displacement, and it is not `ESP`, the `r/m` field gives its register value, and the SIB byte is absent. If the `r/m` field is 4 (which would encode `ESP`), the SIB byte is present and gives the combination and scaling of registers to be added to the displacement.

If the SIB byte is present, it describes the combination of registers (an optional base register, and an optional index register scaled by multiplication by 1, 2, 4 or 8) to be added to the displacement. The SIB byte is divided into the `scale` field, in the top two bits, the `index` field

in the next three, and the `base` field in the bottom three. The general rules are:

- The `base` field encodes the register value of the base register.
- The `index` field encodes the register value of the index register, unless it is 4, in which case no index register is used (so `ESP` cannot be used as an index register).
- The `scale` field encodes the multiplier by which the index register is scaled before adding it to the base and displacement: 0 encodes a multiplier of 1, 1 encodes 2, 2 encodes 4 and 3 encodes 8.

The exceptions to the 32-bit encoding rules are:

- If `mod` is 0 and `r/m` is 5, the effective address encoded is not `[EBP]` as the above rules would suggest, but instead `[disp32]`: the displacement field is present and is four bytes long, and no registers are added to the displacement.
- If `mod` is 0, `r/m` is 4 (meaning the SIB byte is present) and `base` is 4, the effective address encoded is not `[EBP+index]` as the above rules would suggest, but instead `[disp32+index]`: the displacement field is present and is four bytes long, and there is no base register (but the index register is still processed in the normal way).

A.3. Key to Instruction Flags

Given along with each instruction in this appendix is a set of flags, denoting the type of the instruction. The types are as follows:

- `8086`, `186`, `286`, `386`, `486`, `PENT` and `P6` denote the lowest processor type that supports the instruction. Most instructions run on all processors above the given type; those that do not are documented. The Pentium II contains no additional instructions beyond the P6 (Pentium Pro); from the point of view of its instruction set, it can be thought of as a P6 with MMX capability.
- `CYRIX` indicates that the instruction is specific to Cyrix processors, for example the extra MMX instructions in the Cyrix extended MMX instruction set.
- `FPU` indicates that the instruction is a floating-point one, and will only run on machines with a coprocessor (automatically including 486DX, Pentium and above).
- `MMX` indicates that the instruction is an MMX one, and will run on MMX-capable Pentium processors and the Pentium II.
- `PRIV` indicates that the instruction is a protected-mode management instruction. Many of these may only be used in protected mode, or only at privilege level zero.
- `UNDOC` indicates that the instruction is an undocumented one, and not part of the official Intel Architecture; it may or may not be supported on any given machine.

A.4. AAA, AAS, AAM, AAD: ASCII Adjustments

```
AAA                             ; 37                    [8086]

AAS                             ; 3F                    [8086]

AAD                             ; D5 0A                 [8086]
AAD imm                         ; D5 ib                 [8086]

AAM                             ; D4 0A                 [8086]
AAM imm                         ; D4 ib                 [8086]
```

These instructions are used in conjunction with the add, subtract, multiply and divide instructions to perform binary-coded decimal arithmetic in *unpacked* (one BCD digit per byte – easy to translate to and from ASCII, hence the instruction names) form. There are also packed BCD instructions `DAA` and `DAS`: see section A.23.

`AAA` should be used after a one-byte `ADD` instruction whose destination was the `AL` register: by means of examining the value in the low nibble of `AL` and also the auxiliary carry flag `AF`, it determines whether the addition has overflowed, and adjusts it (and sets the carry flag) if so. You can add long BCD strings together by doing `ADD`/`AAA` on the low digits, then doing `ADC`/`AAA` on each subsequent digit.

`AAS` works similarly to `AAA`, but is for use after `SUB` instructions rather than `ADD`.

`AAM` is for use after you have multiplied two decimal digits together and left the result in `AL`: it divides `AL` by ten and stores the quotient in `AH`, leaving the remainder in `AL`. The divisor 10 can be changed by specifying an operand to the instruction: a particularly handy use of this is `AAM 16`, causing the two nibbles in `AL` to be separated into `AH` and `AL`.

`AAD` performs the inverse operation to `AAM`: it multiplies `AH` by ten, adds it to `AL`, and sets `AH` to zero. Again, the multiplier 10 can be changed.

A.5. ADC: Add with Carry

```
ADC r/m8,reg8                   ; 10 /r                 [8086]
ADC r/m16,reg16                 ; o16 11 /r             [8086]
ADC r/m32,reg32                 ; o32 11 /r             [386]

ADC reg8,r/m8                   ; 12 /r                 [8086]
ADC reg16,r/m16                 ; o16 13 /r             [8086]
ADC reg32,r/m32                 ; o32 13 /r             [386]

ADC r/m8,imm8                   ; 80 /2 ib              [8086]
ADC r/m16,imm16                 ; o16 81 /2 iw          [8086]
ADC r/m32,imm32                 ; o32 81 /2 id          [386]

ADC r/m16,imm8                  ; o16 83 /2 ib          [8086]
ADC r/m32,imm8                  ; o32 83 /2 ib          [386]

ADC AL,imm8                     ; 14 ib                 [8086]
ADC AX,imm16                    ; o16 15 iw             [8086]
ADC EAX,imm32                   ; o32 15 id             [386]
```

`ADC` performs integer addition: it adds its two operands together, plus the value of the carry flag, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent `ADC` instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

To add two numbers without also adding the contents of the carry flag, use `ADD` (section A.6).

A.6. ADD: Add Integers

```
ADD r/m8,reg8                   ; 00 /r                 [8086]
ADD r/m16,reg16                 ; o16 01 /r             [8086]
ADD r/m32,reg32                 ; o32 01 /r             [386]

ADD reg8,r/m8                   ; 02 /r                 [8086]
ADD reg16,r/m16                 ; o16 03 /r             [8086]
ADD reg32,r/m32                 ; o32 03 /r             [386]

ADD r/m8,imm8                   ; 80 /0 ib              [8086]
ADD r/m16,imm16                 ; o16 81 /0 iw          [8086]
ADD r/m32,imm32                 ; o32 81 /0 id          [386]

ADD r/m16,imm8                  ; o16 83 /0 ib          [8086]
ADD r/m32,imm8                  ; o32 83 /0 ib          [386]

ADD AL,imm8                     ; 04 ib                 [8086]
ADD AX,imm16                    ; o16 05 iw             [8086]
ADD EAX,imm32                   ; o32 05 id             [386]
```

ADD performs integer addition: it adds its two operands together, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent ADC instruction (section A.5).

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

A.7. AND: Bitwise AND

```
AND r/m8,reg8                   ; 20 /r                 [8086]
AND r/m16,reg16                 ; o16 21 /r             [8086]
AND r/m32,reg32                 ; o32 21 /r             [386]

AND reg8,r/m8                   ; 22 /r                 [8086]
AND reg16,r/m16                 ; o16 23 /r             [8086]
AND reg32,r/m32                 ; o32 23 /r             [386]

AND r/m8,imm8                   ; 80 /4 ib              [8086]
AND r/m16,imm16                 ; o16 81 /4 iw          [8086]
AND r/m32,imm32                 ; o32 81 /4 id          [386]

AND r/m16,imm8                  ; o16 83 /4 ib          [8086]
AND r/m32,imm8                  ; o32 83 /4 ib          [386]

AND AL,imm8                     ; 24 ib                 [8086]
AND AX,imm16                    ; o16 25 iw             [8086]
AND EAX,imm32                   ; o32 25 id             [386]
```

`AND` performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `PAND` (see section A.116) performs the same operation on the 64-bit MMX registers.

A.8. `ARPL`: Adjust RPL Field of Selector

```
ARPL r/m16,reg16              ; 63 /r                 [286,PRIV]
```

`ARPL` expects its two word operands to be segment selectors. It adjusts the RPL (requested privilege level – stored in the bottom two bits of the selector) field of the destination (first) operand to ensure that it is no less (i.e. no more privileged than) the RPL field of the source operand. The zero flag is set if and only if a change had to be made.

A.9. BOUND: Check Array Index against Bounds

```
BOUND reg16,mem                 ; o16 62 /r             [186]
BOUND reg32,mem                 ; o32 62 /r             [386]
```

BOUND expects its second operand to point to an area of memory containing two signed values of the same size as its first operand (i.e. two words for the 16-bit form; two doublewords for the 32-bit form). It performs two signed comparisons: if the value in the register passed as its first operand is less than the first of the in-memory values, or is greater than or equal to the second, it throws a BR exception. Otherwise, it does nothing.

A.10. BSF, BSR: Bit Scan

```
BSF reg16,r/m16                 ; o16 0F BC /r          [386]
BSF reg32,r/m32                 ; o32 0F BC /r          [386]

BSR reg16,r/m16                 ; o16 0F BD /r          [386]
BSR reg32,r/m32                 ; o32 0F BD /r          [386]
```

BSF searches for a set bit in its source (second) operand, starting from the bottom, and if it finds one, stores the index in its destination (first) operand. If no set bit is found, the contents of the destination operand are undefined.

BSR performs the same function, but searches from the top instead, so it finds the most significant set bit.

Bit indices are from 0 (least significant) to 15 or 31 (most significant).

A.11. `BSWAP`: Byte Swap

```
BSWAP reg32                    ; o32 0F C8+r          [486]
```

`BSWAP` swaps the order of the four bytes of a 32-bit register: bits 0-7 exchange places with bits 24-31, and bits 8-15 swap with bits 16-23. There is no explicit 16-bit equivalent: to byte-swap `AX`, `BX`, `CX` or `DX`, `XCHG` can be used.

A.12. `BT`, `BTC`, `BTR`, `BTS`: Bit Test

```
BT r/m16,reg16                  ; o16 0F A3 /r          [386]
BT r/m32,reg32                  ; o32 0F A3 /r          [386]
BT r/m16,imm8                   ; o16 0F BA /4 ib       [386]
BT r/m32,imm8                   ; o32 0F BA /4 ib       [386]

BTC r/m16,reg16                 ; o16 0F BB /r          [386]
BTC r/m32,reg32                 ; o32 0F BB /r          [386]
BTC r/m16,imm8                  ; o16 0F BA /7 ib       [386]
BTC r/m32,imm8                  ; o32 0F BA /7 ib       [386]

BTR r/m16,reg16                 ; o16 0F B3 /r          [386]
BTR r/m32,reg32                 ; o32 0F B3 /r          [386]
BTR r/m16,imm8                  ; o16 0F BA /6 ib       [386]
BTR r/m32,imm8                  ; o32 0F BA /6 ib       [386]

BTS r/m16,reg16                 ; o16 0F AB /r          [386]
BTS r/m32,reg32                 ; o32 0F AB /r          [386]
BTS r/m16,imm                   ; o16 0F BA /5 ib       [386]
BTS r/m32,imm                   ; o32 0F BA /5 ib       [386]
```

These instructions all test one bit of their first operand, whose index is given by the second operand, and store the value of that bit into the carry flag. Bit indices are from 0 (least significant) to 15 or 31 (most significant).

In addition to storing the original value of the bit into the carry flag, `BTR` also resets (clears) the bit in the operand itself. `BTS` sets the bit, and `BTC` complements the bit. `BT` does not modify its operands.

The bit offset should be no greater than the size of the operand.

A.13. CALL: Call Subroutine

```
CALL imm                        ; E8 rw/rd              [8086]
CALL imm:imm16                  ; o16 9A iw iw          [8086]
CALL imm:imm32                  ; o32 9A id iw          [386]
CALL FAR mem16                  ; o16 FF /3             [8086]
CALL FAR mem32                  ; o32 FF /3             [386]
CALL r/m16                      ; o16 FF /2             [8086]
CALL r/m32                      ; o32 FF /2             [386]
```

CALL calls a subroutine, by means of pushing the current instruction pointer (IP) and optionally CS as well on the stack, and then jumping to a given address.

CS is pushed as well as IP if and only if the call is a far call, i.e. a destination segment address is specified in the instruction. The forms involving two colon-separated arguments are far calls; so are the CALL FAR mem forms.

You can choose between the two immediate far call forms (CALL imm:imm) by the use of the WORD and DWORD keywords: CALL WORD 0x1234:0x5678) or CALL DWORD 0x1234:0x56789abc.

The CALL FAR mem forms execute a far call by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using CALL WORD FAR mem or CALL DWORD FAR mem.

The CALL r/m forms execute a near call (within the same segment), loading the destination address out of memory or out of a register. The keyword NEAR may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using CALL WORD mem or CALL DWORD mem.

As a convenience, NASM does not require you to call a far procedure symbol by coding the cumbersome CALL SEG routine:routine, but instead allows the easier synonym CALL FAR routine.

The CALL r/m forms given above are near calls; NASM will accept the NEAR keyword (e.g. CALL NEAR [address]), even though it is not strictly necessary.

A.14. CBW, CWD, CDQ, CWDE: Sign Extensions

```
CBW                             ; o16 98                [8086]
CWD                             ; o16 99                [8086]
CDQ                             ; o32 99                [386]
CWDE                            ; o32 98                [386]
```

All these instructions sign-extend a short value into a longer one, by replicating the top bit of the original value to fill the extended one.

`CBW` extends `AL` into `AX` by repeating the top bit of `AL` in every bit of `AH`. `CWD` extends `AX` into `DX:AX` by repeating the top bit of `AX` throughout `DX`. `CWDE` extends `AX` into `EAX`, and `CDQ` extends `EAX` into `EDX:EAX`.

A.15. `CLC`, `CLD`, `CLI`, `CLTS`: Clear Flags

```
CLC                             ; F8                    [8086]
CLD                             ; FC                    [8086]
CLI                             ; FA                    [8086]
CLTS                            ; 0F 06                 [286,PRIV]
```

These instructions clear various flags. `CLC` clears the carry flag; `CLD` clears the direction flag; `CLI` clears the interrupt flag (thus disabling interrupts); and `CLTS` clears the task-switched (`TS`) flag in `CR0`.

To set the carry, direction, or interrupt flags, use the `STC`, `STD` and `STI` instructions (section A.156). To invert the carry flag, use `CMC` (section A.16).

A.16. CMC: Complement Carry Flag

```
CMC                                ; F5                          [8086]
```

CMC changes the value of the carry flag: if it was 0, it sets it to 1, and vice versa.

A.17. CMOVcc: Conditional Move

```
CMOVcc reg16,r/m16              ; o16 0F 40+cc /r       [P6]
CMOVcc reg32,r/m32              ; o32 0F 40+cc /r       [P6]
```

CMOV moves its source (second) operand into its destination (first) operand if the given condition code is satisfied; otherwise it does nothing.

For a list of condition codes, see section A.2.2.

Although the CMOV instructions are flagged P6 above, they may not be supported by all Pentium Pro processors; the CPUID instruction (section A.22) will return a bit which indicates whether conditional moves are supported.

A.18. `CMP`: Compare Integers

```
CMP r/m8,reg8                   ; 38 /r                 [8086]
CMP r/m16,reg16                 ; o16 39 /r             [8086]
CMP r/m32,reg32                 ; o32 39 /r             [386]

CMP reg8,r/m8                   ; 3A /r                 [8086]
CMP reg16,r/m16                 ; o16 3B /r             [8086]
CMP reg32,r/m32                 ; o32 3B /r             [386]

CMP r/m8,imm8                   ; 80 /0 ib              [8086]
CMP r/m16,imm16                 ; o16 81 /0 iw          [8086]
CMP r/m32,imm32                 ; o32 81 /0 id          [386]

CMP r/m16,imm8                  ; o16 83 /0 ib          [8086]
CMP r/m32,imm8                  ; o32 83 /0 ib          [386]

CMP AL,imm8                     ; 3C ib                 [8086]
CMP AX,imm16                    ; o16 3D iw             [8086]
CMP EAX,imm32                   ; o32 3D id             [386]
```

`CMP` performs a `mental' subtraction of its second operand from its first operand, and affects the flags as if the subtraction had taken place, but does not store the result of the subtraction anywhere.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

A.19. CMPSB, CMPSW, CMPSD: Compare Strings

```
CMPSB                           ; A6                    [8086]
CMPSW                           ; o16 A7                [8086]
CMPSD                           ; o32 A7                [386]
```

CMPSB compares the byte at [DS:SI] or [DS:ESI] with the byte at [ES:DI] or [ES:EDI], and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) SI and DI (or ESI and EDI).

The registers used are SI and DI if the address size is 16 bits, and ESI and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

The segment register used to load from [SI] or [ESI] can be overridden by using a segment register name as a prefix (for example, es cmpsb). The use of ES for the load from [DI] or [EDI] cannot be overridden.

CMPSW and CMPSD work in the same way, but they compare a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to CX (or ECX – again, the address size chooses which) times until the first unequal or equal byte is found.

A.20. CMPXCHG, CMPXCHG486: Compare and Exchange

```
CMPXCHG r/m8,reg8             ; 0F B0 /r              [PENT]
CMPXCHG r/m16,reg16           ; o16 0F B1 /r          [PENT]
CMPXCHG r/m32,reg32           ; o32 0F B1 /r          [PENT]

CMPXCHG486 r/m8,reg8          ; 0F A6 /r              [486,UNDOC]
CMPXCHG486 r/m16,reg16        ; o16 0F A7 /r          [486,UNDOC]
CMPXCHG486 r/m32,reg32        ; o32 0F A7 /r          [486,UNDOC]
```

These two instructions perform exactly the same operation; however, apparently some (not all) 486 processors support it under a non-standard opcode, so NASM provides the undocumented `CMPXCHG486` form to generate the non-standard opcode.

`CMPXCHG` compares its destination (first) operand to the value in `AL`, `AX` or `EAX` (depending on the size of the instruction). If they are equal, it copies its source (second) operand into the destination and sets the zero flag. Otherwise, it clears the zero flag and leaves the destination alone.

`CMPXCHG` is intended to be used for atomic operations in multitasking or multiprocessor environments. To safely update a value in shared memory, for example, you might load the value into `EAX`, load the updated value into `EBX`, and then execute the instruction `lock cmpxchg [value],ebx`. If `value` has not changed since being loaded, it is updated with your desired new value, and the zero flag is set to let you know it has worked. (The `LOCK` prefix prevents another processor doing anything in the middle of this operation: it guarantees atomicity.) However, if another processor has modified the value in between your load and your attempted store, the store does not happen, and you are notified of the failure by a cleared zero flag, so you can go round and try again.

A.21. `CMPXCHG8B`: Compare and Exchange Eight Bytes

```
CMPXCHG8B mem                   ; 0F C7 /1              [PENT]
```

This is a larger and more unwieldy version of `CMPXCHG`: it compares the 64-bit (eight-byte) value stored at `[mem]` with the value in `EDX:EAX`. If they are equal, it sets the zero flag and stores `ECX:EBX` into the memory area. If they are unequal, it clears the zero flag and leaves the memory area untouched.

A.22. `CPUID`: Get CPU Identification Code

```
CPUID                           ; 0F A2                 [PENT]
```

`CPUID` returns various information about the processor it is being executed on. It fills the four registers `EAX`, `EBX`, `ECX` and `EDX` with information, which varies depending on the input contents of `EAX`.

`CPUID` also acts as a barrier to serialise instruction execution: executing the `CPUID` instruction guarantees that all the effects (memory modification, flag modification, register modification) of previous instructions have been completed before the next instruction gets fetched.

The information returned is as follows:

- If `EAX` is zero on input, `EAX` on output holds the maximum acceptable input value of `EAX`, and `EBX:EDX:ECX` contain the string `"GenuineIntel"` (or not, if you have a clone processor). That is to say, `EBX` contains `"Genu"` (in NASM's own sense of character constants, described in section 3.4.2), `EDX` contains `"ineI"` and `ECX` contains `"ntel"`.
- If `EAX` is one on input, `EAX` on output contains version information about the processor, and `EDX` contains a set of feature flags, showing the presence and absence of various features. For example, bit 8 is set if the `CMPXCHG8B` instruction (section A.21) is supported, bit 15 is set if the conditional move instructions (section A.17 and section A.34) are supported, and bit 23 is set if MMX instructions are supported.
- If `EAX` is two on input, `EAX`, `EBX`, `ECX` and `EDX` all contain information about caches and TLBs (Translation Lookahead Buffers).

For more information on the data returned from `CPUID`, see the documentation on Intel's web site.

A.23. `DAA`, `DAS`: Decimal Adjustments

```
DAA                             ; 27                    [8086]
DAS                             ; 2F                    [8086]
```

These instructions are used in conjunction with the add and subtract instructions to perform binary-coded decimal arithmetic in *packed* (one BCD digit per nibble) form. For the unpacked equivalents, see section A.4.

`DAA` should be used after a one-byte `ADD` instruction whose destination was the `AL` register: by means of examining the value in the `AL` and also the auxiliary carry flag `AF`, it determines whether either digit of the addition has overflowed, and adjusts it (and sets the carry and auxiliary-carry flags) if so. You can add long BCD strings together by doing `ADD`/`DAA` on the low two digits, then doing `ADC`/`DAA` on each subsequent pair of digits.

`DAS` works similarly to `DAA`, but is for use after `SUB` instructions rather than `ADD`.

A.24. DEC: Decrement Integer

```
DEC reg16                       ; o16 48+r              [8086]
DEC reg32                       ; o32 48+r              [386]
DEC r/m8                        ; FE /1                 [8086]
DEC r/m16                       ; o16 FF /1             [8086]
DEC r/m32                       ; o32 FF /1             [386]
```

`DEC` subtracts 1 from its operand. It does *not* affect the carry flag: to affect the carry flag, use `SUB something,1` (see section A.159). See also `INC` (section A.79).

A.25. DIV: Unsigned Integer Divide

```
DIV r/m8                        ; F6 /6                 [8086]
DIV r/m16                       ; o16 F7 /6             [8086]
DIV r/m32                       ; o32 F7 /6             [386]
```

DIV performs unsigned integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For DIV r/m8, AX is divided by the given operand; the quotient is stored in AL and the remainder in AH.
- For DIV r/m16, DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX.
- For DIV r/m32, EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX.

Signed integer division is performed by the IDIV instruction: see section A.76.

A.26. EMMS: Empty MMX State

```
EMMS                            ; 0F 77                 [PENT,MMX]
```

EMMS sets the FPU tag word (marking which floating-point registers are available) to all ones, meaning all registers are available for the FPU to use. It should be used after executing MMX instructions and before executing any subsequent floating-point operations.

A.27. ENTER: Create Stack Frame

```
ENTER imm,imm                   ; C8 iw ib              [186]
```

ENTER constructs a stack frame for a high-level language procedure call. The first operand (the iw in the opcode definition above refers to the first operand) gives the amount of stack space to allocate for local variables; the second (the ib above) gives the nesting level of the procedure (for languages like Pascal, with nested procedures).

The function of ENTER, with a nesting level of zero, is equivalent to

```
PUSH EBP              ; or PUSH BP          in 16 bits
MOV EBP,ESP           ; or MOV BP,SP        in 16 bits
SUB ESP,operand1      ; or SUB SP,operand1  in 16 bits
```

This creates a stack frame with the procedure parameters accessible upwards from EBP, and local variables accessible downwards from EBP.

With a nesting level of one, the stack frame created is 4 (or 2) bytes bigger, and the value of the final frame pointer EBP is accessible in memory at [EBP4].

This allows ENTER, when called with a nesting level of two, to look at the stack frame described by the *previous* value of EBP, find the frame pointer at offset –4 from that, and push it along with its new frame pointer, so that when a level-two procedure is called from within a level-one procedure, [EBP4] holds the frame pointer of the most recent level-one procedure call and [EBP8] holds that of the most recent level-two call. And so on, for nesting levels up to 31.

Stack frames created by ENTER can be destroyed by the LEAVE instruction: see section A.94.

A.28. `F2XM1`: Calculate 2**X-1

```
F2XM1                           ; D9 F0                 [8086,FPU]
```

`F2XM1` raises 2 to the power of `ST0`, subtracts one, and stores the result back into `ST0`. The initial contents of `ST0` must be a number in the range –1 to +1.

A.29. `FABS`: Floating-Point Absolute Value

```
FABS                            ; D9 E1                 [8086,FPU]
```

`FABS` computes the absolute value of `ST0`, storing the result back in `ST0`.

A.30. `FADD`, `FADDP`: Floating-Point Addition

```
FADD mem32                      ; D8 /0                 [8086,FPU]
FADD mem64                      ; DC /0                 [8086,FPU]

FADD fpureg                     ; D8 C0+r               [8086,FPU]
FADD ST0,fpureg                 ; D8 C0+r               [8086,FPU]

FADD TO fpureg                  ; DC C0+r               [8086,FPU]
FADD fpureg,ST0                 ; DC C0+r               [8086,FPU]

FADDP fpureg                    ; DE C0+r               [8086,FPU]
FADDP fpureg,ST0                ; DE C0+r               [8086,FPU]
```

`FADD`, given one operand, adds the operand to `ST0` and stores the result back in `ST0`. If the operand has the `TO` modifier, the result is stored in the register given rather than in `ST0`.

`FADDP` performs the same function as `FADD TO`, but pops the register stack after storing the result.

The given two-operand forms are synonyms for the one-operand forms.

A.31. `FBLD`, `FBSTP`: BCD Floating-Point Load and Store

```
FBLD mem80                      ; DF /4                 [8086,FPU]
FBSTP mem80                     ; DF /6                 [8086,FPU]
```

`FBLD` loads an 80-bit (ten-byte) packed binary-coded decimal number from the given memory address, converts it to a real, and pushes it on the register stack. `FBSTP` stores the value of `ST0`, in packed BCD, at the given address and then pops the register stack.

A.32. `FCHS`: Floating-Point Change Sign

```
FCHS                            ; D9 E0                 [8086,FPU]
```

`FCHS` negates the number in `ST0`: negative numbers become positive, and vice versa.

A.33. FCLEX, {FNCLEX}: Clear Floating-Point Exceptions

```
FCLEX                           ; 9B DB E2              [8086,FPU]
FNCLEX                          ; DB E2                 [8086,FPU]
```

FCLEX clears any floating-point exceptions which may be pending. FNCLEX does the same thing but doesn't wait for previous floating-point operations (including the *handling* of pending exceptions) to finish first.

A.34. FCMOVcc: Floating-Point Conditional Move

```
FCMOVB fpureg                   ; DA C0+r               [P6,FPU]
FCMOVB ST0,fpureg               ; DA C0+r               [P6,FPU]

FCMOVBE fpureg                  ; DA D0+r               [P6,FPU]
FCMOVBE ST0,fpureg              ; DA D0+r               [P6,FPU]

FCMOVE fpureg                   ; DA C8+r               [P6,FPU]
FCMOVE ST0,fpureg               ; DA C8+r               [P6,FPU]

FCMOVNB fpureg                  ; DB C0+r               [P6,FPU]
FCMOVNB ST0,fpureg              ; DB C0+r               [P6,FPU]

FCMOVNBE fpureg                 ; DB D0+r               [P6,FPU]
FCMOVNBE ST0,fpureg             ; DB D0+r               [P6,FPU]

FCMOVNE fpureg                  ; DB C8+r               [P6,FPU]
FCMOVNE ST0,fpureg              ; DB C8+r               [P6,FPU]

FCMOVNU fpureg                  ; DB D8+r               [P6,FPU]
FCMOVNU ST0,fpureg              ; DB D8+r               [P6,FPU]

FCMOVU fpureg                   ; DA D8+r               [P6,FPU]
FCMOVU ST0,fpureg               ; DA D8+r               [P6,FPU]
```

The FCMOV instructions perform conditional move operations: each of them moves the contents of the given register into ST0 if its condition is satisfied, and does nothing if not.

The conditions are not the same as the standard condition codes used with conditional jump instructions. The conditions B, BE, NB, NBE, E and NE are exactly as normal, but none of the other standard ones are supported. Instead, the condition U and its counterpart NU are provided; the U condition is satisfied if the last two floating-point numbers compared were *unordered*, i.e. they were not equal but neither one could be said to be greater than the other, for example if they were NaNs. (The flag state which signals this is the setting of the parity flag: so the U condition is notionally equivalent to PE, and NU is equivalent to PO.)

The FCMOV conditions test the main processor's status flags, not the FPU status flags, so using FCMOV directly after FCOM will not work. Instead, you should either use FCOMI which writes directly to the main CPU flags word, or use FSTSW to extract the FPU flags.

Although the FCMOV instructions are flagged P6 above, they may not be supported by all Pentium Pro processors; the CPUID instruction (section A.22) will return a bit which indicates whether conditional moves are supported.

A.35. `FCOM`, `FCOMP`, `FCOMPP`, `FCOMI`, `FCOMIP`: Floating-Point Compare

```
FCOM mem32                      ; D8 /2                 [8086,FPU]
FCOM mem64                      ; DC /2                 [8086,FPU]
FCOM fpureg                     ; D8 D0+r               [8086,FPU]
FCOM ST0,fpureg                 ; D8 D0+r               [8086,FPU]

FCOMP mem32                     ; D8 /3                 [8086,FPU]
FCOMP mem64                     ; DC /3                 [8086,FPU]
FCOMP fpureg                    ; D8 D8+r               [8086,FPU]
FCOMP ST0,fpureg                ; D8 D8+r               [8086,FPU]

FCOMPP                          ; DE D9                 [8086,FPU]

FCOMI fpureg                    ; DB F0+r               [P6,FPU]
FCOMI ST0,fpureg                ; DB F0+r               [P6,FPU]

FCOMIP fpureg                   ; DF F0+r               [P6,FPU]
FCOMIP ST0,fpureg               ; DF F0+r               [P6,FPU]
```

`FCOM` compares `ST0` with the given operand, and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that the carry flag is set (for a `less-than' result) if `ST0` is less than the given operand.

`FCOMP` does the same as `FCOM`, but pops the register stack afterwards. `FCOMPP` compares `ST0` with `ST1` and then pops the register stack twice.

`FCOMI` and `FCOMIP` work like the corresponding forms of `FCOM` and `FCOMP`, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The `FCOM` instructions differ from the `FUCOM` instructions (section A.69) only in the way they handle quiet NaNs: `FUCOM` will handle them silently and set the condition code flags to an `unordered' result, whereas `FCOM` will generate an exception.

A.36. FCOS: Cosine

```
FCOS                                ; D9 FF                 [386,FPU]
```

FCOS computes the cosine of ST0 (in radians), and stores the result in ST0. See also FSINCOS (section A.61).

A.37. `FDECSTP`: Decrement Floating-Point Stack Pointer

```
FDECSTP                         ; D9 F6                 [8086,FPU]
```

`FDECSTP` decrements the `top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the contents of `ST7` had been pushed on the stack. See also `FINCSTP` (section A.46).

A.38. `FxDISI`, `FxENI`: Disable and Enable Floating-Point Interrupts

```
FDISI                           ; 9B DB E1              [8086,FPU]
FNDISI                          ; DB E1                 [8086,FPU]

FENI                            ; 9B DB E0              [8086,FPU]
FNENI                           ; DB E0                 [8086,FPU]
```

`FDISI` and `FENI` disable and enable floating-point interrupts. These instructions are only meaningful on original 8087 processors: the 287 and above treat them as no-operation instructions.

`FNDISI` and `FNENI` do the same thing as `FDISI` and `FENI` respectively, but without waiting for the floating-point processor to finish what it was doing first.

A.39. `FDIV`, `FDIVP`, `FDIVR`, `FDIVRP`: Floating-Point Division

```
FDIV mem32                      ; D8 /6                 [8086,FPU]
FDIV mem64                      ; DC /6                 [8086,FPU]

FDIV fpureg                     ; D8 F0+r               [8086,FPU]
FDIV ST0,fpureg                 ; D8 F0+r               [8086,FPU]

FDIV TO fpureg                  ; DC F8+r               [8086,FPU]
FDIV fpureg,ST0                 ; DC F8+r               [8086,FPU]

FDIVR mem32                     ; D8 /0                 [8086,FPU]
FDIVR mem64                     ; DC /0                 [8086,FPU]

FDIVR fpureg                    ; D8 F8+r               [8086,FPU]
FDIVR ST0,fpureg                ; D8 F8+r               [8086,FPU]

FDIVR TO fpureg                 ; DC F0+r               [8086,FPU]
FDIVR fpureg,ST0                ; DC F0+r               [8086,FPU]

FDIVP fpureg                    ; DE F8+r               [8086,FPU]
FDIVP fpureg,ST0                ; DE F8+r               [8086,FPU]

FDIVRP fpureg                   ; DE F0+r               [8086,FPU]
FDIVRP fpureg,ST0               ; DE F0+r               [8086,FPU]
```

`FDIV` divides `ST0` by the given operand and stores the result back in `ST0`, unless the `TO` qualifier is given, in which case it divides the given operand by `ST0` and stores the result in the operand.

`FDIVR` does the same thing, but does the division the other way up: so if `TO` is not given, it divides the given operand by `ST0` and stores the result in `ST0`, whereas if `TO` is given it divides `ST0` by its operand and stores the result in the operand.

`FDIVP` operates like `FDIV TO`, but pops the register stack once it has finished. `FDIVRP` operates like `FDIVR TO`, but pops the register stack once it has finished.

A.40. `FFREE`: Flag Floating-Point Register as Unused

```
FFREE fpureg                    ; DD C0+r               [8086,FPU]
```

`FFREE` marks the given register as being empty.

A.41. `FIADD`: Floating-Point/Integer Addition

```
FIADD mem16                    ; DE /0                  [8086,FPU]
FIADD mem32                    ; DA /0                  [8086,FPU]
```

`FIADD` adds the 16-bit or 32-bit integer stored in the given memory location to `ST0`, storing the result in `ST0`.

A.42. `FICOM`, `FICOMP`: Floating-Point/Integer Compare

```
FICOM mem16                     ; DE /2                 [8086,FPU]
FICOM mem32                     ; DA /2                 [8086,FPU]

FICOMP mem16                    ; DE /3                 [8086,FPU]
FICOMP mem32                    ; DA /3                 [8086,FPU]
```

`FICOM` compares `ST0` with the 16-bit or 32-bit integer stored in the given memory location, and sets the FPU flags accordingly. `FICOMP` does the same, but pops the register stack afterwards.

A.43. `FIDIV`, `FIDIVR`: Floating-Point/Integer Division

```
FIDIV mem16                     ; DE /6                 [8086,FPU]
FIDIV mem32                     ; DA /6                 [8086,FPU]

FIDIVR mem16                    ; DE /0                 [8086,FPU]
FIDIVR mem32                    ; DA /0                 [8086,FPU]
```

`FIDIV` divides `ST0` by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in `ST0`. `FIDIVR` does the division the other way up: it divides the integer by `ST0`, but still stores the result in `ST0`.

A.44. `FILD`, `FIST`, `FISTP`: Floating-Point/Integer Conversion

```
FILD mem16                      ; DF /0                 [8086,FPU]
FILD mem32                      ; DB /0                 [8086,FPU]
FILD mem64                      ; DF /5                 [8086,FPU]

FIST mem16                      ; DF /2                 [8086,FPU]
FIST mem32                      ; DB /2                 [8086,FPU]

FISTP mem16                     ; DF /3                 [8086,FPU]
FISTP mem32                     ; DB /3                 [8086,FPU]
FISTP mem64                     ; DF /0                 [8086,FPU]
```

`FILD` loads an integer out of a memory location, converts it to a real, and pushes it on the FPU register stack. `FIST` converts `ST0` to an integer and stores that in memory; `FISTP` does the same as `FIST`, but pops the register stack afterwards.

A.45. `FIMUL`: Floating-Point/Integer Multiplication

```
FIMUL mem16                    ; DE /1                [8086,FPU]
FIMUL mem32                    ; DA /1                [8086,FPU]
```

`FIMUL` multiplies `ST0` by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in `ST0`.

A.46. `FINCSTP`: Increment Floating-Point Stack Pointer

```
FINCSTP                         ; D9 F7                 [8086,FPU]
```

`FINCSTP` increments the `top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the register stack had been popped; however, unlike the popping of the stack performed by many FPU instructions, it does not flag the new `ST7` (previously `ST0`) as empty. See also `FDECSTP` (section A.37).

A.47. `FINIT`, `FNINIT`: Initialise Floating-Point Unit

```
FINIT                           ; 9B DB E3              [8086,FPU]
FNINIT                          ; DB E3                 [8086,FPU]
```

`FINIT` initialises the FPU to its default state. It flags all registers as empty, though it does not actually change their values. `FNINIT` does the same, without first waiting for pending exceptions to clear.

A.48. `FISUB`: Floating-Point/Integer Subtraction

```
FISUB mem16                     ; DE /4                 [8086,FPU]
FISUB mem32                     ; DA /4                 [8086,FPU]

FISUBR mem16                    ; DE /5                 [8086,FPU]
FISUBR mem32                    ; DA /5                 [8086,FPU]
```

`FISUB` subtracts the 16-bit or 32-bit integer stored in the given memory location from `ST0`, and stores the result in `ST0`. `FISUBR` does the subtraction the other way round, i.e. it subtracts `ST0` from the given integer, but still stores the result in `ST0`.

A.49. `FLD`: Floating-Point Load

```
FLD mem32                       ; D9 /0                 [8086,FPU]
FLD mem64                       ; DD /0                 [8086,FPU]
FLD mem80                       ; DB /5                 [8086,FPU]
FLD fpureg                      ; D9 C0+r               [8086,FPU]
```

`FLD` loads a floating-point value out of the given register or memory location, and pushes it on the FPU register stack.

A.50. `FLDxx`: Floating-Point Load Constants

```
FLD1                            ; D9 E8                 [8086,FPU]
FLDL2E                          ; D9 EA                 [8086,FPU]
FLDL2T                          ; D9 E9                 [8086,FPU]
FLDLG2                          ; D9 EC                 [8086,FPU]
FLDLN2                          ; D9 ED                 [8086,FPU]
FLDPI                           ; D9 EB                 [8086,FPU]
FLDZ                            ; D9 EE                 [8086,FPU]
```

These instructions push specific standard constants on the FPU register stack. `FLD1` pushes the value 1; `FLDL2E` pushes the base-2 logarithm of e; `FLDL2T` pushes the base-2 log of 10; `FLDLG2` pushes the base-10 log of 2; `FLDLN2` pushes the base-e log of 2; `FLDPI` pushes pi; and `FLDZ` pushes zero.

A.51. `FLDCW`: Load Floating-Point Control Word

```
FLDCW mem16                     ; D9 /5                 [8086,FPU]
```

`FLDCW` loads a 16-bit value out of memory and stores it into the FPU control word (governing things like the rounding mode, the precision, and the exception masks). See also `FSTCW` (section A.64).

A.52. `FLDENV`: Load Floating-Point Environment

```
FLDENV mem                      ; D9 /4                 [8086,FPU]
```

`FLDENV` loads the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) from memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also `FSTENV` (section A.65).

A.53. `FMUL`, `FMULP`: Floating-Point Multiply

```
FMUL mem32                      ; D8 /1                 [8086,FPU]
FMUL mem64                      ; DC /1                 [8086,FPU]

FMUL fpureg                     ; D8 C8+r               [8086,FPU]
FMUL ST0,fpureg                 ; D8 C8+r               [8086,FPU]

FMUL TO fpureg                  ; DC C8+r               [8086,FPU]
FMUL fpureg,ST0                 ; DC C8+r               [8086,FPU]

FMULP fpureg                    ; DE C8+r               [8086,FPU]
FMULP fpureg,ST0                ; DE C8+r               [8086,FPU]
```

`FMUL` multiplies `ST0` by the given operand, and stores the result in `ST0`, unless the `TO` qualifier is used in which case it stores the result in the operand. `FMULP` performs the same operation as `FMUL TO`, and then pops the register stack.

A.54. `FNOP`: Floating-Point No Operation

```
FNOP                                ; D9 D0                 [8086,FPU]
```

`FNOP` does nothing.

A.55. `FPATAN`, `FPTAN`: Arctangent and Tangent

```
FPATAN                          ; D9 F3                 [8086,FPU]
FPTAN                           ; D9 F2                 [8086,FPU]
```

`FPATAN` computes the arctangent, in radians, of the result of dividing `ST1` by `ST0`, stores the result in `ST1`, and pops the register stack. It works like the C `atan2` function, in that changing the sign of both `ST0` and `ST1` changes the output value by pi (so it performs true rectangular-to-polar coordinate conversion, with `ST1` being the Y coordinate and `ST0` being the X coordinate, not merely an arctangent).

`FPTAN` computes the tangent of the value in `ST0` (in radians), and stores the result back into `ST0`.

A.56. `FPREM`, `FPREM1`: Floating-Point Partial Remainder

```
FPREM                           ; D9 F8                 [8086,FPU]
FPREM1                          ; D9 F5                 [386,FPU]
```

These instructions both produce the remainder obtained by dividing `ST0` by `ST1`. This is calculated, notionally, by dividing `ST0` by `ST1`, rounding the result to an integer, multiplying by `ST1` again, and computing the value which would need to be added back on to the result to get back to the original value in `ST0`.

The two instructions differ in the way the notional round-to-integer operation is performed. `FPREM` does it by rounding towards zero, so that the remainder it returns always has the same sign as the original value in `ST0`; `FPREM1` does it by rounding to the nearest integer, so that the remainder always has at most half the magnitude of `ST1`.

Both instructions calculate *partial* remainders, meaning that they may not manage to provide the final result, but might leave intermediate results in `ST0` instead. If this happens, they will set the C2 flag in the FPU status word; therefore, to calculate a remainder, you should repeatedly execute `FPREM` or `FPREM1` until C2 becomes clear.

A.57. `FRNDINT`: Floating-Point Round to Integer

```
FRNDINT                         ; D9 FC                 [8086,FPU]
```

`FRNDINT` rounds the contents of `ST0` to an integer, according to the current rounding mode set in the FPU control word, and stores the result back in `ST0`.

A.58. FSAVE, FRSTOR: Save/Restore Floating-Point State

```
FSAVE mem                       ; 9B DD /6              [8086,FPU]
FNSAVE mem                      ; DD /6                 [8086,FPU]

FRSTOR mem                      ; DD /4                 [8086,FPU]
```

FSAVE saves the entire floating-point unit state, including all the information saved by FSTENV (section A.65) plus the contents of all the registers, to a 94 or 108 byte area of memory (depending on the CPU mode). FRSTOR restores the floating-point state from the same area of memory.

FNSAVE does the same as FSAVE, without first waiting for pending floating-point exceptions to clear.

A.59. `FSCALE`: Scale Floating-Point Value by Power of Two

```
FSCALE                          ; D9 FD                 [8086,FPU]
```

`FSCALE` scales a number by a power of two: it rounds `ST1` towards zero to obtain an integer, then multiplies `ST0` by two to the power of that integer, and stores the result in `ST0`.

A.60. `FSETPM`: Set Protected Mode

```
FSETPM                          ; DB E4                 [286,FPU]
```

This instruction initalises protected mode on the 287 floating-point coprocessor. It is only meaningful on that processor: the 387 and above treat the instruction as a no-operation.

A.61. FSIN, FSINCOS: Sine and Cosine

```
FSIN                            ; D9 FE                 [386,FPU]
FSINCOS                         ; D9 FB                 [386,FPU]
```

FSIN calculates the sine of ST0 (in radians) and stores the result in ST0. FSINCOS does the same, but then pushes the cosine of the same value on the register stack, so that the sine ends up in ST1 and the cosine in ST0. FSINCOS is faster than executing FSIN and FCOS (see section A.36) in succession.

A.62. `FSQRT`: Floating-Point Square Root

```
FSQRT                           ; D9 FA                 [8086,FPU]
```

`FSQRT` calculates the square root of `ST0` and stores the result in `ST0`.

A.63. `FST`, `FSTP`: Floating-Point Store

```
FST mem32                       ; D9 /2                 [8086,FPU]
FST mem64                       ; DD /2                 [8086,FPU]
FST fpureg                      ; DD D0+r               [8086,FPU]

FSTP mem32                      ; D9 /3                 [8086,FPU]
FSTP mem64                      ; DD /3                 [8086,FPU]
FSTP mem80                      ; DB /0                 [8086,FPU]
FSTP fpureg                     ; DD D8+r               [8086,FPU]
```

`FST` stores the value in `ST0` into the given memory location or other FPU register. `FSTP` does the same, but then pops the register stack.

A.64. `FSTCW`: Store Floating-Point Control Word

```
FSTCW mem16                     ; 9B D9 /0              [8086,FPU]
FNSTCW mem16                    ; D9 /0                 [8086,FPU]
```

`FSTCW` stores the FPU control word (governing things like the rounding mode, the precision, and the exception masks) into a 2-byte memory area. See also `FLDCW` (section A.51).

`FNSTCW` does the same thing as `FSTCW`, without first waiting for pending floating-point exceptions to clear.

A.65. `FSTENV`: Store Floating-Point Environment

```
FSTENV mem                      ; 9B D9 /6              [8086,FPU]
FNSTENV mem                     ; D9 /6                 [8086,FPU]
```

`FSTENV` stores the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) into memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also `FLDENV` (section A.52).

`FNSTENV` does the same thing as `FSTENV`, without first waiting for pending floating-point exceptions to clear.

A.66. `FSTSW`: Store Floating-Point Status Word

```
FSTSW mem16                     ; 9B DD /0              [8086,FPU]
FSTSW AX                        ; 9B DF E0              [286,FPU]

FNSTSW mem16                    ; DD /0                 [8086,FPU]
FNSTSW AX                       ; DF E0                 [286,FPU]
```

`FSTSW` stores the FPU status word into `AX` or into a 2-byte memory area.

`FNSTSW` does the same thing as `FSTSW`, without first waiting for pending floating-point exceptions to clear.

A.67. `FSUB`, `FSUBP`, `FSUBR`, `FSUBRP`: Floating-Point Subtract

```
FSUB mem32                      ; D8 /4                 [8086,FPU]
FSUB mem64                      ; DC /4                 [8086,FPU]

FSUB fpureg                     ; D8 E0+r               [8086,FPU]
FSUB ST0,fpureg                 ; D8 E0+r               [8086,FPU]

FSUB TO fpureg                  ; DC E8+r               [8086,FPU]
FSUB fpureg,ST0                 ; DC E8+r               [8086,FPU]

FSUBR mem32                     ; D8 /5                 [8086,FPU]
FSUBR mem64                     ; DC /5                 [8086,FPU]

FSUBR fpureg                    ; D8 E8+r               [8086,FPU]
FSUBR ST0,fpureg                ; D8 E8+r               [8086,FPU]

FSUBR TO fpureg                 ; DC E0+r               [8086,FPU]
FSUBR fpureg,ST0                ; DC E0+r               [8086,FPU]

FSUBP fpureg                    ; DE E8+r               [8086,FPU]
FSUBP fpureg,ST0                ; DE E8+r               [8086,FPU]

FSUBRP fpureg                   ; DE E0+r               [8086,FPU]
FSUBRP fpureg,ST0               ; DE E0+r               [8086,FPU]
```

`FSUB` subtracts the given operand from `ST0` and stores the result back in `ST0`, unless the `TO` qualifier is given, in which case it subtracts `ST0` from the given operand and stores the result in the operand.

`FSUBR` does the same thing, but does the subtraction the other way up: so if `TO` is not given, it subtracts `ST0` from the given operand and stores the result in `ST0`, whereas if `TO` is given it subtracts its operand from `ST0` and stores the result in the operand.

`FSUBP` operates like `FSUB TO`, but pops the register stack once it has finished. `FSUBRP` operates like `FSUBR TO`, but pops the register stack once it has finished.

A.68. `FTST`: Test `ST0` Against Zero

```
FTST                            ; D9 E4                 [8086,FPU]
```

`FTST` compares `ST0` with zero and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that a `less-than' result is generated if `ST0` is negative.

A.69. FUCOMxx: Floating-Point Unordered Compare

```
FUCOM fpureg                    ; DD E0+r               [386,FPU]
FUCOM ST0,fpureg                ; DD E0+r               [386,FPU]

FUCOMP fpureg                   ; DD E8+r               [386,FPU]
FUCOMP ST0,fpureg               ; DD E8+r               [386,FPU]

FUCOMPP                         ; DA E9                 [386,FPU]

FUCOMI fpureg                   ; DB E8+r               [P6,FPU]
FUCOMI ST0,fpureg               ; DB E8+r               [P6,FPU]

FUCOMIP fpureg                  ; DF E8+r               [P6,FPU]
FUCOMIP ST0,fpureg              ; DF E8+r               [P6,FPU]
```

`FUCOM` compares `ST0` with the given operand, and sets the FPU flags accordingly. `ST0` is treated as the left-hand side of the comparison, so that the carry flag is set (for a `less-than' result) if `ST0` is less than the given operand.

`FUCOMP` does the same as `FUCOM`, but pops the register stack afterwards. `FUCOMPP` compares `ST0` with `ST1` and then pops the register stack twice.

`FUCOMI` and `FUCOMIP` work like the corresponding forms of `FUCOM` and `FUCOMP`, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.

The `FUCOM` instructions differ from the `FCOM` instructions (section A.35) only in the way they handle quiet NaNs: `FUCOM` will handle them silently and set the condition code flags to an `unordered' result, whereas `FCOM` will generate an exception.

A.70. `FXAM`: Examine Class of Value in `ST0`

```
FXAM                            ; D9 E5                 [8086,FPU]
```

`FXAM` sets the FPU flags C3, C2 and C0 depending on the type of value stored in `ST0`: 000 (respectively) for an unsupported format, 001 for a NaN, 010 for a normal finite number, 011 for an infinity, 100 for a zero, 101 for an empty register, and 110 for a denormal. It also sets the C1 flag to the sign of the number.

A.71. FXCH: Floating-Point Exchange

```
FXCH                            ; D9 C9                 [8086,FPU]
FXCH fpureg                     ; D9 C8+r               [8086,FPU]
FXCH fpureg,ST0                 ; D9 C8+r               [8086,FPU]
FXCH ST0,fpureg                 ; D9 C8+r               [8086,FPU]
```

FXCH exchanges ST0 with a given FPU register. The no-operand form exchanges ST0 with ST1.

A.72. `FXTRACT`: Extract Exponent and Significand

```
FXTRACT                         ; D9 F4                 [8086,FPU]
```

`FXTRACT` separates the number in `ST0` into its exponent and significand (mantissa), stores the exponent back into `ST0`, and then pushes the significand on the register stack (so that the significand ends up in `ST0`, and the exponent in `ST1`).

A.73. `FYL2X`, `FYL2XP1`: Compute Y times Log2(X) or Log2(X+1)

```
FYL2X                           ; D9 F1                 [8086,FPU]
FYL2XP1                         ; D9 F9                 [8086,FPU]
```

`FYL2X` multiplies `ST1` by the base-2 logarithm of `ST0`, stores the result in `ST1`, and pops the register stack (so that the result ends up in `ST0`). `ST0` must be non-zero and positive.

`FYL2XP1` works the same way, but replacing the base-2 log of `ST0` with that of `ST0` plus one. This time, `ST0` must have magnitude no greater than 1 minus half the square root of two.

A.74. `HLT`: Halt Processor

```
HLT                                 ; F4                        [8086]
```

`HLT` puts the processor into a halted state, where it will perform no more operations until restarted by an interrupt or a reset.

A.75. IBTS: Insert Bit String

```
IBTS r/m16,reg16              ; o16 0F A7 /r          [386,UNDOC]
IBTS r/m32,reg32              ; o32 0F A7 /r          [386,UNDOC]
```

No clear documentation seems to be available for this instruction: the best I've been able to find reads `Takes a string of bits from the second operand and puts them in the first operand'. It is present only in early 386 processors, and conflicts with the opcodes for CMPXCHG486. NASM supports it only for completeness. Its counterpart is XBTS (see section A.167).

A.76. IDIV: Signed Integer Divide

```
IDIV r/m8                       ; F6 /7                 [8086]
IDIV r/m16                      ; o16 F7 /7             [8086]
IDIV r/m32                      ; o32 F7 /7             [386]
```

IDIV performs signed integer division. The explicit operand provided is the divisor; the dividend and destination operands are implicit, in the following way:

- For IDIV r/m8, AX is divided by the given operand; the quotient is stored in AL and the remainder in AH.
- For IDIV r/m16, DX:AX is divided by the given operand; the quotient is stored in AX and the remainder in DX.
- For IDIV r/m32, EDX:EAX is divided by the given operand; the quotient is stored in EAX and the remainder in EDX.

Unsigned integer division is performed by the DIV instruction: see section A.25.

A.77. IMUL: Signed Integer Multiply

```
IMUL r/m8                       ; F6 /5               [8086]
IMUL r/m16                      ; o16 F7 /5           [8086]
IMUL r/m32                      ; o32 F7 /5           [386]

IMUL reg16,r/m16                ; o16 0F AF /r        [386]
IMUL reg32,r/m32                ; o32 0F AF /r        [386]

IMUL reg16,imm8                 ; o16 6B /r ib        [286]
IMUL reg16,imm16                ; o16 69 /r iw        [286]
IMUL reg32,imm8                 ; o32 6B /r ib        [386]
IMUL reg32,imm32                ; o32 69 /r id        [386]

IMUL reg16,r/m16,imm8           ; o16 6B /r ib        [286]
IMUL reg16,r/m16,imm16          ; o16 69 /r iw        [286]
IMUL reg32,r/m32,imm8           ; o32 6B /r ib        [386]
IMUL reg32,r/m32,imm32          ; o32 69 /r id        [386]
```

IMUL performs signed integer multiplication. For the single-operand form, the other operand and destination are implicit, in the following way:

- For IMUL r/m8, AL is multiplied by the given operand; the product is stored in AX.
- For IMUL r/m16, AX is multiplied by the given operand; the product is stored in DX:AX.
- For IMUL r/m32, EAX is multiplied by the given operand; the product is stored in EDX:EAX.

The two-operand form multiplies its two operands and stores the result in the destination (first) operand. The three-operand form multiplies its last two operands and stores the result in the first operand.

The two-operand form is in fact a shorthand for the three-operand form, as can be seen by examining the opcode descriptions: in the two-operand form, the code /r takes both its register and r/m parts from the same operand (the first one).

In the forms with an 8-bit immediate operand and another longer source operand, the immediate operand is considered to be signed, and is sign-extended to the length of the other source operand. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

Unsigned integer multiplication is performed by the MUL instruction: see section A.107.

A.78. IN: Input from I/O Port

```
IN AL,imm8                      ; E4 ib                 [8086]
IN AX,imm8                      ; o16 E5 ib             [8086]
IN EAX,imm8                     ; o32 E5 ib             [386]
IN AL,DX                        ; EC                    [8086]
IN AX,DX                        ; o16 ED                [8086]
IN EAX,DX                       ; o32 ED                [386]
```

IN reads a byte, word or doubleword from the specified I/O port, and stores it in the given destination register. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in DX. See also OUT (section A.111).

A.79. `INC`: Increment Integer

```
INC reg16                       ; o16 40+r              [8086]
INC reg32                       ; o32 40+r              [386]
INC r/m8                        ; FE /0                 [8086]
INC r/m16                       ; o16 FF /0             [8086]
INC r/m32                       ; o32 FF /0             [386]
```

`INC` adds 1 to its operand. It does *not* affect the carry flag: to affect the carry flag, use `ADD something,1` (see section A.6). See also `DEC` (section A.24).

A.80. `INSB`, `INSW`, `INSD`: Input String from I/O Port

```
INSB                            ; 6C                    [186]
INSW                            ; o16 6D                [186]
INSD                            ; o32 6D                [386]
```

`INSB` inputs a byte from the I/O port specified in `DX` and stores it at `[ES:DI]` or `[ES:EDI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` or `EDI`.

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the load from `[DI]` or `[EDI]` cannot be overridden.

`INSW` and `INSD` work in the same way, but they input a word or a doubleword instead of a byte, and increment or decrement the addressing register by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` – again, the address size chooses which) times.

See also `OUTSB`, `OUTSW` and `OUTSD` (section A.112).

A.81. `INT`: Software Interrupt

```
INT imm8                        ; CD ib                 [8086]
```

`INT` causes a software interrupt through a specified vector number from 0 to 255.

The code generated by the `INT` instruction is always two bytes long: although there are short forms for some `INT` instructions, NASM does not generate them when it sees the `INT` mnemonic. In order to generate single-byte breakpoint instructions, use the `INT3` or `INT1` instructions (see section A.82) instead.

A.82. `INT3`, `INT1`, `ICEBP`, `INT01`: Breakpoints

```
INT1                            ; F1                    [P6]
ICEBP                           ; F1                    [P6]
INT01                           ; F1                    [P6]

INT3                            ; CC                    [8086]
```

`INT1` and `INT3` are short one-byte forms of the instructions `INT 1` and `INT 3` (see section A.81). They perform a similar function to their longer counterparts, but take up less code space. They are used as breakpoints by debuggers.

`INT1`, and its alternative synonyms `INT01` and `ICEBP`, is an instruction used by in-circuit emulators (ICEs). It is present, though not documented, on some processors down to the 286, but is only documented for the Pentium Pro. `INT3` is the instruction normally used as a breakpoint by debuggers.

`INT3` is not precisely equivalent to `INT 3`: the short form, since it is designed to be used as a breakpoint, bypasses the normal IOPL checks in virtual-8086 mode, and also does not go through interrupt redirection.

A.83. `INTO`: Interrupt if Overflow

```
INTO                                ; CE                        [8086]
```

`INTO` performs an `INT 4` software interrupt (see section A.81) if and only if the overflow flag is set.

A.84. INVD: Invalidate Internal Caches

```
INVD                            ; 0F 08                 [486]
```

INVD invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It does not write the contents of the caches back to memory first: any modified data held in the caches will be lost. To write the data back first, use WBINVD (section A.164).

A.85. `INVLPG`: Invalidate TLB Entry

```
INVLPG mem                      ; 0F 01 /0              [486]
```

`INVLPG` invalidates the translation lookahead buffer (TLB) entry associated with the supplied memory address.

A.86. `IRET`, `IRETW`, `IRETD`: Return from Interrupt

```
IRET                            ; CF                    [8086]
IRETW                           ; o16 CF                [8086]
IRETD                           ; o32 CF                [386]
```

`IRET` returns from an interrupt (hardware or software) by means of popping `IP` (or `EIP`), `CS` and the flags off the stack and then continuing execution from the new `CS:IP`.

`IRETW` pops `IP`, `CS` and the flags as 2 bytes each, taking 6 bytes off the stack in total. `IRETD` pops `EIP` as 4 bytes, pops a further 4 bytes of which the top two are discarded and the bottom two go into `CS`, and pops the flags as 4 bytes as well, taking 12 bytes off the stack.

`IRET` is a shorthand for either `IRETW` or `IRETD`, depending on the default `BITS` setting at the time.

A.87. JCXZ, JECXZ: Jump if CX/ECX Zero

```
JCXZ imm                        ; o16 E3 rb             [8086]
JECXZ imm                       ; o32 E3 rb             [386]
```

JCXZ performs a short jump (with maximum range 128 bytes) if and only if the contents of the CX register is 0. JECXZ does the same thing, but with ECX.

A.88. JMP: Jump

```
JMP imm                         ; E9 rw/rd              [8086]
JMP SHORT imm                   ; EB rb                 [8086]
JMP imm:imm16                   ; o16 EA iw iw          [8086]
JMP imm:imm32                   ; o32 EA id iw          [386]
JMP FAR mem                     ; o16 FF /5             [8086]
JMP FAR mem                     ; o32 FF /5             [386]
JMP r/m16                       ; o16 FF /4             [8086]
JMP r/m32                       ; o32 FF /4             [386]
```

`JMP` jumps to a given address. The address may be specified as an absolute segment and offset, or as a relative jump within the current segment.

`JMP SHORT imm` has a maximum range of 128 bytes, since the displacement is specified as only 8 bits, but takes up less code space. NASM does not choose when to generate `JMP SHORT` for you: you must explicitly code `SHORT` every time you want a short jump.

You can choose between the two immediate far jump forms (`JMP imm:imm`) by the use of the `WORD` and `DWORD` keywords: `JMP WORD 0x1234:0x5678`) or `JMP DWORD 0x1234:0x56789abc`.

The `JMP FAR mem` forms execute a far jump by loading the destination address out of memory. The address loaded consists of 16 or 32 bits of offset (depending on the operand size), and 16 bits of segment. The operand size may be overridden using `JMP WORD FAR mem` or `JMP DWORD FAR mem`.

The `JMP r/m` forms execute a near jump (within the same segment), loading the destination address out of memory or out of a register. The keyword `NEAR` may be specified, for clarity, in these forms, but is not necessary. Again, operand size can be overridden using `JMP WORD mem` or `JMP DWORD mem`.

As a convenience, NASM does not require you to jump to a far symbol by coding the cumbersome `JMP SEG routine:routine`, but instead allows the easier synonym `JMP FAR routine`.

The `CALL r/m` forms given above are near calls; NASM will accept the `NEAR` keyword (e.g. `CALL NEAR [address]`), even though it is not strictly necessary.

A.89. Jcc: Conditional Branch

```
Jcc imm                         ; 70+cc rb              [8086]
Jcc NEAR imm                    ; 0F 80+cc rw/rd        [386]
```

The conditional jump instructions execute a near (same segment) jump if and only if their conditions are satisfied. For example, JNZ jumps only if the zero flag is not set.

The ordinary form of the instructions has only a 128-byte range; the NEAR form is a 386 extension to the instruction set, and can span the full size of a segment. NASM will not override your choice of jump instruction: if you want Jcc NEAR, you have to use the NEAR keyword.

The SHORT keyword is allowed on the first form of the instruction, for clarity, but is not necessary.

A.90. `LAHF`: Load AH from Flags

```
LAHF                            ; 9F                    [8086]
```

`LAHF` sets the `AH` register according to the contents of the low byte of the flags word. See also `SAHF` (section A.145).

A.91. LAR: Load Access Rights

```
LAR reg16,r/m16                 ; o16 0F 02 /r          [286,PRIV]
LAR reg32,r/m32                 ; o32 0F 02 /r          [286,PRIV]
```

LAR takes the segment selector specified by its source (second) operand, finds the corresponding segment descriptor in the GDT or LDT, and loads the access-rights byte of the descriptor into its destination (first) operand.

A.92. `LDS`, `LES`, `LFS`, `LGS`, `LSS`: Load Far Pointer

```
LDS reg16,mem                   ; o16 C5 /r             [8086]
LDS reg32,mem                   ; o32 C5 /r             [8086]

LES reg16,mem                   ; o16 C4 /r             [8086]
LES reg32,mem                   ; o32 C4 /r             [8086]

LFS reg16,mem                   ; o16 0F B4 /r          [386]
LFS reg32,mem                   ; o32 0F B4 /r          [386]

LGS reg16,mem                   ; o16 0F B5 /r          [386]
LGS reg32,mem                   ; o32 0F B5 /r          [386]

LSS reg16,mem                   ; o16 0F B2 /r          [386]
LSS reg32,mem                   ; o32 0F B2 /r          [386]
```

These instructions load an entire far pointer (16 or 32 bits of offset, plus 16 bits of segment) out of memory in one go. `LDS`, for example, loads 16 or 32 bits from the given memory address into the given register (depending on the size of the register), then loads the *next* 16 bits from memory into `DS`. `LES`, `LFS`, `LGS` and `LSS` work in the same way but use the other segment registers.

A.93. `LEA`: Load Effective Address

```
LEA reg16,mem                   ; o16 8D /r             [8086]
LEA reg32,mem                   ; o32 8D /r             [8086]
```

`LEA`, despite its syntax, does not access memory. It calculates the effective address specified by its second operand as if it were going to load or store data from it, but instead it stores the calculated address into the register specified by its first operand. This can be used to perform quite complex calculations (e.g. `LEA EAX,[EBX+ECX*4+100]`) in one instruction.

`LEA`, despite being a purely arithmetic instruction which accesses no memory, still requires square brackets around its second operand, as if it were a memory reference.

A.94. `LEAVE`: Destroy Stack Frame

```
LEAVE                           ; C9                    [186]
```

`LEAVE` destroys a stack frame of the form created by the `ENTER` instruction (see section A.27). It is functionally equivalent to `MOV ESP,EBP` followed by `POP EBP` (or `MOV SP,BP` followed by `POP BP` in 16-bit mode).

A.95. `LGDT`, `LIDT`, `LLDT`: Load Descriptor Tables

```
LGDT mem                        ; 0F 01 /2              [286,PRIV]
LIDT mem                        ; 0F 01 /3              [286,PRIV]
LLDT r/m16                      ; 0F 00 /2              [286,PRIV]
```

`LGDT` and `LIDT` both take a 6-byte memory area as an operand: they load a 32-bit linear address and a 16-bit size limit from that area (in the opposite order) into the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register). These are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

`LLDT` takes a segment selector as an operand. The processor looks up that selector in the GDT and stores the limit and base address given there into the LDTR (local descriptor table register).

See also `SGDT`, `SIDT` and `SLDT` (section A.151).

A.96. LMSW: Load/Store Machine Status Word

```
LMSW r/m16                      ; 0F 01 /6              [286,PRIV]
```

LMSW loads the bottom four bits of the source operand into the bottom four bits of the CR0 control register (or the Machine Status Word, on 286 processors). See also SMSW (section A.155).

A.97. `LOADALL`, `LOADALL286`: Load Processor State

```
LOADALL                         ; 0F 07                 [386,UNDOC]
LOADALL286                      ; 0F 05                 [286,UNDOC]
```

This instruction, in its two different-opcode forms, is apparently supported on most 286 processors, some 386 and possibly some 486. The opcode differs between the 286 and the 386.

The function of the instruction is to load all information relating to the state of the processor out of a block of memory: on the 286, this block is located implicitly at absolute address `0x800`, and on the 386 and 486 it is at `[ES:EDI]`.

A.98. `LODSB`, `LODSW`, `LODSD`: Load from String

```
LODSB                           ; AC                    [8086]
LODSW                           ; o16 AD                [8086]
LODSD                           ; o32 AD                [386]
```

`LODSB` loads a byte from `[DS:SI]` or `[DS:ESI]` into `AL`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` or `ESI`.

The register used is `SI` if the address size is 16 bits, and `ESI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es lodsb`).

`LODSW` and `LODSD` work in the same way, but they load a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

A.99. `LOOP`, `LOOPE`, `LOOPZ`, `LOOPNE`, `LOOPNZ`: Loop with Counter

```
LOOP imm                        ; E2 rb                 [8086]
LOOP imm,CX                     ; a16 E2 rb             [8086]
LOOP imm,ECX                    ; a32 E2 rb             [386]

LOOPE imm                       ; E1 rb                 [8086]
LOOPE imm,CX                    ; a16 E1 rb             [8086]
LOOPE imm,ECX                   ; a32 E1 rb             [386]
LOOPZ imm                       ; E1 rb                 [8086]
LOOPZ imm,CX                    ; a16 E1 rb             [8086]
LOOPZ imm,ECX                   ; a32 E1 rb             [386]

LOOPNE imm                      ; E0 rb                 [8086]
LOOPNE imm,CX                   ; a16 E0 rb             [8086]
LOOPNE imm,ECX                  ; a32 E0 rb             [386]
LOOPNZ imm                      ; E0 rb                 [8086]
LOOPNZ imm,CX                   ; a16 E0 rb             [8086]
LOOPNZ imm,ECX                  ; a32 E0 rb             [386]
```

`LOOP` decrements its counter register (either `CX` or `ECX` – if one is not specified explicitly, the `BITS` setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.

`LOOPE` (or its synonym `LOOPZ`) adds the additional condition that it only jumps if the counter is nonzero *and* the zero flag is set. Similarly, `LOOPNE` (and `LOOPNZ`) jumps only if the counter is nonzero and the zero flag is clear.

A.100. `LSL`: Load Segment Limit

```
LSL reg16,r/m16                 ; o16 0F 03 /r          [286,PRIV]
LSL reg32,r/m32                 ; o32 0F 03 /r          [286,PRIV]
```

`LSL` is given a segment selector in its source (second) operand; it computes the segment limit value by loading the segment limit field from the associated segment descriptor in the GDT or LDT. (This involves shifting left by 12 bits if the segment limit is page-granular, and not if it is byte-granular; so you end up with a byte limit in either case.) The segment limit obtained is then loaded into the destination (first) operand.

A.101. `LTR`: Load Task Register

```
LTR r/m16                       ; 0F 00 /3              [286,PRIV]
```

`LTR` looks up the segment base and limit in the GDT or LDT descriptor specified by the segment selector given as its operand, and loads them into the Task Register.

A.102. MOV: Move Data

```
MOV r/m8,reg8                   ; 88 /r                 [8086]
MOV r/m16,reg16                 ; o16 89 /r             [8086]
MOV r/m32,reg32                 ; o32 89 /r             [386]
MOV reg8,r/m8                   ; 8A /r                 [8086]
MOV reg16,r/m16                 ; o16 8B /r             [8086]
MOV reg32,r/m32                 ; o32 8B /r             [386]

MOV reg8,imm8                   ; B0+r ib               [8086]
MOV reg16,imm16                 ; o16 B8+r iw           [8086]
MOV reg32,imm32                 ; o32 B8+r id           [386]
MOV r/m8,imm8                   ; C6 /0 ib              [8086]
MOV r/m16,imm16                 ; o16 C7 /0 iw          [8086]
MOV r/m32,imm32                 ; o32 C7 /0 id          [386]

MOV AL,memoffs8                 ; A0 ow/od              [8086]
MOV AX,memoffs16                ; o16 A1 ow/od          [8086]
MOV EAX,memoffs32               ; o32 A1 ow/od          [386]
MOV memoffs8,AL                 ; A2 ow/od              [8086]
MOV memoffs16,AX                ; o16 A3 ow/od          [8086]
MOV memoffs32,EAX               ; o32 A3 ow/od          [386]

MOV r/m16,segreg                ; o16 8C /r             [8086]
MOV r/m32,segreg                ; o32 8C /r             [386]
MOV segreg,r/m16                ; o16 8E /r             [8086]
MOV segreg,r/m32                ; o32 8E /r             [386]

MOV reg32,CR0/2/3/4             ; 0F 20 /r              [386]
MOV reg32,DR0/1/2/3/6/7         ; 0F 21 /r              [386]
MOV reg32,TR3/4/5/6/7           ; 0F 24 /r              [386]
MOV CR0/2/3/4,reg32             ; 0F 22 /r              [386]
MOV DR0/1/2/3/6/7,reg32         ; 0F 23 /r              [386]
MOV TR3/4/5/6/7,reg32           ; 0F 26 /r              [386]
```

`MOV` copies the contents of its source (second) operand into its destination (first) operand.

In all forms of the `MOV` instruction, the two operands are the same size, except for moving between a segment register and an `r/m32` operand. These instructions are treated exactly like the corresponding 16-bit equivalent (so that, for example, `MOV DS,EAX` functions identically to `MOV DS,AX` but saves a prefix when in 32-bit mode), except that when a segment register is moved into a 32-bit destination, the top two bytes of the result are undefined.

`MOV` may not use `CS` as a destination.

`CR4` is only a supported register on the Pentium and above.

A.103. MOVD: Move Doubleword to/from MMX Register

```
MOVD mmxreg,r/m32               ; 0F 6E /r              [PENT,MMX]
MOVD r/m32,mmxreg               ; 0F 7E /r              [PENT,MMX]
```

MOVD copies 32 bits from its source (second) operand into its destination (first) operand. When the destination is a 64-bit MMX register, the top 32 bits are set to zero.

A.104. MOVQ: Move Quadword to/from MMX Register

```
MOVQ mmxreg,r/m64               ; 0F 6F /r              [PENT,MMX]
MOVQ r/m64,mmxreg               ; 0F 7F /r              [PENT,MMX]
```

MOVQ copies 64 bits from its source (second) operand into its destination (first) operand.

A.105. MOVSB, MOVSW, MOVSD: Move String

```
MOVSB                           ; A4                    [8086]
MOVSW                           ; o16 A5                [8086]
MOVSD                           ; o32 A5                [386]
```

`MOVSB` copies the byte at `[ES:DI]` or `[ES:EDI]` to `[DS:SI]` or `[DS:ESI]`. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `SI` and `DI` (or `ESI` and `EDI`).

The registers used are `SI` and `DI` if the address size is 16 bits, and `ESI` and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[SI]` or `[ESI]` can be overridden by using a segment register name as a prefix (for example, `es movsb`). The use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.

`MOVSW` and `MOVSD` work in the same way, but they copy a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` – again, the address size chooses which) times.

A.106. MOVSX, MOVZX: Move Data with Sign or Zero Extend

```
MOVSX reg16,r/m8              ; o16 0F BE /r          [386]
MOVSX reg32,r/m8              ; o32 0F BE /r          [386]
MOVSX reg32,r/m16             ; o32 0F BF /r          [386]

MOVZX reg16,r/m8              ; o16 0F B6 /r          [386]
MOVZX reg32,r/m8              ; o32 0F B6 /r          [386]
MOVZX reg32,r/m16             ; o32 0F B7 /r          [386]
```

MOVSX sign-extends its source (second) operand to the length of its destination (first) operand, and copies the result into the destination operand. MOVZX does the same, but zero-extends rather than sign-extending.

A.107. MUL: Unsigned Integer Multiply

```
MUL r/m8                        ; F6 /4                 [8086]
MUL r/m16                       ; o16 F7 /4             [8086]
MUL r/m32                       ; o32 F7 /4             [386]
```

MUL performs unsigned integer multiplication. The other operand to the multiplication, and the destination operand, are implicit, in the following way:

- For MUL r/m8, AL is multiplied by the given operand; the product is stored in AX.
- For MUL r/m16, AX is multiplied by the given operand; the product is stored in DX:AX.
- For MUL r/m32, EAX is multiplied by the given operand; the product is stored in EDX:EAX.

Signed integer multiplication is performed by the IMUL instruction: see section A.77.

A.108. NEG, NOT: Two's and One's Complement

```
NEG r/m8                        ; F6 /3                 [8086]
NEG r/m16                       ; o16 F7 /3             [8086]
NEG r/m32                       ; o32 F7 /3             [386]

NOT r/m8                        ; F6 /2                 [8086]
NOT r/m16                       ; o16 F7 /2             [8086]
NOT r/m32                       ; o32 F7 /2             [386]
```

NEG replaces the contents of its operand by the two's complement negation (invert all the bits and then add one) of the original value. NOT, similarly, performs one's complement (inverts all the bits).

A.109. `NOP`: No Operation

```
NOP                                 ; 90                        [8086]
```

`NOP` performs no operation. Its opcode is the same as that generated by `XCHG AX,AX` or `XCHG EAX,EAX` (depending on the processor mode; see section A.168).

A.110. OR: Bitwise OR

```
OR r/m8,reg8                    ; 08 /r                 [8086]
OR r/m16,reg16                  ; o16 09 /r             [8086]
OR r/m32,reg32                  ; o32 09 /r             [386]

OR reg8,r/m8                    ; 0A /r                 [8086]
OR reg16,r/m16                  ; o16 0B /r             [8086]
OR reg32,r/m32                  ; o32 0B /r             [386]

OR r/m8,imm8                    ; 80 /1 ib              [8086]
OR r/m16,imm16                  ; o16 81 /1 iw          [8086]
OR r/m32,imm32                  ; o32 81 /1 id          [386]

OR r/m16,imm8                   ; o16 83 /1 ib          [8086]
OR r/m32,imm8                   ; o32 83 /1 ib          [386]

OR AL,imm8                      ; 0C ib                 [8086]
OR AX,imm16                     ; o16 0D iw             [8086]
OR EAX,imm32                    ; o32 0D id             [386]
```

`OR` performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `POR` (see section A.129) performs the same operation on the 64-bit MMX registers.

A.111. OUT: Output Data to I/O Port

```
OUT imm8,AL                     ; E6 ib                 [8086]
OUT imm8,AX                     ; o16 E7 ib             [8086]
OUT imm8,EAX                    ; o32 E7 ib             [386]
OUT DX,AL                       ; EE                    [8086]
OUT DX,AX                       ; o16 EF                [8086]
OUT DX,EAX                      ; o32 EF                [386]
```

`IN` writes the contents of the given source register to the specified I/O port. The port number may be specified as an immediate value if it is between 0 and 255, and otherwise must be stored in `DX`. See also `IN` (section A.78).

A.112. OUTSB, OUTSW, OUTSD: Output String to I/O Port

```
OUTSB                           ; 6E                    [186]
OUTSW                           ; o16 6F                [186]
OUTSD                           ; o32 6F                [386]
```

OUTSB loads a byte from [DS:SI] or [DS:ESI] and writes it to the I/O port specified in DX. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) SI or ESI.

The register used is SI if the address size is 16 bits, and ESI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

The segment register used to load from [SI] or [ESI] can be overridden by using a segment register name as a prefix (for example, es outsb).

OUTSW and OUTSD work in the same way, but they output a word or a doubleword instead of a byte, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REP prefix may be used to repeat the instruction CX (or ECX – again, the address size chooses which) times.

A.113. PACKSSDW, PACKSSWB, PACKUSWB: Pack Data

```
PACKSSDW mmxreg,r/m64          ; 0F 6B /r              [PENT,MMX]
PACKSSWB mmxreg,r/m64          ; 0F 63 /r              [PENT,MMX]
PACKUSWB mmxreg,r/m64          ; 0F 67 /r              [PENT,MMX]
```

All these instructions start by forming a notional 128-bit word by placing the source (second) operand on the left of the destination (first) operand. `PACKSSDW` then splits this 128-bit word into four doublewords, converts each to a word, and loads them side by side into the destination register; `PACKSSWB` and `PACKUSWB` both split the 128-bit word into eight words, converts each to a byte, and loads *those* side by side into the destination register.

`PACKSSDW` and `PACKSSWB` perform signed saturation when reducing the length of numbers: if the number is too large to fit into the reduced space, they replace it by the largest signed number (`7FFFh` or `7Fh`) that *will* fit, and if it is too small then they replace it by the smallest signed number (`8000h` or `80h`) that will fit. `PACKUSWB` performs unsigned saturation: it treats its input as unsigned, and replaces it by the largest unsigned number that will fit.

A.114. PADDxx: MMX Packed Addition

```
PADDB mmxreg,r/m64              ; 0F FC /r              [PENT,MMX]
PADDW mmxreg,r/m64              ; 0F FD /r              [PENT,MMX]
PADDD mmxreg,r/m64              ; 0F FE /r              [PENT,MMX]

PADDSB mmxreg,r/m64             ; 0F EC /r              [PENT,MMX]
PADDSW mmxreg,r/m64             ; 0F ED /r              [PENT,MMX]

PADDUSB mmxreg,r/m64            ; 0F DC /r              [PENT,MMX]
PADDUSW mmxreg,r/m64            ; 0F DD /r              [PENT,MMX]
```

`PADDxx` all perform packed addition between their two 64-bit operands, storing the result in the destination (first) operand. The `PADDxB` forms treat the 64-bit operands as vectors of eight bytes, and add each byte individually; `PADDxW` treat the operands as vectors of four words; and `PADDD` treats its operands as vectors of two doublewords.

`PADDSB` and `PADDSW` perform signed saturation on the sum of each pair of bytes or words: if the result of an addition is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. `PADDUSB` and `PADDUSW` similarly perform unsigned saturation, clipping to `0FFh` or `0FFFFh` if the result is larger than that.

A.115. PADDSIW: MMX Packed Addition to Implicit Destination

```
PADDSIW mmxreg,r/m64          ; 0F 51 /r              [CYRIX,MMX]
```

PADDSIW, specific to the Cyrix extensions to the MMX instruction set, performs the same function as PADDSW, except that the result is not placed in the register specified by the first operand, but instead in the register whose number differs from the first operand only in the last bit. So PADDSIW MM0,MM2 would put the result in MM1, but PADDSIW MM1,MM2 would put the result in MM0.

A.116. PAND, PANDN: MMX Bitwise AND and AND-NOT

```
PAND mmxreg,r/m64               ; 0F DB /r              [PENT,MMX]
PANDN mmxreg,r/m64              ; 0F DF /r              [PENT,MMX]
```

PAND performs a bitwise AND operation between its two operands (i.e. each bit of the result is 1 if and only if the corresponding bits of the two inputs were both 1), and stores the result in the destination (first) operand.

PANDN performs the same operation, but performs a one's complement operation on the destination (first) operand first.

A.117. PAVEB: MMX Packed Average

```
PAVEB mmxreg,r/m64              ; 0F 50 /r              [CYRIX,MMX]
```

PAVEB, specific to the Cyrix MMX extensions, treats its two operands as vectors of eight unsigned bytes, and calculates the average of the corresponding bytes in the operands. The resulting vector of eight averages is stored in the first operand.

A.118. PCMPxx: MMX Packed Comparison

```
PCMPEQB mmxreg,r/m64          ; 0F 74 /r              [PENT,MMX]
PCMPEQW mmxreg,r/m64          ; 0F 75 /r              [PENT,MMX]
PCMPEQD mmxreg,r/m64          ; 0F 76 /r              [PENT,MMX]

PCMPGTB mmxreg,r/m64          ; 0F 64 /r              [PENT,MMX]
PCMPGTW mmxreg,r/m64          ; 0F 65 /r              [PENT,MMX]
PCMPGTD mmxreg,r/m64          ; 0F 66 /r              [PENT,MMX]
```

The PCMPxx instructions all treat their operands as vectors of bytes, words, or doublewords; corresponding elements of the source and destination are compared, and the corresponding element of the destination (first) operand is set to all zeros or all ones depending on the result of the comparison.

PCMPxxB treats the operands as vectors of eight bytes, PCMPxxW treats them as vectors of four words, and PCMPxxD as two doublewords.

PCMPEQx sets the corresponding element of the destination operand to all ones if the two elements compared are equal; PCMPGTx sets the destination element to all ones if the element of the first (destination) operand is greater (treated as a signed integer) than that of the second (source) operand.

A.119. PDISTIB: MMX Packed Distance and Accumulate with Implied Register

```
PDISTIB mmxreg,mem64          ; 0F 54 /r              [CYRIX,MMX]
```

PDISTIB, specific to the Cyrix MMX extensions, treats its two input operands as vectors of eight unsigned bytes. For each byte position, it finds the absolute difference between the bytes in that position in the two input operands, and adds that value to the byte in the same position in the implied output register. The addition is saturated to an unsigned byte in the same way as PADDUSB.

The implied output register is found in the same way as PADDSIW (section A.115).

Note that PDISTIB cannot take a register as its second source operand.

A.120. PMACHRIW: MMX Packed Multiply and Accumulate with Rounding

```
PMACHRIW mmxreg,mem64          ; 0F 5E /r              [CYRIX,MMX]
```

PMACHRIW acts almost identically to PMULHRIW (section A.123), but instead of *storing* its result in the implied destination register, it *adds* its result, as four packed words, to the implied destination register. No saturation is done: the addition can wrap around.

Note that PMACHRIW cannot take a register as its second source operand.

A.121. PMADDWD: MMX Packed Multiply and Add

```
PMADDWD mmxreg,r/m64          ; 0F F5 /r              [PENT,MMX]
```

PMADDWD treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed doubleword results. The top two of these are added and placed in the top 32 bits of the destination (first) operand; the bottom two are added and placed in the bottom 32 bits.

A.122. PMAGW: MMX Packed Magnitude

```
PMAGW mmxreg,r/m64              ; 0F 52 /r              [CYRIX,MMX]
```

PMAGW, specific to the Cyrix MMX extensions, treats both its operands as vectors of four signed words. It compares the absolute values of the words in corresponding positions, and sets each word of the destination (first) operand to whichever of the two words in that position had the larger absolute value.

A.123. PMULHRW, PMULHRIW: MMX Packed Multiply High with Rounding

```
PMULHRW mmxreg,r/m64          ; 0F 59 /r              [CYRIX,MMX]
PMULHRIW mmxreg,r/m64         ; 0F 5D /r              [CYRIX,MMX]
```

These instructions, specific to the Cyrix MMX extensions, treat their operands as vectors of four signed words. Words in corresponding positions are multiplied, to give a 32-bit value in which bits 30 and 31 are guaranteed equal. Bits 30 to 15 of this value (bit mask `0x7FFF8000`) are taken and stored in the corresponding position of the destination operand, after first rounding the low bit (equivalent to adding `0x4000` before extracting bits 30 to 15).

For `PMULHRW`, the destination operand is the first operand; for `PMULHRIW` the destination operand is implied by the first operand in the manner of `PADDSIW` (section A.115).

A.124. PMULHW, PMULLW: MMX Packed Multiply

```
PMULHW mmxreg,r/m64             ; 0F E5 /r              [PENT,MMX]
PMULLW mmxreg,r/m64             ; 0F D5 /r              [PENT,MMX]
```

PMULxW treats its two inputs as vectors of four signed words. It multiplies corresponding elements of the two operands, giving four signed doubleword results.

PMULHW then stores the top 16 bits of each doubleword in the destination (first) operand; PMULLW stores the bottom 16 bits of each doubleword in the destination operand.

A.125. PMVccZB: MMX Packed Conditional Move

```
PMVZB mmxreg,mem64              ; 0F 58 /r              [CYRIX,MMX]
PMVNZB mmxreg,mem64             ; 0F 5A /r              [CYRIX,MMX]
PMVLZB mmxreg,mem64             ; 0F 5B /r              [CYRIX,MMX]
PMVGEZB mmxreg,mem64            ; 0F 5C /r              [CYRIX,MMX]
```

These instructions, specific to the Cyrix MMX extensions, perform parallel conditional moves. The two input operands are treated as vectors of eight bytes. Each byte of the destination (first) operand is either written from the corresponding byte of the source (second) operand, or left alone, depending on the value of the byte in the *implied* operand (specified in the same way as PADDSIW, in section A.115).

PMVZB performs each move if the corresponding byte in the implied operand is zero. PMVNZB moves if the byte is non-zero. PMVLZB moves if the byte is less than zero, and PMVGEZB moves if the byte is greater than or equal to zero.

Note that these instructions cannot take a register as their second source operand.

A.126. `POP`: Pop Data from Stack

```
POP reg16                       ; o16 58+r              [8086]
POP reg32                       ; o32 58+r              [386]

POP r/m16                       ; o16 8F /0             [8086]
POP r/m32                       ; o32 8F /0             [386]

POP CS                          ; 0F                    [8086,UNDOC]
POP DS                          ; 1F                    [8086]
POP ES                          ; 07                    [8086]
POP SS                          ; 17                    [8086]
POP FS                          ; 0F A1                 [386]
POP GS                          ; 0F A9                 [386]
```

`POP` loads a value from the stack (from `[SS:SP]` or `[SS:ESP]`) and then increments the stack pointer.

The address-size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.

The operand-size attribute of the instruction determines whether the stack pointer is incremented by 2 or 4: this means that segment register pops in `BITS 32` mode will pop 4 bytes off the stack and discard the upper two of them. If you need to override that, you can use an `o16` or `o32` prefix.

The above opcode listings give two forms for general-purpose register pop instructions: for example, `POP BX` has the two forms `5B` and `8F C3`. NASM will always generate the shorter form when given `POP BX`. NDISASM will disassemble both.

`POP CS` is not a documented instruction, and is not supported on any processor above the 8086 (since they use `0Fh` as an opcode prefix for instruction set extensions). However, at least some 8086 processors do support it, and so NASM generates it for completeness.

A.127. `POPAx`: Pop All General-Purpose Registers

```
POPA                            ; 61                    [186]
POPAW                           ; o16 61                [186]
POPAD                           ; o32 61                [386]
```

`POPAW` pops a word from the stack into each of, successively, `DI`, `SI`, `BP`, nothing (it discards a word from the stack which was a placeholder for `SP`), `BX`, `DX`, `CX` and `AX`. It is intended to reverse the operation of `PUSHAW` (see section A.135), but it ignores the value for `SP` that was pushed on the stack by `PUSHAW`.

`POPAD` pops twice as much data, and places the results in `EDI`, `ESI`, `EBP`, nothing (placeholder for `ESP`), `EBX`, `EDX`, `ECX` and `EAX`. It reverses the operation of `PUSHAD`.

`POPA` is an alias mnemonic for either `POPAW` or `POPAD`, depending on the current `BITS` setting.

Note that the registers are popped in reverse order of their numeric values in opcodes (see section A.2.1).

A.128. `POPFx`: Pop Flags Register

```
POPF                            ; 9D                    [186]
POPFW                           ; o16 9D                [186]
POPFD                           ; o32 9D                [386]
```

`POPFW` pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386). `POPFD` pops a doubleword and stores it in the entire flags register.

`POPF` is an alias mnemonic for either `POPFW` or `POPFD`, depending on the current `BITS` setting.

See also `PUSHF` (section A.136).

A.129. POR: MMX Bitwise OR

```
POR mmxreg,r/m64                ; 0F EB /r              [PENT,MMX]
```

POR performs a bitwise OR operation between its two operands (i.e. each bit of the result is 1 if and only if at least one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.130. PSLLx, PSRLx, PSRAx: MMX Bit Shifts

```
PSLLW mmxreg,r/m64          ; 0F F1 /r              [PENT,MMX]
PSLLW mmxreg,imm8           ; 0F 71 /6 ib           [PENT,MMX]

PSLLD mmxreg,r/m64          ; 0F F2 /r              [PENT,MMX]
PSLLD mmxreg,imm8           ; 0F 72 /6 ib           [PENT,MMX]

PSLLQ mmxreg,r/m64          ; 0F F3 /r              [PENT,MMX]
PSLLQ mmxreg,imm8           ; 0F 73 /6 ib           [PENT,MMX]

PSRAW mmxreg,r/m64          ; 0F E1 /r              [PENT,MMX]
PSRAW mmxreg,imm8           ; 0F 71 /4 ib           [PENT,MMX]

PSRAD mmxreg,r/m64          ; 0F E2 /r              [PENT,MMX]
PSRAD mmxreg,imm8           ; 0F 72 /4 ib           [PENT,MMX]

PSRLW mmxreg,r/m64          ; 0F D1 /r              [PENT,MMX]
PSRLW mmxreg,imm8           ; 0F 71 /2 ib           [PENT,MMX]

PSRLD mmxreg,r/m64          ; 0F D2 /r              [PENT,MMX]
PSRLD mmxreg,imm8           ; 0F 72 /2 ib           [PENT,MMX]

PSRLQ mmxreg,r/m64          ; 0F D3 /r              [PENT,MMX]
PSRLQ mmxreg,imm8           ; 0F 73 /2 ib           [PENT,MMX]
```

`PSxxQ` perform simple bit shifts on the 64-bit MMX registers: the destination (first) operand is shifted left or right by the number of bits given in the source (second) operand, and the vacated bits are filled in with zeros (for a logical shift) or copies of the original sign bit (for an arithmetic right shift).

`PSxxW` and `PSxxD` perform packed bit shifts: the destination operand is treated as a vector of four words or two doublewords, and each element is shifted individually, so bits shifted out of one element do not interfere with empty bits coming into the next.

`PSLLx` and `PSRLx` perform logical shifts: the vacated bits at one end of the shifted number are filled with zeros. `PSRAx` performs an arithmetic right shift: the vacated bits at the top of the shifted number are filled with copies of the original top (sign) bit.

A.131. PSUBxx: MMX Packed Subtraction

```
PSUBB mmxreg,r/m64            ; 0F F8 /r              [PENT,MMX]
PSUBW mmxreg,r/m64            ; 0F F9 /r              [PENT,MMX]
PSUBD mmxreg,r/m64            ; 0F FA /r              [PENT,MMX]

PSUBSB mmxreg,r/m64           ; 0F E8 /r              [PENT,MMX]
PSUBSW mmxreg,r/m64           ; 0F E9 /r              [PENT,MMX]

PSUBUSB mmxreg,r/m64          ; 0F D8 /r              [PENT,MMX]
PSUBUSW mmxreg,r/m64          ; 0F D9 /r              [PENT,MMX]
```

`PSUBxx` all perform packed subtraction between their two 64-bit operands, storing the result in the destination (first) operand. The `PSUBxB` forms treat the 64-bit operands as vectors of eight bytes, and subtract each byte individually; `PSUBxW` treat the operands as vectors of four words; and `PSUBD` treats its operands as vectors of two doublewords.

In all cases, the elements of the operand on the right are subtracted from the corresponding elements of the operand on the left, not the other way round.

`PSUBSB` and `PSUBSW` perform signed saturation on the sum of each pair of bytes or words: if the result of a subtraction is too large or too small to fit into a signed byte or word result, it is clipped (saturated) to the largest or smallest value which *will* fit. `PSUBUSB` and `PSUBUSW` similarly perform unsigned saturation, clipping to `0FFh` or `0FFFFh` if the result is larger than that.

A.132. PSUBSIW: MMX Packed Subtract with Saturation to Implied Destination

```
PSUBSIW mmxreg,r/m64          ; 0F 55 /r              [CYRIX,MMX]
```

PSUBSIW, specific to the Cyrix extensions to the MMX instruction set, performs the same function as PSUBSW, except that the result is not placed in the register specified by the first operand, but instead in the implied destination register, specified as for PADDSIW (section A.115).

A.133. PUNPCKxxx: Unpack Data

```
PUNPCKHBW mmxreg,r/m64        ; 0F 68 /r              [PENT,MMX]
PUNPCKHWD mmxreg,r/m64        ; 0F 69 /r              [PENT,MMX]
PUNPCKHDQ mmxreg,r/m64        ; 0F 6A /r              [PENT,MMX]

PUNPCKLBW mmxreg,r/m64        ; 0F 60 /r              [PENT,MMX]
PUNPCKLWD mmxreg,r/m64        ; 0F 61 /r              [PENT,MMX]
PUNPCKLDQ mmxreg,r/m64        ; 0F 62 /r              [PENT,MMX]
```

PUNPCKxx all treat their operands as vectors, and produce a new vector generated by interleaving elements from the two inputs. The PUNPCKHxx instructions start by throwing away the bottom half of each input operand, and the PUNPCKLxx instructions throw away the top half.

The remaining elements, totalling 64 bits, are then interleaved into the destination, alternating elements from the second (source) operand and the first (destination) operand: so the leftmost element in the result always comes from the second operand, and the rightmost from the destination.

PUNPCKxBW works a byte at a time, PUNPCKxWD a word at a time, and PUNPCKxDQ a doubleword at a time.

So, for example, if the first operand held 0x7A6A5A4A3A2A1A0A and the second held 0x7B6B5B4B3B2B1B0B, then:

- PUNPCKHBW would return 0x7B7A6B6A5B5A4B4A.
- PUNPCKHWD would return 0x7B6B7A6A5B4B5A4A.
- PUNPCKHDQ would return 0x7B6B5B4B7A6A5A4A.
- PUNPCKLBW would return 0x3B3A2B2A1B1A0B0A.
- PUNPCKLWD would return 0x3B2B3A2A1B0B1A0A.
- PUNPCKLDQ would return 0x3B2B1B0B3A2A1A0A.

A.134. PUSH: Push Data on Stack

```
PUSH reg16                      ; o16 50+r              [8086]
PUSH reg32                      ; o32 50+r              [386]

PUSH r/m16                      ; o16 FF /6             [8086]
PUSH r/m32                      ; o32 FF /6             [386]

PUSH CS                         ; 0E                    [8086]
PUSH DS                         ; 1E                    [8086]
PUSH ES                         ; 06                    [8086]
PUSH SS                         ; 16                    [8086]
PUSH FS                         ; 0F A0                 [386]
PUSH GS                         ; 0F A8                 [386]

PUSH imm8                       ; 6A ib                 [286]
PUSH imm16                      ; o16 68 iw             [286]
PUSH imm32                      ; o32 68 id             [386]
```

`PUSH` decrements the stack pointer (`SP` or `ESP`) by 2 or 4, and then stores the given value at `[SS:SP]` or `[SS:ESP]`.

The address-size attribute of the instruction determines whether `SP` or `ESP` is used as the stack pointer: to deliberately override the default given by the `BITS` setting, you can use an `a16` or `a32` prefix.

The operand-size attribute of the instruction determines whether the stack pointer is decremented by 2 or 4: this means that segment register pushes in `BITS 32` mode will push 4 bytes on the stack, of which the upper two are undefined. If you need to override that, you can use an `o16` or `o32` prefix.

The above opcode listings give two forms for general-purpose register push instructions: for example, `PUSH BX` has the two forms `53` and `FF F3`. NASM will always generate the shorter form when given `PUSH BX`. NDISASM will disassemble both.

Unlike the undocumented and barely supported `POP CS`, `PUSH CS` is a perfectly valid and sensible instruction, supported on all processors.

The instruction `PUSH SP` may be used to distinguish an 8086 from later processors: on an 8086, the value of `SP` stored is the value it has *after* the push instruction, whereas on later processors it is the value *before* the push instruction.

A.135. `PUSHAx`: Push All General-Purpose Registers

```
PUSHA                           ; 60                    [186]
PUSHAD                          ; o32 60                [386]
PUSHAW                          ; o16 60                [186]
```

`PUSHAW` pushes, in succession, `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI` and `DI` on the stack, decrementing the stack pointer by a total of 16.

`PUSHAD` pushes, in succession, `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` and `EDI` on the stack, decrementing the stack pointer by a total of 32.

In both cases, the value of `SP` or `ESP` pushed is its *original* value, as it had before the instruction was executed.

`PUSHA` is an alias mnemonic for either `PUSHAW` or `PUSHAD`, depending on the current `BITS` setting.

Note that the registers are pushed in order of their numeric values in opcodes (see section A.2.1).

See also `POPA` (section A.127).

A.136. `PUSHFx`: Push Flags Register

```
PUSHF                           ; 9C                    [186]
PUSHFD                          ; o32 9C                [386]
PUSHFW                          ; o16 9C                [186]
```

`PUSHFW` pops a word from the stack and stores it in the bottom 16 bits of the flags register (or the whole flags register, on processors below a 386). `PUSHFD` pops a doubleword and stores it in the entire flags register.

`PUSHF` is an alias mnemonic for either `PUSHFW` or `PUSHFD`, depending on the current `BITS` setting.

See also `POPF` (section A.128).

A.137. PXOR: MMX Bitwise XOR

```
PXOR mmxreg,r/m64              ; 0F EF /r              [PENT,MMX]
```

PXOR performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

A.138. RCL, RCR: Bitwise Rotate through Carry Bit

```
RCL r/m8,1                      ; D0 /2                 [8086]
RCL r/m8,CL                     ; D2 /2                 [8086]
RCL r/m8,imm8                   ; C0 /2 ib              [286]
RCL r/m16,1                     ; o16 D1 /2             [8086]
RCL r/m16,CL                    ; o16 D3 /2             [8086]
RCL r/m16,imm8                  ; o16 C1 /2 ib          [286]
RCL r/m32,1                     ; o32 D1 /2             [386]
RCL r/m32,CL                    ; o32 D3 /2             [386]
RCL r/m32,imm8                  ; o32 C1 /2 ib          [386]

RCR r/m8,1                      ; D0 /3                 [8086]
RCR r/m8,CL                     ; D2 /3                 [8086]
RCR r/m8,imm8                   ; C0 /3 ib              [286]
RCR r/m16,1                     ; o16 D1 /3             [8086]
RCR r/m16,CL                    ; o16 D3 /3             [8086]
RCR r/m16,imm8                  ; o16 C1 /3 ib          [286]
RCR r/m32,1                     ; o32 D1 /3             [386]
RCR r/m32,CL                    ; o32 D3 /3             [386]
RCR r/m32,imm8                  ; o32 C1 /3 ib          [386]
```

`RCL` and `RCR` perform a 9-bit, 17-bit or 33-bit bitwise rotation operation, involving the given source/destination (first) operand and the carry bit. Thus, for example, in the operation `RCR AL,1`, a 9-bit rotation is performed in which `AL` is shifted left by 1, the top bit of `AL` moves into the carry flag, and the original value of the carry flag is placed in the low bit of `AL`.

The number of bits to rotate by is given by the second operand. Only the bottom five bits of the rotation count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `RCL foo,1` by using a `BYTE` prefix: `RCL foo,BYTE 1`. Similarly with `RCR`.

A.139. RDMSR: Read Model-Specific Registers

```
RDMSR                           ; 0F 32                 [PENT]
```

RDMSR reads the processor Model-Specific Register (MSR) whose index is stored in ECX, and stores the result in EDX:EAX. See also WRMSR (section A.165).

A.140. RDPMC: Read Performance-Monitoring Counters

```
RDPMC                           ; 0F 33                 [P6]
```

RDPMC reads the processor performance-monitoring counter whose index is stored in ECX, and stores the result in EDX:EAX.

A.141. `RDTSC`: Read Time-Stamp Counter

```
RDTSC                           ; 0F 31                 [PENT]
```

`RDTSC` reads the processor's time-stamp counter into `EDX:EAX`.

A.142. `RET`, `RETF`, `RETN`: Return from Procedure Call

```
RET                             ; C3                    [8086]
RET imm16                       ; C2 iw                 [8086]

RETF                            ; CB                    [8086]
RETF imm16                      ; CA iw                 [8086]

RETN                            ; C3                    [8086]
RETN imm16                      ; C2 iw                 [8086]
```

`RET`, and its exact synonym `RETN`, pop `IP` or `EIP` from the stack and transfer control to the new address. Optionally, if a numeric second operand is provided, they increment the stack pointer by a further `imm16` bytes after popping the return address.

`RETF` executes a far return: after popping `IP`/`EIP`, it then pops `CS`, and *then* increments the stack pointer by the optional argument if present.

A.143. ROL, ROR: Bitwise Rotate

```
ROL r/m8,1                      ; D0 /0                 [8086]
ROL r/m8,CL                     ; D2 /0                 [8086]
ROL r/m8,imm8                   ; C0 /0 ib              [286]
ROL r/m16,1                     ; o16 D1 /0             [8086]
ROL r/m16,CL                    ; o16 D3 /0             [8086]
ROL r/m16,imm8                  ; o16 C1 /0 ib          [286]
ROL r/m32,1                     ; o32 D1 /0             [386]
ROL r/m32,CL                    ; o32 D3 /0             [386]
ROL r/m32,imm8                  ; o32 C1 /0 ib          [386]

ROR r/m8,1                      ; D0 /1                 [8086]
ROR r/m8,CL                     ; D2 /1                 [8086]
ROR r/m8,imm8                   ; C0 /1 ib              [286]
ROR r/m16,1                     ; o16 D1 /1             [8086]
ROR r/m16,CL                    ; o16 D3 /1             [8086]
ROR r/m16,imm8                  ; o16 C1 /1 ib          [286]
ROR r/m32,1                     ; o32 D1 /1             [386]
ROR r/m32,CL                    ; o32 D3 /1             [386]
ROR r/m32,imm8                  ; o32 C1 /1 ib          [386]
```

`ROL` and `ROR` perform a bitwise rotation operation on the given source/destination (first) operand. Thus, for example, in the operation `ROR AL,1`, an 8-bit rotation is performed in which `AL` is shifted left by 1 and the original top bit of `AL` moves round into the low bit.

The number of bits to rotate by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the rotation count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `ROL foo,1` by using a `BYTE` prefix: `ROL foo,BYTE 1`. Similarly with `ROR`.

A.144. `RSM`: Resume from System-Management Mode

```
RSM                             ; 0F AA                 [PENT]
```

`RSM` returns the processor to its normal operating mode when it was in System-Management Mode.

A.145. `SAHF`: Store AH to Flags

```
SAHF                            ; 9E                    [8086]
```

`SAHF` sets the low byte of the flags word according to the contents of the `AH` register. See also `LAHF` (section A.90).

A.146. SAL, SAR: Bitwise Arithmetic Shifts

```
SAL r/m8,1                      ; D0 /4                 [8086]
SAL r/m8,CL                     ; D2 /4                 [8086]
SAL r/m8,imm8                   ; C0 /4 ib              [286]
SAL r/m16,1                     ; o16 D1 /4             [8086]
SAL r/m16,CL                    ; o16 D3 /4             [8086]
SAL r/m16,imm8                  ; o16 C1 /4 ib          [286]
SAL r/m32,1                     ; o32 D1 /4             [386]
SAL r/m32,CL                    ; o32 D3 /4             [386]
SAL r/m32,imm8                  ; o32 C1 /4 ib          [386]

SAR r/m8,1                      ; D0 /0                 [8086]
SAR r/m8,CL                     ; D2 /0                 [8086]
SAR r/m8,imm8                   ; C0 /0 ib              [286]
SAR r/m16,1                     ; o16 D1 /0             [8086]
SAR r/m16,CL                    ; o16 D3 /0             [8086]
SAR r/m16,imm8                  ; o16 C1 /0 ib          [286]
SAR r/m32,1                     ; o32 D1 /0             [386]
SAR r/m32,CL                    ; o32 D3 /0             [386]
SAR r/m32,imm8                  ; o32 C1 /0 ib          [386]
```

`SAL` and `SAR` perform an arithmetic shift operation on the given source/destination (first) operand. The vacated bits are filled with zero for `SAL`, and with copies of the original high bit of the source operand for `SAR`.

`SAL` is a synonym for `SHL` (see section A.152). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as `SHL`.

The number of bits to shift by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SAL foo,1` by using a `BYTE` prefix: `SAL foo,BYTE 1`. Similarly with `SAR`.

A.147. `SALC`: Set AL from Carry Flag

```
SALC                            ; D6                        [8086,UNDOC]
```

`SALC` is an early undocumented instruction similar in concept to `SETcc` (section A.150). Its function is to set `AL` to zero if the carry flag is clear, or to `0xFF` if it is set.

A.148. SBB: Subtract with Borrow

```
SBB r/m8,reg8                   ; 18 /r                 [8086]
SBB r/m16,reg16                 ; o16 19 /r             [8086]
SBB r/m32,reg32                 ; o32 19 /r             [386]

SBB reg8,r/m8                   ; 1A /r                 [8086]
SBB reg16,r/m16                 ; o16 1B /r             [8086]
SBB reg32,r/m32                 ; o32 1B /r             [386]

SBB r/m8,imm8                   ; 80 /3 ib              [8086]
SBB r/m16,imm16                 ; o16 81 /3 iw          [8086]
SBB r/m32,imm32                 ; o32 81 /3 id          [386]

SBB r/m16,imm8                  ; o16 83 /3 ib          [8086]
SBB r/m32,imm8                  ; o32 83 /3 ib          [8086]

SBB AL,imm8                     ; 1C ib                 [8086]
SBB AX,imm16                    ; o16 1D iw             [8086]
SBB EAX,imm32                   ; o32 1D id             [386]
```

SBB performs integer subtraction: it subtracts its second operand, plus the value of the carry flag, from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent SBB instruction.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

To subtract one number from another without also subtracting the contents of the carry flag, use SUB (section A.159).

A.149. SCASB, SCASW, SCASD: Scan String

```
SCASB                           ; AE                    [8086]
SCASW                           ; o16 AF                [8086]
SCASD                           ; o32 AF                [386]
```

SCASB compares the byte in AL with the byte at [ES:DI] or [ES:EDI], and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) DI (or EDI).

The register used is DI if the address size is 16 bits, and EDI if it is 32 bits. If you need to use an address size not equal to the current BITS setting, you can use an explicit a16 or a32 prefix.

Segment override prefixes have no effect for this instruction: the use of ES for the load from [DI] or [EDI] cannot be overridden.

SCASW and SCASD work in the same way, but they compare a word to AX or a doubleword to EAX instead of a byte to AL, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The REPE and REPNE prefixes (equivalently, REPZ and REPNZ) may be used to repeat the instruction up to CX (or ECX – again, the address size chooses which) times until the first unequal or equal byte is found.

A.150. `SETcc`: Set Register from Condition

```
SETcc r/m8                      ; 0F 90+cc /2           [386]
```

`SETcc` sets the given 8-bit operand to zero if its condition is not satisfied, and to 1 if it is.

A.151. SGDT, SIDT, SLDT: Store Descriptor Table Pointers

```
SGDT mem                        ; 0F 01 /0              [286,PRIV]
SIDT mem                        ; 0F 01 /1              [286,PRIV]
SLDT r/m16                      ; 0F 00 /0              [286,PRIV]
```

SGDT and SIDT both take a 6-byte memory area as an operand: they store the contents of the GDTR (global descriptor table register) or IDTR (interrupt descriptor table register) into that area as a 32-bit linear address and a 16-bit size limit from that area (in that order). These are the only instructions which directly use *linear* addresses, rather than segment/offset pairs.

SLDT stores the segment selector corresponding to the LDT (local descriptor table) into the given operand.

See also LGDT, LIDT and LLDT (section A.95).

A.152. SHL, SHR: Bitwise Logical Shifts

```
SHL r/m8,1                      ; D0 /4                 [8086]
SHL r/m8,CL                     ; D2 /4                 [8086]
SHL r/m8,imm8                   ; C0 /4 ib              [286]
SHL r/m16,1                     ; o16 D1 /4             [8086]
SHL r/m16,CL                    ; o16 D3 /4             [8086]
SHL r/m16,imm8                  ; o16 C1 /4 ib          [286]
SHL r/m32,1                     ; o32 D1 /4             [386]
SHL r/m32,CL                    ; o32 D3 /4             [386]
SHL r/m32,imm8                  ; o32 C1 /4 ib          [386]

SHR r/m8,1                      ; D0 /5                 [8086]
SHR r/m8,CL                     ; D2 /5                 [8086]
SHR r/m8,imm8                   ; C0 /5 ib              [286]
SHR r/m16,1                     ; o16 D1 /5             [8086]
SHR r/m16,CL                    ; o16 D3 /5             [8086]
SHR r/m16,imm8                  ; o16 C1 /5 ib          [286]
SHR r/m32,1                     ; o32 D1 /5             [386]
SHR r/m32,CL                    ; o32 D3 /5             [386]
SHR r/m32,imm8                  ; o32 C1 /5 ib          [386]
```

`SHL` and `SHR` perform a logical shift operation on the given source/destination (first) operand. The vacated bits are filled with zero.

A synonym for `SHL` is `SAL` (see section A.146). NASM will assemble either one to the same code, but NDISASM will always disassemble that code as `SHL`.

The number of bits to shift by is given by the second operand. Only the bottom 3, 4 or 5 bits (depending on the source operand size) of the shift count are considered by processors above the 8086.

You can force the longer (286 and upwards, beginning with a `C1` byte) form of `SHL foo,1` by using a `BYTE` prefix: `SHL foo,BYTE 1`. Similarly with `SHR`.

A.153. SHLD, SHRD: Bitwise Double-Precision Shifts

```
SHLD r/m16,reg16,imm8          ; o16 0F A4 /r ib      [386]
SHLD r/m16,reg32,imm8          ; o32 0F A4 /r ib      [386]
SHLD r/m16,reg16,CL            ; o16 0F A5 /r         [386]
SHLD r/m16,reg32,CL            ; o32 0F A5 /r         [386]

SHRD r/m16,reg16,imm8          ; o16 0F AC /r ib      [386]
SHRD r/m32,reg32,imm8          ; o32 0F AC /r ib      [386]
SHRD r/m16,reg16,CL            ; o16 0F AD /r         [386]
SHRD r/m32,reg32,CL            ; o32 0F AD /r         [386]
```

`SHLD` performs a double-precision left shift. It notionally places its second operand to the right of its first, then shifts the entire bit string thus generated to the left by a number of bits specified in the third operand. It then updates only the *first* operand according to the result of this. The second operand is not modified.

`SHRD` performs the corresponding right shift: it notionally places the second operand to the *left* of the first, shifts the whole bit string right, and updates only the first operand.

For example, if `EAX` holds `0x01234567` and `EBX` holds `0x89ABCDEF`, then the instruction `SHLD EAX,EBX,4` would update `EAX` to hold `0x12345678`. Under the same conditions, `SHRD EAX,EBX,4` would update `EAX` to hold `0xF0123456`.

The number of bits to shift by is given by the third operand. Only the bottom 5 bits of the shift count are considered.

A.154. `SMI`: System Management Interrupt

```
SMI                             ; F1                        [386,UNDOC]
```

This is an opcode apparently supported by some AMD processors (which is why it can generate the same opcode as `INT1`), and places the machine into system-management mode, a special debugging mode.

A.155. SMSW: Store Machine Status Word

```
SMSW r/m16                      ; 0F 01 /4              [286,PRIV]
```

SMSW stores the bottom half of the CR0 control register (or the Machine Status Word, on 286 processors) into the destination operand. See also LMSW (section A.96).

A.156. `STC`, `STD`, `STI`: Set Flags

```
STC                             ; F9                    [8086]
STD                             ; FD                    [8086]
STI                             ; FB                    [8086]
```

These instructions set various flags. `STC` sets the carry flag; `STD` sets the direction flag; and `STI` sets the interrupt flag (thus enabling interrupts).

To clear the carry, direction, or interrupt flags, use the `CLC`, `CLD` and `CLI` instructions (section A.15). To invert the carry flag, use `CMC` (section A.16).

A.157. `STOSB`, `STOSW`, `STOSD`: Store Byte to String

```
STOSB                           ; AA                    [8086]
STOSW                           ; o16 AB                [8086]
STOSD                           ; o32 AB                [386]
```

`STOSB` stores the byte in `AL` at `[ES:DI]` or `[ES:EDI]`, and sets the flags accordingly. It then increments or decrements (depending on the direction flag: increments if the flag is clear, decrements if it is set) `DI` (or `EDI`).

The register used is `DI` if the address size is 16 bits, and `EDI` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

Segment override prefixes have no effect for this instruction: the use of `ES` for the store to `[DI]` or `[EDI]` cannot be overridden.

`STOSW` and `STOSD` work in the same way, but they store the word in `AX` or the doubleword in `EAX` instead of the byte in `AL`, and increment or decrement the addressing registers by 2 or 4 instead of 1.

The `REP` prefix may be used to repeat the instruction `CX` (or `ECX` – again, the address size chooses which) times.

A.158. `STR`: Store Task Register

```
STR r/m16                       ; 0F 00 /1              [286,PRIV]
```

`STR` stores the segment selector corresponding to the contents of the Task Register into its operand.

A.159. SUB: Subtract Integers

```
SUB r/m8,reg8                   ; 28 /r                 [8086]
SUB r/m16,reg16                 ; o16 29 /r             [8086]
SUB r/m32,reg32                 ; o32 29 /r             [386]

SUB reg8,r/m8                   ; 2A /r                 [8086]
SUB reg16,r/m16                 ; o16 2B /r             [8086]
SUB reg32,r/m32                 ; o32 2B /r             [386]

SUB r/m8,imm8                   ; 80 /5 ib              [8086]
SUB r/m16,imm16                 ; o16 81 /5 iw          [8086]
SUB r/m32,imm32                 ; o32 81 /5 id          [386]

SUB r/m16,imm8                  ; o16 83 /5 ib          [8086]
SUB r/m32,imm8                  ; o32 83 /5 ib          [386]

SUB AL,imm8                     ; 2C ib                 [8086]
SUB AX,imm16                    ; o16 2D iw             [8086]
SUB EAX,imm32                   ; o32 2D id             [386]
```

SUB performs integer subtraction: it subtracts its second operand from its first, and leaves the result in its destination (first) operand. The flags are set according to the result of the operation: in particular, the carry flag is affected and can be used by a subsequent SBB instruction (section A.148).

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the BYTE qualifier is necessary to force NASM to generate this form of the instruction.

A.160. `TEST`: Test Bits (notional bitwise AND)

```
TEST r/m8,reg8                  ; 84 /r                 [8086]
TEST r/m16,reg16                ; o16 85 /r             [8086]
TEST r/m32,reg32                ; o32 85 /r             [386]

TEST r/m8,imm8                  ; F6 /7 ib              [8086]
TEST r/m16,imm16                ; o16 F7 /7 iw          [8086]
TEST r/m32,imm32                ; o32 F7 /7 id          [386]

TEST AL,imm8                    ; A8 ib                 [8086]
TEST AX,imm16                   ; o16 A9 iw             [8086]
TEST EAX,imm32                  ; o32 A9 id             [386]
```

`TEST` performs a `mental' bitwise AND of its two operands, and affects the flags as if the operation had taken place, but does not store the result of the operation anywhere.

A.161. UMOV: User Move Data

```
UMOV r/m8,reg8                ; 0F 10 /r              [386,UNDOC]
UMOV r/m16,reg16              ; o16 0F 11 /r          [386,UNDOC]
UMOV r/m32,reg32              ; o32 0F 11 /r          [386,UNDOC]

UMOV reg8,r/m8                ; 0F 12 /r              [386,UNDOC]
UMOV reg16,r/m16              ; o16 0F 13 /r          [386,UNDOC]
UMOV reg32,r/m32              ; o32 0F 13 /r          [386,UNDOC]
```

This undocumented instruction is used by in-circuit emulators to access user memory (as opposed to host memory). It is used just like an ordinary memory/register or register/register MOV instruction, but accesses user space.

A.162. VERR, VERW: Verify Segment Readability/Writability

```
VERR r/m16                      ; 0F 00 /4              [286,PRIV]
VERW r/m16                      ; 0F 00 /5              [286,PRIV]
```

VERR sets the zero flag if the segment specified by the selector in its operand can be read from at the current privilege level. VERW sets the zero flag if the segment can be written.

A.163. `WAIT`: Wait for Floating-Point Processor

```
WAIT                            ; 9B                    [8086]
```

`WAIT`, on 8086 systems with a separate 8087 FPU, waits for the FPU to have finished any operation it is engaged in before continuing main processor operations, so that (for example) an FPU store to main memory can be guaranteed to have completed before the CPU tries to read the result back out.

On higher processors, `WAIT` is unnecessary for this purpose, and it has the alternative purpose of ensuring that any pending unmasked FPU exceptions have happened before execution continues.

A.164. `WBINVD`: Write Back and Invalidate Cache

```
WBINVD                          ; 0F 09                 [486]
```

`WBINVD` invalidates and empties the processor's internal caches, and causes the processor to instruct external caches to do the same. It writes the contents of the caches back to memory first, so no data is lost. To flush the caches quickly without bothering to write the data back first, use `INVD` (section A.84).

A.165. `WRMSR`: Write Model-Specific Registers

```
WRMSR                           ; 0F 30                 [PENT]
```

`WRMSR` writes the value in `EDX:EAX` to the processor Model-Specific Register (MSR) whose index is stored in `ECX`. See also `RDMSR` (section A.139).

A.166. XADD: Exchange and Add

```
XADD r/m8,reg8                  ; 0F C0 /r              [486]
XADD r/m16,reg16                ; o16 0F C1 /r          [486]
XADD r/m32,reg32                ; o32 0F C1 /r          [486]
```

XADD exchanges the values in its two operands, and then adds them together and writes the result into the destination (first) operand. This instruction can be used with a LOCK prefix for multi-processor synchronisation purposes.

A.167. XBTS: Extract Bit String

```
XBTS reg16,r/m16              ; o16 0F A6 /r          [386,UNDOC]
XBTS reg32,r/m32              ; o32 0F A6 /r          [386,UNDOC]
```

No clear documentation seems to be available for this instruction: the best I've been able to find reads `Takes a string of bits from the first operand and puts them in the second operand'. It is present only in early 386 processors, and conflicts with the opcodes for `CMPXCHG486`. NASM supports it only for completeness. Its counterpart is `IBTS` (see section A.75).

A.168. XCHG: Exchange

```
XCHG reg8,r/m8                  ; 86 /r                 [8086]
XCHG reg16,r/m8                 ; o16 87 /r             [8086]
XCHG reg32,r/m32                ; o32 87 /r             [386]

XCHG r/m8,reg8                  ; 86 /r                 [8086]
XCHG r/m16,reg16                ; o16 87 /r             [8086]
XCHG r/m32,reg32                ; o32 87 /r             [386]

XCHG AX,reg16                   ; o16 90+r              [8086]
XCHG EAX,reg32                  ; o32 90+r              [386]
XCHG reg16,AX                   ; o16 90+r              [8086]
XCHG reg32,EAX                  ; o32 90+r              [386]
```

`XCHG` exchanges the values in its two operands. It can be used with a `LOCK` prefix for purposes of multi-processor synchronisation.

`XCHG AX,AX` or `XCHG EAX,EAX` (depending on the `BITS` setting) generates the opcode `90h`, and so is a synonym for `NOP` (section A.109).

A.169. `XLATB`: Translate Byte in Lookup Table

```
XLATB                           ; D7                    [8086]
```

`XLATB` adds the value in `AL`, treated as an unsigned byte, to `BX` or `EBX`, and loads the byte from the resulting address (in the segment specified by `DS`) back into `AL`.

The base register used is `BX` if the address size is 16 bits, and `EBX` if it is 32 bits. If you need to use an address size not equal to the current `BITS` setting, you can use an explicit `a16` or `a32` prefix.

The segment register used to load from `[BX+AL]` or `[EBX+AL]` can be overridden by using a segment register name as a prefix (for example, `es xlatb`).

A.170. XOR: Bitwise Exclusive OR

```
XOR r/m8,reg8                   ; 30 /r                 [8086]
XOR r/m16,reg16                 ; o16 31 /r             [8086]
XOR r/m32,reg32                 ; o32 31 /r             [386]

XOR reg8,r/m8                   ; 32 /r                 [8086]
XOR reg16,r/m16                 ; o16 33 /r             [8086]
XOR reg32,r/m32                 ; o32 33 /r             [386]

XOR r/m8,imm8                   ; 80 /6 ib              [8086]
XOR r/m16,imm16                 ; o16 81 /6 iw          [8086]
XOR r/m32,imm32                 ; o32 81 /6 id          [386]

XOR r/m16,imm8                  ; o16 83 /6 ib          [8086]
XOR r/m32,imm8                  ; o32 83 /6 ib          [386]

XOR AL,imm8                     ; 34 ib                 [8086]
XOR AX,imm16                    ; o16 35 iw             [8086]
XOR EAX,imm32                   ; o32 35 id             [386]
```

`XOR` performs a bitwise XOR operation between its two operands (i.e. each bit of the result is 1 if and only if exactly one of the corresponding bits of the two inputs was 1), and stores the result in the destination (first) operand.

In the forms with an 8-bit immediate second operand and a longer first operand, the second operand is considered to be signed, and is sign-extended to the length of the first operand. In these cases, the `BYTE` qualifier is necessary to force NASM to generate this form of the instruction.

The MMX instruction `PXOR` (see section A.137) performs the same operation on the 64-bit MMX registers.