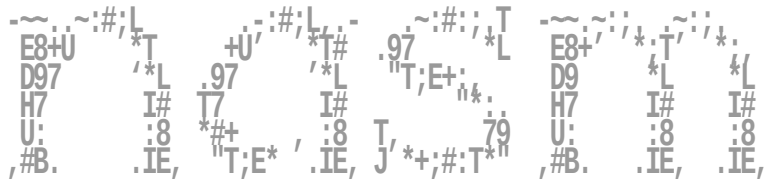
NASM — The Netwide Assembler

version 2.09rc2–20100708



© 1996–2010 The NASM Development Team — All Rights Reserved

This document is redistributable under the license given in the file "LICENSE" distributed in the NASM archive.

This release is dedicated to the memory of Charles A. Crayne. We miss you, Chuck.

Contents

Chapter 1: Introduction 14
1.1 What Is NASM? 14
1.1.1 Why Yet Another Assembler? 14
1.1.2 License Conditions 14
1.2 Contact Information 15
1.3 Installation. 15
1.3.1 Installing NASM under MS−DOS or Windows 15
1.3.2 Installing NASM under Unix 16
Chapter 2: Running NASM 17
2.1 NASM Command−Line Syntax 17
2.1.1 The `-o` Option: Specifying the Output File Name 17
2.1.2 The `-f` Option: Specifying the Output File Format 18
2.1.3 The `-l` Option: Generating a Listing File 18
2.1.4 The `-M` Option: Generate Makefile Dependencies 18
2.1.5 The `-MG` Option: Generate Makefile Dependencies 18
2.1.6 The `-MF` Option: Set Makefile Dependency File 18
2.1.7 The `-MD` Option: Assemble and Generate Dependencies. 18
2.1.8 The `-MT` Option: Dependency Target Name. 19
2.1.9 The `-MQ` Option: Dependency Target Name (Quoted) 19
2.1.10 The `-MP` Option: Emit phony targets. 19
2.1.11 The `-F` Option: Selecting a Debug Information Format 19
2.1.12 The `-g` Option: Enabling Debug Information. 19
2.1.13 The `-X` Option: Selecting an Error Reporting Format. 19
2.1.14 The `-Z` Option: Send Errors to a File. 20
2.1.15 The `-s` Option: Send Errors to `stdout` 20
2.1.16 The `-i` Option: Include File Search Directories 20
2.1.17 The `-p` Option: Pre−Include a File. 20
2.1.18 The `-d` Option: Pre−Define a Macro. 20
2.1.19 The `-u` Option: Undefine a Macro 21
2.1.20 The `-E` Option: Preprocess Only. 21

2.1.21 The -a Option: Don't Preprocess At All 21
2.1.22 The -O Option: Specifying Multipass Optimization 21
2.1.23 The -t Option: Enable TASM Compatibility Mode 22
2.1.24 The -w and -W Options: Enable or Disable Assembly Warnings 22
2.1.25 The -v Option: Display Version Info 23
2.1.26 The -y Option: Display Available Debug Info Formats 23
2.1.27 The --prefix and --postfix Options. 23
2.1.28 The NASMENV Environment Variable 23
2.2 Quick Start for MASM Users 24
2.2.1 NASM Is Case-Sensitive 24
2.2.2 NASM Requires Square Brackets For Memory References 24
2.2.3 NASM Doesn't Store Variable Types 24
2.2.4 NASM Doesn't ASSUME 25
2.2.5 NASM Doesn't Support Memory Models 25
2.2.6 Floating-Point Differences 25
2.2.7 Other Differences 25
Chapter 3: The NASM Language 26
3.1 Layout of a NASM Source Line 26
3.2 Pseudo-Instructions 27
3.2.1 DB and Friends: Declaring Initialized Data 27
3.2.2 RESB and Friends: Declaring Uninitialized Data 27
3.2.3 INCBIN: Including External Binary Files 27
3.2.4 EQU: Defining Constants 28
3.2.5 TIMES: Repeating Instructions or Data 28
3.3 Effective Addresses 28
3.4 Constants 29
3.4.1 Numeric Constants 29
3.4.2 Character Strings 30
3.4.3 Character Constants 31
3.4.4 String Constants 31
3.4.5 Unicode Strings 31
3.4.6 Floating-Point Constants 32
3.4.7 Packed BCD Constants 33
3.5 Expressions 33
3.5.1 |: Bitwise OR Operator 33

3.5.2 ^: Bitwise XOR Operator 33
3.5.3 &: Bitwise AND Operator 33
3.5.4 << and >>: Bit Shift Operators 33
3.5.5 + and −: Addition and Subtraction Operators 33
3.5.6 *, /, //, % and %%: Multiplication and Division 33
3.5.7 Unary Operators: +, −, ~, ! and SEG 34
3.6 SEG and WRT 34
3.7 STRICT: Inhibiting Optimization 34
3.8 Critical Expressions 35
3.9 Local Labels 35
Chapter 4: The NASM Preprocessor 37
4.1 Single−Line Macros 37
4.1.1 The Normal Way: %define 37
4.1.2 Resolving %define: %xdefine 38
4.1.3 Macro Indirection: %[...] 39
4.1.4 Concatenating Single Line Macro Tokens: %+ 39
4.1.5 The Macro Name Itself: %? and %?? 39
4.1.6 Undefining Single−Line Macros: %undef 40
4.1.7 Preprocessor Variables: %assign 40
4.1.8 Defining Strings: %defstr 41
4.1.9 Defining Tokens: %deftok 41
4.2 String Manipulation in Macros 41
4.2.1 Concatenating Strings: %strcat 41
4.2.2 String Length: %strlen 41
4.2.3 Extracting Substrings: %substr 42
4.3 Multi−Line Macros: %macro 42
4.3.1 Overloading Multi−Line Macros 43
4.3.2 Macro−Local Labels 43
4.3.3 Greedy Macro Parameters 44
4.3.4 Macro Parameters Range 44
4.3.5 Default Macro Parameters 45
4.3.6 %0: Macro Parameter Counter 46
4.3.7 %rotate: Rotating Macro Parameters 46
4.3.8 Concatenating Macro Parameters 47
4.3.9 Condition Codes as Macro Parameters 48

4.3.10 Disabling Listing Expansion 48
4.3.11 Undefining Multi-Line Macros: %unmacro. 49
4.4 Conditional Assembly 49
4.4.1 %ifdef: Testing Single-Line Macro Existence. 49
4.4.2 %ifmacro: Testing Multi-Line Macro Existence 50
4.4.3 %ifctx: Testing the Context Stack. 50
4.4.4 %if: Testing Arbitrary Numeric Expressions 50
4.4.5 %ifidn and %ifidni: Testing Exact Text Identity 51
4.4.6 %ifid, %ifnum, %ifstr: Testing Token Types 51
4.4.7 %iftoken: Test for a Single Token 52
4.4.8 %ifempty: Test for Empty Expansion 52
4.5 Preprocessor Loops: %rep. 52
4.6 Source Files and Dependencies 53
4.6.1 %include: Including Other Files 53
4.6.2 %pathsearch: Search the Include Path 53
4.6.3 %depend: Add Dependent Files 54
4.6.4 %use: Include Standard Macro Package 54
4.7 The Context Stack 54
4.7.1 %push and %pop: Creating and Removing Contexts 54
4.7.2 Context-Local Labels. 55
4.7.3 Context-Local Single-Line Macros. 55
4.7.4 %repl: Renaming a Context 55
4.7.5 Example Use of the Context Stack: Block IFs 56
4.8 Stack Relative Preprocessor Directives. 57
4.8.1 %arg Directive 57
4.8.2 %stacksize Directive 58
4.8.3 %local Directive 58
4.9 Reporting User-Defined Errors: %error, %warning, %fatal 59
4.10 Other Preprocessor Directives 59
4.10.1 %line Directive 60
4.10.2 %!<env>: Read an environment variable. 60
4.11 Standard Macros 60
4.11.1 NASM Version Macros 60
4.11.2 __NASM_VERSION_ID__: NASM Version ID. 60
4.11.3 __NASM_VER__: NASM Version string 61

4.11.4 __FILE__ and __LINE__: File Name and Line Number 61
4.11.5 __BITS__: Current BITS Mode 61
4.11.6 __OUTPUT_FORMAT__: Current Output Format 61
4.11.7 Assembly Date and Time Macros 62
4.11.8 __USE_*package*__: Package Include Test 62
4.11.9 __PASS__: Assembly Pass 62
4.11.10 STRUC and ENDSTRUC: Declaring Structure Data Types 62
4.11.11 ISTRUC, AT and IEND: Declaring Instances of Structures 64
4.11.12 ALIGN and ALIGNB: Data Alignment 64
Chapter 5: Standard Macro Packages 66
5.1 altreg: Alternate Register Names 66
5.2 smartalign: Smart ALIGN Macro 66
Chapter 6: Assembler Directives 67
6.1 BITS: Specifying Target Processor Mode 67
6.1.1 USE16 & USE32: Aliases for BITS 68
6.2 DEFAULT: Change the assembler defaults 68
6.3 SECTION or SEGMENT: Changing and Defining Sections 68
6.3.1 The __SECT__ Macro 68
6.4 ABSOLUTE: Defining Absolute Labels 69
6.5 EXTERN: Importing Symbols from Other Modules 70
6.6 GLOBAL: Exporting Symbols to Other Modules 70
6.7 COMMON: Defining Common Data Areas 70
6.8 CPU: Defining CPU Dependencies 71
6.9 FLOAT: Handling of floating−point constants 71
Chapter 7: Output Formats 73
7.1 bin: Flat−Form Binary Output 73
7.1.1 ORG: Binary File Program Origin 73
7.1.2 bin Extensions to the SECTION Directive 73
7.1.3 Multisection Support for the bin Format 74
7.1.4 Map Files 74
7.2 ith: Intel Hex Output 74
7.3 srec: Motorola S−Records Output 74
7.4 obj: Microsoft OMF Object Files 75
7.4.1 obj Extensions to the SEGMENT Directive 75
7.4.2 GROUP: Defining Groups of Segments 76

7.4.3 UPPERCASE: Disabling Case Sensitivity in Output 77
7.4.4 IMPORT: Importing DLL Symbols 77
7.4.5 EXPORT: Exporting DLL Symbols 77
7.4.6 ..start: Defining the Program Entry Point 78
7.4.7 obj Extensions to the EXTERN Directive 78
7.4.8 obj Extensions to the COMMON Directive 78
7.5 win32: Microsoft Win32 Object Files 79
7.5.1 win32 Extensions to the SECTION Directive 79
7.5.2 win32: Safe Structured Exception Handling 80
7.6 win64: Microsoft Win64 Object Files 81
7.6.1 win64: Writing Position–Independent Code 81
7.6.2 win64: Structured Exception Handling 82
7.7 coff: Common Object File Format 85
7.8 macho32 and macho64: Mach Object File Format 85
7.9 elf32 and elf64: Executable and Linkable Format Object Files 85
7.9.1 ELF specific directive osabi 85
7.9.2 elf Extensions to the SECTION Directive 85
7.9.3 Position–Independent Code: elf Special Symbols and WRT 86
7.9.4 Thread Local Storage: elf Special Symbols and WRT 87
7.9.5 elf Extensions to the GLOBAL Directive 87
7.9.6 elf Extensions to the COMMON Directive 87
7.9.7 16–bit code and ELF 88
7.9.8 Debug formats and ELF 88
7.10 aout: Linux a.out Object Files 88
7.11 aoutb: NetBSD/FreeBSD/OpenBSD a.out Object Files 88
7.12 as86: Minix/Linux as86 Object Files 88
7.13 rdf: Relocatable Dynamic Object File Format 88
7.13.1 Requiring a Library: The LIBRARY Directive 89
7.13.2 Specifying a Module Name: The MODULE Directive 89
7.13.3 rdf Extensions to the GLOBAL Directive 89
7.13.4 rdf Extensions to the EXTERN Directive 89
7.14 dbg: Debugging Format 90
Chapter 8: Writing 16–bit Code (DOS, Windows 3/3.1) 91
8.1 Producing .EXE Files 91
8.1.1 Using the obj Format To Generate .EXE Files 91

8.1.2 Using the `bin` Format To Generate `.EXE` Files 92
8.2 Producing `.COM` Files 93
8.2.1 Using the `bin` Format To Generate `.COM` Files 93
8.2.2 Using the `obj` Format To Generate `.COM` Files 93
8.3 Producing `.SYS` Files 94
8.4 Interfacing to 16−bit C Programs. 94
8.4.1 External Symbol Names 94
8.4.2 Memory Models 95
8.4.3 Function Definitions and Function Calls. 96
8.4.4 Accessing Data Items 98
8.4.5 `c16.mac`: Helper Macros for the 16−bit C Interface 98
8.5 Interfacing to Borland Pascal Programs 99
8.5.1 The Pascal Calling Convention 100
8.5.2 Borland Pascal Segment Name Restrictions 101
8.5.3 Using `c16.mac` With Pascal Programs 101
Chapter 9: Writing 32−bit Code (Unix, Win32, DJGPP). 103
9.1 Interfacing to 32−bit C Programs. 103
9.1.1 External Symbol Names 103
9.1.2 Function Definitions and Function Calls. 103
9.1.3 Accessing Data Items 105
9.1.4 `c32.mac`: Helper Macros for the 32−bit C Interface 105
9.2 Writing NetBSD/FreeBSD/OpenBSD and Linux/ELF Shared Libraries. 106
9.2.1 Obtaining the Address of the GOT 106
9.2.2 Finding Your Local Data Items 107
9.2.3 Finding External and Common Data Items 107
9.2.4 Exporting Symbols to the Library User 108
9.2.5 Calling Procedures Outside the Library 109
9.2.6 Generating the Library File 109
Chapter 10: Mixing 16 and 32 Bit Code. 110
10.1 Mixed−Size Jumps 110
10.2 Addressing Between Different−Size Segments 110
10.3 Other Mixed−Size Instructions 111
Chapter 11: Writing 64−bit Code (Unix, Win64) 113
11.1 Register Names in 64−bit Mode 113
11.2 Immediates and Displacements in 64−bit Mode 113

11.3 Interfacing to 64-bit C Programs (Unix) . . . 114
11.4 Interfacing to 64-bit C Programs (Win64) . . . 114
Chapter 12: Troubleshooting . . . 116
12.1 Common Problems . . . 116
12.1.1 NASM Generates Inefficient Code . . . 116
12.1.2 My Jumps are Out of Range . . . 116
12.1.3 `ORG` Doesn't Work . . . 116
12.1.4 `TIMES` Doesn't Work . . . 117
12.2 Bugs . . . 117
Appendix A: Ndisasm . . . 119
A.1 Introduction . . . 119
A.2 Getting Started: Installation . . . 119
A.3 Running NDISASM . . . 119
A.3.1 COM Files: Specifying an Origin . . . 119
A.3.2 Code Following Data: Synchronisation . . . 119
A.3.3 Mixed Code and Data: Automatic (Intelligent) Synchronisation . . . 120
A.3.4 Other Options . . . 121
A.4 Bugs and Improvements . . . 121
Appendix B: Instruction List . . . 122
B.1 Introduction . . . 122
B.1.1 Special instructions... . . . 122
B.1.2 Conventional instructions . . . 122
B.1.3 Katmai Streaming SIMD instructions (SSE — a.k.a. KNI, XMM, MMX2) . . . 148
B.1.4 Introduced in Deschutes but necessary for SSE support . . . 149
B.1.5 XSAVE group (AVX and extended state) . . . 149
B.1.6 Generic memory operations . . . 150
B.1.7 New MMX instructions introduced in Katmai . . . 150
B.1.8 AMD Enhanced 3DNow! (Athlon) instructions . . . 150
B.1.9 Willamette SSE2 Cacheability Instructions . . . 150
B.1.10 Willamette MMX instructions (SSE2 SIMD Integer Instructions) . . . 150
B.1.11 Willamette Streaming SIMD instructions (SSE2) . . . 152
B.1.12 Prescott New Instructions (SSE3) . . . 154
B.1.13 VMX Instructions . . . 154
B.1.14 Extended Page Tables VMX instructions . . . 155
B.1.15 Tejas New Instructions (SSSE3) . . . 155

B.1.16 AMD SSE4A . . . 155
B.1.17 New instructions in Barcelona . . . 155
B.1.18 Penryn New Instructions (SSE4.1) . . . 156
B.1.19 Nehalem New Instructions (SSE4.2) . . . 157
B.1.20 Intel SMX . . . 157
B.1.21 Geode (Cyrix) 3DNow! additions . . . 157
B.1.22 Intel new instructions in ??? . . . 157
B.1.23 Intel AES instructions . . . 157
B.1.24 Intel AVX AES instructions . . . 157
B.1.25 Intel AVX instructions . . . 158
B.1.26 Intel Carry−Less Multiplication instructions (CLMUL) . . . 170
B.1.27 Intel AVX Carry−Less Multiplication instructions (CLMUL) . . . 170
B.1.28 Intel Fused Multiply−Add instructions (FMA) . . . 170
B.1.29 Intel post−32 nm processor instructions . . . 174
B.1.30 VIA (Centaur) security instructions . . . 174
B.1.31 AMD Lightweight Profiling (LWP) instructions . . . 174
B.1.32 AMD XOP and FMA4 instructions (SSE5) . . . 174
B.1.33 Systematic names for the hinting nop instructions . . . 177
Appendix C: NASM Version History . . . 182
C.1 NASM 2 Series . . . 182
C.1.1 Version 2.09 . . . 182
C.1.2 Version 2.08 . . . 182
C.1.3 Version 2.07 . . . 183
C.1.4 Version 2.06 . . . 183
C.1.5 Version 2.05.01 . . . 184
C.1.6 Version 2.05 . . . 184
C.1.7 Version 2.04 . . . 184
C.1.8 Version 2.03.01 . . . 185
C.1.9 Version 2.03 . . . 185
C.1.10 Version 2.02 . . . 186
C.1.11 Version 2.01 . . . 186
C.1.12 Version 2.00 . . . 187
C.2 NASM 0.98 Series . . . 187
C.2.1 Version 0.98.39 . . . 188
C.2.2 Version 0.98.38 . . . 188

C.2.3 Version 0.98.37 . . . 188
C.2.4 Version 0.98.36 . . . 188
C.2.5 Version 0.98.35 . . . 189
C.2.6 Version 0.98.34 . . . 189
C.2.7 Version 0.98.33 . . . 189
C.2.8 Version 0.98.32 . . . 189
C.2.9 Version 0.98.31 . . . 190
C.2.10 Version 0.98.30 . . . 190
C.2.11 Version 0.98.28 . . . 190
C.2.12 Version 0.98.26 . . . 190
C.2.13 Version 0.98.25alt . . . 190
C.2.14 Version 0.98.25 . . . 191
C.2.15 Version 0.98.24p1 . . . 191
C.2.16 Version 0.98.24 . . . 191
C.2.17 Version 0.98.23 . . . 191
C.2.18 Version 0.98.22 . . . 191
C.2.19 Version 0.98.21 . . . 191
C.2.20 Version 0.98.20 . . . 191
C.2.21 Version 0.98.19 . . . 191
C.2.22 Version 0.98.18 . . . 191
C.2.23 Version 0.98.17 . . . 191
C.2.24 Version 0.98.16 . . . 191
C.2.25 Version 0.98.15 . . . 191
C.2.26 Version 0.98.14 . . . 191
C.2.27 Version 0.98.13 . . . 191
C.2.28 Version 0.98.12 . . . 191
C.2.29 Version 0.98.11 . . . 192
C.2.30 Version 0.98.10 . . . 192
C.2.31 Version 0.98.09 . . . 192
C.2.32 Version 0.98.08 . . . 192
C.2.33 Version 0.98.09b with John Coffman patches released 28−Oct−2001 . . . 192
C.2.34 Version 0.98.07 released 01/28/01 . . . 193
C.2.35 Version 0.98.06f released 01/18/01 . . . 193
C.2.36 Version 0.98.06e released 01/09/01 . . . 193
C.2.37 Version 0.98p1 . . . 193

C.2.38 Version 0.98bf (bug−fixed) . . . 193
C.2.39 Version 0.98.03 with John Coffman's changes released 27−Jul−2000 . . . 193
C.2.40 Version 0.98.03 . . . 194
C.2.41 Version 0.98 . . . 197
C.2.42 Version 0.98p9 . . . 197
C.2.43 Version 0.98p8 . . . 197
C.2.44 Version 0.98p7 . . . 198
C.2.45 Version 0.98p6 . . . 198
C.2.46 Version 0.98p3.7 . . . 198
C.2.47 Version 0.98p3.6 . . . 199
C.2.48 Version 0.98p3.5 . . . 199
C.2.49 Version 0.98p3.4 . . . 199
C.2.50 Version 0.98p3.3 . . . 199
C.2.51 Version 0.98p3.2 . . . 199
C.2.52 Version 0.98p3−hpa . . . 200
C.2.53 Version 0.98 pre−release 3 . . . 200
C.2.54 Version 0.98 pre−release 2 . . . 200
C.2.55 Version 0.98 pre−release 1 . . . 200
C.3 NASM 0.9 Series . . . 201
C.3.1 Version 0.97 released December 1997 . . . 201
C.3.2 Version 0.96 released November 1997 . . . 202
C.3.3 Version 0.95 released July 1997 . . . 204
C.3.4 Version 0.94 released April 1997 . . . 205
C.3.5 Version 0.93 released January 1997 . . . 206
C.3.6 Version 0.92 released January 1997 . . . 206
C.3.7 Version 0.91 released November 1996 . . . 206
C.3.8 Version 0.90 released October 1996 . . . 207

Chapter 1: Introduction

1.1 What Is NASM?

The Netwide Assembler, NASM, is an 80x86 and x86–64 assembler designed for portability and modularity. It supports a range of object file formats, including Linux and *BSD `a.out`, `ELF`, `COFF`, `Mach-O`, Microsoft 16–bit `OBJ`, `Win32` and `Win64`. 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 all currently known x86 architectural extensions, and has strong support for macros.

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 to 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 specific to Minix and Linux, and (my version at least) doesn't seem to have much (or any) documentation.
- `MASM` isn't very good, and it's (was) 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 License Conditions

Please see the file `LICENSE`, supplied as part of any NASM distribution archive, for the license conditions under which you may use NASM. NASM is now under the so–called 2–clause BSD license, also known as the simplified BSD license.

Copyright 1996–2010 the NASM Authors – All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.2 Contact Information

The current version of NASM (since about 0.98.08) is maintained by a team of developers, accessible through the `nasm-devel` mailing list (see below for the link). If you want to report a bug, please read section 12.2 first.

NASM has a website at `http://www.nasm.us/`. If it's not there, google for us!

New releases, release candidates, and daily development snapshots of NASM are available from the official web site.

Announcements are posted to `comp.lang.asm.x86`, and to the web site `http://www.freshmeat.net/`.

If you want information about the current development status, please subscribe to the `nasm-devel` email list; see link from the website.

1.3 Installation

1.3.1 Installing NASM under MS-DOS or Windows

Once you've obtained the appropriate archive for NASM, `nasm-XXX-dos.zip` or `nasm-XXX-win32.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 a set of executable files: the NASM executable file `nasm.exe`, the NDISASM executable file `ndisasm.exe`, and possibly additional utilities to handle the RDOFF file format.

The only file NASM needs to run is its own executable, so copy `nasm.exe` to a directory on your PATH, or alternatively edit `autoexec.bat` to add the `nasm` directory to your `PATH` (to do that under Windows XP, go to Start > Control Panel > System > Advanced > Environment Variables; these instructions may work under other versions of Windows as well.)

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, `nasm-XXX.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. See the file `INSTALL` in the source archive.

Note that a number of files are generated from other files by Perl scripts. Although the NASM source distribution includes these generated files, you will need to rebuild them (and hence, will need a Perl interpreter) if you change insns.dat, standard.mac or the documentation. It is possible future source distributions may not include these files at all. Ports of Perl for a variety of platforms, including DOS and Windows, are available from www.cpan.org.

1.3.2 Installing NASM under Unix

Once you've obtained the Unix source archive for NASM, `nasm-XXX.tar.gz` (where `XXX` 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-XXX`.

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.

Chapter 2: Running NASM

2.1 NASM Command-Line 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
```

As `-hf`, 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 (Linux `a.out` systems have long been obsolete, and are rare these days.)

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`, `win32` and `win64`), 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`, `as86`, `coff`, `elf32`, `elf64`, `ieee`, `macho32` and `macho64`) it will substitute `.o`. For `dbg`, `rdf`, `ith` and `srec`, it will use `.dbg`, `.rdf`, `.ith` and `.srec`, respectively, 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
```

Note that this is a small o, and is different from a capital O , which is used to specify the number of optimisation passes required. See section 2.1.22.

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 -hf`.

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 source-listing 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 4.3.10) on the right. For example:

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

If a list file is selected, you may turn off listing for a section of your source with `[list -]`, and turn it back on with `[list +]`, (the default, obviously). There is no "user form" (without the brackets). This can be used to list only sections of interest, avoiding excessively long listings.

2.1.4 The `-M` Option: Generate Makefile Dependencies

This option can be used to generate makefile dependencies on stdout. This can be redirected to a file for further processing. For example:

```
nasm -M myfile.asm > myfile.dep
```

2.1.5 The `-MG` Option: Generate Makefile Dependencies

This option can be used to generate makefile dependencies on stdout. This differs from the `-M` option in that if a nonexisting file is encountered, it is assumed to be a generated file and is added to the dependency list without a prefix.

2.1.6 The `-MF` Option: Set Makefile Dependency File

This option can be used with the `-M` or `-MG` options to send the output to a file, rather than to stdout. For example:

```
nasm -M -MF myfile.dep myfile.asm
```

2.1.7 The `-MD` Option: Assemble and Generate Dependencies

The `-MD` option acts as the combination of the `-M` and `-MF` options (i.e. a filename has to be specified.) However, unlike the `-M` or `-MG` options, `-MD` does *not* inhibit the normal operation of the assembler. Use this to automatically generate updated dependencies with every assembly session. For example:

```
nasm -f elf -o myfile.o -MD myfile.dep myfile.asm
```

2.1.8 The `-MT` Option: Dependency Target Name

The `-MT` option can be used to override the default name of the dependency target. This is normally the same as the output filename, specified by the `-o` option.

2.1.9 The `-MQ` Option: Dependency Target Name (Quoted)

The `-MQ` option acts as the `-MT` option, except it tries to quote characters that have special meaning in Makefile syntax. This is not foolproof, as not all characters with special meaning are quotable in Make.

2.1.10 The `-MP` Option: Emit phony targets

When used with any of the dependency generation options, the `-MP` option causes NASM to emit a phony target without dependencies for each header file. This prevents Make from complaining if a header file has been removed.

2.1.11 The `-F` Option: Selecting a Debug Information Format

This option is used to select the format of the debug information emitted into the output file, to be used by a debugger (or *will* be). Prior to version 2.03.01, the use of this switch did *not* enable output of the selected debug info format. Use `-g`, see section 2.1.12, to enable output. Versions 2.03.01 and later automatically enable `-g` if `-F` is specified.

A complete list of the available debug file formats for an output format can be seen by issuing the command `nasm -f <format> -y`. Not all output formats currently support debugging output. See section 2.1.26.

This should not be confused with the `-f dbg` output format option which is not built into NASM by default. For information on how to enable it when building from the sources, see section 7.14.

2.1.12 The `-g` Option: Enabling Debug Information.

This option can be used to generate debugging information in the specified format. See section 2.1.11. Using `-g` without `-F` results in emitting debug info in the default format, if any, for the selected output format. If no debug information is currently implemented in the selected output format, `-g` is *silently ignored*.

2.1.13 The `-X` Option: Selecting an Error Reporting Format

This option can be used to select an error reporting format for any error messages that might be produced by NASM.

Currently, two error reporting formats may be selected. They are the `-Xvc` option and the `-Xgnu` option. The GNU format is the default and looks like this:

```
filename.asm:65: error: specific error message
```

where `filename.asm` is the name of the source file in which the error was detected, `65` is the source file line number on which the error was detected, `error` is the severity of the error (this could be `warning`), and `specific error message` is a more detailed text message which should help pinpoint the exact problem.

The other format, specified by `-Xvc` is the style used by Microsoft Visual C++ and some other programs. It looks like this:

```
filename.asm(65) : error: specific error message
```

where the only difference is that the line number is in parentheses instead of being delimited by colons.

See also the `Visual C++` output format, section 7.5.

2.1.14 The -Z Option: Send Errors to a File

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 -Z option, taking a filename argument which causes errors to be sent to the specified files rather than standard error. Therefore you can redirect the errors into a file by typing

```
nasm -Z myfile.err -f obj myfile.asm
```

In earlier versions of NASM, this option was called -E, but it was changed since -E is an option conventionally used for preprocessing only, with disastrous results. See section 2.1.20.

2.1.15 The -s Option: Send Errors to stdout

The -s option redirects error messages to stdout rather than stderr, so it can be redirected under MS-DOS. To assemble the file myfile.asm and pipe its output to the more program, you can type:

```
nasm -s -f obj myfile.asm | more
```

See also the -Z option, section 2.1.14.

2.1.16 The -i Option: Include File Search Directories

When NASM sees the %include or %pathsearch directive in a source file (see section 4.6.1, section 4.6.2 or section 3.2.3), 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 NASMENV environment variable (see section 2.1.28).

For Makefile compatibility with many C compilers, this option can also be specified as -I.

2.1.17 The -p Option: Pre-Include a File

NASM allows you to specify files to be *pre-included* 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.

For consistency with the -I, -D and -U options, this option can also be specified as -P.

2.1.18 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 assembly–time options which are then tested using `%ifdef`, for example `-dDEBUG`.

For Makefile compatibility with many C compilers, this option can also be specified as `-D`.

2.1.19 The `-u` Option: Undefine a Macro

The `-u` option undefines a macro that would otherwise have been pre–defined, either automatically or by a `-p` or `-d` option specified earlier on the command lines.

For example, the following command line:

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

would result in `FOO` *not* being a predefined macro in the program. This is useful to override options specified at a different point in a Makefile.

For Makefile compatibility with many C compilers, this option can also be specified as `-U`.

2.1.20 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 preprocess–only mode.

For compatiblity with older version of NASM, this option can also be written `-e`. `-E` in older versions of NASM was the equivalent of the current `-Z` option, section 2.1.14.

2.1.21 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.

2.1.22 The `-O` Option: Specifying Multipass Optimization

NASM defaults to not optimizing operands which can fit into a signed byte. This means that if you want the shortest possible object code, you have to enable optimization.

Using the `-O` option, you can tell NASM to carry out different levels of optimization. The syntax is:

- `-O0`: No optimization. All operands take their long forms, if a short form is not specified, except conditional jumps. This is intended to match NASM 0.98 behavior.
- `-O1`: Minimal optimization. As above, but immediate operands which will fit in a signed byte are optimized, unless the long form is specified. Conditional jumps default to the long form unless otherwise specified.

- `-Ox` (where `x` is the actual letter `x`): Multipass optimization. Minimize branch offsets and signed immediate bytes, overriding size specification unless the `strict` keyword has been used (see section 3.7). For compatability with earlier releases, the letter `x` may also be any number greater than one. This number has no effect on the actual number of passes.

The `-Ox` mode is recommended for most uses.

Note that this is a capital `O`, and is different from a small `o`, which is used to specify the output file name. See section 2.1.1.

2.1.23 The `-t` Option: Enable TASM Compatibility Mode

NASM includes a limited form of compatibility with Borland's `TASM`. When NASM's `-t` option is used, the following changes are made:

- local labels may be prefixed with `@@` instead of `.`
- size override is supported within brackets. In TASM compatible mode, a size override inside square brackets changes the size of the operand, and not the address type of the operand as it does in NASM syntax. E.g. `mov eax,[DWORD val]` is valid syntax in TASM compatibility mode. Note that you lose the ability to override the default address type for the instruction.
- unprefixed forms of some directives supported (`arg`, `elif`, `else`, `endif`, `if`, `ifdef`, `ifdifi`, `ifndef`, `include`, `local`)

2.1.24 The `-w` and `-W` Options: 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 `orphan-labels`; you can enable warnings of this class by the command–line option `-w+orphan-labels` and disable it by `-w-orphan-labels`.

The suppressible warning classes are:

- `macro-params` covers warnings about multi–line macros being invoked with the wrong number of parameters. This warning class is enabled by default; see section 4.3.1 for an example of why you might want to disable it.
- `macro-selfref` warns if a macro references itself. This warning class is disabled by default.
- `macro-defaults` warns when a macro has more default parameters than optional parameters. This warning class is enabled by default; see section 4.3.5 for why you might want to disable it.
- `orphan-labels` covers warnings about source lines which contain no instruction but define a label without a trailing colon. NASM warns about this somewhat obscure condition by default; see section 3.1 for more information.
- `number-overflow` covers warnings about numeric constants which don't fit in 64 bits. This warning class is enabled by default.
- `gnu-elf-extensions` warns if 8–bit or 16–bit relocations are used in `-f elf` format. The GNU extensions allow this. This warning class is disabled by default.
- `float-overflow` warns about floating point overflow. Enabled by default.
- `float-denorm` warns about floating point denormals. Disabled by default.

- `float-underflow` warns about floating point underflow. Disabled by default.
- `float-toolong` warns about too many digits in floating-point numbers. Enabled by default.
- `user` controls `%warning` directives (see section 4.9). Enabled by default.
- `error` causes warnings to be treated as errors. Disabled by default.
- `all` is an alias for *all* suppressible warning classes (not including `error`). Thus, `-w+all` enables all available warnings.

In addition, you can set warning classes across sections. Warning classes may be enabled with `[warning +warning-name]`, disabled with `[warning -warning-name]` or reset to their original value with `[warning *warning-name]`. No "user form" (without the brackets) exists.

Since version 2.00, NASM has also supported the gcc-like syntax `-Wwarning` and `-Wno-warning` instead of `-w+warning` and `-w-warning`, respectively.

2.1.25 The `-v` Option: Display Version Info

Typing `NASM -v` will display the version of NASM which you are using, and the date on which it was compiled.

You will need the version number if you report a bug.

2.1.26 The `-y` Option: Display Available Debug Info Formats

Typing `nasm -f <option> -y` will display a list of the available debug info formats for the given output format. The default format is indicated by an asterisk. For example:

```
nasm -f elf -y
```

```
valid debug formats for 'elf32' output format are
  ('*' denotes default):
  * stabs     ELF32 (i386) stabs debug format for Linux
    dwarf     elf32 (i386) dwarf debug format for Linux
```

2.1.27 The `--prefix` and `--postfix` Options.

The `--prefix` and `--postfix` options prepend or append (respectively) the given argument to all `global` or `extern` variables. E.g. `--prefix _` will prepend the underscore to all global and external variables, as C sometimes (but not always) likes it.

2.1.28 The `NASMENV` Environment Variable

If you define an environment variable called `NASMENV`, 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 `NASMENV` 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 `NASMENV` 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 `NASMENV` variable to the value `!-s!-ic:\nasmlib\` is equivalent to setting it to `-s -ic:\nasmlib\`, but `!-dNAME="my name"` will work.

This environment variable was previously called `NASM`. This was changed with version 0.98.31.

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 7.4) 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 Floating−Point 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 uninitialized 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 6 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 6) 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 uses backslash (\) as the line continuation character; if a line ends with backslash, the next line is considered to be a part of the backslash−ended line.

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+orphan-labels` 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, _, `$`, `#`, `@`, `~`, `.`, and `?`. The only characters which may be used as the *first* character of an identifier are letters, `.` (with special meaning: see section 3.9), _ and `?`. An identifier may also be prefixed with a `$` 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. Maximum length of an identifier is 4095 characters.

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 operand−size prefixes `A16`, `A32`, `A64`, `O16` and `O32`, `O64` are provided – one example of their use is given in chapter 10. 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 x87 floating−point 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. 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 x87 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 Pseudo−Instructions

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`, `DT`, `DO` and `DY`; their uninitialized counterparts `RESB`, `RESW`, `RESD`, `RESQ`, `REST`, `RESO` and `RESY`; the `INCBIN` command, the `EQU` command, and the `TIMES` prefix.

3.2.1 `DB` and Friends: Declaring Initialized Data

`DB`, `DW`, `DD`, `DQ`, `DT`, `DO` and `DY` are used, much as in MASM, to declare initialized 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'                 ; 0x61 0x00 (it's just a number)
dw    'ab'                ; 0x61 0x62 (character constant)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; floating-point constant
dq    0x123456789abcdef0  ; eight byte constant
dq    1.234567e20         ; double-precision float
dt    1.234567e20         ; extended-precision float
```

`DT`, `DO` and `DY` do not accept numeric constants as operands.

3.2.2 `RESB` and Friends: Declaring Uninitialized Data

`RESB`, `RESW`, `RESD`, `RESQ`, `REST`, `RESO` and `RESY` are designed to be used in the BSS section of a module: they declare *uninitialized* 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 uninitialized 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 3.8.

For example:

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

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
```

INCBIN is both a directive and a standard macro; the standard macro version searches for the file in the include file search path and adds the file to the dependency lists. This macro can be overridden if desired.

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.

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 is a critical expression (section 3.8).

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 3.8 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.

The form described in the previous paragraph is also useful if you are trying to access data in a 32–bit segment from within 16 bit code. For more information on this see the section on mixed–size addressing (section 10.2). In particular, if you need to access data with a known offset that is larger than will fit in a 16–bit value, if you don't specify that it is a dword offset, nasm will cause the high word of the offset to be lost.

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.

In 64–bit mode, NASM will by default generate absolute addresses. The `REL` keyword makes it produce `RIP`–relative addresses. Since this is frequently the normally desired behaviour, see the `DEFAULT` directive (section 6.2). The keyword `ABS` overrides `REL`.

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` or `X`, `D` or `T`, `Q` or `O`, and `B` or `Y` for hexadecimal, decimal, octal and binary respectively, or you can prefix `0x`, for hexadecimal in the style of C, or you can prefix `$` for hexadecimal in the style of Borland Pascal or Motorola Assemblers. Note, though, that the `$` 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. In addition, current versions of NASM accept the prefix `0h` for

hexadecimal, `0d` or `0t` for decimal, `0o` or `0q` for octal, and `0b` or `0y` for binary. Please note that unlike C, a `0` prefix by itself does *not* imply an octal constant!

Numeric constants can have underscores (_) interspersed to break up long strings.

Some examples (all producing exactly the same code):

```
mov     ax,200          ; decimal
mov     ax,0200         ; still decimal
mov     ax,0200d        ; explicitly decimal
mov     ax,0d200        ; also decimal
mov     ax,0c8h         ; hex
mov     ax,$0c8         ; hex again: the 0 is required
mov     ax,0xc8         ; hex yet again
mov     ax,0hc8         ; still hex
mov     ax,310q         ; octal
mov     ax,310o         ; octal again
mov     ax,0o310        ; octal yet again
mov     ax,0q310        ; hex yet again
mov     ax,11001000b    ; binary
mov     ax,1100_1000b   ; same binary constant
mov     ax,1100_1000y   ; same binary constant once more
mov     ax,0b1100_1000  ; same binary constant yet again
mov     ax,0y1100_1000  ; same binary constant yet again
```

3.4.2 Character Strings

A character string consists of up to eight characters enclosed in either single quotes (`'...'`), double quotes (`"..."`) or backquotes (`` `...` ``). Single or double quotes are equivalent to NASM (except of course that surrounding the constant with single quotes allows double quotes to appear within it and vice versa); the contents of those are represented verbatim. Strings enclosed in backquotes support C–style `\`–escapes for special characters.

The following escape sequences are recognized by backquoted strings:

```
\'          single quote (')
\"          double quote (")
\`          backquote (`)
\\          backslash (\)
\?          question mark (?)
\a          BEL (ASCII 7)
\b          BS  (ASCII 8)
\t          TAB (ASCII 9)
\n          LF  (ASCII 10)
\v          VT  (ASCII 11)
\f          FF  (ASCII 12)
\r          CR  (ASCII 13)
\e          ESC (ASCII 27)
\377        Up to 3 octal digits - literal byte
\xFF        Up to 2 hexadecimal digits - literal byte
\u1234      4 hexadecimal digits - Unicode character
\U12345678  8 hexadecimal digits - Unicode character
```

All other escape sequences are reserved. Note that `\0`, meaning a `NUL` character (ASCII 0), is a special case of the octal escape sequence.

Unicode characters specified with `\u` or `\U` are converted to UTF−8. For example, the following lines are all equivalent:

```
db `\u263a`             ; UTF-8 smiley face
db `\xe2\x98\xba`       ; UTF-8 smiley face
db 0E2h, 098h, 0BAh     ; UTF-8 smiley face
```

3.4.3 Character Constants

A character constant consists of a string up to eight bytes long, used in an expression context. It is treated as if it was an integer.

A character constant with more than one byte will be arranged with little−endian 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.

3.4.4 String Constants

String constants are character strings used in the context of some pseudo−instructions, namely the `DB` family and `INCBIN` (where it represents a filename.) They are also used in certain preprocessor directives.

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 in a string−supporting context, quoted strings are treated as a string constants even if they are 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`, and so forth.

3.4.5 Unicode Strings

The special operators `__utf16__` and `__utf32__` allows definition of Unicode strings. They take a string in UTF−8 format and converts it to (littleendian) UTF−16 or UTF−32, respectively.

For example:

```
%define u(x) __utf16__(x)
%define w(x) __utf32__(x)

      dw u('C:\WINDOWS'), 0       ; Pathname in UTF-16
      dd w(`A + B = \u206a`), 0   ; String in UTF-32
```

`__utf16__` and `__utf32__` can be applied either to strings passed to the `DB` family instructions, or to character constants in an expression context.

3.4.6 Floating−Point Constants

Floating−point constants are acceptable only as arguments to `DB`, `DW`, `DD`, `DQ`, `DT`, and `DO`, or as arguments to the special operators `__float8__`, `__float16__`, `__float32__`, `__float64__`, `__float80m__`, `__float80e__`, `__float128l__`, and `__float128h__`.

Floating−point constants 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.

NASM also support C99−style hexadecimal floating−point: `0x`, hexadecimal digits, period, optionally more hexadeximal digits, then optionally a `P` followed by a *binary* (not hexadecimal) exponent in decimal notation. As an extension, NASM additionally supports the `0h` and `$` prefixes for hexadecimal, as well binary and octal floating−point, using the `0b` or `0y` and `0o` or `0q` prefixes, respectively.

Underscores to break up groups of digits are permitted in floating−point constants as well.

Some examples:

```
db    -0.2                    ; "Quarter precision"
dw    -0.5                    ; IEEE 754r/SSE5 half precision
dd    1.2                     ; an easy one
dd    1.222_222_222           ; underscores are permitted
dd    0x1p+2                  ; 1.0x2^2 = 4.0
dq    0x1p+32                 ; 1.0x2^32 = 4 294 967 296.0
dq    1.e10                   ; 10 000 000 000.0
dq    1.e+10                  ; synonymous with 1.e10
dq    1.e-10                  ; 0.000 000 000 1
dt    3.14159265358979323846 ; pi
do    1.e+4000                ; IEEE 754r quad precision
```

The 8−bit "quarter−precision" floating−point format is sign:exponent:mantissa = 1:4:3 with an exponent bias of 7. This appears to be the most frequently used 8−bit floating−point format, although it is not covered by any formal standard. This is sometimes called a "minifloat."

The special operators are used to produce floating−point numbers in other contexts. They produce the binary representation of a specific floating−point number as an integer, and can use anywhere integer constants are used in an expression. `__float80m__` and `__float80e__` produce the 64−bit mantissa and 16−bit exponent of an 80−bit floating−point number, and `__float128l__` and `__float128h__` produce the lower and upper 64−bit halves of a 128−bit floating−point number, respectively.

For example:

```
mov     rax,__float64__(3.14159265358979323846)
```

... would assign the binary representation of pi as a 64−bit floating point number into `RAX`. This is exactly equivalent to:

```
mov     rax,0x400921fb54442d18
```

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.

The special tokens `__Infinity__`, `__QNaN__` (or `__NaN__`) and `__SNaN__` can be used to generate infinities, quiet NaNs, and signalling NaNs, respectively. These are normally used as macros:

```
%define Inf __Infinity__
%define NaN __QNaN__

      dq    +1.5, -Inf, NaN         ; Double-precision constants
```

3.4.7 Packed BCD Constants

x87-style packed BCD constants can be used in the same contexts as 80-bit floating-point numbers. They are suffixed with `p` or prefixed with `0p`, and can include up to 18 decimal digits.

As with other numeric constants, underscores can be used to separate digits.

For example:

```
dt 12_345_678_901_245_678p
dt -12_345_678_901_245_678p
dt +0p33
dt 33p
```

3.5 Expressions

Expressions in NASM are similar in syntax to those in C. Expressions are evaluated as 64-bit integers which are then adjusted to the appropriate size.

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, ! is the logical negation operator, 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 STRICT: Inhibiting Optimization

When assembling with the optimizer set to level 2 or higher (see section 2.1.22), NASM will use size specifiers (BYTE, WORD, DWORD, QWORD, TWORD, OWORD or YWORD), but will give them the smallest possible size. The keyword STRICT can be used to inhibit optimization and force a particular operand to be emitted in the specified size. For example, with the optimizer on, and in BITS 16 mode,

```
        push dword 33
```

is encoded in three bytes `66 6A 21`, whereas

```
        push strict dword 33
```

is encoded in six bytes, with a full dword immediate operand `66 68 21 00 00 00`.

With the optimizer off, the same code (six bytes) is generated whether the `STRICT` keyword was used or not.

3.8 Critical Expressions

Although NASM has an optional multi–pass optimizer, there are some expressions which must be resolvable on the first pass. These are called *Critical Expressions*.

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.

3.9 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 7.4.6), `..imagebase` is used to find out the offset from a base address of the current image in the `win64` output format (see section 7.6.1). So just keep in mind that symbols beginning with a double period are special.

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.

The preprocessor collapses all lines which end with a backslash (\) character into a single line. Thus:

```
%define THIS_VERY_LONG_MACRO_NAME_IS_DEFINED_TO \
        THIS_VALUE
```

will work like a single−line macro without the backslash−newline sequence.

4.1 Single−Line 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 9.1 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 4.1.7).

You can pre−define single−line macros using the '−d' option on the NASM command line: see section 2.1.18.

4.1.2 Resolving `%define`: `%xdefine`

To have a reference to an embedded single−line macro resolved at the time that the embedding macro is *defined*, as opposed to when the embedding macro is *expanded*, you need a different mechanism to the one offered by `%define`. The solution is to use `%xdefine`, or it's case−insensitive counterpart `%ixdefine`.

Suppose you have the following code:

```
%define  isTrue  1
%define  isFalse isTrue
%define  isTrue  0

val1:    db      isFalse

%define  isTrue  1

val2:    db      isFalse
```

In this case, `val1` is equal to 0, and `val2` is equal to 1. This is because, when a single−line macro is defined using `%define`, it is expanded only when it is called. As `isFalse` expands to `isTrue`, the expansion will be the current value of `isTrue`. The first time it is called that is 0, and the second time it is 1.

If you wanted `isFalse` to expand to the value assigned to the embedded macro `isTrue` at the time that `isFalse` was defined, you need to change the above code to use `%xdefine`.

```
%xdefine isTrue  1
%xdefine isFalse isTrue
%xdefine isTrue  0

val1:    db      isFalse

%xdefine isTrue  1

val2:    db      isFalse
```

Now, each time that `isFalse` is called, it expands to 1, as that is what the embedded macro `isTrue` expanded to at the time that `isFalse` was defined.

4.1.3 Macro Indirection: %[. . .]

The `%[...]` construct can be used to expand macros in contexts where macro expansion would otherwise not occur, including in the names other macros. For example, if you have a set of macros named `Foo16`, `Foo32` and `Foo64`, you could write:

```
     mov ax,Foo%[__BITS__]    ; The Foo value
```

to use the builtin macro `__BITS__` (see section 4.11.5) to automatically select between them. Similarly, the two statements:

```
%xdefine Bar          Quux    ; Expands due to %xdefine
%define  Bar          %[Quux] ; Expands due to %[...]
```

have, in fact, exactly the same effect.

`%[...]` concatenates to adjacent tokens in the same way that multi−line macro parameters do, see section 4.3.8 for details.

4.1.4 Concatenating Single Line Macro Tokens: %+

Individual tokens in single line macros can be concatenated, to produce longer tokens for later processing. This can be useful if there are several similar macros that perform similar functions.

Please note that a space is required after `%+`, in order to disambiguate it from the syntax `%+1` used in multiline macros.

As an example, consider the following:

```
%define BDASTART 400h                ; Start of BIOS data area

struc   tBIOSDA                      ; its structure
        .COM1addr       RESW    1
        .COM2addr       RESW    1
        ; ..and so on
endstruc
```

Now, if we need to access the elements of tBIOSDA in different places, we can end up with:

```
        mov     ax,BDASTART + tBIOSDA.COM1addr
        mov     bx,BDASTART + tBIOSDA.COM2addr
```

This will become pretty ugly (and tedious) if used in many places, and can be reduced in size significantly by using the following macro:

```
; Macro to access BIOS variables by their names (from tBDA):

%define BDA(x)  BDASTART + tBIOSDA. %+ x
```

Now the above code can be written as:

```
        mov     ax,BDA(COM1addr)
        mov     bx,BDA(COM2addr)
```

Using this feature, we can simplify references to a lot of macros (and, in turn, reduce typing errors).

4.1.5 The Macro Name Itself: %? and %??

The special symbols `%?` and `%??` can be used to reference the macro name itself inside a macro expansion, this is supported for both single−and multi−line macros. `%?` refers to the macro name as *invoked*, whereas

`%??` refers to the macro name as *declared*. The two are always the same for case−sensitive macros, but for case−insensitive macros, they can differ.

For example:

```
%idefine Foo mov %?,%??

        foo
        FOO
```

will expand to:

```
        mov foo,Foo
        mov FOO,Foo
```

The sequence:

```
%idefine keyword $%?
```

can be used to make a keyword "disappear", for example in case a new instruction has been used as a label in older code. For example:

```
%idefine pause $%?                  ; Hide the PAUSE instruction
```

4.1.6 Undefining Single−Line Macros: `%undef`

Single−line macros can be removed with the `%undef` directive. For example, the following sequence:

```
%define foo bar
%undef  foo

        mov     eax, foo
```

will expand to the instruction `mov eax, foo`, since after `%undef` the macro `foo` is no longer defined.

Macros that would otherwise be pre−defined can be undefined on the command−line using the '−u' option on the NASM command line: see section 2.1.19.

4.1.7 Preprocessor Variables: `%assign`

An alternative way to define single−line macros is by means of the `%assign` command (and its case−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 4.5 for an example of this. Another use for `%assign` is given in section 8.4 and section 9.1.

The expression passed to `%assign` is a critical expression (see section 3.8), 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).

4.1.8 Defining Strings: %defstr

%defstr, and its case–insensitive counterpart %idefstr, define or redefine a single–line macro without parameters but converts the entire right–hand side, after macro expansion, to a quoted string before definition.

For example:

```
%defstr test TEST
```

is equivalent to

```
%define test 'TEST'
```

This can be used, for example, with the %! construct (see section 4.10.2):

```
%defstr PATH %!PATH          ; The operating system PATH variable
```

4.1.9 Defining Tokens: %deftok

%deftok, and its case–insensitive counterpart %ideftok, define or redefine a single–line macro without parameters but converts the second parameter, after string conversion, to a sequence of tokens.

For example:

```
%deftok test 'TEST'
```

is equivalent to

```
%define test TEST
```

4.2 String Manipulation in Macros

It's often useful to be able to handle strings in macros. NASM supports a few simple string handling macro operators from which more complex operations can be constructed.

All the string operators define or redefine a value (either a string or a numeric value) to a single–line macro. When producing a string value, it may change the style of quoting of the input string or strings, and possibly use \–escapes inside `–quoted strings.

4.2.1 Concatenating Strings: %strcat

The %strcat operator concatenates quoted strings and assign them to a single–line macro.

For example:

```
%strcat alpha "Alpha: ", '12" screen'
```

... would assign the value 'Alpha: 12" screen' to alpha. Similarly:

```
%strcat beta '"foo"\', "'bar'"
```

... would assign the value `'"foo"\\'bar'` to beta.

The use of commas to separate strings is permitted but optional.

4.2.2 String Length: %strlen

The %strlen operator assigns the length of a string to a macro. For example:

```
%strlen charcnt 'my string'
```

In this example, charcnt would receive the value 9, just as if an %assign had been used. In this example, 'my string' was a literal string but it could also have been a single–line macro that expands to a string, as in the following example:

```
%define sometext 'my string'
%strlen charcnt sometext
```

As in the first case, this would result in `charcnt` being assigned the value of 9.

4.2.3 Extracting Substrings: `%substr`

Individual letters or substrings in strings can be extracted using the `%substr` operator. An example of its use is probably more useful than the description:

```
%substr mychar 'xyzw' 1       ; equivalent to %define mychar 'x'
%substr mychar 'xyzw' 2       ; equivalent to %define mychar 'y'
%substr mychar 'xyzw' 3       ; equivalent to %define mychar 'z'
%substr mychar 'xyzw' 2,2     ; equivalent to %define mychar 'yz'
%substr mychar 'xyzw' 2,-1    ; equivalent to %define mychar 'yzw'
%substr mychar 'xyzw' 2,-2    ; equivalent to %define mychar 'yz'
```

As with `%strlen` (see section 4.2.2), the first parameter is the single−line macro to be created and the second is the string. The third parameter specifies the first character to be selected, and the optional fourth parameter preceeded by comma) is the length. Note that the first index is 1, not 0 and the last index is equal to the value that `%strlen` would assign given the same string. Index values out of range result in an empty string. A negative length means "until N−1 characters before the end of string", i.e. `-1` means until end of string, `-2` until one character before, etc.

4.3 Multi−Line 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 case−sensitive, 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
```

4.3.1 Overloading Multi–Line 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 `-w-macro-params` command–line option (see section 2.1.24).

4.3.2 Macro–Local 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 3.9. 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.

4.3.3 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 6.3.1 for a better way to write the above macro.

4.3.4 Macro Parameters Range

NASM also allows you to expand parameters via special construction `%{x:y}` where `x` is the first parameter index and `y` is the last. Any index can be either negative or positive. Though the indices must never be zero.

For example

```
%macro mpar 1-*
     db %{3:5}
%endmacro

mpar 1,2,3,4,5,6
```

expands to `3,4,5` range.

Even more, the parameters can be reversed so that

```
%macro mpar 1-*
     db %{5:3}
%endmacro

mpar 1,2,3,4,5,6
```

expands to `5,4,3` range.

But even this is not the last. The parameters can be addressed via negative indices so NASM will count them reversed. The ones who know Python may see the analogue here.

```
%macro mpar 1-*
     db %{-1:-3}
%endmacro

mpar 1,2,3,4,5,6
```

expands to `6,5,4` range.

Note that NASM uses comma to separate parameters being expanded.

By the way, here is a trick – you might use the index `%{-1:-1}` gives you the last argument passed to a macro.

4.3.5 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 4.3.3) 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 can provide extra information to a macro by providing too many default parameters:

```
%macro quux 1 something
```

This will trigger a warning by default; see section 2.1.24 for more information. When quux is invoked, it receives not one but two parameters. something can be referred to as %2. The difference between passing something this way and writing something in the macro body is that with this way something is evaluated when the macro is defined, not when it is expanded.

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 4.3.6) 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 4.3.7.

4.3.6 %0: Macro Parameter Counter

The parameter reference %0 will return a numeric constant giving the number of parameters received, that is, if %0 is n then %n is the last parameter. %0 is mostly useful for macros that can take a variable number of parameters. It can be used as an argument to %rep (see section 4.5) in order to iterate through all the parameters of a macro. Examples are given in section 4.3.7.

4.3.7 %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.

4.3.8 Concatenating Macro Parameters

NASM can concatenate macro parameters and macro indirection constructs 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 4.3.2) and context–local labels (section 4.7.2). 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.)

The single–line macro indirection construct, `%[...]` (section 4.1.3), behaves the same way as macro parameters for the purpose of concatenation.

See also the `%+` operator, section 4.1.4.

4.3.9 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 4.3.2 can be replaced by a general conditional–return 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.

4.3.10 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
```

4.3.11 Undefining Multi-Line Macros: %unmacro

Multi-line macros can be removed with the %unmacro directive. Unlike the %undef directive, however, %unmacro takes an argument specification, and will only remove exact matches with that argument specification.

For example:

```
%macro foo 1-3
        ; Do something
%endmacro
%unmacro foo 1-3
```

removes the previously defined macro foo, but

```
%macro bar 1-3
        ; Do something
%endmacro
%unmacro bar 1
```

does *not* remove the macro bar, since the argument specification does not match exactly.

4.4 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 inverse forms %ifn and %elifn are also supported.

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

There are a number of variants of the %if directive. Each has its corresponding %elif, %ifn, and %elifn directives; for example, the equivalents to the %ifdef directive are %elifdef, %ifndef, and %elifndef.

4.4.1 %ifdef: Testing Single-Line 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`.

4.4.2 `%ifmacro`: Testing Multi–Line Macro Existence

The `%ifmacro` directive operates in the same way as the `%ifdef` directive, except that it checks for the existence of a multi–line macro.

For example, you may be working with a large project and not have control over the macros in a library. You may want to create a macro with one name if it doesn't already exist, and another name if one with that name does exist.

The `%ifmacro` is considered true if defining a macro with the given name and number of arguments would cause a definitions conflict. For example:

```
%ifmacro MyMacro 1-3

     %error "MyMacro 1-3" causes a conflict with an existing macro.

%else

     %macro MyMacro 1-3

             ; insert code to define the macro

     %endmacro

%endif
```

This will create the macro "MyMacro 1–3" if no macro already exists which would conflict with it, and emits a warning if there would be a definition conflict.

You can test for the macro not existing by using the `%ifnmacro` instead of `%ifmacro`. Additional tests can be performed in `%elif` blocks by using `%elifmacro` and `%elifnmacro`.

4.4.3 `%ifctx`: Testing the Context Stack

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

For more details of the context stack, see section 4.7. For a sample use of `%ifctx`, see section 4.7.5.

4.4.4 `%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 4.5 for a detailed example.

The expression given to `%if`, and its counterpart `%elif`, is a critical expression (see section 3.8).

`%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.

Like other `%if` constructs, `%if` has a counterpart `%elif`, and negative forms `%ifn` and `%elifn`.

4.4.5 `%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 case–insensitive.

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

4.4.6 `%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 4.3.3 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 `%elif`..., `%ifn`..., and `%elifn`... versions exist for each of `%ifid`, `%ifnum` and `%ifstr`.

4.4.7 `%iftoken`: Test for a Single Token

Some macros will want to do different things depending on if it is passed a single token (e.g. paste it to something else using `%+`) versus a multi−token sequence.

The conditional assembly construct `%iftoken` assembles the subsequent code if and only if the expanded parameters consist of exactly one token, possibly surrounded by whitespace.

For example:

```
%iftoken 1
```

will assemble the subsequent code, but

```
%iftoken -1
```

will not, since `-1` contains two tokens: the unary minus operator `-`, and the number `1`.

The usual `%eliftoken`, `%ifntoken`, and `%elifntoken` variants are also provided.

4.4.8 `%ifempty`: Test for Empty Expansion

The conditional assembly construct `%ifempty` assembles the subsequent code if and only if the expanded parameters do not contain any tokens at all, whitespace excepted.

The usual `%elifempty`, `%ifnempty`, and `%elifnempty` variants are also provided.

4.5 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.

4.6 Source Files and Dependencies

These commands allow you to split your sources into multiple files.

4.6.1 `%include`: 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 2.1.17).

4.6.2 `%pathsearch`: Search the Include Path

The `%pathsearch` directive takes a single−line macro name and a filename, and declare or redefines the specified single−line macro to be the include−path−resolved version of the filename, if the file exists (otherwise, it is passed unchanged.)

For example,

```
%pathsearch MyFoo "foo.bin"
```

... with `-Ibins/` in the include path may end up defining the macro `MyFoo` to be `"bins/foo.bin"`.

4.6.3 %depend: Add Dependent Files

The `%depend` directive takes a filename and adds it to the list of files to be emitted as dependency generation when the `-M` options and its relatives (see section 2.1.4) are used. It produces no output.

This is generally used in conjunction with `%pathsearch`. For example, a simplified version of the standard macro wrapper for the `INCBIN` directive looks like:

```
%imacro incbin 1-2+ 0
%pathsearch dep %1
%depend dep
        incbin dep,%2
%endmacro
```

This first resolves the location of the file into the macro `dep`, then adds it to the dependency lists, and finally issues the assembler-level `INCBIN` directive.

4.6.4 %use: Include Standard Macro Package

The `%use` directive is similar to `%include`, but rather than including the contents of a file, it includes a named standard macro package. The standard macro packages are part of NASM, and are described in chapter 5.

Unlike the `%include` directive, package names for the `%use` directive do not require quotes, but quotes are permitted. In NASM 2.04 and 2.05 the unquoted form would be macro-expanded; this is no longer true. Thus, the following lines are equivalent:

```
%use altreg
%use 'altreg'
```

Standard macro packages are protected from multiple inclusion. When a standard macro package is used, a testable single-line macro of the form `__USE_`*package*`__` is also defined, see section 4.11.8.

4.7 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 characterized 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.

4.7.1 %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` takes an optional 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. If no name is given, the context is unnamed (this is normally used when both the `%push` and the `%pop` are inside a single macro definition.)

The directive `%pop`, taking one optional argument, removes the top context from the context stack and destroys it, along with any labels associated with it. If an argument is given, it must match the name of the current context, otherwise it will issue an error.

4.7.2 Context–Local 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.

4.7.3 Context–Local Single–Line 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`.

4.7.4 `%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`.

4.7.5 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 4.7.2, 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`.

4.8 Stack Relative Preprocessor Directives

The following preprocessor directives provide a way to use labels to refer to local variables allocated on the stack.

- `%arg` (see section 4.8.1)
- `%stacksize` (see section 4.8.2)
- `%local` (see section 4.8.3)

4.8.1 `%arg` Directive

The `%arg` directive is used to simplify the handling of parameters passed on the stack. Stack based parameter passing is used by many high level languages, including C, C++ and Pascal.

While NASM has macros which attempt to duplicate this functionality (see section 8.4.5), the syntax is not particularly convenient to use and is not TASM compatible. Here is an example which shows the use of `%arg` without any external macros:

```
some_function:

    %push     mycontext        ; save the current context
    %stacksize large           ; tell NASM to use bp
    %arg      i:word, j_ptr:word

        mov     ax,[i]
        mov     bx,[j_ptr]
        add     ax,[bx]
        ret

    %pop                       ; restore original context
```

This is similar to the procedure defined in section 8.4.5 and adds the value in i to the value pointed to by j_ptr and returns the sum in the ax register. See section 4.7.1 for an explanation of `push` and `pop` and the use of context stacks.

4.8.2 `%stacksize` Directive

The `%stacksize` directive is used in conjunction with the `%arg` (see section 4.8.1) and the `%local` (see section 4.8.3) directives. It tells NASM the default size to use for subsequent `%arg` and `%local` directives. The `%stacksize` directive takes one required argument which is one of `flat`, `flat64`, `large` or `small`.

```
%stacksize flat
```

This form causes NASM to use stack–based parameter addressing relative to `ebp` and it assumes that a near form of call was used to get to this label (i.e. that `eip` is on the stack).

```
%stacksize flat64
```

This form causes NASM to use stack–based parameter addressing relative to `rbp` and it assumes that a near form of call was used to get to this label (i.e. that `rip` is on the stack).

```
%stacksize large
```

This form uses `bp` to do stack–based parameter addressing and assumes that a far form of call was used to get to this address (i.e. that `ip` and `cs` are on the stack).

```
%stacksize small
```

This form also uses `bp` to address stack parameters, but it is different from `large` because it also assumes that the old value of bp is pushed onto the stack (i.e. it expects an `ENTER` instruction). In other words, it expects that `bp`, `ip` and `cs` are on the top of the stack, underneath any local space which may have been allocated by `ENTER`. This form is probably most useful when used in combination with the `%local` directive (see section 4.8.3).

4.8.3 `%local` Directive

The `%local` directive is used to simplify the use of local temporary stack variables allocated in a stack frame. Automatic local variables in C are an example of this kind of variable. The `%local` directive is most useful when used with the `%stacksize` (see section 4.8.2 and is also compatible with the `%arg` directive (see section 4.8.1). It allows simplified reference to variables on the stack which have been allocated typically by using the `ENTER` instruction. An example of its use is the following:

```
silly_swap:

    %push mycontext             ; save the current context
    %stacksize small            ; tell NASM to use bp
    %assign %$localsize 0       ; see text for explanation
    %local old_ax:word, old_dx:word

        enter   %$localsize,0   ; see text for explanation
        mov     [old_ax],ax     ; swap ax & bx
        mov     [old_dx],dx     ; and swap dx & cx
        mov     ax,bx
        mov     dx,cx
        mov     bx,[old_ax]
        mov     cx,[old_dx]
        leave                   ; restore old bp
        ret                     ;

    %pop                        ; restore original context
```

The `%$localsize` variable is used internally by the `%local` directive and *must* be defined within the current context before the `%local` directive may be used. Failure to do so will result in one expression

syntax error for each `%local` variable declared. It then may be used in the construction of an appropriately sized ENTER instruction as shown in the example.

4.9 Reporting User–Defined Errors: `%error`, `%warning`, `%fatal`

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 F1
    ; do some setup
%elifdef F2
    ; do some different setup
%else
    %error "Neither F1 nor F2 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.

Similarly, `%warning` issues a warning, but allows assembly to continue:

```
%ifdef F1
    ; do some setup
%elifdef F2
    ; do some different setup
%else
    %warning "Neither F1 nor F2 was defined, assuming F1."
    %define F1
%endif
```

`%error` and `%warning` are issued only on the final assembly pass. This makes them safe to use in conjunction with tests that depend on symbol values.

`%fatal` terminates assembly immediately, regardless of pass. This is useful when there is no point in continuing the assembly further, and doing so is likely just going to cause a spew of confusing error messages.

It is optional for the message string after `%error`, `%warning` or `%fatal` to be quoted. If it is *not*, then single–line macros are expanded in it, which can be used to display more information to the user. For example:

```
%if foo > 64
    %assign foo_over foo-64
    %error foo is foo_over bytes too large
%endif
```

4.10 Other Preprocessor Directives

NASM also has preprocessor directives which allow access to information from external sources. Currently they include:

- `%line` enables NASM to correctly handle the output of another preprocessor (see section 4.10.1).
- `%!` enables NASM to read in the value of an environment variable, which can then be used in your program (see section 4.10.2).

4.10.1 `%line` Directive

The `%line` directive is used to notify NASM that the input line corresponds to a specific line number in another file. Typically this other file would be an original source file, with the current NASM input being the output of a pre−processor. The `%line` directive allows NASM to output messages which indicate the line number of the original source file, instead of the file that is being read by NASM.

This preprocessor directive is not generally of use to programmers, by may be of interest to preprocessor authors. The usage of the `%line` preprocessor directive is as follows:

```
%line nnn[+mmm] [filename]
```

In this directive, `nnn` identifies the line of the original source file which this line corresponds to. `mmm` is an optional parameter which specifies a line increment value; each line of the input file read in is considered to correspond to `mmm` lines of the original source file. Finally, `filename` is an optional parameter which specifies the file name of the original source file.

After reading a `%line` preprocessor directive, NASM will report all file name and line numbers relative to the values specified therein.

4.10.2 `%!<env>`: Read an environment variable.

The `%!<env>` directive makes it possible to read the value of an environment variable at assembly time. This could, for example, be used to store the contents of an environment variable into a string, which could be used at some other point in your code.

For example, suppose that you have an environment variable `FOO`, and you want the contents of `FOO` to be embedded in your program. You could do that as follows:

```
%defstr FOO     %!FOO
```

See section 4.1.8 for notes on the `%defstr` directive.

4.11 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 but context−local preprocessor variables and single−line macros.

Most user−level assembler directives (see chapter 6) are implemented as macros which invoke primitive directives; these are described in chapter 6. The rest of the standard macro set is described here.

4.11.1 NASM Version Macros

The single−line macros `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `____NASM_PATCHLEVEL__` expand to the major, minor, subminor and patch level parts of the version number of NASM being used. So, under NASM 0.98.32p1 for example, `__NASM_MAJOR__` would be defined to be 0, `__NASM_MINOR__` would be defined as 98, `__NASM_SUBMINOR__` would be defined to 32, and `____NASM_PATCHLEVEL__` would be defined as 1.

Additionally, the macro `__NASM_SNAPSHOT__` is defined for automatically generated snapshot releases *only*.

4.11.2 `__NASM_VERSION_ID__`: NASM Version ID

The single−line macro `__NASM_VERSION_ID__` expands to a dword integer representing the full version number of the version of nasm being used. The value is the equivalent to `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `____NASM_PATCHLEVEL__` concatenated to produce a single doubleword. Hence, for 0.98.32p1, the returned number would be equivalent to:

```
        dd      0x00622001
```

or

```
        db      1,32,98,0
```

Note that the above lines are generate exactly the same code, the second line is used just to give an indication of the order that the separate values will be present in memory.

4.11.3 __NASM_VER__: NASM Version string

The single−line macro __NASM_VER__ expands to a string which defines the version number of nasm being used. So, under NASM 0.98.32 for example,

```
        db      __NASM_VER__
```

would expand to

```
        db      "0.98.32"
```

4.11.4 __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.

4.11.5 __BITS__: Current BITS Mode

The __BITS__ standard macro is updated every time that the BITS mode is set using the `BITS XX` or `[BITS XX]` directive, where XX is a valid mode number of 16, 32 or 64. __BITS__ receives the specified mode number and makes it globally available. This can be very useful for those who utilize mode−dependent macros.

4.11.6 __OUTPUT_FORMAT__: Current Output Format

The __OUTPUT_FORMAT__ standard macro holds the current Output Format, as given by the `-f` option or NASM's default. Type `nasm -hf` for a list.

```
%ifidn __OUTPUT_FORMAT__, win32
 %define NEWLINE 13, 10
%elifidn __OUTPUT_FORMAT__, elf32
 %define NEWLINE 10
%endif
```

4.11.7 Assembly Date and Time Macros

NASM provides a variety of macros that represent the timestamp of the assembly session.

- The __DATE__ and __TIME__ macros give the assembly date and time as strings, in ISO 8601 format ("YYYY-MM-DD" and "HH:MM:SS", respectively.)
- The __DATE_NUM__ and __TIME_NUM__ macros give the assembly date and time in numeric form; in the format YYYYMMDD and HHMMSS respectively.
- The __UTC_DATE__ and __UTC_TIME__ macros give the assembly date and time in universal time (UTC) as strings, in ISO 8601 format ("YYYY-MM-DD" and "HH:MM:SS", respectively.) If the host platform doesn't provide UTC time, these macros are undefined.
- The __UTC_DATE_NUM__ and __UTC_TIME_NUM__ macros give the assembly date and time universal time (UTC) in numeric form; in the format YYYYMMDD and HHMMSS respectively. If the host platform doesn't provide UTC time, these macros are undefined.
- The __POSIX_TIME__ macro is defined as a number containing the number of seconds since the POSIX epoch, 1 January 1970 00:00:00 UTC; excluding any leap seconds. This is computed using UTC time if available on the host platform, otherwise it is computed using the local time as if it was UTC.

All instances of time and date macros in the same assembly session produce consistent output. For example, in an assembly session started at 42 seconds after midnight on January 1, 2010 in Moscow (timezone UTC+3) these macros would have the following values, assuming, of course, a properly configured environment with a correct clock:

```
__DATE__              "2010-01-01"
__TIME__              "00:00:42"
__DATE_NUM__          20100101
__TIME_NUM__          000042
__UTC_DATE__          "2009-12-31"
__UTC_TIME__          "21:00:42"
__UTC_DATE_NUM__      20091231
__UTC_TIME_NUM__      210042
__POSIX_TIME__        1262293242
```

4.11.8 __USE_*package*__: Package Include Test

When a standard macro package (see chapter 5) is included with the %use directive (see section 4.6.4), a single-line macro of the form __USE_*package*__ is automatically defined. This allows testing if a particular package is invoked or not.

For example, if the altreg package is included (see section 5.1), then the macro __USE_ALTREG__ is defined.

4.11.9 __PASS__: Assembly Pass

The macro __PASS__ is defined to be 1 on preparatory passes, and 2 on the final pass. In preprocess-only mode, it is set to 3, and when running only to generate dependencies (due to the -M or -MG option, see section 2.1.4) it is set to 0.

Avoid using this macro if at all possible. It is tremendously easy to generate very strange errors by misusing it, and the semantics may change in future versions of NASM.

4.11.10 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 or two parameters. The first parameter is the name of the data type. The second, optional parameter is the base offset of the structure. The name of the data type is defined as a symbol with the value of the base offset, and the name of the data type with the suffix _size appended to it is 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 by default 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].

Sometimes you only have the address of the structure displaced by an offset. For example, consider this standard stack frame setup:

```
push ebp
mov ebp, esp
sub esp, 40
```

In this case, you could access an element by subtracting the offset:

```
mov [ebp - 40 + mytype.word], ax
```

However, if you do not want to repeat this offset, you can use –40 as a base offset:

```
struc mytype, -40
```

And access an element this way:

```
mov [ebp + mytype.word], ax
```

4.11.11 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
```

4.11.12 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.

See also the smartalign standard macro package, section 5.2.

Chapter 5: Standard Macro Packages

The %use directive (see section 4.6.4) includes one of the standard macro packages included with the NASM distribution and compiled into the NASM binary. It operates like the %include directive (see section 4.6.1), but the included contents is provided by NASM itself.

The names of standard macro packages are case insensitive, and can be quoted or not.

5.1 altreg: Alternate Register Names

The altreg standard macro package provides alternate register names. It provides numeric register names for all registers (not just R8–R15), the Intel–defined aliases R8L–R15L for the low bytes of register (as opposed to the NASM/AMD standard names R8B–R15B), and the names R0H–R3H (by analogy with R0L–R3L) for AH, CH, DH, and BH.

Example use:

```
%use altreg

proc:
      mov r0l,r3h                     ; mov al,bh
      ret
```

See also section 11.1.

5.2 smartalign: Smart ALIGN Macro

The smartalign standard macro package provides for an ALIGN macro which is more powerful than the default (and backwards–compatible) one (see section 4.11.12). When the smartalign package is enabled, when ALIGN is used without a second argument, NASM will generate a sequence of instructions more efficient than a series of NOP. Furthermore, if the padding exceeds a specific threshold, then NASM will generate a jump over the entire padding sequence.

The specific instructions generated can be controlled with the new ALIGNMODE macro. This macro takes two parameters: one mode, and an optional jump threshold override. If (for any reason) you need to turn off the jump completely just set jump threshold value to –1. The following modes are possible:

- generic: Works on all x86 CPUs and should have reasonable performance. The default jump threshold is 8. This is the default.
- nop: Pad out with NOP instructions. The only difference compared to the standard ALIGN macro is that NASM can still jump over a large padding area. The default jump threshold is 16.
- k7: Optimize for the AMD K7 (Athlon/Althon XP). These instructions should still work on all x86 CPUs. The default jump threshold is 16.
- k8: Optimize for the AMD K8 (Opteron/Althon 64). These instructions should still work on all x86 CPUs. The default jump threshold is 16.
- p6: Optimize for Intel CPUs. This uses the long NOP instructions first introduced in Pentium Pro. This is incompatible with all CPUs of family 5 or lower, as well as some VIA CPUs and several virtualization solutions. The default jump threshold is 16.

The macro __ALIGNMODE__ is defined to contain the current alignment mode. A number of other macros beginning with __ALIGN_ are used internally by this macro package.

Chapter 6: 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: *user−level* directives and *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 *format−specific* directives are documented along with the formats that implement them, in chapter 7.

6.1 `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, 32−bit mode or 64−bit mode. The syntax is `BITS XX`, where XX is 16, 32 or 64.

In most cases, you should not need to use `BITS` explicitly. The `aout`, `coff`, `elf`, `macho`, `win32` and `win64` object formats, which are designed for use in 32−bit or 64−bit operating systems, all cause NASM to select 32−bit or 64−bit mode, respectively, 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 or 64−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` mode, 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` mode, the reverse is true: 32−bit instructions require no prefixes, whereas instructions using 16−bit data need an 0x66 and those working on 16−bit addresses need an 0x67.

When NASM is in `BITS 64` mode, most instructions operate the same as they do for `BITS 32` mode. However, there are 8 more general and SSE registers, and 16−bit addressing is no longer supported.

The default address size is 64 bits; 32−bit addressing can be selected with the 0x67 prefix. The default operand size is still 32 bits, however, and the 0x66 prefix selects 16−bit operand size. The `REX` prefix is used both to select 64−bit operand size, and to access the new registers. NASM automatically inserts REX prefixes when necessary.

When the `REX` prefix is used, the processor does not know how to address the AH, BH, CH or DH (high 8−bit legacy) registers. Instead, it is possible to access the the low 8−bits of the SP, BP SI and DI registers as SPL, BPL, SIL and DIL, respectively; but only when the REX prefix is used.

The `BITS` directive has an exactly equivalent primitive form, `[BITS 16]`, `[BITS 32]` and `[BITS 64]`. The user−level form is a macro which has no function other than to call the primitive form.

Note that the space is neccessary, e.g. `BITS32` will *not* work!

6.1.1 USE16 & USE32: Aliases for BITS

The 'USE16' and 'USE32' directives can be used in place of 'BITS 16' and 'BITS 32', for compatibility with other assemblers.

6.2 DEFAULT: Change the assembler defaults

The DEFAULT directive changes the assembler defaults. Normally, NASM defaults to a mode where the programmer is expected to explicitly specify most features directly. However, this is occationally obnoxious, as the explicit form is pretty much the only one one wishes to use.

Currently, the only DEFAULT that is settable is whether or not registerless instructions in 64–bit mode are RIP–relative or not. By default, they are absolute unless overridden with the REL specifier (see section 3.3). However, if DEFAULT REL is specified, REL is default, unless overridden with the ABS specifier, *except when used with an FS or GS segment override*.

The special handling of FS and GS overrides are due to the fact that these registers are generally used as thread pointers or other special functions in 64–bit mode, and generating RIP–relative addresses would be extremely confusing.

DEFAULT REL is disabled with DEFAULT ABS.

6.3 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 (but see section 7.1.3, all support the standardized section names .text, .data and .bss for the code, data and uninitialized–data sections. The obj format, by contrast, does not recognize these section names as being special, and indeed will strip off the leading period of any section name that has one.

6.3.1 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 4.3.3 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.

6.4 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 3.8) 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.

6.5 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.

6.6 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.

6.7 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 uninitialized 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.

6.8 `CPU`: Defining CPU Dependencies

The `CPU` directive restricts assembly to those instructions which are available on the specified CPU.

Options are:

- `CPU 8086` Assemble only 8086 instruction set
- `CPU 186` Assemble instructions up to the 80186 instruction set
- `CPU 286` Assemble instructions up to the 286 instruction set
- `CPU 386` Assemble instructions up to the 386 instruction set
- `CPU 486` 486 instruction set
- `CPU 586` Pentium instruction set
- `CPU PENTIUM` Same as 586
- `CPU 686` P6 instruction set
- `CPU PPRO` Same as 686
- `CPU P2` Same as 686
- `CPU P3` Pentium III (Katmai) instruction sets
- `CPU KATMAI` Same as P3
- `CPU P4` Pentium 4 (Willamette) instruction set
- `CPU WILLAMETTE` Same as P4
- `CPU PRESCOTT` Prescott instruction set
- `CPU X64` x86−64 (x64/AMD64/Intel 64) instruction set
- `CPU IA64` IA64 CPU (in x86 mode) instruction set

All options are case insensitive. All instructions will be selected only if they apply to the selected CPU or lower. By default, all instructions are available.

6.9 `FLOAT`: Handling of floating−point constants

By default, floating−point constants are rounded to nearest, and IEEE denormals are supported. The following options can be set to alter this behaviour:

- `FLOAT DAZ` Flush denormals to zero
- `FLOAT NODAZ` Do not flush denormals to zero (default)
- `FLOAT NEAR` Round to nearest (default)
- `FLOAT UP` Round up (toward +Infinity)
- `FLOAT DOWN` Round down (toward –Infinity)
- `FLOAT ZERO` Round toward zero
- `FLOAT DEFAULT` Restore default settings

The standard macros `__FLOAT_DAZ__`, `__FLOAT_ROUND__`, and `__FLOAT__` contain the current state, as long as the programmer has avoided the use of the brackeded primitive form, (`[FLOAT]`).

`__FLOAT__` contains the full set of floating−point settings; this value can be saved away and invoked later to restore the setting.

Chapter 7: 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.

7.1 `bin`: Flat–Form 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 MS–DOS: `.COM` executables and `.SYS` device drivers are pure binary files. Pure binary output is also useful for operating system and boot loader development.

The `bin` format supports multiple section names. For details of how NASM handles sections in the `bin` format, see section 7.1.3.

Using the `bin` format puts NASM by default into 16–bit mode (see section 6.1). In order to use `bin` to write 32–bit or 64–bit code, such as an OS kernel, you need to explicitly issue the `BITS 32` or `BITS 64` 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`.

7.1.1 `ORG`: Binary File Program Origin

The `bin` format provides an additional directive to the list given in chapter 6: `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 section; it does not permit any of the trickery that MASM's version does. See section 12.1.3 for further comments.

7.1.2 `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.

7.1.3 Multisection Support for the bin Format

The bin format allows the use of multiple sections, of arbitrary names, besides the "known" .text, .data, and .bss names.

- Sections may be designated progbits or nobits. Default is progbits (except .bss, which defaults to nobits, of course).
- Sections can be aligned at a specified boundary following the previous section with align=, or at an arbitrary byte−granular position with start=.
- Sections can be given a virtual start address, which will be used for the calculation of all memory references within that section with vstart=.
- Sections can be ordered using follows=<section> or vfollows=<section> as an alternative to specifying an explicit start address.
- Arguments to org, start, vstart, and align= are critical expressions. See section 3.8. E.g. align=(1 << ALIGN_SHIFT) − ALIGN_SHIFT must be defined before it is used here.
- Any code which comes before an explicit SECTION directive is directed by default into the .text section.
- If an ORG statement is not given, ORG 0 is used by default.
- The .bss section will be placed after the last progbits section, unless start=, vstart=, follows=, or vfollows= has been specified.
- All sections are aligned on dword boundaries, unless a different alignment has been specified.
- Sections may not overlap.
- NASM creates the section.<secname>.start for each section, which may be used in your code.

7.1.4 Map Files

Map files can be generated in −f bin format by means of the [map] option. Map types of all (default), brief, sections, segments, or symbols may be specified. Output may be directed to stdout (default), stderr, or a specified file. E.g. [map symbols myfile.map]. No "user form" exists, the square brackets must be used.

7.2 ith: Intel Hex Output

The ith file format produces Intel hex−format files. Just as the bin format, this is a flat memory image format with no support for relocation or linking. It is usually used with ROM programmers and similar utilities.

All extensions supported by the bin file format is also supported by the ith file format.

ith provides a default output file−name extension of .ith.

7.3 srec: Motorola S−Records Output

The srec file format produces Motorola S−records files. Just as the bin format, this is a flat memory image format with no support for relocation or linking. It is usually used with ROM programmers and similar utilities.

All extensions supported by the bin file format is also supported by the srec file format.

srec provides a default output file−name extension of .srec.

7.4 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
```

7.4.1 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.

7.4.2 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 uninitialized 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.

7.4.3 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.

7.4.4 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
```

7.4.5 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 initialized 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
```

7.4.6 `..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.

7.4.7 `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 7.4.8.

7.4.8 `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 7.4.7). 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
```

7.5 `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 7.4).

`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.

7.5.1 `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 initialized data section, whereas `bss` declares an uninitialized data section.
- `rdata` declares an initialized data section that is readable but not writable. Microsoft compilers use this section to place constants in it.
- `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, 8-byte alignment for rdata 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 .rdata   rdata align=8
section .bss     bss   align=4
```

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

7.5.2 `win32`: Safe Structured Exception Handling

Among other improvements in Windows XP SP2 and Windows Server 2003 Microsoft has introduced concept of "safe structured exception handling." General idea is to collect handlers' entry points in designated read−only table and have alleged entry point verified against this table prior exception control is passed to the handler. In order for an executable module to be equipped with such "safe exception handler table," all object modules on linker command line has to comply with certain criteria. If one single module among them does not, then the table in question is omitted and above mentioned run−time checks will not be performed for application in question. Table omission is by default silent and therefore can be easily overlooked. One can instruct linker to refuse to produce binary without such table by passing `/safeseh` command line option.

Without regard to this run−time check merits it's natural to expect NASM to be capable of generating modules suitable for `/safeseh` linking. From developer's viewpoint the problem is two−fold:

- how to adapt modules not deploying exception handlers of their own;
- how to adapt/develop modules utilizing custom exception handling;

Former can be easily achieved with any NASM version by adding following line to source code:

```
$@feat.00 equ 1
```

As of version 2.03 NASM adds this absolute symbol automatically. If it's not already present to be precise. I.e. if for whatever reason developer would choose to assign another value in source file, it would still be perfectly possible.

Registering custom exception handler on the other hand requires certain "magic." As of version 2.03 additional directive is implemented, `safeseh`, which instructs the assembler to produce appropriately formatted input data for above mentioned "safe exception handler table." Its typical use would be:

```
section .text
extern  _MessageBoxA@16
%if     __NASM_VERSION_ID__ >= 0x02030000
safeseh handler         ; register handler as "safe handler"
%endif
handler:
        push    DWORD 1 ; MB_OKCANCEL
        push    DWORD caption
        push    DWORD text
        push    DWORD 0
        call    _MessageBoxA@16
        sub     eax,1   ; incidentally suits as return value
                        ; for exception handler
        ret
global  _main
_main:
        push    DWORD handler
        push    DWORD [fs:0]
        mov     DWORD [fs:0],esp ; engage exception handler
        xor     eax,eax
        mov     eax,DWORD[eax]   ; cause exception
        pop     DWORD [fs:0]     ; disengage exception handler
        add     esp,4
        ret
text:   db      'OK to rethrow, CANCEL to generate core dump',0
caption:db      'SEGV',0
```

```
section .drectve info
        db      '/defaultlib:user32.lib /defaultlib:msvcrt.lib '
```

As you might imagine, it's perfectly possible to produce .exe binary with "safe exception handler table" and yet engage unregistered exception handler. Indeed, handler is engaged by simply manipulating `[fs:0]` location at run–time, something linker has no power over, run–time that is. It should be explicitly mentioned that such failure to register handler's entry point with `safeseh` directive has undesired side effect at run–time. If exception is raised and unregistered handler is to be executed, the application is abruptly terminated without any notification whatsoever. One can argue that system could at least have logged some kind "non–safe exception handler in x.exe at address n" message in event log, but no, literally no notification is provided and user is left with no clue on what caused application failure.

Finally, all mentions of linker in this paragraph refer to Microsoft linker version 7.x and later. Presence of `@feat.00` symbol and input data for "safe exception handler table" causes no backward incompatibilities and "safeseh" modules generated by NASM 2.03 and later can still be linked by earlier versions or non–Microsoft linkers.

7.6 `win64`: Microsoft Win64 Object Files

The `win64` output format generates Microsoft Win64 object files, which is nearly 100% identical to the `win32` object format (section 7.5) with the exception that it is meant to target 64–bit code and the x86–64 platform altogether. This object file is used exactly the same as the `win32` object format (section 7.5), in NASM, with regard to this exception.

7.6.1 `win64`: Writing Position–Independent Code

While `REL` takes good care of RIP–relative addressing, there is one aspect that is easy to overlook for a Win64 programmer: indirect references. Consider a switch dispatch table:

```
        jmp     QWORD[dsptch+rax*8]
        ...
dsptch: dq      case0
        dq      case1
        ...
```

Even novice Win64 assembler programmer will soon realize that the code is not 64–bit savvy. Most notably linker will refuse to link it with "`'ADDR32' relocation to '.text' invalid without /LARGEADDRESSAWARE:NO`". So [s]he will have to split jmp instruction as following:

```
        lea     rbx,[rel dsptch]
        jmp     QWORD[rbx+rax*8]
```

What happens behind the scene is that effective address in `lea` is encoded relative to instruction pointer, or in perfectly position–independent manner. But this is only part of the problem! Trouble is that in .dll context `caseN` relocations will make their way to the final module and might have to be adjusted at .dll load time. To be specific when it can't be loaded at preferred address. And when this occurs, pages with such relocations will be rendered private to current process, which kind of undermines the idea of sharing .dll. But no worry, it's trivial to fix:

```
        lea     rbx,[rel dsptch]
        add     rbx,QWORD[rbx+rax*8]
        jmp     rbx
        ...
dsptch: dq      case0-dsptch
        dq      case1-dsptch
        ...
```

NASM version 2.03 and later provides another alternative, `wrt ..imagebase` operator, which returns offset from base address of the current image, be it .exe or .dll module, therefore the name. For those acquainted with PE–COFF format base address denotes start of `IMAGE_DOS_HEADER` structure. Here is how to implement switch with these image–relative references:

```
        lea     rbx,[rel dsptch]
        mov     eax,DWORD[rbx+rax*4]
        sub     rbx,dsptch wrt ..imagebase
        add     rbx,rax
        jmp     rbx
        ...
dsptch: dd      case0 wrt ..imagebase
        dd      case1 wrt ..imagebase
```

One can argue that the operator is redundant. Indeed, snippet before last works just fine with any NASM version and is not even Windows specific... The real reason for implementing `wrt ..imagebase` will become apparent in next paragraph.

It should be noted that `wrt ..imagebase` is defined as 32–bit operand only:

```
        dd      label wrt ..imagebase           ; ok
        dq      label wrt ..imagebase           ; bad
        mov     eax,label wrt ..imagebase       ; ok
        mov     rax,label wrt ..imagebase       ; bad
```

7.6.2 `win64`: Structured Exception Handling

Structured exception handing in Win64 is completely different matter from Win32. Upon exception program counter value is noted, and linker–generated table comprising start and end addresses of all the functions [in given executable module] is traversed and compared to the saved program counter. Thus so called `UNWIND_INFO` structure is identified. If it's not found, then offending subroutine is assumed to be "leaf" and just mentioned lookup procedure is attempted for its caller. In Win64 leaf function is such function that does not call any other function *nor* modifies any Win64 non–volatile registers, including stack pointer. The latter ensures that it's possible to identify leaf function's caller by simply pulling the value from the top of the stack.

While majority of subroutines written in assembler are not calling any other function, requirement for non–volatile registers' immutability leaves developer with not more than 7 registers and no stack frame, which is not necessarily what [s]he counted with. Customarily one would meet the requirement by saving non–volatile registers on stack and restoring them upon return, so what can go wrong? If [and only if] an exception is raised at run–time and no `UNWIND_INFO` structure is associated with such "leaf" function, the stack unwind procedure will expect to find caller's return address on the top of stack immediately followed by its frame. Given that developer pushed caller's non–volatile registers on stack, would the value on top point at some code segment or even addressable space? Well, developer can attempt copying caller's return address to the top of stack and this would actually work in some very specific circumstances. But unless developer can guarantee that these circumstances are always met, it's more appropriate to assume worst case scenario, i.e. stack unwind procedure going berserk. Relevant question is what happens then? Application is abruptly terminated without any notification whatsoever. Just like in Win32 case, one can argue that system could at least have logged "unwind procedure went berserk in x.exe at address n" in event log, but no, no trace of failure is left.

Now, when we understand significance of the `UNWIND_INFO` structure, let's discuss what's in it and/or how it's processed. First of all it is checked for presence of reference to custom language–specific exception handler. If there is one, then it's invoked. Depending on the return value, execution flow is resumed (exception is said to be "handled"), *or* rest of `UNWIND_INFO` structure is processed as following. Beside optional reference to custom handler, it carries information about current callee's stack frame and where non–volatile registers are saved. Information is detailed enough to be able to reconstruct contents of caller's non–volatile registers upon call to current callee. And so caller's context is reconstructed, and then unwind

procedure is repeated, i.e. another UNWIND_INFO structure is associated, this time, with caller's instruction pointer, which is then checked for presence of reference to language−specific handler, etc. The procedure is recursively repeated till exception is handled. As last resort system "handles" it by generating memory core dump and terminating the application.

As for the moment of this writing NASM unfortunately does not facilitate generation of above mentioned detailed information about stack frame layout. But as of version 2.03 it implements building blocks for generating structures involved in stack unwinding. As simplest example, here is how to deploy custom exception handler for leaf function:

```
default rel
section .text
extern  MessageBoxA
handler:
        sub     rsp,40
        mov     rcx,0
        lea     rdx,[text]
        lea     r8,[caption]
        mov     r9,1    ; MB_OKCANCEL
        call    MessageBoxA
        sub     eax,1   ; incidentally suits as return value
                        ; for exception handler
        add     rsp,40
        ret
global  main
main:
        xor     rax,rax
        mov     rax,QWORD[rax]  ; cause exception
        ret
main_end:
text:   db      'OK to rethrow, CANCEL to generate core dump',0
caption:db      'SEGV',0

section .pdata  rdata align=4
        dd      main wrt ..imagebase
        dd      main_end wrt ..imagebase
        dd      xmain wrt ..imagebase
section .xdata  rdata align=8
xmain:  db      9,0,0,0
        dd      handler wrt ..imagebase
section .drectve info
        db      '/defaultlib:user32.lib /defaultlib:msvcrt.lib '
```

What you see in .pdata section is element of the "table comprising start and end addresses of function" along with reference to associated UNWIND_INFO structure. And what you see in .xdata section is UNWIND_INFO structure describing function with no frame, but with designated exception handler. References are *required* to be image−relative (which is the real reason for implementing wrt ..imagebase operator). It should be noted that rdata align=n, as well as wrt ..imagebase, are optional in these two segments' contexts, i.e. can be omitted. Latter means that *all* 32−bit references, not only above listed required ones, placed into these two segments turn out image−relative. Why is it important to understand? Developer is allowed to append handler−specific data to UNWIND_INFO structure, and if [s]he adds a 32−bit reference, then [s]he will have to remember to adjust its value to obtain the real pointer.

As already mentioned, in Win64 terms leaf function is one that does not call any other function *nor* modifies any non−volatile register, including stack pointer. But it's not uncommon that assembler programmer plans to utilize every single register and sometimes even have variable stack frame. Is there anything one can do with bare building blocks? I.e. besides manually composing fully−fledged `UNWIND_INFO` structure, which would surely be considered error−prone? Yes, there is. Recall that exception handler is called first, before stack layout is analyzed. As it turned out, it's perfectly possible to manipulate current callee's context in custom handler in manner that permits further stack unwinding. General idea is that handler would not actually "handle" the exception, but instead restore callee's context, as it was at its entry point and thus mimic leaf function. In other words, handler would simply undertake part of unwinding procedure. Consider following example:

```
function:
        mov     rax,rsp         ; copy rsp to volatile register
        push    r15             ; save non-volatile registers
        push    rbx
        push    rbp
        mov     r11,rsp         ; prepare variable stack frame
        sub     r11,rcx
        and     r11,-64
        mov     QWORD[r11],rax  ; check for exceptions
        mov     rsp,r11         ; allocate stack frame
        mov     QWORD[rsp],rax  ; save original rsp value
magic_point:
        ...
        mov     r11,QWORD[rsp]  ; pull original rsp value
        mov     rbp,QWORD[r11-24]
        mov     rbx,QWORD[r11-16]
        mov     r15,QWORD[r11-8]
        mov     rsp,r11         ; destroy frame
        ret
```

The keyword is that up to `magic_point` original `rsp` value remains in chosen volatile register and no non−volatile register, except for `rsp`, is modified. While past `magic_point` `rsp` remains constant till the very end of the `function`. In this case custom language−specific exception handler would look like this:

```
EXCEPTION_DISPOSITION handler (EXCEPTION_RECORD *rec,ULONG64 frame,
        CONTEXT *context,DISPATCHER_CONTEXT *disp)
{   ULONG64 *rsp;
    if (context->Rip<(ULONG64)magic_point)
        rsp = (ULONG64 *)context->Rax;
    else
    {   rsp = ((ULONG64 **)context->Rsp)[0];
        context->Rbp = rsp[-3];
        context->Rbx = rsp[-2];
        context->R15 = rsp[-1];
    }
    context->Rsp = (ULONG64)rsp;

    memcpy (disp->ContextRecord,context,sizeof(CONTEXT));
    RtlVirtualUnwind(UNW_FLAG_NHANDLER,disp->ImageBase,
        dips->ControlPc,disp->FunctionEntry,disp->ContextRecord,
        &disp->HandlerData,&disp->EstablisherFrame,NULL);
    return ExceptionContinueSearch;
}
```

As custom handler mimics leaf function, corresponding UNWIND_INFO structure does not have to contain any information about stack frame and its layout.

7.7 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.

7.8 macho32 and macho64: Mach Object File Format

The macho32 and macho64 output formts produces Mach-O object files suitable for linking with the MacOS X linker. macho is a synonym for macho32.

macho provides a default output file-name extension of .o.

7.9 elf32 and elf64: Executable and Linkable Format Object Files

The elf32 and elf64 output formats generate ELF32 and ELF64 (Executable and Linkable Format) object files, as used by Linux as well as Unix System V, including Solaris x86, UnixWare and SCO Unix. elf provides a default output file-name extension of .o. elf is a synonym for elf32.

7.9.1 ELF specific directive osabi

The ELF header specifies the application binary interface for the target operating system (OSABI). This field can be set by using the osabi directive with the numeric value (0-255) of the target system. If this directive is not used, the default value will be "UNIX System V ABI" (0) which will work on most systems which support ELF.

7.9.2 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, 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.
- tls defines the section to be one which contains thread local variables.

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

```
section .text    progbits  alloc   exec    nowrite  align=16
section .rodata  progbits  alloc   noexec  nowrite  align=4
section .lrodata progbits  alloc   noexec  nowrite  align=4
section .data    progbits  alloc   noexec  write    align=4
section .ldata   progbits  alloc   noexec  write    align=4
section .bss     nobits    alloc   noexec  write    align=4
section .lbss    nobits    alloc   noexec  write    align=4
section .tdata   progbits  alloc   noexec  write    align=4    tls
section .tbss    nobits    alloc   noexec  write    align=4    tls
section .comment progbits  noalloc noexec  nowrite  align=1
section other    progbits  alloc   noexec  nowrite  align=1
```

(Any section name other than those in the above table is treated by default like `other` in the above table. Please note that section names are case sensitive.)

7.9.3 Position–Independent 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 ELF specific 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 summarized 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 9.2.

7.9.4 Thread Local Storage: `elf` Special Symbols and `WRT`

- In ELF32 mode, referring to an external or global symbol using `wrt ..tlsie` causes the linker to build an entry *in* the GOT containing the offset of the symbol within the TLS block, so you can access the value of the symbol with code such as:

```
mov  eax,[tid wrt ..tlsie]
mov  [gs:eax],ebx
```

- In ELF64 mode, referring to an external or global symbol using `wrt ..gottpoff` causes the linker to build an entry *in* the GOT containing the offset of the symbol within the TLS block, so you can access the value of the symbol with code such as:

```
mov   rax,[rel tid wrt ..gottpoff]
mov   rcx,[fs:rax]
```

7.9.5 `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.

Optionally, you can control the ELF visibility of the symbol. Just add one of the visibility keywords: `default`, `internal`, `hidden`, or `protected`. The default is `default` of course. For example, to make `hashlookup` hidden:

```
global   hashlookup:function hidden
```

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 9.2.4.

7.9.6 `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.

7.9.7 16–bit code and ELF

The ELF32 specification doesn't provide relocations for 8– and 16–bit values, but the GNU ld linker adds these as an extension. NASM can generate GNU–compatible relocations, to allow 16–bit code to be linked as ELF using GNU ld. If NASM is used with the -w+gnu-elf-extensions option, a warning is issued when one of these relocations is generated.

7.9.8 Debug formats and ELF

ELF32 and ELF64 provide debug information in STABS and DWARF formats. Line number information is generated for all executable sections, but please note that only the ".text" section is executable by default.

7.10 aout: Linux a.out Object Files

The aout format generates a.out object files, in the form used by early Linux systems (current Linux systems use ELF, see section 7.9.) 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 does not.

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.

7.11 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 position–independent 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 7.9.3 for full documentation of this feature.

aoutb also supports the same extensions to the GLOBAL directive as elf does: see section 7.9.5 for documentation of this.

7.12 as86: Minix/Linux as86 Object Files

The Minix/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 use of SEG or WRT, and no extensions to any standard directives. It supports only the three standard section names .text, .data and .bss. The only special symbol supported is ..start.

7.13 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.

7.13.1 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
```

7.13.2 Specifying a Module Name: The MODULE Directive

Special RDOFF header record is used to store the name of the module. It can be used, for example, by run−time loader to perform dynamic linking. MODULE directive takes one argument which is the name of current module:

```
module  mymodname
```

Note that when you statically link modules and tell linker to strip the symbols from output file, all module names will be stripped too. To avoid it, you should start module names with $, like:

```
module  $kernel.core
```

7.13.3 rdf Extensions to the GLOBAL Directive

RDOFF global symbols can contain additional information needed by the static linker. You can mark a global symbol as exported, thus telling the linker do not strip it from target executable or library file. Like in ELF, you can also specify whether an exported symbol is a procedure (function) or data object.

Suffixing the name with a colon and the word export you make the symbol exported:

```
global  sys_open:export
```

To specify that exported symbol is a procedure (function), you add the word proc or function after declaration:

```
global  sys_open:export proc
```

Similarly, to specify exported data object, add the word data or object to the directive:

```
global  kernel_ticks:export data
```

7.13.4 rdf Extensions to the EXTERN Directive

By default the EXTERN directive in RDOFF declares a "pure external" symbol (i.e. the static linker will complain if such a symbol is not resolved). To declare an "imported" symbol, which must be resolved later during a dynamic linking phase, RDOFF offers an additional import modifier. As in GLOBAL, you can also specify whether an imported symbol is a procedure (function) or data object. For example:

```
library $libc
extern  _open:import
extern  _printf:import proc
extern  _errno:import data
```

Here the directive LIBRARY is also included, which gives the dynamic linker a hint as to where to find requested symbols.

7.14 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 output/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 8: 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.

8.1 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.

8.1.1 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. A fourth linker, ALINK, written by Anthony A.J. Williams, is available at alink.sourceforge.net.

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 7.4.6). 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 initializes 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 uninitialized 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.

8.1.2 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`.

8.2 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.

8.2.1 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 uninitialized 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 (uninitialized 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 initialized 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.

8.2.2 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.

8.3 Producing `.SYS` Files

MS–DOS 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`.

8.4 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 mixed–language program.

8.4.1 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`.

Also see section 2.1.27.

8.4.2 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.

8.4.3 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 8.5.1). 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 initialize 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.

8.4.4 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 8.4.1.) 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 organizes 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.

8.4.5 `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 alternative, TASM compatible form of `arg` is also now built into NASM's preprocessor. See section 4.8 for details.)

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.

8.5 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.

8.5.1 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);
```

8.5.2 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`.
- initialized data must be in a segment whose name is either `CONST` or something ending in `_DATA`.
- Uninitialized 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.

8.5.3 Using `c16.mac` With Pascal Programs

The `c16.mac` macro package, described in section 8.4.5, 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 8.4.5: 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 9: 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.

9.1 Interfacing to 32−bit C Programs

A lot of the discussion in section 8.4, 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.

9.1.1 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 8.4.1, will still work. For `ELF`, though, the leading underscore should not be used.

See also section 2.1.27.

9.1.2 Function Definitions and Function Calls

The 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);
```

9.1.3 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 9.1.1.) 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 organizes 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.

9.1.4 `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.

9.2 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 position−independent 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 9.2.4 for a caveat).

9.2.1 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
```

9.2.2 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.

9.2.3 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.

9.2.4 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.

9.2.5 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`.

9.2.6 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 10: 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.

10.1 Mixed−Size Jumps

The most common form of mixed−size 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.

10.2 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 10.1, 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 10.1, 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.

10.3 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`, `a32` and `a64` 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`, `a32` and `a64` 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`, `SCASx`, `LODSx`, `STOSx`, `MOVSx`, `INSx`, `OUTSx`, and `XLATB`. Also, the various push and pop instructions (`PUSHA` and `POPF` as well as the more usual `PUSH` and `POP`) can accept `a16`, `a32` or `a64` prefixes to force a particular one of `SP`, `ESP` or `RSP` 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 11: Writing 64–bit Code (Unix, Win64)

This chapter attempts to cover some of the common issues involved when writing 64–bit code, to run under Win64 or Unix. It covers how to write assembly code to interface with 64–bit C routines, and how to write position–independent code for shared libraries.

All 64–bit code uses a flat memory model, since segmentation is not available in 64–bit mode. The one exception is the `FS` and `GS` registers, which still add their bases.

Position independence in 64–bit mode is significantly simpler, since the processor supports `RIP`–relative addressing directly; see the `REL` keyword (section 3.3). On most 64–bit platforms, it is probably desirable to make that the default, using the directive `DEFAULT REL` (section 6.2).

64–bit programming is relatively similar to 32–bit programming, but of course pointers are 64 bits long; additionally, all existing platforms pass arguments in registers rather than on the stack. Furthermore, 64–bit platforms use SSE2 by default for floating point. Please see the ABI documentation for your platform.

64–bit platforms differ in the sizes of the fundamental datatypes, not just from 32–bit platforms but from each other. If a specific size data type is desired, it is probably best to use the types defined in the Standard C header `<inttypes.h>`.

In 64–bit mode, the default instruction size is still 32 bits. When loading a value into a 32–bit register (but not an 8– or 16–bit register), the upper 32 bits of the corresponding 64–bit register are set to zero.

11.1 Register Names in 64–bit Mode

NASM uses the following names for general–purpose registers in 64–bit mode, for 8–, 16–, 32– and 64–bit references, respecitively:

```
AL/AH, CL/CH, DL/DH, BL/BH, SPL, BPL, SIL, DIL, R8B-R15B
AX, CX, DX, BX, SP, BP, SI, DI, R8W-R15W
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D-R15D
RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8-R15
```

This is consistent with the AMD documentation and most other assemblers. The Intel documentation, however, uses the names `R8L-R15L` for 8–bit references to the higher registers. It is possible to use those names by definiting them as macros; similarly, if one wants to use numeric names for the low 8 registers, define them as macros. The standard macro package `altreg` (see section 5.1) can be used for this purpose.

11.2 Immediates and Displacements in 64–bit Mode

In 64–bit mode, immediates and displacements are generally only 32 bits wide. NASM will therefore truncate most displacements and immediates to 32 bits.

The only instruction which takes a full 64–bit immediate is:

```
MOV reg64,imm64
```

NASM will produce this instruction whenever the programmer uses `MOV` with an immediate into a 64–bit register. If this is not desirable, simply specify the equivalent 32–bit register, which will be automatically zero–extended by the processor, or specify the immediate as `DWORD`:

```
mov rax,foo             ; 64-bit immediate
mov rax,qword foo       ; (identical)
```

```
mov eax,foo             ; 32-bit immediate, zero-extended
mov rax,dword foo       ; 32-bit immediate, sign-extended
```

The length of these instructions are 10, 5 and 7 bytes, respectively.

The only instructions which take a full 64–bit *displacement* is loading or storing, using `MOV`, `AL`, `AX`, `EAX` or `RAX` (but no other registers) to an absolute 64–bit address. Since this is a relatively rarely used instruction (64–bit code generally uses relative addressing), the programmer has to explicitly declare the displacement size as `QWORD`:

```
default abs

mov eax,[foo]           ; 32-bit absolute disp, sign-extended
mov eax,[a32 foo]       ; 32-bit absolute disp, zero-extended
mov eax,[qword foo]     ; 64-bit absolute disp

default rel

mov eax,[foo]           ; 32-bit relative disp
mov eax,[a32 foo]       ; d:o, address truncated to 32 bits(!)
mov eax,[qword foo]     ; error
mov eax,[abs qword foo] ; 64-bit absolute disp
```

A sign–extended absolute displacement can access from –2 GB to +2 GB; a zero–extended absolute displacement can access from 0 to 4 GB.

11.3 Interfacing to 64–bit C Programs (Unix)

On Unix, the 64–bit ABI is defined by the document:

`http://www.nasm.us/links/unix64abi`

Although written for AT&T–syntax assembly, the concepts apply equally well for NASM–style assembly. What follows is a simplified summary.

The first six integer arguments (from the left) are passed in `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`, in that order. Additional integer arguments are passed on the stack. These registers, plus `RAX`, `R10` and `R11` are destroyed by function calls, and thus are available for use by the function without saving.

Integer return values are passed in `RAX` and `RDX`, in that order.

Floating point is done using SSE registers, except for `long double`. Floating–point arguments are passed in `XMM0` to `XMM7`; return is `XMM0` and `XMM1`. `long double` are passed on the stack, and returned in `ST0` and `ST1`.

All SSE and x87 registers are destroyed by function calls.

On 64–bit Unix, `long` is 64 bits.

Integer and SSE register arguments are counted separately, so for the case of

```
void foo(long a, double b, int c)
```

`a` is passed in `RDI`, `b` in `XMM0`, and `c` in `ESI`.

11.4 Interfacing to 64–bit C Programs (Win64)

The Win64 ABI is described at:

`http://www.nasm.us/links/win64abi`

What follows is a simplified summary.

The first four integer arguments are passed in RCX, RDX, R8 and R9, in that order. Additional integer arguments are passed on the stack. These registers, plus RAX, R10 and R11 are destroyed by function calls, and thus are available for use by the function without saving.

Integer return values are passed in RAX only.

Floating point is done using SSE registers, except for long double. Floating−point arguments are passed in XMM0 to XMM3; return is XMM0 only.

On Win64, long is 32 bits; long long or _int64 is 64 bits.

Integer and SSE register arguments are counted together, so for the case of

```
void foo(long long a, double b, int c)
```

a is passed in RCX, b in XMM1, and c in R8D.

Chapter 12: 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.

12.1 Common Problems

12.1.1 NASM Generates Inefficient Code

We sometimes get '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, it's user error: if you prefer to have NASM produce the more efficient code automatically enable optimization with the `-O` option (see section 2.1.22).

12.1.2 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. See section 2.1.22.

12.1.3 `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.

12.1.4 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.

12.2 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 firstly via the bugtracker at https://sourceforge.net/projects/nasm/ (click on "Bugs"), or if that fails then through one of the contacts in section 1.2.

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 12.1, 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 NASMENV 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: Ndisasm

The Netwide Disassembler, NDISASM

A.1 Introduction

The Netwide Disassembler is a small companion program to the Netwide Assembler, NASM. It seemed a shame to have an x86 assembler, complete with a full instruction table, and not make as much use of it as possible, so here's a disassembler which shares the instruction table (and some other bits of code) with NASM.

The Netwide Disassembler does nothing except to produce disassemblies of *binary* source files. NDISASM does not have any understanding of object file formats, like `objdump`, and it will not understand `DOS .EXE` files like `debug` will. It just disassembles.

A.2 Getting Started: Installation

See section 1.3 for installation instructions. NDISASM, like NASM, has a `man page` which you may want to put somewhere useful, if you are on a Unix system.

A.3 Running NDISASM

To disassemble a file, you will typically use a command of the form

```
ndisasm -b {16|32|64} filename
```

NDISASM can disassemble 16–, 32– or 64–bit code equally easily, provided of course that you remember to specify which it is to work with. If no `-b` switch is present, NDISASM works in 16–bit mode by default. The `-u` switch (for USE32) also invokes 32–bit mode.

Two more command line options are `-r` which reports the version number of NDISASM you are running, and `-h` which gives a short summary of command line options.

A.3.1 COM Files: Specifying an Origin

To disassemble a `DOS .COM` file correctly, a disassembler must assume that the first instruction in the file is loaded at address `0x100`, rather than at zero. NDISASM, which assumes by default that any file you give it is loaded at zero, will therefore need to be informed of this.

The `-o` option allows you to declare a different origin for the file you are disassembling. Its argument may be expressed in any of the NASM numeric formats: decimal by default, if it begins with '`$`' or '`0x`' or ends in '`H`' it's `hex`, if it ends in '`Q`' it's `octal`, and if it ends in '`B`' it's `binary`.

Hence, to disassemble a `.COM` file:

```
ndisasm -o100h filename.com
```

will do the trick.

A.3.2 Code Following Data: Synchronisation

Suppose you are disassembling a file which contains some data which isn't machine code, and *then* contains some machine code. NDISASM will faithfully plough through the data section, producing machine instructions wherever it can (although most of them will look bizarre, and some may have unusual prefixes, e.g. '`FS OR AX,0x240A`'), and generating 'DB' instructions ever so often if it's totally stumped. Then it will reach the code section.

Supposing NDISASM has just finished generating a strange machine instruction from part of the data section, and its file position is now one byte *before* the beginning of the code section. It's entirely possible that another spurious instruction will get generated, starting with the final byte of the data section, and then the correct first instruction in the code section will not be seen because the starting point skipped over it. This isn't really ideal.

To avoid this, you can specify a '`synchronisation`' point, or indeed as many synchronisation points as you like (although NDISASM can only handle 2147483647 sync points internally). The definition of a sync point is this: NDISASM guarantees to hit sync points exactly during disassembly. If it is thinking about generating an instruction which would cause it to jump over a sync point, it will discard that instruction and output a '`db`' instead. So it *will* start disassembly exactly from the sync point, and so you *will* see all the instructions in your code section.

Sync points are specified using the `-s` option: they are measured in terms of the program origin, not the file position. So if you want to synchronize after 32 bytes of a `.COM` file, you would have to do

```
ndisasm -o100h -s120h file.com
```

rather than

```
ndisasm -o100h -s20h file.com
```

As stated above, you can specify multiple sync markers if you need to, just by repeating the `-s` option.

A.3.3 Mixed Code and Data: Automatic (Intelligent) Synchronisation

Suppose you are disassembling the boot sector of a `DOS` floppy (maybe it has a virus, and you need to understand the virus so that you know what kinds of damage it might have done you). Typically, this will contain a `JMP` instruction, then some data, then the rest of the code. So there is a very good chance of NDISASM being *misaligned* when the data ends and the code begins. Hence a sync point is needed.

On the other hand, why should you have to specify the sync point manually? What you'd do in order to find where the sync point would be, surely, would be to read the `JMP` instruction, and then to use its target address as a sync point. So can NDISASM do that for you?

The answer, of course, is yes: using either of the synonymous switches `-a` (for automatic sync) or `-i` (for intelligent sync) will enable `auto-sync` mode. Auto-sync mode automatically generates a sync point for any forward-referring PC-relative jump or call instruction that NDISASM encounters. (Since NDISASM is one-pass, if it encounters a PC-relative jump whose target has already been processed, there isn't much it can do about it...)

Only PC-relative jumps are processed, since an absolute jump is either through a register (in which case NDISASM doesn't know what the register contains) or involves a segment address (in which case the target code isn't in the same segment that NDISASM is working in, and so the sync point can't be placed anywhere useful).

For some kinds of file, this mechanism will automatically put sync points in all the right places, and save you from having to place any sync points manually. However, it should be stressed that auto-sync mode is *not* guaranteed to catch all the sync points, and you may still have to place some manually.

Auto-sync mode doesn't prevent you from declaring manual sync points: it just adds automatically generated ones to the ones you provide. It's perfectly feasible to specify `-i` *and* some `-s` options.

Another caveat with auto-sync mode is that if, by some unpleasant fluke, something in your data section should disassemble to a PC-relative call or jump instruction, NDISASM may obediently place a sync point in a totally random place, for example in the middle of one of the instructions in your code section. So you may end up with a wrong disassembly even if you use auto-sync. Again, there isn't much I can do about this. If you have problems, you'll have to use manual sync points, or use the `-k` option (documented below) to suppress disassembly of the data area.

A.3.4 Other Options

The `-e` option skips a header on the file, by ignoring the first N bytes. This means that the header is *not* counted towards the disassembly offset: if you give `-e10 -o10`, disassembly will start at byte 10 in the file, and this will be given offset 10, not 20.

The `-k` option is provided with two comma-separated numeric arguments, the first of which is an assembly offset and the second is a number of bytes to skip. This *will* count the skipped bytes towards the assembly offset: its use is to suppress disassembly of a data section which wouldn't contain anything you wanted to see anyway.

A.4 Bugs and Improvements

There are no known bugs. However, any you find, with patches if possible, should be sent to `nasm-bugs@lists.sourceforge.net`, or to the developer's site at `https://sourceforge.net/projects/nasm/` and we'll try to fix them. Feel free to send contributions and new features as well.

Appendix B: Instruction List

B.1 Introduction

The following sections show the instructions which NASM currently supports. For each instruction, there is a separate entry for each supported addressing mode. The third column shows the processor type in which the instruction was introduced and, when appropriate, one or more usage flags.

B.1.1 Special instructions...

```
DB
DW
DD
DQ
DT
DO
DY
RESB            imm                     8086
RESW
RESD
RESQ
REST
RESO
RESY
```

B.1.2 Conventional instructions

```
AAA                                     8086,NOLONG
AAD                                     8086,NOLONG
AAD             imm                     8086,NOLONG
AAM                                     8086,NOLONG
AAM             imm                     8086,NOLONG
AAS                                     8086,NOLONG
ADC             mem,reg8                8086
ADC             reg8,reg8               8086
ADC             mem,reg16               8086
ADC             reg16,reg16             8086
ADC             mem,reg32               386
ADC             reg32,reg32             386
ADC             mem,reg64               X64
ADC             reg64,reg64             X64
ADC             reg8,mem                8086
ADC             reg8,reg8               8086
ADC             reg16,mem               8086
ADC             reg16,reg16             8086
ADC             reg32,mem               386
ADC             reg32,reg32             386
ADC             reg64,mem               X64
ADC             reg64,reg64             X64
ADC             rm16,imm8               8086
```

```
ADC             rm32,imm8               386
ADC             rm64,imm8               X64
ADC             reg_al,imm              8086
ADC             reg_ax,sbyte16          8086
ADC             reg_ax,imm              8086
ADC             reg_eax,sbyte32         386
ADC             reg_eax,imm             386
ADC             reg_rax,sbyte64         X64
ADC             reg_rax,imm             X64
ADC             rm8,imm                 8086
ADC             rm16,imm                8086
ADC             rm32,imm                386
ADC             rm64,imm                X64
ADC             mem,imm8                8086
ADC             mem,imm16               8086
ADC             mem,imm32               386
ADD             mem,reg8                8086
ADD             reg8,reg8               8086
ADD             mem,reg16               8086
ADD             reg16,reg16             8086
ADD             mem,reg32               386
ADD             reg32,reg32             386
ADD             mem,reg64               X64
ADD             reg64,reg64             X64
ADD             reg8,mem                8086
ADD             reg8,reg8               8086
ADD             reg16,mem               8086
ADD             reg16,reg16             8086
ADD             reg32,mem               386
ADD             reg32,reg32             386
ADD             reg64,mem               X64
ADD             reg64,reg64             X64
ADD             rm16,imm8               8086
ADD             rm32,imm8               386
ADD             rm64,imm8               X64
ADD             reg_al,imm              8086
ADD             reg_ax,sbyte16          8086
ADD             reg_ax,imm              8086
ADD             reg_eax,sbyte32         386
ADD             reg_eax,imm             386
ADD             reg_rax,sbyte64         X64
ADD             reg_rax,imm             X64
ADD             rm8,imm                 8086
ADD             rm16,imm                8086
ADD             rm32,imm                386
ADD             rm64,imm                X64
ADD             mem,imm8                8086
ADD             mem,imm16               8086
ADD             mem,imm32               386
AND             mem,reg8                8086
AND             reg8,reg8               8086
AND             mem,reg16               8086
AND             reg16,reg16             8086
```

```
AND             mem,reg32               386
AND             reg32,reg32             386
AND             mem,reg64               X64
AND             reg64,reg64             X64
AND             reg8,mem                8086
AND             reg8,reg8               8086
AND             reg16,mem               8086
AND             reg16,reg16             8086
AND             reg32,mem               386
AND             reg32,reg32             386
AND             reg64,mem               X64
AND             reg64,reg64             X64
AND             rm16,imm8               8086
AND             rm32,imm8               386
AND             rm64,imm8               X64
AND             reg_al,imm              8086
AND             reg_ax,sbyte16          8086
AND             reg_ax,imm              8086
AND             reg_eax,sbyte32         386
AND             reg_eax,imm             386
AND             reg_rax,sbyte64         X64
AND             reg_rax,imm             X64
AND             rm8,imm                 8086
AND             rm16,imm                8086
AND             rm32,imm                386
AND             rm64,imm                X64
AND             mem,imm8                8086
AND             mem,imm16               8086
AND             mem,imm32               386
ARPL            mem,reg16               286,PROT,NOLONG
ARPL            reg16,reg16             286,PROT,NOLONG
BB0_RESET                               PENT,CYRIX,ND
BB1_RESET                               PENT,CYRIX,ND
BOUND           reg16,mem               186,NOLONG
BOUND           reg32,mem               386,NOLONG
BSF             reg16,mem               386
BSF             reg16,reg16             386
BSF             reg32,mem               386
BSF             reg32,reg32             386
BSF             reg64,mem               X64
BSF             reg64,reg64             X64
BSR             reg16,mem               386
BSR             reg16,reg16             386
BSR             reg32,mem               386
BSR             reg32,reg32             386
BSR             reg64,mem               X64
BSR             reg64,reg64             X64
BSWAP           reg32                   486
BSWAP           reg64                   X64
BT              mem,reg16               386
BT              reg16,reg16             386
BT              mem,reg32               386
BT              reg32,reg32             386
```

```
BT              mem,reg64               X64
BT              reg64,reg64             X64
BT              rm16,imm                386
BT              rm32,imm                386
BT              rm64,imm                X64
BTC             mem,reg16               386
BTC             reg16,reg16             386
BTC             mem,reg32               386
BTC             reg32,reg32             386
BTC             mem,reg64               X64
BTC             reg64,reg64             X64
BTC             rm16,imm                386
BTC             rm32,imm                386
BTC             rm64,imm                X64
BTR             mem,reg16               386
BTR             reg16,reg16             386
BTR             mem,reg32               386
BTR             reg32,reg32             386
BTR             mem,reg64               X64
BTR             reg64,reg64             X64
BTR             rm16,imm                386
BTR             rm32,imm                386
BTR             rm64,imm                X64
BTS             mem,reg16               386
BTS             reg16,reg16             386
BTS             mem,reg32               386
BTS             reg32,reg32             386
BTS             mem,reg64               X64
BTS             reg64,reg64             X64
BTS             rm16,imm                386
BTS             rm32,imm                386
BTS             rm64,imm                X64
CALL            imm                     8086
CALL            imm|near                8086
CALL            imm|far                 8086,ND,NOLONG
CALL            imm16                   8086
CALL            imm16|near              8086
CALL            imm16|far               8086,ND,NOLONG
CALL            imm32                   386
CALL            imm32|near              386
CALL            imm32|far               386,ND,NOLONG
CALL            imm:imm                 8086,NOLONG
CALL            imm16:imm               8086,NOLONG
CALL            imm:imm16               8086,NOLONG
CALL            imm32:imm               386,NOLONG
CALL            imm:imm32               386,NOLONG
CALL            mem|far                 8086,NOLONG
CALL            mem|far                 X64
CALL            mem16|far               8086
CALL            mem32|far               386
CALL            mem64|far               X64
CALL            mem|near                8086
CALL            mem16|near              8086
```

```
CALL            mem32|near              386,NOLONG
CALL            mem64|near              X64
CALL            reg16                   8086
CALL            reg32                   386,NOLONG
CALL            reg64                   X64
CALL            mem                     8086
CALL            mem16                   8086
CALL            mem32                   386,NOLONG
CALL            mem64                   X64
CBW                                     8086
CDQ                                     386
CDQE                                    X64
CLC                                     8086
CLD                                     8086
CLGI                                    X64,AMD
CLI                                     8086
CLTS                                    286,PRIV
CMC                                     8086
CMP             mem,reg8                8086
CMP             reg8,reg8               8086
CMP             mem,reg16               8086
CMP             reg16,reg16             8086
CMP             mem,reg32               386
CMP             reg32,reg32             386
CMP             mem,reg64               X64
CMP             reg64,reg64             X64
CMP             reg8,mem                8086
CMP             reg8,reg8               8086
CMP             reg16,mem               8086
CMP             reg16,reg16             8086
CMP             reg32,mem               386
CMP             reg32,reg32             386
CMP             reg64,mem               X64
CMP             reg64,reg64             X64
CMP             rm16,imm8               8086
CMP             rm32,imm8               386
CMP             rm64,imm8               X64
CMP             reg_al,imm              8086
CMP             reg_ax,sbyte16          8086
CMP             reg_ax,imm              8086
CMP             reg_eax,sbyte32         386
CMP             reg_eax,imm             386
CMP             reg_rax,sbyte64         X64
CMP             reg_rax,imm             X64
CMP             rm8,imm                 8086
CMP             rm16,imm                8086
CMP             rm32,imm                386
CMP             rm64,imm                X64
CMP             mem,imm8                8086
CMP             mem,imm16               8086
CMP             mem,imm32               386
CMPSB                                   8086
CMPSD                                   386
```

```
CMPSQ                                   X64
CMPSW                                   8086
CMPXCHG         mem,reg8                PENT
CMPXCHG         reg8,reg8               PENT
CMPXCHG         mem,reg16               PENT
CMPXCHG         reg16,reg16             PENT
CMPXCHG         mem,reg32               PENT
CMPXCHG         reg32,reg32             PENT
CMPXCHG         mem,reg64               X64
CMPXCHG         reg64,reg64             X64
CMPXCHG486      mem,reg8                486,UNDOC,ND
CMPXCHG486      reg8,reg8               486,UNDOC,ND
CMPXCHG486      mem,reg16               486,UNDOC,ND
CMPXCHG486      reg16,reg16             486,UNDOC,ND
CMPXCHG486      mem,reg32               486,UNDOC,ND
CMPXCHG486      reg32,reg32             486,UNDOC,ND
CMPXCHG8B       mem                     PENT
CMPXCHG16B      mem                     X64
CPUID                                   PENT
CPU_READ                                PENT,CYRIX
CPU_WRITE                               PENT,CYRIX
CQO                                     X64
CWD                                     8086
CWDE                                    386
DAA                                     8086,NOLONG
DAS                                     8086,NOLONG
DEC             reg16                   8086,NOLONG
DEC             reg32                   386,NOLONG
DEC             rm8                     8086
DEC             rm16                    8086
DEC             rm32                    386
DEC             rm64                    X64
DIV             rm8                     8086
DIV             rm16                    8086
DIV             rm32                    386
DIV             rm64                    X64
DMINT                                   P6,CYRIX
EMMS                                    PENT,MMX
ENTER           imm,imm                 186
EQU             imm                     8086
EQU             imm:imm                 8086
F2XM1                                   8086,FPU
FABS                                    8086,FPU
FADD            mem32                   8086,FPU
FADD            mem64                   8086,FPU
FADD            fpureg|to               8086,FPU
FADD            fpureg                  8086,FPU
FADD            fpureg,fpu0             8086,FPU
FADD            fpu0,fpureg             8086,FPU
FADD                                    8086,FPU,ND
FADDP           fpureg                  8086,FPU
FADDP           fpureg,fpu0             8086,FPU
FADDP                                   8086,FPU,ND
```

```
FBLD            mem80                   8086,FPU
FBLD            mem                     8086,FPU
FBSTP           mem80                   8086,FPU
FBSTP           mem                     8086,FPU
FCHS                                    8086,FPU
FCLEX                                   8086,FPU
FCMOVB          fpureg                  P6,FPU
FCMOVB          fpu0,fpureg             P6,FPU
FCMOVB                                  P6,FPU,ND
FCMOVBE         fpureg                  P6,FPU
FCMOVBE         fpu0,fpureg             P6,FPU
FCMOVBE                                 P6,FPU,ND
FCMOVE          fpureg                  P6,FPU
FCMOVE          fpu0,fpureg             P6,FPU
FCMOVE                                  P6,FPU,ND
FCMOVNB         fpureg                  P6,FPU
FCMOVNB         fpu0,fpureg             P6,FPU
FCMOVNB                                 P6,FPU,ND
FCMOVNBE        fpureg                  P6,FPU
FCMOVNBE        fpu0,fpureg             P6,FPU
FCMOVNBE                                P6,FPU,ND
FCMOVNE         fpureg                  P6,FPU
FCMOVNE         fpu0,fpureg             P6,FPU
FCMOVNE                                 P6,FPU,ND
FCMOVNU         fpureg                  P6,FPU
FCMOVNU         fpu0,fpureg             P6,FPU
FCMOVNU                                 P6,FPU,ND
FCMOVU          fpureg                  P6,FPU
FCMOVU          fpu0,fpureg             P6,FPU
FCMOVU                                  P6,FPU,ND
FCOM            mem32                   8086,FPU
FCOM            mem64                   8086,FPU
FCOM            fpureg                  8086,FPU
FCOM            fpu0,fpureg             8086,FPU
FCOM                                    8086,FPU,ND
FCOMI           fpureg                  P6,FPU
FCOMI           fpu0,fpureg             P6,FPU
FCOMI                                   P6,FPU,ND
FCOMIP          fpureg                  P6,FPU
FCOMIP          fpu0,fpureg             P6,FPU
FCOMIP                                  P6,FPU,ND
FCOMP           mem32                   8086,FPU
FCOMP           mem64                   8086,FPU
FCOMP           fpureg                  8086,FPU
FCOMP           fpu0,fpureg             8086,FPU
FCOMP                                   8086,FPU,ND
FCOMPP                                  8086,FPU
FCOS                                    386,FPU
FDECSTP                                 8086,FPU
FDISI                                   8086,FPU
FDIV            mem32                   8086,FPU
FDIV            mem64                   8086,FPU
FDIV            fpureg|to               8086,FPU
```

```
FDIV            fpureg                  8086,FPU
FDIV            fpureg,fpu0             8086,FPU
FDIV            fpu0,fpureg             8086,FPU
FDIV                                    8086,FPU,ND
FDIVP           fpureg                  8086,FPU
FDIVP           fpureg,fpu0             8086,FPU
FDIVP                                   8086,FPU,ND
FDIVR           mem32                   8086,FPU
FDIVR           mem64                   8086,FPU
FDIVR           fpureg|to               8086,FPU
FDIVR           fpureg,fpu0             8086,FPU
FDIVR           fpureg                  8086,FPU
FDIVR           fpu0,fpureg             8086,FPU
FDIVR                                   8086,FPU,ND
FDIVRP          fpureg                  8086,FPU
FDIVRP          fpureg,fpu0             8086,FPU
FDIVRP                                  8086,FPU,ND
FEMMS                                   PENT,3DNOW
FENI                                    8086,FPU
FFREE           fpureg                  8086,FPU
FFREE                                   8086,FPU
FFREEP          fpureg                  286,FPU,UNDOC
FFREEP                                  286,FPU,UNDOC
FIADD           mem32                   8086,FPU
FIADD           mem16                   8086,FPU
FICOM           mem32                   8086,FPU
FICOM           mem16                   8086,FPU
FICOMP          mem32                   8086,FPU
FICOMP          mem16                   8086,FPU
FIDIV           mem32                   8086,FPU
FIDIV           mem16                   8086,FPU
FIDIVR          mem32                   8086,FPU
FIDIVR          mem16                   8086,FPU
FILD            mem32                   8086,FPU
FILD            mem16                   8086,FPU
FILD            mem64                   8086,FPU
FIMUL           mem32                   8086,FPU
FIMUL           mem16                   8086,FPU
FINCSTP                                 8086,FPU
FINIT                                   8086,FPU
FIST            mem32                   8086,FPU
FIST            mem16                   8086,FPU
FISTP           mem32                   8086,FPU
FISTP           mem16                   8086,FPU
FISTP           mem64                   8086,FPU
FISTTP          mem16                   PRESCOTT,FPU
FISTTP          mem32                   PRESCOTT,FPU
FISTTP          mem64                   PRESCOTT,FPU
FISUB           mem32                   8086,FPU
FISUB           mem16                   8086,FPU
FISUBR          mem32                   8086,FPU
FISUBR          mem16                   8086,FPU
FLD             mem32                   8086,FPU
```

```
FLD             mem64                   8086,FPU
FLD             mem80                   8086,FPU
FLD             fpureg                  8086,FPU
FLD                                     8086,FPU,ND
FLD1                                    8086,FPU
FLDCW           mem                     8086,FPU,SW
FLDENV          mem                     8086,FPU
FLDL2E                                  8086,FPU
FLDL2T                                  8086,FPU
FLDLG2                                  8086,FPU
FLDLN2                                  8086,FPU
FLDPI                                   8086,FPU
FLDZ                                    8086,FPU
FMUL            mem32                   8086,FPU
FMUL            mem64                   8086,FPU
FMUL            fpureg|to               8086,FPU
FMUL            fpureg,fpu0             8086,FPU
FMUL            fpureg                  8086,FPU
FMUL            fpu0,fpureg             8086,FPU
FMUL                                    8086,FPU,ND
FMULP           fpureg                  8086,FPU
FMULP           fpureg,fpu0             8086,FPU
FMULP                                   8086,FPU,ND
FNCLEX                                  8086,FPU
FNDISI                                  8086,FPU
FNENI                                   8086,FPU
FNINIT                                  8086,FPU
FNOP                                    8086,FPU
FNSAVE          mem                     8086,FPU
FNSTCW          mem                     8086,FPU,SW
FNSTENV         mem                     8086,FPU
FNSTSW          mem                     8086,FPU,SW
FNSTSW          reg_ax                  286,FPU
FPATAN                                  8086,FPU
FPREM                                   8086,FPU
FPREM1                                  386,FPU
FPTAN                                   8086,FPU
FRNDINT                                 8086,FPU
FRSTOR          mem                     8086,FPU
FSAVE           mem                     8086,FPU
FSCALE                                  8086,FPU
FSETPM                                  286,FPU
FSIN                                    386,FPU
FSINCOS                                 386,FPU
FSQRT                                   8086,FPU
FST             mem32                   8086,FPU
FST             mem64                   8086,FPU
FST             fpureg                  8086,FPU
FST                                     8086,FPU,ND
FSTCW           mem                     8086,FPU,SW
FSTENV          mem                     8086,FPU
FSTP            mem32                   8086,FPU
FSTP            mem64                   8086,FPU
```

```
FSTP            mem80                   8086,FPU
FSTP            fpureg                  8086,FPU
FSTP                                    8086,FPU,ND
FSTSW           mem                     8086,FPU,SW
FSTSW           reg_ax                  286,FPU
FSUB            mem32                   8086,FPU
FSUB            mem64                   8086,FPU
FSUB            fpureg|to               8086,FPU
FSUB            fpureg,fpu0             8086,FPU
FSUB            fpureg                  8086,FPU
FSUB            fpu0,fpureg             8086,FPU
FSUB                                    8086,FPU,ND
FSUBP           fpureg                  8086,FPU
FSUBP           fpureg,fpu0             8086,FPU
FSUBP                                   8086,FPU,ND
FSUBR           mem32                   8086,FPU
FSUBR           mem64                   8086,FPU
FSUBR           fpureg|to               8086,FPU
FSUBR           fpureg,fpu0             8086,FPU
FSUBR           fpureg                  8086,FPU
FSUBR           fpu0,fpureg             8086,FPU
FSUBR                                   8086,FPU,ND
FSUBRP          fpureg                  8086,FPU
FSUBRP          fpureg,fpu0             8086,FPU
FSUBRP                                  8086,FPU,ND
FTST                                    8086,FPU
FUCOM           fpureg                  386,FPU
FUCOM           fpu0,fpureg             386,FPU
FUCOM                                   386,FPU,ND
FUCOMI          fpureg                  P6,FPU
FUCOMI          fpu0,fpureg             P6,FPU
FUCOMI                                  P6,FPU,ND
FUCOMIP         fpureg                  P6,FPU
FUCOMIP         fpu0,fpureg             P6,FPU
FUCOMIP                                 P6,FPU,ND
FUCOMP          fpureg                  386,FPU
FUCOMP          fpu0,fpureg             386,FPU
FUCOMP                                  386,FPU,ND
FUCOMPP                                 386,FPU
FXAM                                    8086,FPU
FXCH            fpureg                  8086,FPU
FXCH            fpureg,fpu0             8086,FPU
FXCH            fpu0,fpureg             8086,FPU
FXCH                                    8086,FPU,ND
FXTRACT                                 8086,FPU
FYL2X                                   8086,FPU
FYL2XP1                                 8086,FPU
HLT                                     8086,PRIV
IBTS            mem,reg16               386,SW,UNDOC,ND
IBTS            reg16,reg16             386,UNDOC,ND
IBTS            mem,reg32               386,SD,UNDOC,ND
IBTS            reg32,reg32             386,UNDOC,ND
ICEBP                                   386,ND
```

```
IDIV            rm8                     8086
IDIV            rm16                    8086
IDIV            rm32                    386
IDIV            rm64                    X64
IMUL            rm8                     8086
IMUL            rm16                    8086
IMUL            rm32                    386
IMUL            rm64                    X64
IMUL            reg16,mem               386
IMUL            reg16,reg16             386
IMUL            reg32,mem               386
IMUL            reg32,reg32             386
IMUL            reg64,mem               X64
IMUL            reg64,reg64             X64
IMUL            reg16,mem,imm8          186
IMUL            reg16,mem,sbyte16       186,ND
IMUL            reg16,mem,imm16         186
IMUL            reg16,mem,imm           186,ND
IMUL            reg16,reg16,imm8        186
IMUL            reg16,reg16,sbyte16     186,ND
IMUL            reg16,reg16,imm16       186
IMUL            reg16,reg16,imm         186,ND
IMUL            reg32,mem,imm8          386
IMUL            reg32,mem,sbyte32       386,ND
IMUL            reg32,mem,imm32         386
IMUL            reg32,mem,imm           386,ND
IMUL            reg32,reg32,imm8        386
IMUL            reg32,reg32,sbyte32     386,ND
IMUL            reg32,reg32,imm32       386
IMUL            reg32,reg32,imm         386,ND
IMUL            reg64,mem,imm8          X64
IMUL            reg64,mem,sbyte64       X64,ND
IMUL            reg64,mem,imm32         X64
IMUL            reg64,mem,imm           X64,ND
IMUL            reg64,reg64,imm8        X64
IMUL            reg64,reg64,sbyte64     X64,ND
IMUL            reg64,reg64,imm32       X64
IMUL            reg64,reg64,imm         X64,ND
IMUL            reg16,imm8              186
IMUL            reg16,sbyte16           186,ND
IMUL            reg16,imm16             186
IMUL            reg16,imm               186,ND
IMUL            reg32,imm8              386
IMUL            reg32,sbyte32           386,ND
IMUL            reg32,imm32             386
IMUL            reg32,imm               386,ND
IMUL            reg64,imm8              X64
IMUL            reg64,sbyte64           X64,ND
IMUL            reg64,imm32             X64
IMUL            reg64,imm               X64,ND
IN              reg_al,imm              8086
IN              reg_ax,imm              8086
IN              reg_eax,imm             386
```

```
IN              reg_al,reg_dx           8086
IN              reg_ax,reg_dx           8086
IN              reg_eax,reg_dx          386
INC             reg16                   8086,NOLONG
INC             reg32                   386,NOLONG
INC             rm8                     8086
INC             rm16                    8086
INC             rm32                    386
INC             rm64                    X64
INCBIN
INSB                                    186
INSD                                    386
INSW                                    186
INT             imm                     8086
INT01                                   386,ND
INT1                                    386
INT03                                   8086,ND
INT3                                    8086
INTO                                    8086,NOLONG
INVD                                    486,PRIV
INVLPG          mem                     486,PRIV
INVLPGA         reg_ax,reg_ecx          X86_64,AMD,NOLONG
INVLPGA         reg_eax,reg_ecx         X86_64,AMD
INVLPGA         reg_rax,reg_ecx         X64,AMD
INVLPGA                                 X86_64,AMD
IRET                                    8086
IRETD                                   386
IRETQ                                   X64
IRETW                                   8086
JCXZ            imm                     8086,NOLONG
JECXZ           imm                     386
JRCXZ           imm                     X64
JMP             imm|short               8086
JMP             imm                     8086,ND
JMP             imm                     8086
JMP             imm|near                8086,ND
JMP             imm|far                 8086,ND,NOLONG
JMP             imm16                   8086
JMP             imm16|near              8086,ND
JMP             imm16|far               8086,ND,NOLONG
JMP             imm32                   386
JMP             imm32|near              386,ND
JMP             imm32|far               386,ND,NOLONG
JMP             imm:imm                 8086,NOLONG
JMP             imm16:imm               8086,NOLONG
JMP             imm:imm16               8086,NOLONG
JMP             imm32:imm               386,NOLONG
JMP             imm:imm32               386,NOLONG
JMP             mem|far                 8086,NOLONG
JMP             mem|far                 X64
JMP             mem16|far               8086
JMP             mem32|far               386
JMP             mem64|far               X64
```

```
JMP             mem|near                8086
JMP             mem16|near              8086
JMP             mem32|near              386,NOLONG
JMP             mem64|near              X64
JMP             reg16                   8086
JMP             reg32                   386,NOLONG
JMP             reg64                   X64
JMP             mem                     8086
JMP             mem16                   8086
JMP             mem32                   386,NOLONG
JMP             mem64                   X64
JMPE            imm                     IA64
JMPE            imm16                   IA64
JMPE            imm32                   IA64
JMPE            rm16                    IA64
JMPE            rm32                    IA64
LAHF                                    8086
LAR             reg16,mem               286,PROT,SW
LAR             reg16,reg16             286,PROT
LAR             reg16,reg32             386,PROT
LAR             reg16,reg64             X64,PROT,ND
LAR             reg32,mem               386,PROT,SW
LAR             reg32,reg16             386,PROT
LAR             reg32,reg32             386,PROT
LAR             reg32,reg64             X64,PROT,ND
LAR             reg64,mem               X64,PROT,SW
LAR             reg64,reg16             X64,PROT
LAR             reg64,reg32             X64,PROT
LAR             reg64,reg64             X64,PROT
LDS             reg16,mem               8086,NOLONG
LDS             reg32,mem               386,NOLONG
LEA             reg16,mem               8086
LEA             reg32,mem               386
LEA             reg64,mem               X64
LEAVE                                   186
LES             reg16,mem               8086,NOLONG
LES             reg32,mem               386,NOLONG
LFENCE                                  X64,AMD
LFS             reg16,mem               386
LFS             reg32,mem               386
LFS             reg64,mem               X64
LGDT            mem                     286,PRIV
LGS             reg16,mem               386
LGS             reg32,mem               386
LGS             reg64,mem               X64
LIDT            mem                     286,PRIV
LLDT            mem                     286,PROT,PRIV
LLDT            mem16                   286,PROT,PRIV
LLDT            reg16                   286,PROT,PRIV
LMSW            mem                     286,PRIV
LMSW            mem16                   286,PRIV
LMSW            reg16                   286,PRIV
LOADALL                                 386,UNDOC
```

```
LOADALL286                              286,UNDOC
LODSB                                   8086
LODSD                                   386
LODSQ                                   X64
LODSW                                   8086
LOOP            imm                     8086
LOOP            imm,reg_cx              8086,NOLONG
LOOP            imm,reg_ecx             386
LOOP            imm,reg_rcx             X64
LOOPE           imm                     8086
LOOPE           imm,reg_cx              8086,NOLONG
LOOPE           imm,reg_ecx             386
LOOPE           imm,reg_rcx             X64
LOOPNE          imm                     8086
LOOPNE          imm,reg_cx              8086,NOLONG
LOOPNE          imm,reg_ecx             386
LOOPNE          imm,reg_rcx             X64
LOOPNZ          imm                     8086
LOOPNZ          imm,reg_cx              8086,NOLONG
LOOPNZ          imm,reg_ecx             386
LOOPNZ          imm,reg_rcx             X64
LOOPZ           imm                     8086
LOOPZ           imm,reg_cx              8086,NOLONG
LOOPZ           imm,reg_ecx             386
LOOPZ           imm,reg_rcx             X64
LSL             reg16,mem               286,PROT,SW
LSL             reg16,reg16             286,PROT
LSL             reg16,reg32             386,PROT
LSL             reg16,reg64             X64,PROT,ND
LSL             reg32,mem               386,PROT,SW
LSL             reg32,reg16             386,PROT
LSL             reg32,reg32             386,PROT
LSL             reg32,reg64             X64,PROT,ND
LSL             reg64,mem               X64,PROT,SW
LSL             reg64,reg16             X64,PROT
LSL             reg64,reg32             X64,PROT
LSL             reg64,reg64             X64,PROT
LSS             reg16,mem               386
LSS             reg32,mem               386
LSS             reg64,mem               X64
LTR             mem                     286,PROT,PRIV
LTR             mem16                   286,PROT,PRIV
LTR             reg16                   286,PROT,PRIV
MFENCE                                  X64,AMD
MONITOR                                 PRESCOTT
MONITOR         reg_eax,reg_ecx,reg_edx PRESCOTT,ND
MONITOR         reg_rax,reg_ecx,reg_edx X64,ND
MOV             mem,reg_sreg            8086
MOV             reg16,reg_sreg          8086
MOV             reg32,reg_sreg          386
MOV             reg_sreg,mem            8086
MOV             reg_sreg,reg16          8086
MOV             reg_sreg,reg32          386
```

```
MOV             reg_al,mem_offs         8086
MOV             reg_ax,mem_offs         8086
MOV             reg_eax,mem_offs        386
MOV             reg_rax,mem_offs        X64
MOV             mem_offs,reg_al         8086
MOV             mem_offs,reg_ax         8086
MOV             mem_offs,reg_eax        386
MOV             mem_offs,reg_rax        X64
MOV             reg32,reg_creg          386,PRIV,NOLONG
MOV             reg64,reg_creg          X64,PRIV
MOV             reg_creg,reg32          386,PRIV,NOLONG
MOV             reg_creg,reg64          X64,PRIV
MOV             reg32,reg_dreg          386,PRIV,NOLONG
MOV             reg64,reg_dreg          X64,PRIV
MOV             reg_dreg,reg32          386,PRIV,NOLONG
MOV             reg_dreg,reg64          X64,PRIV
MOV             reg32,reg_treg          386,NOLONG,ND
MOV             reg_treg,reg32          386,NOLONG,ND
MOV             mem,reg8                8086
MOV             reg8,reg8               8086
MOV             mem,reg16               8086
MOV             reg16,reg16             8086
MOV             mem,reg32               386
MOV             reg32,reg32             386
MOV             mem,reg64               X64
MOV             reg64,reg64             X64
MOV             reg8,mem                8086
MOV             reg8,reg8               8086
MOV             reg16,mem               8086
MOV             reg16,reg16             8086
MOV             reg32,mem               386
MOV             reg32,reg32             386
MOV             reg64,mem               X64
MOV             reg64,reg64             X64
MOV             reg8,imm                8086
MOV             reg16,imm               8086
MOV             reg32,imm               386
MOV             reg64,imm               X64
MOV             reg64,imm32             X64
MOV             rm8,imm                 8086
MOV             rm16,imm                8086
MOV             rm32,imm                386
MOV             rm64,imm                X64
MOV             mem,imm8                8086
MOV             mem,imm16               8086
MOV             mem,imm32               386
MOVD            mmxreg,mem              PENT,MMX,SD
MOVD            mmxreg,reg32            PENT,MMX
MOVD            mem,mmxreg              PENT,MMX,SD
MOVD            reg32,mmxreg            PENT,MMX
MOVD            xmmreg,mem              X64,SD
MOVD            xmmreg,reg32            X64
MOVD            mem,xmmreg              X64,SD
```

```
MOVD            reg32,xmmreg            X64,SSE
MOVQ            mmxreg,mmxrm            PENT,MMX
MOVQ            mmxrm,mmxreg            PENT,MMX
MOVQ            mmxreg,rm64             X64,MMX
MOVQ            rm64,mmxreg             X64,MMX
MOVSB                                   8086
MOVSD                                   386
MOVSQ                                   X64
MOVSW                                   8086
MOVSX           reg16,mem               386
MOVSX           reg16,reg8              386
MOVSX           reg32,rm8               386
MOVSX           reg32,rm16              386
MOVSX           reg64,rm8               X64
MOVSX           reg64,rm16              X64
MOVSXD          reg64,rm32              X64
MOVSX           reg64,rm32              X64,ND
MOVZX           reg16,mem               386
MOVZX           reg16,reg8              386
MOVZX           reg32,rm8               386
MOVZX           reg32,rm16              386
MOVZX           reg64,rm8               X64
MOVZX           reg64,rm16              X64
MUL             rm8                     8086
MUL             rm16                    8086
MUL             rm32                    386
MUL             rm64                    X64
MWAIT                                   PRESCOTT
MWAIT           reg_eax,reg_ecx         PRESCOTT,ND
NEG             rm8                     8086
NEG             rm16                    8086
NEG             rm32                    386
NEG             rm64                    X64
NOP                                     8086
NOP             rm16                    P6
NOP             rm32                    P6
NOP             rm64                    X64
NOT             rm8                     8086
NOT             rm16                    8086
NOT             rm32                    386
NOT             rm64                    X64
OR              mem,reg8                8086
OR              reg8,reg8               8086
OR              mem,reg16               8086
OR              reg16,reg16             8086
OR              mem,reg32               386
OR              reg32,reg32             386
OR              mem,reg64               X64
OR              reg64,reg64             X64
OR              reg8,mem                8086
OR              reg8,reg8               8086
OR              reg16,mem               8086
OR              reg16,reg16             8086
```

```
OR              reg32,mem               386
OR              reg32,reg32             386
OR              reg64,mem               X64
OR              reg64,reg64             X64
OR              rm16,imm8               8086
OR              rm32,imm8               386
OR              rm64,imm8               X64
OR              reg_al,imm              8086
OR              reg_ax,sbyte16          8086
OR              reg_ax,imm              8086
OR              reg_eax,sbyte32         386
OR              reg_eax,imm             386
OR              reg_rax,sbyte64         X64
OR              reg_rax,imm             X64
OR              rm8,imm                 8086
OR              rm16,imm                8086
OR              rm32,imm                386
OR              rm64,imm                X64
OR              mem,imm8                8086
OR              mem,imm16               8086
OR              mem,imm32               386
OUT             imm,reg_al              8086
OUT             imm,reg_ax              8086
OUT             imm,reg_eax             386
OUT             reg_dx,reg_al           8086
OUT             reg_dx,reg_ax           8086
OUT             reg_dx,reg_eax          386
OUTSB                                   186
OUTSD                                   386
OUTSW                                   186
PACKSSDW        mmxreg,mmxrm            PENT,MMX
PACKSSWB        mmxreg,mmxrm            PENT,MMX
PACKUSWB        mmxreg,mmxrm            PENT,MMX
PADDB           mmxreg,mmxrm            PENT,MMX
PADDD           mmxreg,mmxrm            PENT,MMX
PADDSB          mmxreg,mmxrm            PENT,MMX
PADDSIW         mmxreg,mmxrm            PENT,MMX,CYRIX
PADDSW          mmxreg,mmxrm            PENT,MMX
PADDUSB         mmxreg,mmxrm            PENT,MMX
PADDUSW         mmxreg,mmxrm            PENT,MMX
PADDW           mmxreg,mmxrm            PENT,MMX
PAND            mmxreg,mmxrm            PENT,MMX
PANDN           mmxreg,mmxrm            PENT,MMX
PAUSE                                   8086
PAVEB           mmxreg,mmxrm            PENT,MMX,CYRIX
PAVGUSB         mmxreg,mmxrm            PENT,3DNOW
PCMPEQB         mmxreg,mmxrm            PENT,MMX
PCMPEQD         mmxreg,mmxrm            PENT,MMX
PCMPEQW         mmxreg,mmxrm            PENT,MMX
PCMPGTB         mmxreg,mmxrm            PENT,MMX
PCMPGTD         mmxreg,mmxrm            PENT,MMX
PCMPGTW         mmxreg,mmxrm            PENT,MMX
PDISTIB         mmxreg,mem              PENT,MMX,CYRIX
```

```
PF2ID           mmxreg,mmxrm            PENT,3DNOW
PFACC           mmxreg,mmxrm            PENT,3DNOW
PFADD           mmxreg,mmxrm            PENT,3DNOW
PFCMPEQ         mmxreg,mmxrm            PENT,3DNOW
PFCMPGE         mmxreg,mmxrm            PENT,3DNOW
PFCMPGT         mmxreg,mmxrm            PENT,3DNOW
PFMAX           mmxreg,mmxrm            PENT,3DNOW
PFMIN           mmxreg,mmxrm            PENT,3DNOW
PFMUL           mmxreg,mmxrm            PENT,3DNOW
PFRCP           mmxreg,mmxrm            PENT,3DNOW
PFRCPIT1        mmxreg,mmxrm            PENT,3DNOW
PFRCPIT2        mmxreg,mmxrm            PENT,3DNOW
PFRSQIT1        mmxreg,mmxrm            PENT,3DNOW
PFRSQRT         mmxreg,mmxrm            PENT,3DNOW
PFSUB           mmxreg,mmxrm            PENT,3DNOW
PFSUBR          mmxreg,mmxrm            PENT,3DNOW
PI2FD           mmxreg,mmxrm            PENT,3DNOW
PMACHRIW        mmxreg,mem              PENT,MMX,CYRIX
PMADDWD         mmxreg,mmxrm            PENT,MMX
PMAGW           mmxreg,mmxrm            PENT,MMX,CYRIX
PMULHRIW        mmxreg,mmxrm            PENT,MMX,CYRIX
PMULHRWA        mmxreg,mmxrm            PENT,3DNOW
PMULHRWC        mmxreg,mmxrm            PENT,MMX,CYRIX
PMULHW          mmxreg,mmxrm            PENT,MMX
PMULLW          mmxreg,mmxrm            PENT,MMX
PMVGEZB         mmxreg,mem              PENT,MMX,CYRIX
PMVLZB          mmxreg,mem              PENT,MMX,CYRIX
PMVNZB          mmxreg,mem              PENT,MMX,CYRIX
PMVZB           mmxreg,mem              PENT,MMX,CYRIX
POP             reg16                   8086
POP             reg32                   386,NOLONG
POP             reg64                   X64
POP             rm16                    8086
POP             rm32                    386,NOLONG
POP             rm64                    X64
POP             reg_cs                  8086,UNDOC,ND
POP             reg_dess                8086,NOLONG
POP             reg_fsgs                386
POPA                                    186,NOLONG
POPAD                                   386,NOLONG
POPAW                                   186,NOLONG
POPF                                    8086
POPFD                                   386,NOLONG
POPFQ                                   X64
POPFW                                   8086
POR             mmxreg,mmxrm            PENT,MMX
PREFETCH        mem                     PENT,3DNOW
PREFETCHW       mem                     PENT,3DNOW
PSLLD           mmxreg,mmxrm            PENT,MMX
PSLLD           mmxreg,imm              PENT,MMX
PSLLQ           mmxreg,mmxrm            PENT,MMX
PSLLQ           mmxreg,imm              PENT,MMX
PSLLW           mmxreg,mmxrm            PENT,MMX
```

```
PSLLW           mmxreg,imm              PENT,MMX
PSRAD           mmxreg,mmxrm            PENT,MMX
PSRAD           mmxreg,imm              PENT,MMX
PSRAW           mmxreg,mmxrm            PENT,MMX
PSRAW           mmxreg,imm              PENT,MMX
PSRLD           mmxreg,mmxrm            PENT,MMX
PSRLD           mmxreg,imm              PENT,MMX
PSRLQ           mmxreg,mmxrm            PENT,MMX
PSRLQ           mmxreg,imm              PENT,MMX
PSRLW           mmxreg,mmxrm            PENT,MMX
PSRLW           mmxreg,imm              PENT,MMX
PSUBB           mmxreg,mmxrm            PENT,MMX
PSUBD           mmxreg,mmxrm            PENT,MMX
PSUBSB          mmxreg,mmxrm            PENT,MMX
PSUBSIW         mmxreg,mmxrm            PENT,MMX,CYRIX
PSUBSW          mmxreg,mmxrm            PENT,MMX
PSUBUSB         mmxreg,mmxrm            PENT,MMX
PSUBUSW         mmxreg,mmxrm            PENT,MMX
PSUBW           mmxreg,mmxrm            PENT,MMX
PUNPCKHBW       mmxreg,mmxrm            PENT,MMX
PUNPCKHDQ       mmxreg,mmxrm            PENT,MMX
PUNPCKHWD       mmxreg,mmxrm            PENT,MMX
PUNPCKLBW       mmxreg,mmxrm            PENT,MMX
PUNPCKLDQ       mmxreg,mmxrm            PENT,MMX
PUNPCKLWD       mmxreg,mmxrm            PENT,MMX
PUSH            reg16                   8086
PUSH            reg32                   386,NOLONG
PUSH            reg64                   X64
PUSH            rm16                    8086
PUSH            rm32                    386,NOLONG
PUSH            rm64                    X64
PUSH            reg_cs                  8086,NOLONG
PUSH            reg_dess                8086,NOLONG
PUSH            reg_fsgs                386
PUSH            imm8                    186
PUSH            imm16                   186,AR0,SZ
PUSH            imm32                   386,NOLONG,AR0,SZ
PUSH            imm32                   386,NOLONG,SD
PUSH            imm32                   X64,AR0,SZ
PUSH            imm64                   X64,AR0,SZ
PUSHA                                   186,NOLONG
PUSHAD                                  386,NOLONG
PUSHAW                                  186,NOLONG
PUSHF                                   8086
PUSHFD                                  386,NOLONG
PUSHFQ                                  X64
PUSHFW                                  8086
PXOR            mmxreg,mmxrm            PENT,MMX
RCL             rm8,unity               8086
RCL             rm8,reg_cl              8086
RCL             rm8,imm                 186
RCL             rm16,unity              8086
RCL             rm16,reg_cl             8086
```

```
RCL             rm16,imm                186
RCL             rm32,unity              386
RCL             rm32,reg_cl             386
RCL             rm32,imm                386
RCL             rm64,unity              X64
RCL             rm64,reg_cl             X64
RCL             rm64,imm                X64
RCR             rm8,unity               8086
RCR             rm8,reg_cl              8086
RCR             rm8,imm                 186
RCR             rm16,unity              8086
RCR             rm16,reg_cl             8086
RCR             rm16,imm                186
RCR             rm32,unity              386
RCR             rm32,reg_cl             386
RCR             rm32,imm                386
RCR             rm64,unity              X64
RCR             rm64,reg_cl             X64
RCR             rm64,imm                X64
RDSHR           rm32                    P6,CYRIXM
RDMSR                                   PENT,PRIV
RDPMC                                   P6
RDTSC                                   PENT
RDTSCP                                  X86_64
RET                                     8086
RET             imm                     8086,SW
RETF                                    8086
RETF            imm                     8086,SW
RETN                                    8086
RETN            imm                     8086,SW
ROL             rm8,unity               8086
ROL             rm8,reg_cl              8086
ROL             rm8,imm                 186
ROL             rm16,unity              8086
ROL             rm16,reg_cl             8086
ROL             rm16,imm                186
ROL             rm32,unity              386
ROL             rm32,reg_cl             386
ROL             rm32,imm                386
ROL             rm64,unity              X64
ROL             rm64,reg_cl             X64
ROL             rm64,imm                X64
ROR             rm8,unity               8086
ROR             rm8,reg_cl              8086
ROR             rm8,imm                 186
ROR             rm16,unity              8086
ROR             rm16,reg_cl             8086
ROR             rm16,imm                186
ROR             rm32,unity              386
ROR             rm32,reg_cl             386
ROR             rm32,imm                386
ROR             rm64,unity              X64
ROR             rm64,reg_cl             X64
```

```
ROR             rm64,imm                X64
RDM                                     P6,CYRIX,ND
RSDC            reg_sreg,mem80          486,CYRIXM
RSLDT           mem80                   486,CYRIXM
RSM                                     PENTM
RSTS            mem80                   486,CYRIXM
SAHF                                    8086
SAL             rm8,unity               8086,ND
SAL             rm8,reg_cl              8086,ND
SAL             rm8,imm                 186,ND
SAL             rm16,unity              8086,ND
SAL             rm16,reg_cl             8086,ND
SAL             rm16,imm                186,ND
SAL             rm32,unity              386,ND
SAL             rm32,reg_cl             386,ND
SAL             rm32,imm                386,ND
SAL             rm64,unity              X64,ND
SAL             rm64,reg_cl             X64,ND
SAL             rm64,imm                X64,ND
SALC                                    8086,UNDOC
SAR             rm8,unity               8086
SAR             rm8,reg_cl              8086
SAR             rm8,imm                 186
SAR             rm16,unity              8086
SAR             rm16,reg_cl             8086
SAR             rm16,imm                186
SAR             rm32,unity              386
SAR             rm32,reg_cl             386
SAR             rm32,imm                386
SAR             rm64,unity              X64
SAR             rm64,reg_cl             X64
SAR             rm64,imm                X64
SBB             mem,reg8                8086
SBB             reg8,reg8               8086
SBB             mem,reg16               8086
SBB             reg16,reg16             8086
SBB             mem,reg32               386
SBB             reg32,reg32             386
SBB             mem,reg64               X64
SBB             reg64,reg64             X64
SBB             reg8,mem                8086
SBB             reg8,reg8               8086
SBB             reg16,mem               8086
SBB             reg16,reg16             8086
SBB             reg32,mem               386
SBB             reg32,reg32             386
SBB             reg64,mem               X64
SBB             reg64,reg64             X64
SBB             rm16,imm8               8086
SBB             rm32,imm8               386
SBB             rm64,imm8               X64
SBB             reg_al,imm              8086
SBB             reg_ax,sbyte16          8086
```

```
SBB             reg_ax,imm              8086
SBB             reg_eax,sbyte32         386
SBB             reg_eax,imm             386
SBB             reg_rax,sbyte64         X64
SBB             reg_rax,imm             X64
SBB             rm8,imm                 8086
SBB             rm16,imm                8086
SBB             rm32,imm                386
SBB             rm64,imm                X64
SBB             mem,imm8                8086
SBB             mem,imm16               8086
SBB             mem,imm32               386
SCASB                                   8086
SCASD                                   386
SCASQ                                   X64
SCASW                                   8086
SFENCE                                  X64,AMD
SGDT            mem                     286
SHL             rm8,unity               8086
SHL             rm8,reg_cl              8086
SHL             rm8,imm                 186
SHL             rm16,unity              8086
SHL             rm16,reg_cl             8086
SHL             rm16,imm                186
SHL             rm32,unity              386
SHL             rm32,reg_cl             386
SHL             rm32,imm                386
SHL             rm64,unity              X64
SHL             rm64,reg_cl             X64
SHL             rm64,imm                X64
SHLD            mem,reg16,imm           3862
SHLD            reg16,reg16,imm         3862
SHLD            mem,reg32,imm           3862
SHLD            reg32,reg32,imm         3862
SHLD            mem,reg64,imm           X642
SHLD            reg64,reg64,imm         X642
SHLD            mem,reg16,reg_cl        386
SHLD            reg16,reg16,reg_cl      386
SHLD            mem,reg32,reg_cl        386
SHLD            reg32,reg32,reg_cl      386
SHLD            mem,reg64,reg_cl        X64
SHLD            reg64,reg64,reg_cl      X64
SHR             rm8,unity               8086
SHR             rm8,reg_cl              8086
SHR             rm8,imm                 186
SHR             rm16,unity              8086
SHR             rm16,reg_cl             8086
SHR             rm16,imm                186
SHR             rm32,unity              386
SHR             rm32,reg_cl             386
SHR             rm32,imm                386
SHR             rm64,unity              X64
SHR             rm64,reg_cl             X64
```

```
SHR             rm64,imm                X64
SHRD            mem,reg16,imm           3862
SHRD            reg16,reg16,imm         3862
SHRD            mem,reg32,imm           3862
SHRD            reg32,reg32,imm         3862
SHRD            mem,reg64,imm           X642
SHRD            reg64,reg64,imm         X642
SHRD            mem,reg16,reg_cl        386
SHRD            reg16,reg16,reg_cl      386
SHRD            mem,reg32,reg_cl        386
SHRD            reg32,reg32,reg_cl      386
SHRD            mem,reg64,reg_cl        X64
SHRD            reg64,reg64,reg_cl      X64
SIDT            mem                     286
SLDT            mem                     286
SLDT            mem16                   286
SLDT            reg16                   286
SLDT            reg32                   386
SLDT            reg64                   X64,ND
SLDT            reg64                   X64
SKINIT                                  X64
SMI                                     386,UNDOC
SMINT                                   P6,CYRIX,ND
SMINTOLD                                486,CYRIX,ND
SMSW            mem                     286
SMSW            mem16                   286
SMSW            reg16                   286
SMSW            reg32                   386
STC                                     8086
STD                                     8086
STGI                                    X64
STI                                     8086
STOSB                                   8086
STOSD                                   386
STOSQ                                   X64
STOSW                                   8086
STR             mem                     286,PROT
STR             mem16                   286,PROT
STR             reg16                   286,PROT
STR             reg32                   386,PROT
STR             reg64                   X64
SUB             mem,reg8                8086
SUB             reg8,reg8               8086
SUB             mem,reg16               8086
SUB             reg16,reg16             8086
SUB             mem,reg32               386
SUB             reg32,reg32             386
SUB             mem,reg64               X64
SUB             reg64,reg64             X64
SUB             reg8,mem                8086
SUB             reg8,reg8               8086
SUB             reg16,mem               8086
SUB             reg16,reg16             8086
```

```
SUB             reg32,mem               386
SUB             reg32,reg32             386
SUB             reg64,mem               X64
SUB             reg64,reg64             X64
SUB             rm16,imm8               8086
SUB             rm32,imm8               386
SUB             rm64,imm8               X64
SUB             reg_al,imm              8086
SUB             reg_ax,sbyte16          8086
SUB             reg_ax,imm              8086
SUB             reg_eax,sbyte32         386
SUB             reg_eax,imm             386
SUB             reg_rax,sbyte64         X64
SUB             reg_rax,imm             X64
SUB             rm8,imm                 8086
SUB             rm16,imm                8086
SUB             rm32,imm                386
SUB             rm64,imm                X64
SUB             mem,imm8                8086
SUB             mem,imm16               8086
SUB             mem,imm32               386
SVDC            mem80,reg_sreg          486,CYRIXM
SVLDT           mem80                   486,CYRIXM,ND
SVTS            mem80                   486,CYRIXM
SWAPGS                                  X64
SYSCALL                                 P6,AMD
SYSENTER                                P6
SYSEXIT                                 P6,PRIV
SYSRET                                  P6,PRIV,AMD
TEST            mem,reg8                8086
TEST            reg8,reg8               8086
TEST            mem,reg16               8086
TEST            reg16,reg16             8086
TEST            mem,reg32               386
TEST            reg32,reg32             386
TEST            mem,reg64               X64
TEST            reg64,reg64             X64
TEST            reg8,mem                8086
TEST            reg16,mem               8086
TEST            reg32,mem               386
TEST            reg64,mem               X64
TEST            reg_al,imm              8086
TEST            reg_ax,imm              8086
TEST            reg_eax,imm             386
TEST            reg_rax,imm             X64
TEST            rm8,imm                 8086
TEST            rm16,imm                8086
TEST            rm32,imm                386
TEST            rm64,imm                X64
TEST            mem,imm8                8086
TEST            mem,imm16               8086
TEST            mem,imm32               386
UD0                                     186,UNDOC
```

```
UD1                             186,UNDOC
UD2B                            186,UNDOC,ND
UD2                             186
UD2A                            186,ND
UMOV            mem,reg8        386,UNDOC,ND
UMOV            reg8,reg8       386,UNDOC,ND
UMOV            mem,reg16       386,UNDOC,ND
UMOV            reg16,reg16     386,UNDOC,ND
UMOV            mem,reg32       386,UNDOC,ND
UMOV            reg32,reg32     386,UNDOC,ND
UMOV            reg8,mem        386,UNDOC,ND
UMOV            reg8,reg8       386,UNDOC,ND
UMOV            reg16,mem       386,UNDOC,ND
UMOV            reg16,reg16     386,UNDOC,ND
UMOV            reg32,mem       386,UNDOC,ND
UMOV            reg32,reg32     386,UNDOC,ND
VERR            mem             286,PROT
VERR            mem16           286,PROT
VERR            reg16           286,PROT
VERW            mem             286,PROT
VERW            mem16           286,PROT
VERW            reg16           286,PROT
FWAIT                           8086
WBINVD                          486,PRIV
WRSHR           rm32            P6,CYRIXM
WRMSR                           PENT,PRIV
XADD            mem,reg8        486
XADD            reg8,reg8       486
XADD            mem,reg16       486
XADD            reg16,reg16     486
XADD            mem,reg32       486
XADD            reg32,reg32     486
XADD            mem,reg64       X64
XADD            reg64,reg64     X64
XBTS            reg16,mem       386,SW,UNDOC,ND
XBTS            reg16,reg16     386,UNDOC,ND
XBTS            reg32,mem       386,SD,UNDOC,ND
XBTS            reg32,reg32     386,UNDOC,ND
XCHG            reg_ax,reg16    8086
XCHG            reg_eax,reg32na 386
XCHG            reg_rax,reg64   X64
XCHG            reg16,reg_ax    8086
XCHG            reg32na,reg_eax 386
XCHG            reg64,reg_rax   X64
XCHG            reg_eax,reg_eax 386,NOLONG
XCHG            reg8,mem        8086
XCHG            reg8,reg8       8086
XCHG            reg16,mem       8086
XCHG            reg16,reg16     8086
XCHG            reg32,mem       386
XCHG            reg32,reg32     386
XCHG            reg64,mem       X64
XCHG            reg64,reg64     X64
```

```
XCHG            mem,reg8                8086
XCHG            reg8,reg8               8086
XCHG            mem,reg16               8086
XCHG            reg16,reg16             8086
XCHG            mem,reg32               386
XCHG            reg32,reg32             386
XCHG            mem,reg64               X64
XCHG            reg64,reg64             X64
XLATB                                   8086
XLAT                                    8086
XOR             mem,reg8                8086
XOR             reg8,reg8               8086
XOR             mem,reg16               8086
XOR             reg16,reg16             8086
XOR             mem,reg32               386
XOR             reg32,reg32             386
XOR             mem,reg64               X64
XOR             reg64,reg64             X64
XOR             reg8,mem                8086
XOR             reg8,reg8               8086
XOR             reg16,mem               8086
XOR             reg16,reg16             8086
XOR             reg32,mem               386
XOR             reg32,reg32             386
XOR             reg64,mem               X64
XOR             reg64,reg64             X64
XOR             rm16,imm8               8086
XOR             rm32,imm8               386
XOR             rm64,imm8               X64
XOR             reg_al,imm              8086
XOR             reg_ax,sbyte16          8086
XOR             reg_ax,imm              8086
XOR             reg_eax,sbyte32         386
XOR             reg_eax,imm             386
XOR             reg_rax,sbyte64         X64
XOR             reg_rax,imm             X64
XOR             rm8,imm                 8086
XOR             rm16,imm                8086
XOR             rm32,imm                386
XOR             rm64,imm                X64
XOR             mem,imm8                8086
XOR             mem,imm16               8086
XOR             mem,imm32               386
CMOVcc          reg16,mem               P6
CMOVcc          reg16,reg16             P6
CMOVcc          reg32,mem               P6
CMOVcc          reg32,reg32             P6
CMOVcc          reg64,mem               X64
CMOVcc          reg64,reg64             X64
Jcc             imm|near                386
Jcc             imm16|near              386
Jcc             imm32|near              386
Jcc             imm|short               8086,ND
```

```
Jcc             imm                     8086,ND
Jcc             imm                     386,ND
Jcc             imm                     8086,ND
Jcc             imm                     8086
SETcc           mem                     386
SETcc           reg8                    386
```

B.1.3 Katmai Streaming SIMD instructions (SSE — a.k.a. KNI, XMM, MMX2)

```
ADDPS           xmmreg,xmmrm            KATMAI,SSE
ADDSS           xmmreg,xmmrm            KATMAI,SSE,SD
ANDNPS          xmmreg,xmmrm            KATMAI,SSE
ANDPS           xmmreg,xmmrm            KATMAI,SSE
CMPEQPS         xmmreg,xmmrm            KATMAI,SSE
CMPEQSS         xmmreg,xmmrm            KATMAI,SSE
CMPLEPS         xmmreg,xmmrm            KATMAI,SSE
CMPLESS         xmmreg,xmmrm            KATMAI,SSE
CMPLTPS         xmmreg,xmmrm            KATMAI,SSE
CMPLTSS         xmmreg,xmmrm            KATMAI,SSE
CMPNEQPS        xmmreg,xmmrm            KATMAI,SSE
CMPNEQSS        xmmreg,xmmrm            KATMAI,SSE
CMPNLEPS        xmmreg,xmmrm            KATMAI,SSE
CMPNLESS        xmmreg,xmmrm            KATMAI,SSE
CMPNLTPS        xmmreg,xmmrm            KATMAI,SSE
CMPNLTSS        xmmreg,xmmrm            KATMAI,SSE
CMPORDPS        xmmreg,xmmrm            KATMAI,SSE
CMPORDSS        xmmreg,xmmrm            KATMAI,SSE
CMPUNORDPS      xmmreg,xmmrm            KATMAI,SSE
CMPUNORDSS      xmmreg,xmmrm            KATMAI,SSE
CMPPS           xmmreg,mem,imm          KATMAI,SSE
CMPPS           xmmreg,xmmreg,imm       KATMAI,SSE
CMPSS           xmmreg,mem,imm          KATMAI,SSE
CMPSS           xmmreg,xmmreg,imm       KATMAI,SSE
COMISS          xmmreg,xmmrm            KATMAI,SSE
CVTPI2PS        xmmreg,mmxrm            KATMAI,SSE,MMX
CVTPS2PI        mmxreg,xmmrm            KATMAI,SSE,MMX
CVTSI2SS        xmmreg,mem              KATMAI,SSE,SD,AR1,ND
CVTSI2SS        xmmreg,rm32             KATMAI,SSE,SD,AR1
CVTSI2SS        xmmreg,rm64             X64,SSE,AR1
CVTSS2SI        reg32,xmmreg            KATMAI,SSE,SD,AR1
CVTSS2SI        reg32,mem               KATMAI,SSE,SD,AR1
CVTSS2SI        reg64,xmmreg            X64,SSE,SD,AR1
CVTSS2SI        reg64,mem               X64,SSE,SD,AR1
CVTTPS2PI       mmxreg,xmmrm            KATMAI,SSE,MMX
CVTTSS2SI       reg32,xmmrm             KATMAI,SSE,SD,AR1
CVTTSS2SI       reg64,xmmrm             X64,SSE,SD,AR1
DIVPS           xmmreg,xmmrm            KATMAI,SSE
DIVSS           xmmreg,xmmrm            KATMAI,SSE
LDMXCSR         mem                     KATMAI,SSE,SD
MAXPS           xmmreg,xmmrm            KATMAI,SSE
MAXSS           xmmreg,xmmrm            KATMAI,SSE
MINPS           xmmreg,xmmrm            KATMAI,SSE
MINSS           xmmreg,xmmrm            KATMAI,SSE
MOVAPS          xmmreg,mem              KATMAI,SSE
```

```
MOVAPS          mem,xmmreg              KATMAI,SSE
MOVAPS          xmmreg,xmmreg           KATMAI,SSE
MOVAPS          xmmreg,xmmreg           KATMAI,SSE
MOVHPS          xmmreg,mem              KATMAI,SSE
MOVHPS          mem,xmmreg              KATMAI,SSE
MOVLHPS         xmmreg,xmmreg           KATMAI,SSE
MOVLPS          xmmreg,mem              KATMAI,SSE
MOVLPS          mem,xmmreg              KATMAI,SSE
MOVHLPS         xmmreg,xmmreg           KATMAI,SSE
MOVMSKPS        reg32,xmmreg            KATMAI,SSE
MOVMSKPS        reg64,xmmreg            X64,SSE
MOVNTPS         mem,xmmreg              KATMAI,SSE
MOVSS           xmmreg,mem              KATMAI,SSE
MOVSS           mem,xmmreg              KATMAI,SSE
MOVSS           xmmreg,xmmreg           KATMAI,SSE
MOVSS           xmmreg,xmmreg           KATMAI,SSE
MOVUPS          xmmreg,mem              KATMAI,SSE
MOVUPS          mem,xmmreg              KATMAI,SSE
MOVUPS          xmmreg,xmmreg           KATMAI,SSE
MOVUPS          xmmreg,xmmreg           KATMAI,SSE
MULPS           xmmreg,xmmrm            KATMAI,SSE
MULSS           xmmreg,xmmrm            KATMAI,SSE
ORPS            xmmreg,xmmrm            KATMAI,SSE
RCPPS           xmmreg,xmmrm            KATMAI,SSE
RCPSS           xmmreg,xmmrm            KATMAI,SSE
RSQRTPS         xmmreg,xmmrm            KATMAI,SSE
RSQRTSS         xmmreg,xmmrm            KATMAI,SSE
SHUFPS          xmmreg,mem,imm          KATMAI,SSE
SHUFPS          xmmreg,xmmreg,imm       KATMAI,SSE
SQRTPS          xmmreg,xmmrm            KATMAI,SSE
SQRTSS          xmmreg,xmmrm            KATMAI,SSE
STMXCSR         mem                     KATMAI,SSE,SD
SUBPS           xmmreg,xmmrm            KATMAI,SSE
SUBSS           xmmreg,xmmrm            KATMAI,SSE
UCOMISS         xmmreg,xmmrm            KATMAI,SSE
UNPCKHPS        xmmreg,xmmrm            KATMAI,SSE
UNPCKLPS        xmmreg,xmmrm            KATMAI,SSE
XORPS           xmmreg,xmmrm            KATMAI,SSE
```

B.1.4 Introduced in Deschutes but necessary for SSE support

```
FXRSTOR         mem                     P6,SSE,FPU
FXRSTOR64       mem                     X64,SSE,FPU
FXSAVE          mem                     P6,SSE,FPU
FXSAVE64        mem                     X64,SSE,FPU
```

B.1.5 XSAVE group (AVX and extended state)

```
XGETBV                                  NEHALEM
XSETBV                                  NEHALEM,PRIV
XSAVE           mem                     NEHALEM
XSAVE64         mem                     LONG,NEHALEM
XSAVEOPT        mem                     FUTURE
XSAVEOPT64      mem                     LONG,FUTURE
```

```
XRSTOR          mem                     NEHALEM
XRSTOR64        mem                     LONG,NEHALEM
```

B.1.6 Generic memory operations

```
PREFETCHNTA     mem                     KATMAI
PREFETCHT0      mem                     KATMAI
PREFETCHT1      mem                     KATMAI
PREFETCHT2      mem                     KATMAI
SFENCE                                  KATMAI
```

B.1.7 New MMX instructions introduced in Katmai

```
MASKMOVQ        mmxreg,mmxreg           KATMAI,MMX
MOVNTQ          mem,mmxreg              KATMAI,MMX
PAVGB           mmxreg,mmxrm            KATMAI,MMX
PAVGW           mmxreg,mmxrm            KATMAI,MMX
PEXTRW          reg32,mmxreg,imm        KATMAI,MMX
PINSRW          mmxreg,mem,imm          KATMAI,MMX
PINSRW          mmxreg,rm16,imm         KATMAI,MMX
PINSRW          mmxreg,reg32,imm        KATMAI,MMX
PMAXSW          mmxreg,mmxrm            KATMAI,MMX
PMAXUB          mmxreg,mmxrm            KATMAI,MMX
PMINSW          mmxreg,mmxrm            KATMAI,MMX
PMINUB          mmxreg,mmxrm            KATMAI,MMX
PMOVMSKB        reg32,mmxreg            KATMAI,MMX
PMULHUW         mmxreg,mmxrm            KATMAI,MMX
PSADBW          mmxreg,mmxrm            KATMAI,MMX
PSHUFW          mmxreg,mmxrm,imm        KATMAI,MMX2
```

B.1.8 AMD Enhanced 3DNow! (Athlon) instructions

```
PF2IW           mmxreg,mmxrm            PENT,3DNOW
PFNACC          mmxreg,mmxrm            PENT,3DNOW
PFPNACC         mmxreg,mmxrm            PENT,3DNOW
PI2FW           mmxreg,mmxrm            PENT,3DNOW
PSWAPD          mmxreg,mmxrm            PENT,3DNOW
```

B.1.9 Willamette SSE2 Cacheability Instructions

```
MASKMOVDQU      xmmreg,xmmreg           WILLAMETTE,SSE2
CLFLUSH         mem                     WILLAMETTE,SSE2
MOVNTDQ         mem,xmmreg              WILLAMETTE,SSE2,SO
MOVNTI          mem,reg32               WILLAMETTE,SD
MOVNTI          mem,reg64               X64
MOVNTPD         mem,xmmreg              WILLAMETTE,SSE2,SO
LFENCE                                  WILLAMETTE,SSE2
MFENCE                                  WILLAMETTE,SSE2
```

B.1.10 Willamette MMX instructions (SSE2 SIMD Integer Instructions)

```
MOVD            mem,xmmreg              WILLAMETTE,SSE2,SD
MOVD            xmmreg,mem              WILLAMETTE,SSE2,SD
MOVD            xmmreg,rm32             WILLAMETTE,SSE2
MOVD            rm32,xmmreg             WILLAMETTE,SSE2
MOVDQA          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVDQA          mem,xmmreg              WILLAMETTE,SSE2,SO
```

```
MOVDQA          xmmreg,mem              WILLAMETTE,SSE2,SO
MOVDQA          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVDQU          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVDQU          mem,xmmreg              WILLAMETTE,SSE2,SO
MOVDQU          xmmreg,mem              WILLAMETTE,SSE2,SO
MOVDQU          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVDQ2Q         mmxreg,xmmreg           WILLAMETTE,SSE2
MOVQ            xmmreg,xmmreg           WILLAMETTE,SSE2
MOVQ            xmmreg,xmmreg           WILLAMETTE,SSE2
MOVQ            mem,xmmreg              WILLAMETTE,SSE2
MOVQ            xmmreg,mem              WILLAMETTE,SSE2
MOVQ            xmmreg,rm64             X64,SSE2
MOVQ            rm64,xmmreg             X64,SSE2
MOVQ2DQ         xmmreg,mmxreg           WILLAMETTE,SSE2
PACKSSWB        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PACKSSDW        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PACKUSWB        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDB           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDQ           mmxreg,mmxrm            WILLAMETTE,MMX
PADDQ           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDSB          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDSW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDUSB         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PADDUSW         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PAND            xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PANDN           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PAVGB           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PAVGW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPEQB         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPEQW         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPEQD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPGTB         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPGTW         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PCMPGTD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PEXTRW          reg32,xmmreg,imm        WILLAMETTE,SSE2
PINSRW          xmmreg,reg16,imm        WILLAMETTE,SSE2
PINSRW          xmmreg,reg32,imm        WILLAMETTE,SSE2,ND
PINSRW          xmmreg,mem,imm          WILLAMETTE,SSE2
PINSRW          xmmreg,mem16,imm        WILLAMETTE,SSE2
PMADDWD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMAXSW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMAXUB          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMINSW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMINUB          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMOVMSKB        reg32,xmmreg            WILLAMETTE,SSE2
PMULHUW         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMULHW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMULLW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PMULUDQ         mmxreg,mmxrm            WILLAMETTE,SSE2,SO
PMULUDQ         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
POR             xmmreg,xmmrm            WILLAMETTE,SSE2,SO
```

```
PSADBW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSHUFD          xmmreg,xmmreg,imm       WILLAMETTE,SSE2
PSHUFD          xmmreg,mem,imm          WILLAMETTE,SSE22
PSHUFHW         xmmreg,xmmreg,imm       WILLAMETTE,SSE2
PSHUFHW         xmmreg,mem,imm          WILLAMETTE,SSE22
PSHUFLW         xmmreg,xmmreg,imm       WILLAMETTE,SSE2
PSHUFLW         xmmreg,mem,imm          WILLAMETTE,SSE22
PSLLDQ          xmmreg,imm              WILLAMETTE,SSE2,AR1
PSLLW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSLLW           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSLLD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSLLD           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSLLQ           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSLLQ           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRAW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSRAW           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRAD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSRAD           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRLDQ          xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRLW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSRLW           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRLD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSRLD           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSRLQ           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSRLQ           xmmreg,imm              WILLAMETTE,SSE2,AR1
PSUBB           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBW           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBQ           mmxreg,mmxrm            WILLAMETTE,SSE2,SO
PSUBQ           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBSB          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBSW          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBUSB         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PSUBUSW         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKHBW       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKHWD       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKHDQ       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKHQDQ      xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKLBW       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKLWD       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKLDQ       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PUNPCKLQDQ      xmmreg,xmmrm            WILLAMETTE,SSE2,SO
PXOR            xmmreg,xmmrm            WILLAMETTE,SSE2,SO
```

B.1.11 Willamette Streaming SIMD instructions (SSE2)

```
ADDPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
ADDSD           xmmreg,xmmrm            WILLAMETTE,SSE2
ANDNPD          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
ANDPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPEQPD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPEQSD         xmmreg,xmmrm            WILLAMETTE,SSE2
CMPLEPD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPLESD         xmmreg,xmmrm            WILLAMETTE,SSE2
```

```
CMPLTPD         xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPLTSD         xmmreg,xmmrm            WILLAMETTE,SSE2
CMPNEQPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPNEQSD        xmmreg,xmmrm            WILLAMETTE,SSE2
CMPNLEPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPNLESD        xmmreg,xmmrm            WILLAMETTE,SSE2
CMPNLTPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPNLTSD        xmmreg,xmmrm            WILLAMETTE,SSE2
CMPORDPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPORDSD        xmmreg,xmmrm            WILLAMETTE,SSE2
CMPUNORDPD      xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CMPUNORDSD      xmmreg,xmmrm            WILLAMETTE,SSE2
CMPPD           xmmreg,xmmrm,imm        WILLAMETTE,SSE22
CMPSD           xmmreg,xmmrm,imm        WILLAMETTE,SSE2
COMISD          xmmreg,xmmrm            WILLAMETTE,SSE2
CVTDQ2PD        xmmreg,xmmrm            WILLAMETTE,SSE2
CVTDQ2PS        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTPD2DQ        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTPD2PI        mmxreg,xmmrm            WILLAMETTE,SSE2,SO
CVTPD2PS        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTPI2PD        xmmreg,mmxrm            WILLAMETTE,SSE2
CVTPS2DQ        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTPS2PD        xmmreg,xmmrm            WILLAMETTE,SSE2
CVTSD2SI        reg32,xmmreg            WILLAMETTE,SSE2,AR1
CVTSD2SI        reg32,mem               WILLAMETTE,SSE2,AR1
CVTSD2SI        reg64,xmmreg            X64,SSE2,AR1
CVTSD2SI        reg64,mem               X64,SSE2,AR1
CVTSD2SS        xmmreg,xmmrm            WILLAMETTE,SSE2
CVTSI2SD        xmmreg,mem              WILLAMETTE,SSE2,SD,AR1,ND
CVTSI2SD        xmmreg,rm32             WILLAMETTE,SSE2,SD,AR1
CVTSI2SD        xmmreg,rm64             X64,SSE2,AR1
CVTSS2SD        xmmreg,xmmrm            WILLAMETTE,SSE2,SD
CVTTPD2PI       mmxreg,xmmrm            WILLAMETTE,SSE2,SO
CVTTPD2DQ       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTTPS2DQ       xmmreg,xmmrm            WILLAMETTE,SSE2,SO
CVTTSD2SI       reg32,xmmreg            WILLAMETTE,SSE2,AR1
CVTTSD2SI       reg32,mem               WILLAMETTE,SSE2,AR1
CVTTSD2SI       reg64,xmmreg            X64,SSE2,AR1
CVTTSD2SI       reg64,mem               X64,SSE2,AR1
DIVPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
DIVSD           xmmreg,xmmrm            WILLAMETTE,SSE2
MAXPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
MAXSD           xmmreg,xmmrm            WILLAMETTE,SSE2
MINPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
MINSD           xmmreg,xmmrm            WILLAMETTE,SSE2
MOVAPD          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVAPD          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVAPD          mem,xmmreg              WILLAMETTE,SSE2,SO
MOVAPD          xmmreg,mem              WILLAMETTE,SSE2,SO
MOVHPD          mem,xmmreg              WILLAMETTE,SSE2
MOVHPD          xmmreg,mem              WILLAMETTE,SSE2
MOVLPD          mem,xmmreg              WILLAMETTE,SSE2
MOVLPD          xmmreg,mem              WILLAMETTE,SSE2
```

```
MOVMSKPD        reg32,xmmreg            WILLAMETTE,SSE2
MOVMSKPD        reg64,xmmreg            X64,SSE2
MOVSD           xmmreg,xmmreg           WILLAMETTE,SSE2
MOVSD           xmmreg,xmmreg           WILLAMETTE,SSE2
MOVSD           mem,xmmreg              WILLAMETTE,SSE2
MOVSD           xmmreg,mem              WILLAMETTE,SSE2
MOVUPD          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVUPD          xmmreg,xmmreg           WILLAMETTE,SSE2
MOVUPD          mem,xmmreg              WILLAMETTE,SSE2,SO
MOVUPD          xmmreg,mem              WILLAMETTE,SSE2,SO
MULPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
MULSD           xmmreg,xmmrm            WILLAMETTE,SSE2
ORPD            xmmreg,xmmrm            WILLAMETTE,SSE2,SO
SHUFPD          xmmreg,xmmreg,imm       WILLAMETTE,SSE2
SHUFPD          xmmreg,mem,imm          WILLAMETTE,SSE2
SQRTPD          xmmreg,xmmrm            WILLAMETTE,SSE2,SO
SQRTSD          xmmreg,xmmrm            WILLAMETTE,SSE2
SUBPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
SUBSD           xmmreg,xmmrm            WILLAMETTE,SSE2
UCOMISD         xmmreg,xmmrm            WILLAMETTE,SSE2
UNPCKHPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
UNPCKLPD        xmmreg,xmmrm            WILLAMETTE,SSE2,SO
XORPD           xmmreg,xmmrm            WILLAMETTE,SSE2,SO
```

B.1.12 Prescott New Instructions (SSE3)

```
ADDSUBPD        xmmreg,xmmrm            PRESCOTT,SSE3,SO
ADDSUBPS        xmmreg,xmmrm            PRESCOTT,SSE3,SO
HADDPD          xmmreg,xmmrm            PRESCOTT,SSE3,SO
HADDPS          xmmreg,xmmrm            PRESCOTT,SSE3,SO
HSUBPD          xmmreg,xmmrm            PRESCOTT,SSE3,SO
HSUBPS          xmmreg,xmmrm            PRESCOTT,SSE3,SO
LDDQU           xmmreg,mem              PRESCOTT,SSE3,SO
MOVDDUP         xmmreg,xmmrm            PRESCOTT,SSE3
MOVSHDUP        xmmreg,xmmrm            PRESCOTT,SSE3
MOVSLDUP        xmmreg,xmmrm            PRESCOTT,SSE3
```

B.1.13 VMX Instructions

```
VMCALL                                  VMX
VMCLEAR         mem                     VMX
VMLAUNCH                                VMX
VMLOAD                                  X64,VMX
VMMCALL                                 X64,VMX
VMPTRLD         mem                     VMX
VMPTRST         mem                     VMX
VMREAD          rm32,reg32              VMX,NOLONG,SD
VMREAD          rm64,reg64              X64,VMX
VMRESUME                                VMX
VMRUN                                   X64,VMX
VMSAVE                                  X64,VMX
VMWRITE         reg32,rm32              VMX,NOLONG,SD
VMWRITE         reg64,rm64              X64,VMX
VMXOFF                                  VMX
VMXON           mem                     VMX
```

B.1.14 Extended Page Tables VMX instructions

```
INVEPT          reg32,mem               VMX,SO,NOLONG
INVEPT          reg64,mem               VMX,SO,LONG
INVVPID         reg32,mem               VMX,SO,NOLONG
INVVPID         reg64,mem               VMX,SO,LONG
```

B.1.15 Tejas New Instructions (SSSE3)

```
PABSB           mmxreg,mmxrm            SSSE3,MMX
PABSB           xmmreg,xmmrm            SSSE3
PABSW           mmxreg,mmxrm            SSSE3,MMX
PABSW           xmmreg,xmmrm            SSSE3
PABSD           mmxreg,mmxrm            SSSE3,MMX
PABSD           xmmreg,xmmrm            SSSE3
PALIGNR         mmxreg,mmxrm,imm        SSSE3,MMX
PALIGNR         xmmreg,xmmrm,imm        SSSE3
PHADDW          mmxreg,mmxrm            SSSE3,MMX
PHADDW          xmmreg,xmmrm            SSSE3
PHADDD          mmxreg,mmxrm            SSSE3,MMX
PHADDD          xmmreg,xmmrm            SSSE3
PHADDSW         mmxreg,mmxrm            SSSE3,MMX
PHADDSW         xmmreg,xmmrm            SSSE3
PHSUBW          mmxreg,mmxrm            SSSE3,MMX
PHSUBW          xmmreg,xmmrm            SSSE3
PHSUBD          mmxreg,mmxrm            SSSE3,MMX
PHSUBD          xmmreg,xmmrm            SSSE3
PHSUBSW         mmxreg,mmxrm            SSSE3,MMX
PHSUBSW         xmmreg,xmmrm            SSSE3
PMADDUBSW       mmxreg,mmxrm            SSSE3,MMX
PMADDUBSW       xmmreg,xmmrm            SSSE3
PMULHRSW        mmxreg,mmxrm            SSSE3,MMX
PMULHRSW        xmmreg,xmmrm            SSSE3
PSHUFB          mmxreg,mmxrm            SSSE3,MMX
PSHUFB          xmmreg,xmmrm            SSSE3
PSIGNB          mmxreg,mmxrm            SSSE3,MMX
PSIGNB          xmmreg,xmmrm            SSSE3
PSIGNW          mmxreg,mmxrm            SSSE3,MMX
PSIGNW          xmmreg,xmmrm            SSSE3
PSIGND          mmxreg,mmxrm            SSSE3,MMX
PSIGND          xmmreg,xmmrm            SSSE3
```

B.1.16 AMD SSE4A

```
EXTRQ           xmmreg,imm,imm          SSE4A,AMD
EXTRQ           xmmreg,xmmreg           SSE4A,AMD
INSERTQ         xmmreg,xmmreg,imm,imm   SSE4A,AMD
INSERTQ         xmmreg,xmmreg           SSE4A,AMD
MOVNTSD         mem,xmmreg              SSE4A,AMD
MOVNTSS         mem,xmmreg              SSE4A,AMD,SD
```

B.1.17 New instructions in Barcelona

```
LZCNT           reg16,rm16              P6,AMD
LZCNT           reg32,rm32              P6,AMD
LZCNT           reg64,rm64              X64,AMD
```

B.1.18 Penryn New Instructions (SSE4.1)

```
BLENDPD         xmmreg,xmmrm,imm        SSE41
BLENDPS         xmmreg,xmmrm,imm        SSE41
BLENDVPD        xmmreg,xmmrm,xmm0       SSE41
BLENDVPS        xmmreg,xmmrm,xmm0       SSE41
DPPD            xmmreg,xmmrm,imm        SSE41
DPPS            xmmreg,xmmrm,imm        SSE41
EXTRACTPS       rm32,xmmreg,imm         SSE41
EXTRACTPS       reg64,xmmreg,imm        SSE41,X64
INSERTPS        xmmreg,xmmrm,imm        SSE41,SD
MOVNTDQA        xmmreg,mem              SSE41
MPSADBW         xmmreg,xmmrm,imm        SSE41
PACKUSDW        xmmreg,xmmrm            SSE41
PBLENDVB        xmmreg,xmmrm,xmm0       SSE41
PBLENDW         xmmreg,xmmrm,imm        SSE41
PCMPEQQ         xmmreg,xmmrm            SSE41
PEXTRB          reg32,xmmreg,imm        SSE41
PEXTRB          mem8,xmmreg,imm         SSE41
PEXTRB          reg64,xmmreg,imm        SSE41,X64
PEXTRD          rm32,xmmreg,imm         SSE41
PEXTRQ          rm64,xmmreg,imm         SSE41,X64
PEXTRW          reg32,xmmreg,imm        SSE41
PEXTRW          mem16,xmmreg,imm        SSE41
PEXTRW          reg64,xmmreg,imm        SSE41,X64
PHMINPOSUW      xmmreg,xmmrm            SSE41
PINSRB          xmmreg,mem,imm          SSE41
PINSRB          xmmreg,rm8,imm          SSE41
PINSRB          xmmreg,reg32,imm        SSE41
PINSRD          xmmreg,mem,imm          SSE41
PINSRD          xmmreg,rm32,imm         SSE41
PINSRQ          xmmreg,mem,imm          SSE41,X64
PINSRQ          xmmreg,rm64,imm         SSE41,X64
PMAXSB          xmmreg,xmmrm            SSE41
PMAXSD          xmmreg,xmmrm            SSE41
PMAXUD          xmmreg,xmmrm            SSE41
PMAXUW          xmmreg,xmmrm            SSE41
PMINSB          xmmreg,xmmrm            SSE41
PMINSD          xmmreg,xmmrm            SSE41
PMINUD          xmmreg,xmmrm            SSE41
PMINUW          xmmreg,xmmrm            SSE41
PMOVSXBW        xmmreg,xmmrm            SSE41
PMOVSXBD        xmmreg,xmmrm            SSE41,SD
PMOVSXBQ        xmmreg,xmmrm            SSE41,SW
PMOVSXWD        xmmreg,xmmrm            SSE41
PMOVSXWQ        xmmreg,xmmrm            SSE41,SD
PMOVSXDQ        xmmreg,xmmrm            SSE41
PMOVZXBW        xmmreg,xmmrm            SSE41
PMOVZXBD        xmmreg,xmmrm            SSE41,SD
PMOVZXBQ        xmmreg,xmmrm            SSE41,SW
PMOVZXWD        xmmreg,xmmrm            SSE41
PMOVZXWQ        xmmreg,xmmrm            SSE41,SD
PMOVZXDQ        xmmreg,xmmrm            SSE41
PMULDQ          xmmreg,xmmrm            SSE41
```

```
PMULLD          xmmreg,xmmrm            SSE41
PTEST           xmmreg,xmmrm            SSE41
ROUNDPD         xmmreg,xmmrm,imm        SSE41
ROUNDPS         xmmreg,xmmrm,imm        SSE41
ROUNDSD         xmmreg,xmmrm,imm        SSE41
ROUNDSS         xmmreg,xmmrm,imm        SSE41
```

B.1.19 Nehalem New Instructions (SSE4.2)

```
CRC32           reg32,rm8               SSE42
CRC32           reg32,rm16              SSE42
CRC32           reg32,rm32              SSE42
CRC32           reg64,rm8               SSE42,X64
CRC32           reg64,rm64              SSE42,X64
PCMPESTRI       xmmreg,xmmrm,imm        SSE42
PCMPESTRM       xmmreg,xmmrm,imm        SSE42
PCMPISTRI       xmmreg,xmmrm,imm        SSE42
PCMPISTRM       xmmreg,xmmrm,imm        SSE42
PCMPGTQ         xmmreg,xmmrm            SSE42
POPCNT          reg16,rm16              NEHALEM,SW
POPCNT          reg32,rm32              NEHALEM,SD
POPCNT          reg64,rm64              NEHALEM,X64
```

B.1.20 Intel SMX

```
GETSEC                                  KATMAI
```

B.1.21 Geode (Cyrix) 3DNow! additions

```
PFRCPV          mmxreg,mmxrm            PENT,3DNOW,CYRIX
PFRSQRTV        mmxreg,mmxrm            PENT,3DNOW,CYRIX
```

B.1.22 Intel new instructions in ???

```
MOVBE           reg16,mem16             NEHALEM
MOVBE           reg32,mem32             NEHALEM
MOVBE           reg64,mem64             NEHALEM
MOVBE           mem16,reg16             NEHALEM
MOVBE           mem32,reg32             NEHALEM
MOVBE           mem64,reg64             NEHALEM
```

B.1.23 Intel AES instructions

```
AESENC          xmmreg,xmmrm128         SSE,WESTMERE
AESENCLAST      xmmreg,xmmrm128         SSE,WESTMERE
AESDEC          xmmreg,xmmrm128         SSE,WESTMERE
AESDECLAST      xmmreg,xmmrm128         SSE,WESTMERE
AESIMC          xmmreg,xmmrm128         SSE,WESTMERE
AESKEYGENASSIST xmmreg,xmmrm128,imm8    SSE,WESTMERE
```

B.1.24 Intel AVX AES instructions

```
VAESENC         xmmreg,xmmreg*,xmmrm128 AVX,SANDYBRIDGE
VAESENCLAST     xmmreg,xmmreg*,xmmrm128 AVX,SANDYBRIDGE
VAESDEC         xmmreg,xmmreg*,xmmrm128 AVX,SANDYBRIDGE
VAESDECLAST     xmmreg,xmmreg*,xmmrm128 AVX,SANDYBRIDGE
VAESIMC         xmmreg,xmmrm128         AVX,SANDYBRIDGE
VAESKEYGENASSIST xmmreg,xmmrm128,imm8   AVX,SANDYBRIDGE
```

B.1.25 Intel AVX instructions

```
VADDPD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VADDPD          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VADDPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VADDPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VADDSD          xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VADDSS          xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VADDSUBPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VADDSUBPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VADDSUBPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VADDSUBPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VANDPD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VANDPD          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VANDPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VANDPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VANDNPD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VANDNPD         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VANDNPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VANDNPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VBLENDPD        xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VBLENDPD        ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VBLENDPS        xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VBLENDPS        ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VBLENDVPD       xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VBLENDVPD       xmmreg,xmmrm128,xmm0     AVX,SANDYBRIDGE
VBLENDVPD       ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VBLENDVPD       ymmreg,ymmrm256,ymm0     AVX,SANDYBRIDGE
VBLENDVPS       xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VBLENDVPS       xmmreg,xmmrm128,xmm0     AVX,SANDYBRIDGE
VBLENDVPS       ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VBLENDVPD       ymmreg,ymmrm256,ymm0     AVX,SANDYBRIDGE
VBROADCASTSS    xmmreg,mem32             AVX,SANDYBRIDGE
VBROADCASTSS    ymmreg,mem32             AVX,SANDYBRIDGE
VBROADCASTSD    ymmreg,mem64             AVX,SANDYBRIDGE
VBROADCASTF128  ymmreg,mem128            AVX,SANDYBRIDGE
VCMPEQPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLTPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLTPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLEPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLEPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPUNORDPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPUNORDPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLTPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLTPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLEPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLEPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPORDPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPORDPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_UQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_UQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
```

```
VCMPNGEPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGEPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGTPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGTPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPFALSEPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPFALSEPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_OQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_OQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGEPD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGEPD         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGTPD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGTPD         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPTRUEPD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPTRUEPD       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_OSPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_OSPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLT_OQPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLT_OQPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLE_OQPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLE_OQPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPUNORD_SPD    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPUNORD_SPD    ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_USPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_USPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLT_UQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLT_UQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLE_UQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLE_UQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPORD_SPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPORD_SPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_USPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_USPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGE_UQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGE_UQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGT_UQPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGT_UQPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPFALSE_OSPD   xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPFALSE_OSPD   ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_OSPD     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_OSPD     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGE_OQPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGE_OQPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGT_OQPD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGT_OQPD      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPTRUE_USPD    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPTRUE_USPD    ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPPD           xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VCMPPD           ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VCMPEQPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLTPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLTPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLEPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
```

```
VCMPLEPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPUNORDPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPUNORDPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLTPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLTPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLEPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLEPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPORDPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPORDPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_UQPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_UQPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGEPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGEPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGTPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGTPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPFALSEPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPFALSEPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_OQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_OQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGEPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGEPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGTPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGTPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPTRUEPS        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPTRUEPS        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_OSPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_OSPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLT_OQPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLT_OQPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPLE_OQPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPLE_OQPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPUNORD_SPS     xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPUNORD_SPS     ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_USPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_USPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLT_UQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLT_UQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNLE_UQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNLE_UQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPORD_SPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPORD_SPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPEQ_USPS       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPEQ_USPS       ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGE_UQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGE_UQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNGT_UQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNGT_UQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPFALSE_OSPS    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPFALSE_OSPS    ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPNEQ_OSPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPNEQ_OSPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
```

```
VCMPGE_OQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGE_OQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPGT_OQPS      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPGT_OQPS      ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPTRUE_USPS    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VCMPTRUE_USPS    ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VCMPPS           xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VCMPPS           ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VCMPEQSD         xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPLTSD         xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPLESD         xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPUNORDSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNEQSD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNLTSD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNLESD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPORDSD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPEQ_UQSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNGESD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNGTSD        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPFALSESD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNEQ_OQSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPGESD         xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPGTSD         xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPTRUESD       xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPEQ_OSSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPLT_OQSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPLE_OQSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPUNORD_SSD    xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNEQ_USSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNLT_UQSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNLE_UQSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPORD_SSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPEQ_USSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNGE_UQSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNGT_UQSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPFALSE_OSSD   xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPNEQ_OSSD     xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPGE_OQSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPGT_OQSD      xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPTRUE_USSD    xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCMPSD           xmmreg,xmmreg*,xmmrm64,imm8 AVX,SANDYBRIDGE
VCMPEQSS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPLTSS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPLESS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPUNORDSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNEQSS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNLTSS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNLESS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPORDSS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPEQ_UQSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNGESS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNGTSS        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPFALSESS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
```

```
VCMPNEQ_OQSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPGESS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPGTSS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPTRUESS       xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPEQ_OSSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPLT_OQSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPLE_OQSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPUNORD_SSS    xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNEQ_USSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNLT_UQSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNLE_UQSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPORD_SSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPEQ_USSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNGE_UQSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNGT_UQSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPFALSE_OSSS   xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPNEQ_OSSS     xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPGE_OQSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPGT_OQSS      xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPTRUE_USSS    xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCMPSS           xmmreg,xmmreg*,xmmrm32,imm8 AVX,SANDYBRIDGE
VCOMISD          xmmreg,xmmrm64           AVX,SANDYBRIDGE
VCOMISS          xmmreg,xmmrm32           AVX,SANDYBRIDGE
VCVTDQ2PD        xmmreg,xmmrm64           AVX,SANDYBRIDGE
VCVTDQ2PD        ymmreg,xmmrm128          AVX,SANDYBRIDGE
VCVTDQ2PS        xmmreg,xmmrm128          AVX,SANDYBRIDGE
VCVTDQ2PS        ymmreg,ymmrm256          AVX,SANDYBRIDGE
VCVTPD2DQ        xmmreg,xmmreg            AVX,SANDYBRIDGE
VCVTPD2DQ        xmmreg,mem128            AVX,SANDYBRIDGE,SO
VCVTPD2DQ        xmmreg,ymmreg            AVX,SANDYBRIDGE
VCVTPD2DQ        xmmreg,mem256            AVX,SANDYBRIDGE,SY
VCVTPD2PS        xmmreg,xmmreg            AVX,SANDYBRIDGE
VCVTPD2PS        xmmreg,mem128            AVX,SANDYBRIDGE,SO
VCVTPD2PS        xmmreg,ymmreg            AVX,SANDYBRIDGE
VCVTPD2PS        xmmreg,mem256            AVX,SANDYBRIDGE,SY
VCVTPS2DQ        xmmreg,xmmrm128          AVX,SANDYBRIDGE
VCVTPS2DQ        ymmreg,ymmrm256          AVX,SANDYBRIDGE
VCVTPS2PD        xmmreg,xmmrm64           AVX,SANDYBRIDGE
VCVTPS2PD        ymmreg,xmmrm128          AVX,SANDYBRIDGE
VCVTSD2SI        reg32,xmmrm64            AVX,SANDYBRIDGE
VCVTSD2SI        reg64,xmmrm64            AVX,SANDYBRIDGE,LONG
VCVTSD2SS        xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VCVTSI2SD        xmmreg,xmmreg*,rm32      AVX,SANDYBRIDGE,SD
VCVTSI2SD        xmmreg,xmmreg*,mem32     AVX,SANDYBRIDGE,ND,SD
VCVTSI2SD        xmmreg,xmmreg*,rm64      AVX,SANDYBRIDGE,LONG
VCVTSI2SS        xmmreg,xmmreg*,rm32      AVX,SANDYBRIDGE,SD
VCVTSI2SS        xmmreg,xmmreg*,mem32     AVX,SANDYBRIDGE,ND,SD
VCVTSI2SS        xmmreg,xmmreg*,rm64      AVX,SANDYBRIDGE,LONG
VCVTSS2SD        xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VCVTSS2SI        reg32,xmmrm32            AVX,SANDYBRIDGE
VCVTSS2SI        reg64,xmmrm32            AVX,SANDYBRIDGE,LONG
VCVTTPD2DQ       xmmreg,xmmreg            AVX,SANDYBRIDGE
VCVTTPD2DQ       xmmreg,mem128            AVX,SANDYBRIDGE,SO
```

```
VCVTTPD2DQ      xmmreg,ymmreg           AVX,SANDYBRIDGE
VCVTTPD2DQ      xmmreg,mem256           AVX,SANDYBRIDGE,SY
VCVTTPS2DQ      xmmreg,xmmrm128         AVX,SANDYBRIDGE
VCVTTPS2DQ      ymmreg,ymmrm256         AVX,SANDYBRIDGE
VCVTTSD2SI      reg32,xmmrm64           AVX,SANDYBRIDGE
VCVTTSD2SI      reg64,xmmrm64           AVX,SANDYBRIDGE,LONG
VCVTTSS2SI      reg32,xmmrm32           AVX,SANDYBRIDGE
VCVTTSS2SI      reg64,xmmrm32           AVX,SANDYBRIDGE,LONG
VDIVPD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VDIVPD          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VDIVPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VDIVPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VDIVSD          xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VDIVSS          xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VDPPD           xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VDPPS           xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VDPPS           ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VEXTRACTF128    xmmrm128,xmmreg,imm8    AVX,SANDYBRIDGE
VEXTRACTPS      rm32,xmmreg,imm8        AVX,SANDYBRIDGE
VHADDPD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VHADDPD         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VHADDPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VHADDPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VHSUBPD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VHSUBPD         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VHSUBPS         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VHSUBPS         ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VINSERTF128     ymmreg,ymmreg,xmmrm128,imm8 AVX,SANDYBRIDGE
VINSERTPS       xmmreg,xmmreg*,xmmrm32,imm8 AVX,SANDYBRIDGE
VLDDQU          xmmreg,mem128           AVX,SANDYBRIDGE
VLDQQU          ymmreg,mem256           AVX,SANDYBRIDGE
VLDDQU          ymmreg,mem256           AVX,SANDYBRIDGE
VLDMXCSR        mem32                   AVX,SANDYBRIDGE
VMASKMOVDQU     xmmreg,xmmreg           AVX,SANDYBRIDGE
VMASKMOVPS      xmmreg,xmmreg,mem128    AVX,SANDYBRIDGE
VMASKMOVPS      ymmreg,ymmreg,mem256    AVX,SANDYBRIDGE
VMASKMOVPS      mem128,xmmreg,xmmreg    AVX,SANDYBRIDGE,SO
VMASKMOVPS      mem256,xmmreg,xmmreg    AVX,SANDYBRIDGE,SY
VMASKMOVPD      xmmreg,xmmreg,mem128    AVX,SANDYBRIDGE
VMASKMOVPD      ymmreg,ymmreg,mem256    AVX,SANDYBRIDGE
VMASKMOVPD      mem128,xmmreg,xmmreg    AVX,SANDYBRIDGE
VMASKMOVPD      mem256,ymmreg,ymmreg    AVX,SANDYBRIDGE
VMAXPD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VMAXPD          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VMAXPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VMAXPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VMAXSD          xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VMAXSS          xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VMINPD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VMINPD          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VMINPS          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VMINPS          ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VMINSD          xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
```

```
VMINSS          xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VMOVAPD         xmmreg,xmmrm128          AVX,SANDYBRIDGE
VMOVAPD         xmmrm128,xmmreg          AVX,SANDYBRIDGE
VMOVAPD         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVAPD         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVAPS         xmmreg,xmmrm128          AVX,SANDYBRIDGE
VMOVAPS         xmmrm128,xmmreg          AVX,SANDYBRIDGE
VMOVAPS         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVAPS         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVQ           xmmreg,xmmrm64           AVX,SANDYBRIDGE
VMOVQ           xmmrm64,xmmreg           AVX,SANDYBRIDGE
VMOVQ           xmmreg,rm64              AVX,SANDYBRIDGE,LONG
VMOVQ           rm64,xmmreg              AVX,SANDYBRIDGE,LONG
VMOVD           xmmreg,rm32              AVX,SANDYBRIDGE
VMOVD           rm32,xmmreg              AVX,SANDYBRIDGE
VMOVDDUP        xmmreg,xmmrm64           AVX,SANDYBRIDGE
VMOVDDUP        ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVDQA         xmmreg,xmmrm128          AVX,SANDYBRIDGE
VMOVDQA         xmmrm128,xmmreg          AVX,SANDYBRIDGE
VMOVQQA         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVQQA         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVDQA         ymmreg,ymmrm             AVX,SANDYBRIDGE
VMOVDQA         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVDQU         xmmreg,xmmrm128          AVX,SANDYBRIDGE
VMOVDQU         xmmrm128,xmmreg          AVX,SANDYBRIDGE
VMOVQQU         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVQQU         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVDQU         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VMOVDQU         ymmrm256,ymmreg          AVX,SANDYBRIDGE
VMOVHLPS        xmmreg,xmmreg*,xmmreg    AVX,SANDYBRIDGE
VMOVHPD         xmmreg,xmmreg*,mem64     AVX,SANDYBRIDGE
VMOVHPD         mem64,xmmreg             AVX,SANDYBRIDGE
VMOVHPS         xmmreg,xmmreg*,mem64     AVX,SANDYBRIDGE
VMOVHPS         mem64,xmmreg             AVX,SANDYBRIDGE
VMOVLHPS        xmmreg,xmmreg*,xmmreg    AVX,SANDYBRIDGE
VMOVLPD         xmmreg,xmmreg*,mem64     AVX,SANDYBRIDGE
VMOVLPD         mem64,xmmreg             AVX,SANDYBRIDGE
VMOVLPS         xmmreg,xmmreg*,mem64     AVX,SANDYBRIDGE
VMOVLPS         mem64,xmmreg             AVX,SANDYBRIDGE
VMOVMSKPD       reg64,xmmreg             AVX,SANDYBRIDGE,LONG
VMOVMSKPD       reg32,xmmreg             AVX,SANDYBRIDGE
VMOVMSKPD       reg64,ymmreg             AVX,SANDYBRIDGE,LONG
VMOVMSKPD       reg32,ymmreg             AVX,SANDYBRIDGE
VMOVMSKPS       reg64,xmmreg             AVX,SANDYBRIDGE,LONG
VMOVMSKPS       reg32,xmmreg             AVX,SANDYBRIDGE
VMOVMSKPS       reg64,ymmreg             AVX,SANDYBRIDGE,LONG
VMOVMSKPS       reg32,ymmreg             AVX,SANDYBRIDGE
VMOVNTDQ        mem128,xmmreg            AVX,SANDYBRIDGE
VMOVNTQQ        mem256,ymmreg            AVX,SANDYBRIDGE
VMOVNTDQ        mem256,ymmreg            AVX,SANDYBRIDGE
VMOVNTDQA       xmmreg,mem128            AVX,SANDYBRIDGE
VMOVNTPD        mem128,xmmreg            AVX,SANDYBRIDGE
VMOVNTPD        mem256,ymmreg            AVX,SANDYBRIDGE
```

```
VMOVNTPS        mem128,xmmreg             AVX,SANDYBRIDGE
VMOVNTPS        mem128,ymmreg             AVX,SANDYBRIDGE
VMOVSD          xmmreg,xmmreg*,xmmreg     AVX,SANDYBRIDGE
VMOVSD          xmmreg,mem64              AVX,SANDYBRIDGE
VMOVSD          xmmreg,xmmreg*,xmmreg     AVX,SANDYBRIDGE
VMOVSD          mem64,xmmreg              AVX,SANDYBRIDGE
VMOVSHDUP       xmmreg,xmmrm128           AVX,SANDYBRIDGE
VMOVSHDUP       ymmreg,ymmrm256           AVX,SANDYBRIDGE
VMOVSLDUP       xmmreg,xmmrm128           AVX,SANDYBRIDGE
VMOVSLDUP       ymmreg,ymmrm256           AVX,SANDYBRIDGE
VMOVSS          xmmreg,xmmreg*,xmmreg     AVX,SANDYBRIDGE
VMOVSS          xmmreg,mem64              AVX,SANDYBRIDGE
VMOVSS          xmmreg,xmmreg*,xmmreg     AVX,SANDYBRIDGE
VMOVSS          mem64,xmmreg              AVX,SANDYBRIDGE
VMOVUPD         xmmreg,xmmrm128           AVX,SANDYBRIDGE
VMOVUPD         xmmrm128,xmmreg           AVX,SANDYBRIDGE
VMOVUPD         ymmreg,ymmrm256           AVX,SANDYBRIDGE
VMOVUPD         ymmrm256,ymmreg           AVX,SANDYBRIDGE
VMOVUPS         xmmreg,xmmrm128           AVX,SANDYBRIDGE
VMOVUPS         xmmrm128,xmmreg           AVX,SANDYBRIDGE
VMOVUPS         ymmreg,ymmrm256           AVX,SANDYBRIDGE
VMOVUPS         ymmrm256,ymmreg           AVX,SANDYBRIDGE
VMPSADBW        xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VMULPD          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VMULPD          ymmreg,ymmreg*,ymmrm256   AVX,SANDYBRIDGE
VMULPS          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VMULPS          ymmreg,ymmreg*,ymmrm256   AVX,SANDYBRIDGE
VMULSD          xmmreg,xmmreg*,xmmrm64    AVX,SANDYBRIDGE
VMULSS          xmmreg,xmmreg*,xmmrm32    AVX,SANDYBRIDGE
VORPD           xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VORPD           ymmreg,ymmreg*,ymmrm256   AVX,SANDYBRIDGE
VORPS           xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VORPS           ymmreg,ymmreg*,ymmrm256   AVX,SANDYBRIDGE
VPABSB          xmmreg,xmmrm128           AVX,SANDYBRIDGE
VPABSW          xmmreg,xmmrm128           AVX,SANDYBRIDGE
VPABSD          xmmreg,xmmrm128           AVX,SANDYBRIDGE
VPACKSSWB       xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPACKSSDW       xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPACKUSWB       xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPACKUSDW       xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDB          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDW          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDD          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDQ          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDSB         xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDSW         xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDUSB        xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPADDUSW        xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPALIGNR        xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VPAND           xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPANDN          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPAVGB          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
VPAVGW          xmmreg,xmmreg*,xmmrm128   AVX,SANDYBRIDGE
```

```
VPBLENDVB       xmmreg,xmmreg*,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPBLENDW        xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VPCMPESTRI      xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPCMPESTRM      xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPCMPISTRI      xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPCMPISTRM      xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPCMPEQB        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPEQW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPEQD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPEQQ        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPGTB        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPGTW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPGTD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCMPGTQ        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPERMILPD       xmmreg,xmmreg,xmmrm128   AVX,SANDYBRIDGE
VPERMILPD       ymmreg,ymmreg,ymmrm256   AVX,SANDYBRIDGE
VPERMILPD       xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPERMILPD       ymmreg,ymmrm256,imm8     AVX,SANDYBRIDGE
VPERMILTD2PD    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILTD2PD    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILTD2PD    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILTD2PD    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMILMO2PD    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILMO2PD    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILMO2PD    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILMO2PD    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMILMZ2PD    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILMZ2PD    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILMZ2PD    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILMZ2PD    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMIL2PD      xmmreg,xmmreg,xmmrm128,xmmreg,imm8 AVX,SANDYBRIDGE
VPERMIL2PD      xmmreg,xmmreg,xmmreg,xmmrm128,imm8 AVX,SANDYBRIDGE
VPERMIL2PD      ymmreg,ymmreg,ymmrm256,ymmreg,imm8 AVX,SANDYBRIDGE
VPERMIL2PD      ymmreg,ymmreg,ymmreg,ymmrm256,imm8 AVX,SANDYBRIDGE
VPERMILPS       xmmreg,xmmreg,xmmrm128   AVX,SANDYBRIDGE
VPERMILPS       ymmreg,ymmreg,ymmrm256   AVX,SANDYBRIDGE
VPERMILPS       xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPERMILPS       ymmreg,ymmrm256,imm8     AVX,SANDYBRIDGE
VPERMILTD2PS    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILTD2PS    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILTD2PS    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILTD2PS    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMILMO2PS    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILMO2PS    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILMO2PS    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILMO2PS    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMILMZ2PS    xmmreg,xmmreg,xmmrm128,xmmreg AVX,SANDYBRIDGE
VPERMILMZ2PS    xmmreg,xmmreg,xmmreg,xmmrm128 AVX,SANDYBRIDGE
VPERMILMZ2PS    ymmreg,ymmreg,ymmrm256,ymmreg AVX,SANDYBRIDGE
VPERMILMZ2PS    ymmreg,ymmreg,ymmreg,ymmrm256 AVX,SANDYBRIDGE
VPERMIL2PS      xmmreg,xmmreg,xmmrm128,xmmreg,imm8 AVX,SANDYBRIDGE
VPERMIL2PS      xmmreg,xmmreg,xmmreg,xmmrm128,imm8 AVX,SANDYBRIDGE
VPERMIL2PS      ymmreg,ymmreg,ymmrm256,ymmreg,imm8 AVX,SANDYBRIDGE
```

```
VPERMIL2PS      ymmreg,ymmreg,ymmreg,ymmrm256,imm8 AVX,SANDYBRIDGE
VPERM2F128      ymmreg,ymmreg,ymmrm256,imm8 AVX,SANDYBRIDGE
VPEXTRB         reg64,xmmreg,imm8        AVX,SANDYBRIDGE,LONG
VPEXTRB         reg32,xmmreg,imm8        AVX,SANDYBRIDGE
VPEXTRB         mem8,xmmreg,imm8         AVX,SANDYBRIDGE
VPEXTRW         reg64,xmmreg,imm8        AVX,SANDYBRIDGE,LONG
VPEXTRW         reg32,xmmreg,imm8        AVX,SANDYBRIDGE
VPEXTRW         mem16,xmmreg,imm8        AVX,SANDYBRIDGE
VPEXTRW         reg64,xmmreg,imm8        AVX,SANDYBRIDGE,LONG
VPEXTRW         reg32,xmmreg,imm8        AVX,SANDYBRIDGE
VPEXTRW         mem16,xmmreg,imm8        AVX,SANDYBRIDGE
VPEXTRD         reg64,xmmreg,imm8        AVX,SANDYBRIDGE,LONG
VPEXTRD         rm32,xmmreg,imm8         AVX,SANDYBRIDGE
VPEXTRQ         rm64,xmmreg,imm8         AVX,SANDYBRIDGE,LONG
VPHADDW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPHADDD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPHADDSW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPHMINPOSUW     xmmreg,xmmrm128          AVX,SANDYBRIDGE
VPHSUBW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPHSUBD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPHSUBSW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPINSRB         xmmreg,xmmreg*,mem8,imm8 AVX,SANDYBRIDGE
VPINSRB         xmmreg,xmmreg*,rm8,imm8  AVX,SANDYBRIDGE
VPINSRB         xmmreg,xmmreg*,reg32,imm8 AVX,SANDYBRIDGE
VPINSRW         xmmreg,xmmreg*,mem16,imm8 AVX,SANDYBRIDGE
VPINSRW         xmmreg,xmmreg*,rm16,imm8 AVX,SANDYBRIDGE
VPINSRW         xmmreg,xmmreg*,reg32,imm8 AVX,SANDYBRIDGE
VPINSRD         xmmreg,xmmreg*,mem32,imm8 AVX,SANDYBRIDGE
VPINSRD         xmmreg,xmmreg*,rm32,imm8 AVX,SANDYBRIDGE
VPINSRQ         xmmreg,xmmreg*,mem64,imm8 AVX,SANDYBRIDGE,LONG
VPINSRQ         xmmreg,xmmreg*,rm64,imm8 AVX,SANDYBRIDGE,LONG
VPMADDWD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMADDUBSW      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXSB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXSW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXSD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXUB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXUW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMAXUD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINSB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINSW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINSD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINUB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINUW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMINUD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMOVMSKB       reg64,xmmreg             AVX,SANDYBRIDGE,LONG
VPMOVMSKB       reg32,xmmreg             AVX,SANDYBRIDGE
VPMOVSXBW       xmmreg,xmmrm64           AVX,SANDYBRIDGE
VPMOVSXBD       xmmreg,xmmrm32           AVX,SANDYBRIDGE
VPMOVSXBQ       xmmreg,xmmrm16           AVX,SANDYBRIDGE
VPMOVSXWD       xmmreg,xmmrm64           AVX,SANDYBRIDGE
VPMOVSXWQ       xmmreg,xmmrm32           AVX,SANDYBRIDGE
VPMOVSXDQ       xmmreg,xmmrm64           AVX,SANDYBRIDGE
```

```
VPMOVZXBW       xmmreg,xmmrm64           AVX,SANDYBRIDGE
VPMOVZXBD       xmmreg,xmmrm32           AVX,SANDYBRIDGE
VPMOVZXBQ       xmmreg,xmmrm16           AVX,SANDYBRIDGE
VPMOVZXWD       xmmreg,xmmrm64           AVX,SANDYBRIDGE
VPMOVZXWQ       xmmreg,xmmrm32           AVX,SANDYBRIDGE
VPMOVZXDQ       xmmreg,xmmrm64           AVX,SANDYBRIDGE
VPMULHUW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULHRSW       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULHW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULLW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULLD         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULUDQ        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPMULDQ         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPOR            xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSADBW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSHUFB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSHUFD         xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPSHUFHW        xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPSHUFLW        xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VPSIGNB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSIGNW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSIGND         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSLLDQ         xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRLDQ         xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSLLW          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSLLW          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSLLD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSLLD          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSLLQ          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSLLQ          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRAW          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSRAW          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRAD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSRAD          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRLW          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSRLW          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRLD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSRLD          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPSRLQ          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSRLQ          xmmreg,xmmreg*,imm8      AVX,SANDYBRIDGE
VPTEST          xmmreg,xmmrm128          AVX,SANDYBRIDGE
VPTEST          ymmreg,ymmrm256          AVX,SANDYBRIDGE
VPSUBB          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBW          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBD          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBQ          xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBSB         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBSW         xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBUSB        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPSUBUSW        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKHBW      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKHWD      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKHDQ      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
```

```
VPUNPCKHQDQ      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKLBW       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKLWD       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKLDQ       xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPUNPCKLQDQ      xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPXOR            xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VRCPPS           xmmreg,xmmrm128          AVX,SANDYBRIDGE
VRCPPS           ymmreg,ymmrm256          AVX,SANDYBRIDGE
VRCPSS           xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VRSQRTPS         xmmreg,xmmrm128          AVX,SANDYBRIDGE
VRSQRTPS         ymmreg,ymmrm256          AVX,SANDYBRIDGE
VRSQRTSS         xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VROUNDPD         xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VROUNDPD         ymmreg,ymmrm256,imm8     AVX,SANDYBRIDGE
VROUNDPS         xmmreg,xmmrm128,imm8     AVX,SANDYBRIDGE
VROUNDPS         ymmreg,ymmrm256,imm8     AVX,SANDYBRIDGE
VROUNDSD         xmmreg,xmmreg*,xmmrm64,imm8 AVX,SANDYBRIDGE
VROUNDSS         xmmreg,xmmreg*,xmmrm32,imm8 AVX,SANDYBRIDGE
VSHUFPD          xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VSHUFPD          ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VSHUFPS          xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
VSHUFPS          ymmreg,ymmreg*,ymmrm256,imm8 AVX,SANDYBRIDGE
VSQRTPD          xmmreg,xmmrm128          AVX,SANDYBRIDGE
VSQRTPD          ymmreg,ymmrm256          AVX,SANDYBRIDGE
VSQRTPS          xmmreg,xmmrm128          AVX,SANDYBRIDGE
VSQRTPS          ymmreg,ymmrm256          AVX,SANDYBRIDGE
VSQRTSD          xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VSQRTSS          xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VSTMXCSR         mem32                    AVX,SANDYBRIDGE
VSUBPD           xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VSUBPD           ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VSUBPS           xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VSUBPS           ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VSUBSD           xmmreg,xmmreg*,xmmrm64   AVX,SANDYBRIDGE
VSUBSS           xmmreg,xmmreg*,xmmrm32   AVX,SANDYBRIDGE
VTESTPS          xmmreg,xmmrm128          AVX,SANDYBRIDGE
VTESTPS          ymmreg,ymmrm256          AVX,SANDYBRIDGE
VTESTPD          xmmreg,xmmrm128          AVX,SANDYBRIDGE
VTESTPD          ymmreg,ymmrm256          AVX,SANDYBRIDGE
VUCOMISD         xmmreg,xmmrm64           AVX,SANDYBRIDGE
VUCOMISS         xmmreg,xmmrm32           AVX,SANDYBRIDGE
VUNPCKHPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VUNPCKHPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VUNPCKHPS        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VUNPCKHPS        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VUNPCKLPD        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VUNPCKLPD        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VUNPCKLPS        xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VUNPCKLPS        ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VXORPD           xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VXORPD           ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
VXORPS           xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VXORPS           ymmreg,ymmreg*,ymmrm256  AVX,SANDYBRIDGE
```

```
VZEROALL                                        AVX,SANDYBRIDGE
VZEROUPPER                                      AVX,SANDYBRIDGE
```

B.1.26 Intel Carry−Less Multiplication instructions (CLMUL)

```
PCLMULLQLQDQ     xmmreg,xmmrm128          SSE,WESTMERE
PCLMULHQLQDQ     xmmreg,xmmrm128          SSE,WESTMERE
PCLMULLQHQDQ     xmmreg,xmmrm128          SSE,WESTMERE
PCLMULHQHQDQ     xmmreg,xmmrm128          SSE,WESTMERE
PCLMULQDQ        xmmreg,xmmrm128,imm8     SSE,WESTMERE
```

B.1.27 Intel AVX Carry−Less Multiplication instructions (CLMUL)

```
VPCLMULLQLQDQ    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCLMULHQLQDQ    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCLMULLQHQDQ    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCLMULHQHQDQ    xmmreg,xmmreg*,xmmrm128  AVX,SANDYBRIDGE
VPCLMULQDQ       xmmreg,xmmreg*,xmmrm128,imm8 AVX,SANDYBRIDGE
```

B.1.28 Intel Fused Multiply−Add instructions (FMA)

```
VFMADD132PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD132PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD132PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD132PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD312PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD312PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD312PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD312PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD213PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD213PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD213PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD213PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD123PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD123PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD123PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD123PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD231PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD231PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD231PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD231PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD321PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD321PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD321PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADD321PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB132PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB132PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB132PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB132PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB312PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB312PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB312PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB312PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB213PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB213PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
```

```
VFMADDSUB213PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB213PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB123PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB123PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB123PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB123PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB231PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB231PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB231PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB231PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB321PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB321PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADDSUB321PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMADDSUB321PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB132PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB132PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB132PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB132PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB312PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB312PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB312PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB312PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB213PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB213PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB213PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB213PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB123PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB123PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB123PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB123PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB231PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB231PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB231PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB231PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB321PS      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB321PS      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUB321PD      xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUB321PD      ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD132PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD132PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD132PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD132PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD312PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD312PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD312PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD312PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD213PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD213PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD213PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD213PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD123PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD123PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD123PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
```

```
VFMSUBADD123PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD231PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD231PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD231PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD231PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD321PS   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD321PS   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMSUBADD321PD   xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFMSUBADD321PD   ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD132PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD132PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD132PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD132PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD312PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD312PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD312PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD312PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD213PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD213PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD213PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD213PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD123PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD123PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD123PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD123PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD231PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD231PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD231PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD231PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD321PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD321PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMADD321PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMADD321PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB132PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB132PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB132PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB132PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB312PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB312PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB312PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB312PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB213PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB213PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB213PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB213PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB123PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB123PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB123PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB123PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB231PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB231PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB231PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB231PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
```

```
VFNMSUB321PS     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB321PS     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFNMSUB321PD     xmmreg,xmmreg,xmmrm128   FMA,FUTURE
VFNMSUB321PD     ymmreg,ymmreg,ymmrm256   FMA,FUTURE
VFMADD132SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD132SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMADD312SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD312SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMADD213SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD213SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMADD123SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD123SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMADD231SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD231SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMADD321SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMADD321SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB132SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB132SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB312SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB312SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB213SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB213SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB123SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB123SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB231SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB231SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFMSUB321SS      xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFMSUB321SD      xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD132SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD132SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD312SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD312SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD213SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD213SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD123SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD123SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD231SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD231SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMADD321SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMADD321SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB132SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB132SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB312SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB312SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB213SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB213SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB123SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB123SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB231SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB231SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
VFNMSUB321SS     xmmreg,xmmreg,xmmrm32    FMA,FUTURE
VFNMSUB321SD     xmmreg,xmmreg,xmmrm64    FMA,FUTURE
```

B.1.29 Intel post–32 nm processor instructions

```
RDFSBASE        reg32                   LONG,FUTURE
RDFSBASE        reg64                   LONG,FUTURE
RDGSBASE        reg32                   LONG,FUTURE
RDGSBASE        reg64                   LONG,FUTURE
RDRAND          reg16                   FUTURE
RDRAND          reg32                   FUTURE
RDRAND          reg64                   LONG,FUTURE
WRFSBASE        reg32                   LONG,FUTURE
WRFSBASE        reg64                   LONG,FUTURE
WRGSBASE        reg32                   LONG,FUTURE
WRGSBASE        reg64                   LONG,FUTURE
VCVTPH2PS       ymmreg,xmmrm128         AVX,FUTURE
VCVTPH2PS       xmmreg,xmmrm64          AVX,FUTURE
VCVTPS2PH       xmmrm128,ymmreg,imm8    AVX,FUTURE
VCVTPS2PH       xmmrm64,xmmreg,imm8     AVX,FUTURE
```

B.1.30 VIA (Centaur) security instructions

```
XSTORE                                  PENT,CYRIX
XCRYPTECB                               PENT,CYRIX
XCRYPTCBC                               PENT,CYRIX
XCRYPTCTR                               PENT,CYRIX
XCRYPTCFB                               PENT,CYRIX
XCRYPTOFB                               PENT,CYRIX
MONTMUL                                 PENT,CYRIX
XSHA1                                   PENT,CYRIX
XSHA256                                 PENT,CYRIX
```

B.1.31 AMD Lightweight Profiling (LWP) instructions

```
LLWPCB          reg32                   AMD,386
LLWPCB          reg64                   AMD,X64
SLWPCB          reg32                   AMD,386
SLWPCB          reg64                   AMD,X64
LWPVAL          reg32,rm32,imm32        AMD,386
LWPVAL          reg64,rm32,imm32        AMD,X64
LWPINS          reg32,rm32,imm32        AMD,386
LWPINS          reg64,rm32,imm32        AMD,X64
```

B.1.32 AMD XOP and FMA4 instructions (SSE5)

```
VFMADDPD        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMADDPD        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMADDPD        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMADDPD        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMADDPS        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMADDPS        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMADDPS        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMADDPS        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMADDSD        xmmreg,xmmreg*,xmmrm64,xmmreg AMD,SSE5
VFMADDSD        xmmreg,xmmreg*,xmmreg,xmmrm64 AMD,SSE5
VFMADDSS        xmmreg,xmmreg*,xmmrm32,xmmreg AMD,SSE5
VFMADDSS        xmmreg,xmmreg*,xmmreg,xmmrm32 AMD,SSE5
VFMADDSUBPD     xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
```

```
VFMADDSUBPD      ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMADDSUBPD      xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMADDSUBPD      ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMADDSUBPS      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMADDSUBPS      ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMADDSUBPS      xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMADDSUBPS      ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMSUBADDPD      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMSUBADDPD      ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMSUBADDPD      xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMSUBADDPD      ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMSUBADDPS      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMSUBADDPS      ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMSUBADDPS      xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMSUBADDPS      ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMSUBPD         xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMSUBPD         ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMSUBPD         xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMSUBPD         ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMSUBPS         xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFMSUBPS         ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFMSUBPS         xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFMSUBPS         ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFMSUBSD         xmmreg,xmmreg*,xmmrm64,xmmreg AMD,SSE5
VFMSUBSD         xmmreg,xmmreg*,xmmreg,xmmrm64 AMD,SSE5
VFMSUBSS         xmmreg,xmmreg*,xmmrm32,xmmreg AMD,SSE5
VFMSUBSS         xmmreg,xmmreg*,xmmreg,xmmrm32 AMD,SSE5
VFNMADDPD        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFNMADDPD        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFNMADDPD        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFNMADDPD        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFNMADDPS        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFNMADDPS        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFNMADDPS        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFNMADDPS        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFNMADDSD        xmmreg,xmmreg*,xmmrm64,xmmreg AMD,SSE5
VFNMADDSD        xmmreg,xmmreg*,xmmreg,xmmrm64 AMD,SSE5
VFNMADDSS        xmmreg,xmmreg*,xmmrm32,xmmreg AMD,SSE5
VFNMADDSS        xmmreg,xmmreg*,xmmreg,xmmrm32 AMD,SSE5
VFNMSUBPD        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFNMSUBPD        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFNMSUBPD        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFNMSUBPD        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFNMSUBPS        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VFNMSUBPS        ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VFNMSUBPS        xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VFNMSUBPS        ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VFNMSUBSD        xmmreg,xmmreg*,xmmrm64,xmmreg AMD,SSE5
VFNMSUBSD        xmmreg,xmmreg*,xmmreg,xmmrm64 AMD,SSE5
VFNMSUBSS        xmmreg,xmmreg*,xmmrm32,xmmreg AMD,SSE5
VFNMSUBSS        xmmreg,xmmreg*,xmmreg,xmmrm32 AMD,SSE5
VFRCZPD          xmmreg,xmmrm128*         AMD,SSE5
VFRCZPD          ymmreg,ymmrm256*         AMD,SSE5
```

```
VFRCZPS         xmmreg,xmmrm128*         AMD,SSE5
VFRCZPS         ymmreg,ymmrm256*         AMD,SSE5
VFRCZSD         xmmreg,xmmrm64*          AMD,SSE5
VFRCZSS         xmmreg,xmmrm32*          AMD,SSE5
VPCMOV          xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPCMOV          ymmreg,ymmreg*,ymmrm256,ymmreg AMD,SSE5
VPCMOV          xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VPCMOV          ymmreg,ymmreg*,ymmreg,ymmrm256 AMD,SSE5
VPCOMB          xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMD          xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMQ          xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMUB         xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMUD         xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMUQ         xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMUW         xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPCOMW          xmmreg,xmmreg*,xmmrm128,imm8 AMD,SSE5
VPHADDBD        xmmreg,xmmrm128*         AMD,SSE5
VPHADDBQ        xmmreg,xmmrm128*         AMD,SSE5
VPHADDBW        xmmreg,xmmrm128*         AMD,SSE5
VPHADDDQ        xmmreg,xmmrm128*         AMD,SSE5
VPHADDUBD       xmmreg,xmmrm128*         AMD,SSE5
VPHADDUBQ       xmmreg,xmmrm128*         AMD,SSE5
VPHADDUBW       xmmreg,xmmrm128*         AMD,SSE5
VPHADDUDQ       xmmreg,xmmrm128*         AMD,SSE5
VPHADDUWD       xmmreg,xmmrm128*         AMD,SSE5
VPHADDUWQ       xmmreg,xmmrm128*         AMD,SSE5
VPHADDWD        xmmreg,xmmrm128*         AMD,SSE5
VPHADDWQ        xmmreg,xmmrm128*         AMD,SSE5
VPHSUBBW        xmmreg,xmmrm128*         AMD,SSE5
VPHSUBDQ        xmmreg,xmmrm128*         AMD,SSE5
VPHSUBWD        xmmreg,xmmrm128*         AMD,SSE5
VPMACSDD        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSDQH       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSDQL       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSSDD       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSSDQH      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSSDQL      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSSWD       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSSWW       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSWD        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMACSWW        xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMADCSSWD      xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPMADCSWD       xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPPERM          xmmreg,xmmreg*,xmmreg,xmmrm128 AMD,SSE5
VPPERM          xmmreg,xmmreg*,xmmrm128,xmmreg AMD,SSE5
VPROTB          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPROTB          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPROTB          xmmreg,xmmrm128*,imm8    AMD,SSE5
VPROTD          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPROTD          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPROTD          xmmreg,xmmrm128*,imm8    AMD,SSE5
VPROTQ          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPROTQ          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
```

```
VPROTQ          xmmreg,xmmrm128*,imm8    AMD,SSE5
VPROTW          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPROTW          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPROTW          xmmreg,xmmrm128*,imm8    AMD,SSE5
VPSHAB          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHAB          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHAD          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHAD          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHAQ          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHAQ          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHAW          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHAW          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHLB          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHLB          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHLD          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHLD          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHLQ          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHLQ          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
VPSHLW          xmmreg,xmmrm128*,xmmreg  AMD,SSE5
VPSHLW          xmmreg,xmmreg*,xmmrm128  AMD,SSE5
```

B.1.33 Systematic names for the hinting nop instructions

```
HINT_NOP0       rm16                     P6,UNDOC
HINT_NOP0       rm32                     P6,UNDOC
HINT_NOP0       rm64                     X64,UNDOC
HINT_NOP1       rm16                     P6,UNDOC
HINT_NOP1       rm32                     P6,UNDOC
HINT_NOP1       rm64                     X64,UNDOC
HINT_NOP2       rm16                     P6,UNDOC
HINT_NOP2       rm32                     P6,UNDOC
HINT_NOP2       rm64                     X64,UNDOC
HINT_NOP3       rm16                     P6,UNDOC
HINT_NOP3       rm32                     P6,UNDOC
HINT_NOP3       rm64                     X64,UNDOC
HINT_NOP4       rm16                     P6,UNDOC
HINT_NOP4       rm32                     P6,UNDOC
HINT_NOP4       rm64                     X64,UNDOC
HINT_NOP5       rm16                     P6,UNDOC
HINT_NOP5       rm32                     P6,UNDOC
HINT_NOP5       rm64                     X64,UNDOC
HINT_NOP6       rm16                     P6,UNDOC
HINT_NOP6       rm32                     P6,UNDOC
HINT_NOP6       rm64                     X64,UNDOC
HINT_NOP7       rm16                     P6,UNDOC
HINT_NOP7       rm32                     P6,UNDOC
HINT_NOP7       rm64                     X64,UNDOC
HINT_NOP8       rm16                     P6,UNDOC
HINT_NOP8       rm32                     P6,UNDOC
HINT_NOP8       rm64                     X64,UNDOC
HINT_NOP9       rm16                     P6,UNDOC
HINT_NOP9       rm32                     P6,UNDOC
HINT_NOP9       rm64                     X64,UNDOC
HINT_NOP10      rm16                     P6,UNDOC
```

```
HINT_NOP10      rm32                    P6,UNDOC
HINT_NOP10      rm64                    X64,UNDOC
HINT_NOP11      rm16                    P6,UNDOC
HINT_NOP11      rm32                    P6,UNDOC
HINT_NOP11      rm64                    X64,UNDOC
HINT_NOP12      rm16                    P6,UNDOC
HINT_NOP12      rm32                    P6,UNDOC
HINT_NOP12      rm64                    X64,UNDOC
HINT_NOP13      rm16                    P6,UNDOC
HINT_NOP13      rm32                    P6,UNDOC
HINT_NOP13      rm64                    X64,UNDOC
HINT_NOP14      rm16                    P6,UNDOC
HINT_NOP14      rm32                    P6,UNDOC
HINT_NOP14      rm64                    X64,UNDOC
HINT_NOP15      rm16                    P6,UNDOC
HINT_NOP15      rm32                    P6,UNDOC
HINT_NOP15      rm64                    X64,UNDOC
HINT_NOP16      rm16                    P6,UNDOC
HINT_NOP16      rm32                    P6,UNDOC
HINT_NOP16      rm64                    X64,UNDOC
HINT_NOP17      rm16                    P6,UNDOC
HINT_NOP17      rm32                    P6,UNDOC
HINT_NOP17      rm64                    X64,UNDOC
HINT_NOP18      rm16                    P6,UNDOC
HINT_NOP18      rm32                    P6,UNDOC
HINT_NOP18      rm64                    X64,UNDOC
HINT_NOP19      rm16                    P6,UNDOC
HINT_NOP19      rm32                    P6,UNDOC
HINT_NOP19      rm64                    X64,UNDOC
HINT_NOP20      rm16                    P6,UNDOC
HINT_NOP20      rm32                    P6,UNDOC
HINT_NOP20      rm64                    X64,UNDOC
HINT_NOP21      rm16                    P6,UNDOC
HINT_NOP21      rm32                    P6,UNDOC
HINT_NOP21      rm64                    X64,UNDOC
HINT_NOP22      rm16                    P6,UNDOC
HINT_NOP22      rm32                    P6,UNDOC
HINT_NOP22      rm64                    X64,UNDOC
HINT_NOP23      rm16                    P6,UNDOC
HINT_NOP23      rm32                    P6,UNDOC
HINT_NOP23      rm64                    X64,UNDOC
HINT_NOP24      rm16                    P6,UNDOC
HINT_NOP24      rm32                    P6,UNDOC
HINT_NOP24      rm64                    X64,UNDOC
HINT_NOP25      rm16                    P6,UNDOC
HINT_NOP25      rm32                    P6,UNDOC
HINT_NOP25      rm64                    X64,UNDOC
HINT_NOP26      rm16                    P6,UNDOC
HINT_NOP26      rm32                    P6,UNDOC
HINT_NOP26      rm64                    X64,UNDOC
HINT_NOP27      rm16                    P6,UNDOC
HINT_NOP27      rm32                    P6,UNDOC
HINT_NOP27      rm64                    X64,UNDOC
```

```
HINT_NOP28      rm16                    P6,UNDOC
HINT_NOP28      rm32                    P6,UNDOC
HINT_NOP28      rm64                    X64,UNDOC
HINT_NOP29      rm16                    P6,UNDOC
HINT_NOP29      rm32                    P6,UNDOC
HINT_NOP29      rm64                    X64,UNDOC
HINT_NOP30      rm16                    P6,UNDOC
HINT_NOP30      rm32                    P6,UNDOC
HINT_NOP30      rm64                    X64,UNDOC
HINT_NOP31      rm16                    P6,UNDOC
HINT_NOP31      rm32                    P6,UNDOC
HINT_NOP31      rm64                    X64,UNDOC
HINT_NOP32      rm16                    P6,UNDOC
HINT_NOP32      rm32                    P6,UNDOC
HINT_NOP32      rm64                    X64,UNDOC
HINT_NOP33      rm16                    P6,UNDOC
HINT_NOP33      rm32                    P6,UNDOC
HINT_NOP33      rm64                    X64,UNDOC
HINT_NOP34      rm16                    P6,UNDOC
HINT_NOP34      rm32                    P6,UNDOC
HINT_NOP34      rm64                    X64,UNDOC
HINT_NOP35      rm16                    P6,UNDOC
HINT_NOP35      rm32                    P6,UNDOC
HINT_NOP35      rm64                    X64,UNDOC
HINT_NOP36      rm16                    P6,UNDOC
HINT_NOP36      rm32                    P6,UNDOC
HINT_NOP36      rm64                    X64,UNDOC
HINT_NOP37      rm16                    P6,UNDOC
HINT_NOP37      rm32                    P6,UNDOC
HINT_NOP37      rm64                    X64,UNDOC
HINT_NOP38      rm16                    P6,UNDOC
HINT_NOP38      rm32                    P6,UNDOC
HINT_NOP38      rm64                    X64,UNDOC
HINT_NOP39      rm16                    P6,UNDOC
HINT_NOP39      rm32                    P6,UNDOC
HINT_NOP39      rm64                    X64,UNDOC
HINT_NOP40      rm16                    P6,UNDOC
HINT_NOP40      rm32                    P6,UNDOC
HINT_NOP40      rm64                    X64,UNDOC
HINT_NOP41      rm16                    P6,UNDOC
HINT_NOP41      rm32                    P6,UNDOC
HINT_NOP41      rm64                    X64,UNDOC
HINT_NOP42      rm16                    P6,UNDOC
HINT_NOP42      rm32                    P6,UNDOC
HINT_NOP42      rm64                    X64,UNDOC
HINT_NOP43      rm16                    P6,UNDOC
HINT_NOP43      rm32                    P6,UNDOC
HINT_NOP43      rm64                    X64,UNDOC
HINT_NOP44      rm16                    P6,UNDOC
HINT_NOP44      rm32                    P6,UNDOC
HINT_NOP44      rm64                    X64,UNDOC
HINT_NOP45      rm16                    P6,UNDOC
HINT_NOP45      rm32                    P6,UNDOC
```

```
HINT_NOP45      rm64                    X64,UNDOC
HINT_NOP46      rm16                    P6,UNDOC
HINT_NOP46      rm32                    P6,UNDOC
HINT_NOP46      rm64                    X64,UNDOC
HINT_NOP47      rm16                    P6,UNDOC
HINT_NOP47      rm32                    P6,UNDOC
HINT_NOP47      rm64                    X64,UNDOC
HINT_NOP48      rm16                    P6,UNDOC
HINT_NOP48      rm32                    P6,UNDOC
HINT_NOP48      rm64                    X64,UNDOC
HINT_NOP49      rm16                    P6,UNDOC
HINT_NOP49      rm32                    P6,UNDOC
HINT_NOP49      rm64                    X64,UNDOC
HINT_NOP50      rm16                    P6,UNDOC
HINT_NOP50      rm32                    P6,UNDOC
HINT_NOP50      rm64                    X64,UNDOC
HINT_NOP51      rm16                    P6,UNDOC
HINT_NOP51      rm32                    P6,UNDOC
HINT_NOP51      rm64                    X64,UNDOC
HINT_NOP52      rm16                    P6,UNDOC
HINT_NOP52      rm32                    P6,UNDOC
HINT_NOP52      rm64                    X64,UNDOC
HINT_NOP53      rm16                    P6,UNDOC
HINT_NOP53      rm32                    P6,UNDOC
HINT_NOP53      rm64                    X64,UNDOC
HINT_NOP54      rm16                    P6,UNDOC
HINT_NOP54      rm32                    P6,UNDOC
HINT_NOP54      rm64                    X64,UNDOC
HINT_NOP55      rm16                    P6,UNDOC
HINT_NOP55      rm32                    P6,UNDOC
HINT_NOP55      rm64                    X64,UNDOC
HINT_NOP56      rm16                    P6,UNDOC
HINT_NOP56      rm32                    P6,UNDOC
HINT_NOP56      rm64                    X64,UNDOC
HINT_NOP57      rm16                    P6,UNDOC
HINT_NOP57      rm32                    P6,UNDOC
HINT_NOP57      rm64                    X64,UNDOC
HINT_NOP58      rm16                    P6,UNDOC
HINT_NOP58      rm32                    P6,UNDOC
HINT_NOP58      rm64                    X64,UNDOC
HINT_NOP59      rm16                    P6,UNDOC
HINT_NOP59      rm32                    P6,UNDOC
HINT_NOP59      rm64                    X64,UNDOC
HINT_NOP60      rm16                    P6,UNDOC
HINT_NOP60      rm32                    P6,UNDOC
HINT_NOP60      rm64                    X64,UNDOC
HINT_NOP61      rm16                    P6,UNDOC
HINT_NOP61      rm32                    P6,UNDOC
HINT_NOP61      rm64                    X64,UNDOC
HINT_NOP62      rm16                    P6,UNDOC
HINT_NOP62      rm32                    P6,UNDOC
HINT_NOP62      rm64                    X64,UNDOC
HINT_NOP63      rm16                    P6,UNDOC
```

```
HINT_NOP63      rm32                    P6,UNDOC
HINT_NOP63      rm64                    X64,UNDOC
```

Appendix C: NASM Version History

C.1 NASM 2 Series

The NASM 2 series support x86−64, and is the production version of NASM since 2007.

C.1.1 Version 2.09

- Updates of new AVX v7 Intel instruciotns.
- PUSH imm32 now officially documented.
- Fix for encoding the LFS, LGS and LSS in 64−bit mode.
- Fixes for compatibility with Watcom compilers and DOS 8.3 file format.
- Macros parameter range expansion introduced.
- Backward compatibility on expanging of local sigle macros restored.
- 8 bit relocations for ELF and BIN targets are introduced.
- Short intersegment jumps are permitted now.
- SECTALIGN directive introduced.
- No−jump mode introduced for smartaligns.
- Short target aliases such as win, elf, macho introduced.
- Various small improvements in documentation.
- No hang anymore if malloc.log fails on creation.
- The environments without vsnprintf function is able to compile nasm again.
- AMD LWP instructions updated.
- Tighten EA checks. We warn a user if there overflow in EA addressing.

C.1.2 Version 2.08

- A number of enhancements/fixes in macros area.
- Support for converting strings to tokens. See section 4.1.9.
- Fuzzy operand size logic introduced.
- Fix COFF stack overrun on too long export identifiers.
- Fix Macho−O alignment bug.
- Fix crashes with –fwin32 on file with many exports.
- Fix stack overrun for too long [DEBUG id].
- Fix incorrect sbyte usage in IMUL (hit only if optimization flag passed).
- Append ending token for `.stabs` records in the ELF output format.
- New NSIS script which uses ModernUI and MultiUser approach.

- Visual Studio 2008 NASM integration (rules file).
- Warn a user if a constant is too long (and as result will be stripped).
- The obsoleted pre−XOP AMD SSE5 instruction set which was never actualized was removed.
- Fix stack overrun on too long error file name passed from the command line.
- Bind symbols to the .text section by default (ie in case if SECTION directive was omitted) in the ELF output format.
- Fix sync points array index wrapping.
- A few fixes for FMA4 and XOP instruction templates.
- Add AMD Lightweight Profiling (LWP) instructions.
- Fix the offset for `%arg` in 64−bit mode.
- An undefined local macro (`%$`) no longer matches a global macro with the same name.
- Fix NULL dereference on too long local labels.

C.1.3 Version 2.07

- NASM is now under the 2−clause BSD license. See section 1.1.2.
- Fix the section type for the `.strtab` section in the `elf64` output format.
- Fix the handling of `COMMON` directives in the `obj` output format.
- New `ith` and `srec` output formats; these are variants of the `bin` output format which output Intel hex and Motorola S−records, respectively. See section 7.2 and section 7.3.
- `rdf2ihx` replaced with an enhanced `rdf2bin`, which can output binary, COM, Intel hex or Motorola S−records.
- The Windows installer now puts the NASM directory first in the `PATH` of the "NASM Shell".
- Revert the early expansion behavior of `%+` to pre−2.06 behavior: `%+` is only expanded late.
- Yet another Mach−O alignment fix.
- Don't delete the list file on errors. Also, include error and warning information in the list file.
- Support for 64−bit Mach−O output, see section 7.8.
- Fix assert failure on certain operations that involve strings with high−bit bytes.

C.1.4 Version 2.06

- This release is dedicated to the memory of Charles A. Crayne, long time NASM developer as well as moderator of `comp.lang.asm.x86` and author of the book *Serious Assembler*. We miss you, Chuck.
- Support for indirect macro expansion (`%[...]`). See section 4.1.3.
- `%pop` can now take an argument, see section 4.7.1.
- The argument to `%use` is no longer macro−expanded. Use `%[...]` if macro expansion is desired.
- Support for thread−local storage in ELF32 and ELF64. See section 7.9.4.
- Fix crash on `%ifmacro` without an argument.
- Correct the arguments to the `POPCNT` instruction.
- Fix section alignment in the Mach−O format.

- Update AVX support to version 5 of the Intel specification.
- Fix the handling of accesses to context−local macros from higher levels in the context stack.
- Treat `WAIT` as a prefix rather than as an instruction, thereby allowing constructs like `O16 FSAVE` to work correctly.
- Support for structures with a non−zero base offset. See section 4.11.10.
- Correctly handle preprocessor token concatenation (see section 4.3.8) involving floating−point numbers.
- The `PINSR` series of instructions have been corrected and rationalized.
- Removed AMD SSE5, replaced with the new XOP/FMA4/CVT16 (rev 3.03) spec.
- The ELF backends no longer automatically generate a `.comment` section.
- Add additional "well−known" ELF sections with default attributes. See section 7.9.2.

C.1.5 Version 2.05.01

- Fix the `−w`/`−W` option parsing, which was broken in NASM 2.05.

C.1.6 Version 2.05

- Fix redundant REX.W prefix on `JMP reg64`.
- Make the behaviour of `−O0` match NASM 0.98 legacy behavior. See section 2.1.22.
- `−w−user` can be used to suppress the output of `%warning` directives. See section 2.1.24.
- Fix bug where `ALIGN` would issue a full alignment datum instead of zero bytes.
- Fix offsets in list files.
- Fix `%include` inside multi−line macros or loops.
- Fix error where NASM would generate a spurious warning on valid optimizations of immediate values.
- Fix arguments to a number of the `CVT` SSE instructions.
- Fix RIP−relative offsets when the instruction carries an immediate.
- Massive overhaul of the ELF64 backend for spec compliance.
- Fix the Geode `PFRCPV` and `PFRSQRTV` instruction.
- Fix the SSE 4.2 `CRC32` instruction.

C.1.7 Version 2.04

- Sanitize macro handing in the `%error` directive.
- New `%warning` directive to issue user−controlled warnings.
- `%error` directives are now deferred to the final assembly phase.
- New `%fatal` directive to immediately terminate assembly.
- New `%strcat` directive to join quoted strings together.
- New `%use` macro directive to support standard macro directives. See section 4.6.4.
- Excess default parameters to `%macro` now issues a warning by default. See section 4.3.
- Fix `%ifn` and `%elifn`.
- Fix nested `%else` clauses.

- Correct the handling of nested `%rep`s.
- New `%unmacro` directive to undeclare a multi−line macro. See section 4.3.11.
- Builtin macro `__PASS__` which expands to the current assembly pass. See section 4.11.9.
- `__utf16__` and `__utf32__` operators to generate UTF−16 and UTF−32 strings. See section 3.4.5.
- Fix bug in case−insensitive matching when compiled on platforms that don't use the `configure` script. Of the official release binaries, that only affected the OS/2 binary.
- Support for x87 packed BCD constants. See section 3.4.7.
- Correct the `LTR` and `SLDT` instructions in 64−bit mode.
- Fix unnecessary REX.W prefix on indirect jumps in 64−bit mode.
- Add AVX versions of the AES instructions (`VAES`...).
- Fix the 256−bit FMA instructions.
- Add 256−bit AVX stores per the latest AVX spec.
- VIA XCRYPT instructions can now be written either with or without `REP`, apparently different versions of the VIA spec wrote them differently.
- Add missing 64−bit `MOVNTI` instruction.
- Fix the operand size of `VMREAD` and `VMWRITE`.
- Numerous bug fixes, especially to the AES, AVX and VTX instructions.
- The optimizer now always runs until it converges. It also runs even when disabled, but doesn't optimize. This allows most forward references to be resolved properly.
- `%push` no longer needs a context identifier; omitting the context identifier results in an anonymous context.

C.1.8 Version 2.03.01

- Fix buffer overflow in the listing module.
- Fix the handling of hexadecimal escape codes in '...' strings.
- The Postscript/PDF documentation has been reformatted.
- The `-F` option now implies `-g`.

C.1.9 Version 2.03

- Add support for Intel AVX, CLMUL and FMA instructions, including YMM registers.
- `dy`, `resy` and `yword` for 32−byte operands.
- Fix some SSE5 instructions.
- Intel `INVEPT`, `INVVPID` and `MOVBE` instructions.
- Fix checking for critical expressions when the optimizer is enabled.
- Support the DWARF debugging format for ELF targets.
- Fix optimizations of signed bytes.
- Fix operation on bigendian machines.
- Fix buffer overflow in the preprocessor.

- SAFESEH support for Win32, IMAGEREL for Win64 (SEH).
- %? and %?? to refer to the name of a macro itself. In particular, %idefine keyword $%? can be used to make a keyword "disappear".
- New options for dependency generation: -MD, -MF, -MP, -MT, -MQ.
- New preprocessor directives %pathsearch and %depend; INCBIN reimplemented as a macro.
- %include now resolves macros in a sane manner.
- %substr can now be used to get other than one-character substrings.
- New type of character/string constants, using backquotes (`...`), which support C-style escape sequences.
- %defstr and %idefstr to stringize macro definitions before creation.
- Fix forward references used in EQU statements.

C.1.10 Version 2.02

- Additional fixes for MMX operands with explicit qword, as well as (hopefully) SSE operands with oword.
- Fix handling of truncated strings with DO.
- Fix segfaults due to memory overwrites when floating-point constants were used.
- Fix segfaults due to missing include files.
- Fix OpenWatcom Makefiles for DOS and OS/2.
- Add autogenerated instruction list back into the documentation.
- ELF: Fix segfault when generating stabs, and no symbols have been defined.
- ELF: Experimental support for DWARF debugging information.
- New compile date and time standard macros.
- %ifnum now returns true for negative numbers.
- New %iftoken test for a single token.
- New %ifempty test for empty expansion.
- Add support for the XSAVE instruction group.
- Makefile for Netware/gcc.
- Fix issue with some warnings getting emitted way too many times.
- Autogenerated instruction list added to the documentation.

C.1.11 Version 2.01

- Fix the handling of MMX registers with explicit qword tags on memory (broken in 2.00 due to 64-bit changes.)
- Fix the PREFETCH instructions.
- Fix the documentation.
- Fix debugging info when using -f elf (backwards compatibility alias for -f elf32).
- Man pages for rdoff tools (from the Debian project.)

- ELF: handle large numbers of sections.
- Fix corrupt output when the optimizer runs out of passes.

C.1.12 Version 2.00

- Added c99 data−type compliance.
- Added general x86−64 support.
- Added win64 (x86−64 COFF) output format.
- Added `__BITS__` standard macro.
- Renamed the `elf` output format to `elf32` for clarity.
- Added `elf64` and `macho` (MacOS X) output formats.
- Added Numeric constants in `dq` directive.
- Added `oword`, `do` and `reso` pseudo operands.
- Allow underscores in numbers.
- Added 8−, 16− and 128−bit floating−point formats.
- Added binary, octal and hexadecimal floating−point.
- Correct the generation of floating−point constants.
- Added floating−point option control.
- Added Infinity and NaN floating point support.
- Added ELF Symbol Visibility support.
- Added setting OSABI value in ELF header directive.
- Added Generate Makefile Dependencies option.
- Added Unlimited Optimization Passes option.
- Added `%IFN` and `%ELIFN` support.
- Added Logical Negation Operator.
- Enhanced Stack Relative Preprocessor Directives.
- Enhanced ELF Debug Formats.
- Enhanced Send Errors to a File option.
- Added SSSE3, SSE4.1, SSE4.2, SSE5 support.
- Added a large number of additional instructions.
- Significant performance improvements.
- `−w+warning` and `−w−warning` can now be written as –Wwarning and –Wno−warning, respectively. See section 2.1.24.
- Add `−w+error` to treat warnings as errors. See section 2.1.24.
- Add `−w+all` and `−w−all` to enable or disable all suppressible warnings. See section 2.1.24.

C.2 NASM 0.98 Series

The 0.98 series was the production versions of NASM from 1999 to 2007.

C.2.1 Version 0.98.39

- fix buffer overflow
- fix outas86's `.bss` handling
- "make spotless" no longer deletes config.h.in.
- `%(el)if(n)idn` insensitivity to string quotes difference (#809300).
- (nasm.c)`__OUTPUT_FORMAT__` changed to string value instead of symbol.

C.2.2 Version 0.98.38

- Add Makefile for 16–bit DOS binaries under OpenWatcom, and modify `mkdep.pl` to be able to generate completely pathless dependencies, as required by OpenWatcom wmake (it supports path searches, but not explicit paths.)
- Fix the `STR` instruction.
- Fix the ELF output format, which was broken under certain circumstances due to the addition of stabs support.
- Quick–fix Borland format debug–info for `-f obj`
- Fix for `%rep` with no arguments (#560568)
- Fix concatenation of preprocessor function call (#794686)
- Fix long label causes coredump (#677841)
- Use autoheader as well as autoconf to keep configure from generating ridiculously long command lines.
- Make sure that all of the formats which support debugging output actually will suppress debugging output when `-g` not specified.

C.2.3 Version 0.98.37

- Paths given in `-I` switch searched for `incbin`–ed as well as `%include`–ed files.
- Added stabs debugging for the ELF output format, patch from Martin Wawro.
- Fix `output/outbin.c` to allow origin > 80000000h.
- Make `-U` switch work.
- Fix the use of relative offsets with explicit prefixes, e.g. `a32 loop foo`.
- Remove `backslash()`.
- Fix the `SMSW` and `SLDT` instructions.
- `-O2` and `-O3` are no longer aliases for `-O10` and `-O15`. If you mean the latter, please say so! :)

C.2.4 Version 0.98.36

- Update rdoff – librarian/archiver – common rec – docs!
- Fix signed/unsigned problems.
- Fix `JMP FAR label` and `CALL FAR label`.
- Add new multisection support – map files – fix align bug
- Fix sysexit, movhps/movlps reg,reg bugs in insns.dat
- `Q` or `O` suffixes indicate octal

- Support Prescott new instructions (PNI).
- Cyrix `XSTORE` instruction.

C.2.5 Version 0.98.35

- Fix build failure on 16−bit DOS (Makefile.bc3 workaround for compiler bug.)
- Fix dependencies and compiler warnings.
- Add "const" in a number of places.
- Add –X option to specify error reporting format (use –Xvc to integrate with Microsoft Visual Studio.)
- Minor changes for code legibility.
- Drop use of tmpnam() in rdoff (security fix.)

C.2.6 Version 0.98.34

- Correct additional address−size vs. operand−size confusions.
- Generate dependencies for all Makefiles automatically.
- Add support for unimplemented (but theoretically available) registers such as tr0 and cr5. Segment registers 6 and 7 are called segr6 and segr7 for the operations which they can be represented.
- Correct some disassembler bugs related to redundant address−size prefixes. Some work still remains in this area.
- Correctly generate an error for things like "SEG eax".
- Add the JMPE instruction, enabled by "CPU IA64".
- Correct compilation on newer gcc/glibc platforms.
- Issue an error on things like "jmp far eax".

C.2.7 Version 0.98.33

- New __NASM_PATCHLEVEL__ and __NASM_VERSION_ID__ standard macros to round out the version−query macros. version.pl now understands X.YYplWW or X.YY.ZZplWW as a version number, equivalent to X.YY.ZZ.WW (or X.YY.0.WW, as appropriate).
- New keyword "strict" to disable the optimization of specific operands.
- Fix the handing of size overrides with JMP instructions (instructions such as "jmp dword foo".)
- Fix the handling of "ABSOLUTE label", where "label" points into a relocatable segment.
- Fix OBJ output format with lots of externs.
- More documentation updates.
- Add –Ov option to get verbose information about optimizations.
- Undo a braindead change which broke `%elif` directives.
- Makefile updates.

C.2.8 Version 0.98.32

- Fix NASM crashing when `%macro` directives were left unterminated.
- Lots of documentation updates.
- Complete rewrite of the PostScript/PDF documentation generator.

- The MS Visual C++ Makefile was updated and corrected.
- Recognize .rodata as a standard section name in ELF.
- Fix some obsolete Perl4−isms in Perl scripts.
- Fix configure.in to work with autoconf 2.5x.
- Fix a couple of "make cleaner" misses.
- Make the normal "./configure && make" work with Cygwin.

C.2.9 Version 0.98.31

- Correctly build in a separate object directory again.
- Derive all references to the version number from the version file.
- New standard macros __NASM_SUBMINOR__ and __NASM_VER__ macros.
- Lots of Makefile updates and bug fixes.
- New `%ifmacro` directive to test for multiline macros.
- Documentation updates.
- Fixes for 16−bit OBJ format output.
- Changed the NASM environment variable to NASMENV.

C.2.10 Version 0.98.30

- Changed doc files a lot: completely removed old README*xx* and Wishlist files, incorporating all information in CHANGES and TODO.
- I waited a long time to rename zoutieee.c to (original) outieee.c
- moved all output modules to output/ subdirectory.
- Added 'make strip' target to strip debug info from nasm & ndisasm.
- Added INSTALL file with installation instructions.
- Added –v option description to nasm man.
- Added dist makefile target to produce source distributions.
- 16−bit support for ELF output format (GNU extension, but useful.)

C.2.11 Version 0.98.28

- Fastcooked this for Debian's Woody release: Frank applied the INCBIN bug patch to 0.98.25alt and called it 0.98.28 to not confuse poor little apt−get.

C.2.12 Version 0.98.26

- Reorganised files even better from 0.98.25alt

C.2.13 Version 0.98.25alt

- Prettified the source tree. Moved files to more reasonable places.
- Added findleak.pl script to misc/ directory.
- Attempted to fix doc.

C.2.14 Version 0.98.25

- Line continuation character \.
- Docs inadvertantly reverted – "dos packaging".

C.2.15 Version 0.98.24p1

- FIXME: Someone, document this please.

C.2.16 Version 0.98.24

- Documentation – Ndisasm doc added to Nasm.doc.

C.2.17 Version 0.98.23

- Attempted to remove rdoff version1
- Lino Mastrodomenico's patches to preproc.c (%$$ bug?).

C.2.18 Version 0.98.22

- Update rdoff2 – attempt to remove v1.

C.2.19 Version 0.98.21

- Optimization fixes.

C.2.20 Version 0.98.20

- Optimization fixes.

C.2.21 Version 0.98.19

- H. J. Lu's patch back out.

C.2.22 Version 0.98.18

- Added ".rdata" to "−f win32".

C.2.23 Version 0.98.17

- H. J. Lu's "bogus elf" patch. (Red Hat problem?)

C.2.24 Version 0.98.16

- Fix whitespace before "[section ..." bug.

C.2.25 Version 0.98.15

- Rdoff changes (?).
- Fix fixes to memory leaks.

C.2.26 Version 0.98.14

- Fix memory leaks.

C.2.27 Version 0.98.13

- There was no 0.98.13

C.2.28 Version 0.98.12

- Update optimization (new function of "−O1")

- Changes to test/bintest.asm (?).

C.2.29 Version 0.98.11

- Optimization changes.
- Ndisasm fixed.

C.2.30 Version 0.98.10

- There was no 0.98.10

C.2.31 Version 0.98.09

- Add multiple sections support to "−f bin".
- Changed GLOBAL_TEMP_BASE in outelf.c from 6 to 15.
- Add "−v" as an alias to the "−r" switch.
- Remove "#ifdef" from Tasm compatibility options.
- Remove redundant size−overrides on "mov ds, ex", etc.
- Fixes to SSE2, other insns.dat (?).
- Enable uppercase "I" and "P" switches.
- Case insinsitive "seg" and "wrt".
- Update install.sh (?).
- Allocate tokens in blocks.
- Improve "invalid effective address" messages.

C.2.32 Version 0.98.08

- Add "`%strlen`" and "`%substr`" macro operators
- Fixed broken c16.mac.
- Unterminated string error reported.
- Fixed bugs as per 0.98bf

C.2.33 Version 0.98.09b with John Coffman patches released 28−Oct−2001

Changes from 0.98.07 release to 98.09b as of 28−Oct−2001

- More closely compatible with 0.98 when –O0 is implied or specified. Not strictly identical, since backward branches in range of short offsets are recognized, and signed byte values with no explicit size specification will be assembled as a single byte.
- More forgiving with the PUSH instruction. 0.98 requires a size to be specified always. 0.98.09b will imply the size from the current BITS setting (16 or 32).
- Changed definition of the optimization flag:

–O0 strict two−pass assembly, JMP and Jcc are handled more like 0.98, except that back− ward JMPs are short, if possible.

–O1 strict two−pass assembly, but forward branches are assembled with code guaranteed to reach; may produce larger code than –O0, but will produce successful assembly more often if branch offset sizes are not specified.

–O2 multi−pass optimization, minimize branch offsets; also will minimize signed immed− iate bytes, overriding size specification.

–O3 like –O2, but more passes taken, if needed

C.2.34 Version 0.98.07 released 01/28/01

- Added Stepane Denis' SSE2 instructions to a *working* version of the code – some earlier versions were based on broken code – sorry 'bout that. version "0.98.07"

01/28/01

- Cosmetic modifications to nasm.c, nasm.h, AUTHORS, MODIFIED

C.2.35 Version 0.98.06f released 01/18/01

- – Add "metalbrain"s jecxz bug fix in insns.dat – alter nasmdoc.src to match – version "0.98.06f"

C.2.36 Version 0.98.06e released 01/09/01

- Removed the "outforms.h" file – it appears to be someone's old backup of "outform.h". version "0.98.06e"

01/09/01

- fbk – finally added the fix for the "multiple %includes bug", known since 7/27/99 – reported originally (?) and sent to us by Austin Lunnen – he reports that John Fine had a fix within the day. Here it is...
- Nelson Rush resigns from the group. Big thanks to Nelson for his leadership and enthusiasm in getting these changes incorporated into Nasm!
- fbk – [list +], [list –] directives – ineptly implemented, should be re−written or removed, perhaps.
- Brian Raiter / fbk – "elfso bug" fix – applied to aoutb format as well – testing might be desirable...

08/07/00

- James Seter – –postfix, –prefix command line switches.
- Yuri Zaporogets – rdoff utility changes.

C.2.37 Version 0.98p1

- GAS−like palign (Panos Minos)
- FIXME: Someone, fill this in with details

C.2.38 Version 0.98bf (bug−fixed)

- Fixed – elf and aoutb bug – shared libraries – multiple "%include" bug in "−f obj" – jcxz, jecxz bug – unrecognized option bug in ndisasm

C.2.39 Version 0.98.03 with John Coffman's changes released 27−Jul−2000

- Added signed byte optimizations for the 0x81/0x83 class of instructions: ADC, ADD, AND, CMP, OR, SBB, SUB, XOR: when used as 'ADD reg16,imm' or 'ADD reg32,imm.' Also optimization of signed byte form of 'PUSH imm' and 'IMUL reg,imm'/'IMUL reg,reg,imm.' No size specification is needed.
- Added multi−pass JMP and Jcc offset optimization. Offsets on forward references will preferentially use the short form, without the need to code a specific size (short or near) for the branch. Added instructions for 'Jcc label' to use the form 'Jnotcc $+3/JMP label', in cases where a short offset is out of bounds. If compiling for a 386 or higher CPU, then the 386 form of Jcc will be used instead.

This feature is controlled by a new command−line switch: "O", (upper case letter O). "−O0" reverts the assembler to no extra optimization passes, "−O1" allows up to 5 extra passes, and "−O2"(default), allows up to 10 extra optimization passes.

- Added a new directive: 'cpu XXX', where XXX is any of: 8086, 186, 286, 386, 486, 586, pentium, 686, PPro, P2, P3 or Katmai. All are case insensitive. All instructions will be selected only if they apply to the selected cpu or lower. Corrected a couple of bugs in cpu−dependence in 'insns.dat'.
- Added to 'standard.mac', the "use16" and "use32" forms of the "bits 16/32" directive. This is nothing new, just conforms to a lot of other assemblers. (minor)
- Changed label allocation from 320/32 (10000 labels @ 200K+) to 32/37 (1000 labels); makes running under DOS much easier. Since additional label space is allocated dynamically, this should have no effect on large programs with lots of labels. The 37 is a prime, believed to be better for hashing. (minor)

C.2.40 Version 0.98.03

"Integrated patchfile 0.98−0.98.01. I call this version 0.98.03 for historical reasons: 0.98.02 was trashed." —John Coffman <johninsd@san.rr.com>, 27−Jul−2000

- Kendall Bennett's SciTech MGL changes
- Note that you must define "TASM_COMPAT" at compile−time to get the Tasm Ideal Mode compatibility.
- All changes can be compiled in and out using the TASM_COMPAT macros, and when compiled without TASM_COMPAT defined we get the exact same binary as the unmodified 0.98 sources.
- standard.mac, macros.c: Added macros to ignore TASM directives before first include
- nasm.h: Added extern declaration for tasm_compatible_mode
- nasm.c: Added global variable tasm_compatible_mode
- Added command line switch for TASM compatible mode (−t)
- Changed version command line to reflect when compiled with TASM additions
- Added response file processing to allow all arguments on a single line (response file is @resp rather than −@resp for NASM format).
- labels.c: Changes islocal() macro to support TASM style @@local labels.
- Added islocalchar() macro to support TASM style @@local labels.
- parser.c: Added support for TASM style memory references (ie: mov [DWORD eax],10 rather than the NASM style mov DWORD [eax],10).
- preproc.c: Added new directives, `%arg`, `%local`, `%stacksize` to directives table
- Added support for TASM style directives without a leading % symbol.
- Integrated a block of changes from Andrew Zabolotny <bit@eltech.ru>:
- A new keyword `%xdefine` and its case−insensitive counterpart `%ixdefine`. They work almost the same way as `%define` and `%idefine` but expand the definition immediately, not on the invocation. Something like a cross between `%define` and `%assign`. The "x" suffix stands for "eXpand", so "xdefine" can be deciphered as "expand−and−define". Thus you can do things like this:

```
%assign ofs     0

%macro  arg     1
        %xdefine %1 dword [esp+ofs]
```

```
        %assign ofs ofs+4
%endmacro
```

- Changed the place where the expansion of %$name macros are expanded. Now they are converted into ..@ctxnum.name form when detokenizing, so there are no quirks as before when using %$name arguments to macros, in macros etc. For example:

```
%macro  abc     1
        %define %1 hello
%endm

abc     %$here
%$here
```

Now last line will be expanded into "hello" as expected. This also allows for lots of goodies, a good example are extended "proc" macros included in this archive.

- Added a check for "cstk" in smacro_defined() before calling get_ctx() – this allows for things like:

```
%ifdef %$abc
%endif
```

to work without warnings even in no context.

- Added a check for "cstk" in %if*ctx and %elif*ctx directives – this allows to use `%ifctx` without excessive warnings. If there is no active context, `%ifctx` goes through "false" branch.
- Removed "user error: " prefix with `%error` directive: it just clobbers the output and has absolutely no functionality. Besides, this allows to write macros that does not differ from built–in functions in any way.
- Added expansion of string that is output by `%error` directive. Now you can do things like:

```
%define hello(x) Hello, x!

%define %$name andy
%error "hello(%$name)"
```

Same happened with `%include` directive.

- Now all directives that expect an identifier will try to expand and concatenate everything without whitespaces in between before usage. For example, with "unfixed" nasm the commands

```
%define %$abc hello
%define __%$abc goodbye
__%$abc
```

would produce "incorrect" output: last line will expand to

```
hello goodbyehello
```

Not quite what you expected, eh? :–) The answer is that preprocessor treats the `%define` construct as if it would be

```
%define __ %$abc goodbye
```

(note the white space between __ and %$abc). After my "fix" it will "correctly" expand into

```
goodbye
```

as expected. Note that I use quotes around words "correct", "incorrect" etc because this is rather a feature not a bug; however current behaviour is more logical (and allows more advanced macro usage :–).

Same change was applied to: %push,%macro,%imacro,%define,%idefine,%xdefine,%ixdefine, %assign,%iassign,%undef

- A new directive [WARNING {+|–}warning–id] have been added. It works only if the assembly phase is enabled (i.e. it doesn't work with nasm –e).
- A new warning type: macro–selfref. By default this warning is disabled; when enabled NASM warns when a macro self–references itself; for example the following source:

```
[WARNING macro-selfref]

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

                push    eax,ebx,ecx
```

will produce a warning, but if we remove the first line we won't see it anymore (which is The Right Thing To Do {tm} IMHO since C preprocessor eats such constructs without warnings at all).

- Added a "error" routine to preprocessor which always will set ERR_PASS1 bit in severity_code. This removes annoying repeated errors on first and second passes from preprocessor.
- Added the %+ operator in single–line macros for concatenating two identifiers. Usage example:

```
%define _myfunc _otherfunc
%define cextern(x) _ %+ x
cextern (myfunc)
```

After first expansion, third line will become "_myfunc". After this expansion is performed again so it becomes "_otherunc".

- Now if preprocessor is in a non–emitting state, no warning or error will be emitted. Example:

```
%if 1
        mov     eax,ebx
%else
        put anything you want between these two brackets,
        even macro-parameter references %1 or local
        labels %$zz or macro-local labels %%zz - no
        warning will be emitted.
%endif
```

- Context–local variables on expansion as a last resort are looked up in outer contexts. For example, the following piece:

```
%push   outer
%define %$a [esp]

        %push   inner
        %$a
        %pop
%pop
```

will expand correctly the fourth line to [esp]; if we'll define another %$a inside the "inner" context, it will take precedence over outer definition. However, this modification has been applied only to expand_smacro

and not to smacro_define: as a consequence expansion looks in outer contexts, but `%ifdef` won't look in outer contexts.

This behaviour is needed because we don't want nested contexts to act on already defined local macros. Example:

```
%define %$arg1  [esp+4]
test    eax,eax
if      nz
        mov     eax,%$arg1
endif
```

In this example the "if" mmacro enters into the "if" context, so %$arg1 is not valid anymore inside "if". Of course it could be worked around by using explicitely %$$arg1 but this is ugly IMHO.

- Fixed memory leak in `%undef`. The origline wasn't freed before exiting on success.
- Fixed trap in preprocessor when line expanded to empty set of tokens. This happens, for example, in the following case:

```
#define SOMETHING
SOMETHING
```

C.2.41 Version 0.98

All changes since NASM 0.98p3 have been produced by H. Peter Anvin <hpa@zytor.com>.

- The documentation comment delimiter is
- Allow EQU definitions to refer to external labels; reported by Pedro Gimeno.
- Re−enable support for RDOFF v1; reported by Pedro Gimeno.
- Updated License file per OK from Simon and Julian.

C.2.42 Version 0.98p9

- Update documentation (although the instruction set reference will have to wait; I don't want to hold up the 0.98 release for it.)
- Verified that the NASM implementation of the PEXTRW and PMOVMSKB instructions is correct. The encoding differs from what the Intel manuals document, but the Pentium III behaviour matches NASM, not the Intel manuals.
- Fix handling of implicit sizes in PSHUFW and PINSRW, reported by Stefan Hoffmeister.
- Resurrect the –s option, which was removed when changing the diagnostic output to stdout.

C.2.43 Version 0.98p8

- Fix for "DB" when NASM is running on a bigendian machine.
- Invoke insns.pl once for each output script, making Makefile.in legal for "make –j".
- Improve the Unix configure−based makefiles to make package creation easier.
- Included an RPM .spec file for building RPM (RedHat Package Manager) packages on Linux or Unix systems.
- Fix Makefile dependency problems.
- Change src/rdsrc.pl to include sectioning information in info output; required for install−info to work.

- Updated the RDOFF distribution to version 2 from Jules; minor massaging to make it compile in my environment.
- Split doc files that can be built by anyone with a Perl interpreter off into a separate archive.
- "Dress rehearsal" release!

C.2.44 Version 0.98p7

- Fixed opcodes with a third byte−sized immediate argument to not complain if given "byte" on the immediate.
- Allow `%undef` to remove single−line macros with arguments. This matches the behaviour of #undef in the C preprocessor.
- Allow –d, –u, –i and –p to be specified as –D, –U, –I and –P for compatibility with most C compilers and preprocessors. This allows Makefile options to be shared between cc and nasm, for example.
- Minor cleanups.
- Went through the list of Katmai instructions and hopefully fixed the (rather few) mistakes in it.
- (Hopefully) fixed a number of disassembler bugs related to ambiguous instructions (disambiguated by –p) and SSE instructions with REP.
- Fix for bug reported by Mark Junger: "call dword 0x12345678" should work and may add an OSP (affected CALL, JMP, Jcc).
- Fix for environments when "stderr" isn't a compile−time constant.

C.2.45 Version 0.98p6

- Took officially over coordination of the 0.98 release; so drop the p3.x notation. Skipped p4 and p5 to avoid confusion with John Fine's J4 and J5 releases.
- Update the documentation; however, it still doesn't include documentation for the various new instructions. I somehow wonder if it makes sense to have an instruction set reference in the assembler manual when Intel et al have PDF versions of their manuals online.
- Recognize "idt" or "centaur" for the –p option to ndisasm.
- Changed error messages back to stderr where they belong, but add an –E option to redirect them elsewhere (the DOS shell cannot redirect stderr.)
- –M option to generate Makefile dependencies (based on code from Alex Verstak.)
- `%undef` preprocessor directive, and –u option, that undefines a single−line macro.
- OS/2 Makefile (Mkfiles/Makefile.os2) for Borland under OS/2; from Chuck Crayne.
- Various minor bugfixes (reported by): – Dangling `%s` in preproc.c (Martin Junker)
- THERE ARE KNOWN BUGS IN SSE AND THE OTHER KATMAI INSTRUCTIONS. I am on a trip and didn't bring the Katmai instruction reference, so I can't work on them right now.
- Updated the License file per agreement with Simon and Jules to include a GPL distribution clause.

C.2.46 Version 0.98p3.7

- (Hopefully) fixed the canned Makefiles to include the outrdf2 and zoutieee modules.
- Renamed changes.asm to changed.asm.

C.2.47 Version 0.98p3.6

- Fixed a bunch of instructions that were added in 0.98p3.5 which had memory operands, and the address−size prefix was missing from the instruction pattern.

C.2.48 Version 0.98p3.5

- Merged in changes from John S. Fine's 0.98−J5 release. John's based 0.98−J5 on my 0.98p3.3 release; this merges the changes.
- Expanded the instructions flag field to a long so we can fit more flags; mark SSE (KNI) and AMD or Katmai−specific instructions as such.
- Fix the "PRIV" flag on a bunch of instructions, and create new "PROT" flag for protected−mode−only instructions (orthogonal to if the instruction is privileged!) and new "SMM" flag for SMM−only instructions.
- Added AMD−only SYSCALL and SYSRET instructions.
- Make SSE actually work, and add new Katmai MMX instructions.
- Added a –p (preferred vendor) option to ndisasm so that it can distinguish e.g. Cyrix opcodes also used in SSE. For example:

```
ndisasm -p cyrix aliased.bin
00000000  670F514310        paddsiw mm0,[ebx+0x10]
00000005  670F514320        paddsiw mm0,[ebx+0x20]
ndisasm -p intel aliased.bin
00000000  670F514310        sqrtps xmm0,[ebx+0x10]
00000005  670F514320        sqrtps xmm0,[ebx+0x20]
```

- Added a bunch of Cyrix−specific instructions.

C.2.49 Version 0.98p3.4

- Made at least an attempt to modify all the additional Makefiles (in the Mkfiles directory). I can't test it, but this was the best I could do.
- DOS DJGPP+"Opus Make" Makefile from John S. Fine.
- changes.asm changes from John S. Fine.

C.2.50 Version 0.98p3.3

- Patch from Conan Brink to allow nesting of `%rep` directives.
- If we're going to allow INT01 as an alias for INT1/ICEBP (one of Jules 0.98p3 changes), then we should allow INT03 as an alias for INT3 as well.
- Updated changes.asm to include the latest changes.
- Tried to clean up the <CR>s that had snuck in from a DOS/Windows environment into my Unix environment, and try to make sure than DOS/Windows users get them back.
- We would silently generate broken tools if insns.dat wasn't sorted properly. Change insns.pl so that the order doesn't matter.
- Fix bug in insns.pl (introduced by me) which would cause conditional instructions to have an extra "cc" in disassembly, e.g. "jnz" disassembled as "jccnz".

C.2.51 Version 0.98p3.2

- Merged in John S. Fine's changes from his 0.98−J4 prerelease; see http://www.csoft.net/cz/johnfine/

- Changed previous "spotless" Makefile target (appropriate for distribution) to "distclean", and added "cleaner" target which is same as "clean" except deletes files generated by Perl scripts; "spotless" is union.
- Removed BASIC programs from distribution. Get a Perl interpreter instead (see below.)
- Calling this "pre−release 3.2" rather than "p3−hpa2" because of John's contributions.
- Actually link in the IEEE output format (zoutieee.c); fix a bunch of compiler warnings in that file. Note I don't know what IEEE output is supposed to look like, so these changes were made "blind".

C.2.52 Version 0.98p3−hpa

- Merged nasm098p3.zip with nasm−0.97.tar.gz to create a fully buildable version for Unix systems (Makefile.in updates, etc.)
- Changed insns.pl to create the instruction tables in nasm.h and names.c, so that a new instruction can be added by adding it *only* to insns.dat.
- Added the following new instructions: SYSENTER, SYSEXIT, FXSAVE, FXRSTOR, UD1, UD2 (the latter two are two opcodes that Intel guarantee will never be used; one of them is documented as UD2 in Intel documentation, the other one just as "Undefined Opcode" — calling it UD1 seemed to make sense.)
- MAX_SYMBOL was defined to be 9, but LOADALL286 and LOADALL386 are 10 characters long. Now MAX_SYMBOL is derived from insns.dat.
- A note on the BASIC programs included: forget them. insns.bas is already out of date. Get yourself a Perl interpreter for your platform of choice at http://www.cpan.org/ports/index.html.

C.2.53 Version 0.98 pre−release 3

- added response file support, improved command line handling, new layout help screen
- fixed limit checking bug, 'OUT byte nn, reg' bug, and a couple of rdoff related bugs, updated Wishlist; 0.98 Prerelease 3.

C.2.54 Version 0.98 pre−release 2

- fixed bug in outcoff.c to do with truncating section names longer than 8 characters, referencing beyond end of string; 0.98 pre−release 2

C.2.55 Version 0.98 pre−release 1

- Fixed a bug whereby STRUC didn't work at all in RDF.
- Fixed a problem with group specification in PUBDEFs in OBJ.
- Improved ease of adding new output formats. Contribution due to Fox Cutter.
- Fixed a bug in relocations in the 'bin' format: was showing up when a relocatable reference crossed an 8192−byte boundary in any output section.
- Fixed a bug in local labels: local−label lookups were inconsistent between passes one and two if an EQU occurred between the definition of a global label and the subsequent use of a local label local to that global.
- Fixed a seg−fault in the preprocessor (again) which happened when you use a blank line as the first line of a multi−line macro definition and then defined a label on the same line as a call to that macro.
- Fixed a stale−pointer bug in the handling of the NASM environment variable. Thanks to Thomas McWilliams.
- ELF had a hard limit on the number of sections which caused segfaults when transgressed. Fixed.
- Added ability for ndisasm to read from stdin by using '−' as the filename.

- ndisasm wasn't outputting the TO keyword. Fixed.
- Fixed error cascade on bogus expression in `%if` – an error in evaluation was causing the entire `%if` to be discarded, thus creating trouble later when the `%else` or `%endif` was encountered.
- Forward reference tracking was instruction–granular not operand– granular, which was causing 286–specific code to be generated needlessly on code of the form 'shr word [forwardref],1'. Thanks to Jim Hague for sending a patch.
- All messages now appear on stdout, as sending them to stderr serves no useful purpose other than to make redirection difficult.
- Fixed the problem with EQUs pointing to an external symbol – this now generates an error message.
- Allowed multiple size prefixes to an operand, of which only the first is taken into account.
- Incorporated John Fine's changes, including fixes of a large number of preprocessor bugs, some small problems in OBJ, and a reworking of label handling to define labels before their line is assembled, rather than after.
- Reformatted a lot of the source code to be more readable. Included 'coding.txt' as a guideline for how to format code for contributors.
- Stopped nested `%reps` causing a panic – they now cause a slightly more friendly error message instead.
- Fixed floating point constant problems (patch by Pedro Gimeno)
- Fixed the return value of insn_size() not being checked for –1, indicating an error.
- Incorporated 3Dnow! instructions.
- Fixed the 'mov eax, eax + ebx' bug.
- Fixed the GLOBAL EQU bug in ELF. Released developers release 3.
- Incorporated John Fine's command line parsing changes
- Incorporated David Lindauer's OMF debug support
- Made changes for LCC 4.0 support (`__NASM_CDecl__`, removed register size specification warning when sizes agree).

C.3 NASM 0.9 Series

Revisions before 0.98.

C.3.1 Version 0.97 released December 1997

- This was entirely a bug–fix release to 0.96, which seems to have got cursed. Silly me.
- Fixed stupid mistake in OBJ which caused 'MOV EAX,<constant>' to fail. Caused by an error in the 'MOV EAX,' support.
- ndisasm hung at EOF when compiled with lcc on Linux because lcc on Linux somehow breaks feof(). ndisasm now does not rely on feof().
- A heading in the documentation was missing due to a markup error in the indexing. Fixed.
- Fixed failure to update all pointers on realloc() within extended– operand code in parser.c. Was causing wrong behaviour and seg faults on lines such as 'dd 0.0,0.0,0.0,0.0,...'
- Fixed a subtle preprocessor bug whereby invoking one multi–line macro on the first line of the expansion of another, when the second had been invoked with a label defined before it, didn't expand the inner macro.
- Added internal.doc back in to the distribution archives – it was missing in 0.96 *blush*

- Fixed bug causing 0.96 to be unable to assemble its own test files, specifically objtest.asm. *blush again*
- Fixed seg−faults and bogus error messages caused by mismatching `%rep` and `%endrep` within multi−line macro definitions.
- Fixed a problem with buffer overrun in OBJ, which was causing corruption at ends of long PUBDEF records.
- Separated DOS archives into main−program and documentation to reduce download size.

C.3.2 Version 0.96 released November 1997

- Fixed a bug whereby, if 'nasm sourcefile' would cause a filename collision warning and put output into 'nasm.out', then 'nasm sourcefile –o outputfile' still gave the warning even though the '−o' was honoured. Fixed name pollution under Digital UNIX: one of its header files defined R_SP, which broke the enum in nasm.h.
- Fixed minor instruction table problems: FUCOM and FUCOMP didn't have two−operand forms; NDISASM didn't recognise the longer register forms of PUSH and POP (eg FF F3 for PUSH BX); TEST mem,imm32 was flagged as undocumented; the 32−bit forms of CMOV had 16−bit operand size prefixes; 'AAD imm' and 'AAM imm' are no longer flagged as undocumented because the Intel Architecture reference documents them.
- Fixed a problem with the local−label mechanism, whereby strange types of symbol (EQUs, auto−defined OBJ segment base symbols) interfered with the 'previous global label' value and screwed up local labels.
- Fixed a bug whereby the stub preprocessor didn't communicate with the listing file generator, so that the –a and –l options in conjunction would produce a useless listing file.
- Merged 'os2' object file format back into 'obj', after discovering that 'obj' _also_ shouldn't have a link pass separator in a module containing a non−trivial MODEND. Flat segments are now declared using the FLAT attribute. 'os2' is no longer a valid object format name: use 'obj'.
- Removed the fixed−size temporary storage in the evaluator. Very very long expressions (like 'mov ax,1+1+1+1+...' for two hundred 1s or so) should now no longer crash NASM.
- Fixed a bug involving segfaults on disassembly of MMX instructions, by changing the meaning of one of the operand−type flags in nasm.h. This may cause other apparently unrelated MMX problems; it needs to be tested thoroughly.
- Fixed some buffer overrun problems with large OBJ output files. Thanks to DJ Delorie for the bug report and fix.
- Made preprocess−only mode actually listen to the `%line` markers as it prints them, so that it can report errors more sanely.
- Re−designed the evaluator to keep more sensible track of expressions involving forward references: can now cope with previously−nightmare situations such as:

```
mov ax,foo | bar
foo equ 1
bar equ 2
```

- Added the ALIGN and ALIGNB standard macros.
- Added PIC support in ELF: use of WRT to obtain the four extra relocation types needed.
- Added the ability for output file formats to define their own extensions to the GLOBAL, COMMON and EXTERN directives.
- Implemented common−variable alignment, and global−symbol type and size declarations, in ELF.

- Implemented NEAR and FAR keywords for common variables, plus far−common element size specification, in OBJ.
- Added a feature whereby EXTERNs and COMMONs in OBJ can be given a default WRT specification (either a segment or a group).
- Transformed the Unix NASM archive into an auto−configuring package.
- Added a sanity−check for people applying SEG to things which are already segment bases: this previously went unnoticed by the SEG processing and caused OBJ−driver panics later.
- Added the ability, in OBJ format, to deal with 'MOV EAX,' type references: OBJ doesn't directly support dword−size segment base fixups, but as long as the low two bytes of the constant term are zero, a word−size fixup can be generated instead and it will work.
- Added the ability to specify sections' alignment requirements in Win32 object files and pure binary files.
- Added preprocess−time expression evaluation: the `%assign` (and `%iassign`) directive and the bare `%if` (and `%elif`) conditional. Added relational operators to the evaluator, for use only in `%if` constructs: the standard relationals = < > <= >= <> (and C−like synonyms == and !=) plus low−precedence logical operators &&, ^^ and ||.
- Added a preprocessor repeat construct: `%rep` / `%exitrep` / `%endrep`.
- Added the __FILE__ and __LINE__ standard macros.
- Added a sanity check for number constants being greater than 0xFFFFFFFF. The warning can be disabled.
- Added the %0 token whereby a variadic multi−line macro can tell how many parameters it's been given in a specific invocation.
- Added `%rotate`, allowing multi−line macro parameters to be cycled.
- Added the '*' option for the maximum parameter count on multi−line macros, allowing them to take arbitrarily many parameters.
- Added the ability for the user−level forms of EXTERN, GLOBAL and COMMON to take more than one argument.
- Added the IMPORT and EXPORT directives in OBJ format, to deal with Windows DLLs.
- Added some more preprocessor `%if` constructs: `%ifidn` / `%ifidni` (exact textual identity), and `%ifid` / `%ifnum` / `%ifstr` (token type testing).
- Added the ability to distinguish SHL AX,1 (the 8086 version) from SHL AX,BYTE 1 (the 286−and−upwards version whose constant happens to be 1).
- Added NetBSD/FreeBSD/OpenBSD's variant of a.out format, complete with PIC shared library features.
- Changed NASM's idiosyncratic handling of FCLEX, FDISI, FENI, FINIT, FSAVE, FSTCW, FSTENV, and FSTSW to bring it into line with the otherwise accepted standard. The previous behaviour, though it was a deliberate feature, was a deliberate feature based on a misunderstanding. Apologies for the inconvenience.
- Improved the flexibility of ABSOLUTE: you can now give it an expression rather than being restricted to a constant, and it can take relocatable arguments as well.
- Added the ability for a variable to be declared as EXTERN multiple times, and the subsequent definitions are just ignored.
- We now allow instruction prefixes (CS, DS, LOCK, REPZ etc) to be alone on a line (without a following instruction).

- Improved sanity checks on whether the arguments to EXTERN, GLOBAL and COMMON are valid identifiers.
- Added misc/exebin.mac to allow direct generation of .EXE files by hacking up an EXE header using DB and DW; also added test/binexe.asm to demonstrate the use of this. Thanks to Yann Guidon for contributing the EXE header code.
- ndisasm forgot to check whether the input file had been successfully opened. Now it does. Doh!
- Added the Cyrix extensions to the MMX instruction set.
- Added a hinting mechanism to allow [EAX+EBX] and [EBX+EAX] to be assembled differently. This is important since [ESI+EBP] and [EBP+ESI] have different default base segment registers.
- Added support for the PharLap OMF extension for 4096−byte segment alignment.

C.3.3 Version 0.95 released July 1997

- Fixed yet another ELF bug. This one manifested if the user relied on the default segment, and attempted to define global symbols without first explicitly declaring the target segment.
- Added makefiles (for NASM and the RDF tools) to build Win32 console apps under Symantec C++. Donated by Mark Junker.
- Added ‘macros.bas’ and ‘insns.bas’, QBasic versions of the Perl scripts that convert ‘standard.mac’ to ‘macros.c’ and convert ‘insns.dat’ to ‘insnsa.c’ and ‘insnsd.c’. Also thanks to Mark Junker.
- Changed the diassembled forms of the conditional instructions so that JB is now emitted as JC, and other similar changes. Suggested list by Ulrich Doewich.
- Added ‘@’ to the list of valid characters to begin an identifier with.
- Documentary changes, notably the addition of the ‘Common Problems’ section in nasm.doc.
- Fixed a bug relating to 32−bit PC−relative fixups in OBJ.
- Fixed a bug in perm_copy() in labels.c which was causing exceptions in cleanup_labels() on some systems.
- Positivity sanity check in TIMES argument changed from a warning to an error following a further complaint.
- Changed the acceptable limits on byte and word operands to allow things like ‘~10111001b’ to work.
- Fixed a major problem in the preprocessor which caused seg−faults if macro definitions contained blank lines or comment−only lines.
- Fixed inadequate error checking on the commas separating the arguments to ‘db’, ‘dw’ etc.
- Fixed a crippling bug in the handling of macros with operand counts defined with a ‘+’ modifier.
- Fixed a bug whereby object file formats which stored the input file name in the output file (such as OBJ and COFF) weren’t doing so correctly when the output file name was specified on the command line.
- Removed [INC] and [INCLUDE] support for good, since they were obsolete anyway.
- Fixed a bug in OBJ which caused all fixups to be output in 16−bit (old−format) FIXUPP records, rather than putting the 32−bit ones in FIXUPP32 (new−format) records.
- Added, tentatively, OS/2 object file support (as a minor variant on OBJ).
- Updates to Fox Cutter’s Borland C makefile, Makefile.bc2.
- Removed a spurious second fclose() on the output file.

- Added the ‘−s’ command line option to redirect all messages which would go to stderr (errors, help text) to stdout instead.
- Added the ‘−w’ command line option to selectively suppress some classes of assembly warning messages.
- Added the ‘−p’ pre−include and ‘−d’ pre−define command−line options.
- Added an include file search path: the ‘−i’ command line option.
- Fixed a silly little preprocessor bug whereby starting a line with a ‘%!’ environment−variable reference caused an ‘unknown directive’ error.
- Added the long−awaited listing file support: the ‘−l’ command line option.
- Fixed a problem with OBJ format whereby, in the absence of any explicit segment definition, non−global symbols declared in the implicit default segment generated spurious EXTDEF records in the output.
- Added the NASM environment variable.
- From this version forward, Win32 console−mode binaries will be included in the DOS distribution in addition to the 16−bit binaries. Added Makefile.vc for this purpose.
- Added ‘return 0;’ to test/objlink.c to prevent compiler warnings.
- Added the __NASM_MAJOR__ and __NASM_MINOR__ standard defines.
- Added an alternative memory−reference syntax in which prefixing an operand with ‘&’ is equivalent to enclosing it in square brackets, at the request of Fox Cutter.
- Errors in pass two now cause the program to return a non−zero error code, which they didn’t before.
- Fixed the single−line macro cycle detection, which didn’t work at all on macros with no parameters (caused an infinite loop). Also changed the behaviour of single−line macro cycle detection to work like cpp, so that macros like ‘extrn’ as given in the documentation can be implemented.
- Fixed the implementation of WRT, which was too restrictive in that you couldn’t do ‘mov ax,[di+abc wrt dgroup]’ because (di+abc) wasn’t a relocatable reference.

C.3.4 Version 0.94 released April 1997

- Major item: added the macro processor.
- Added undocumented instructions SMI, IBTS, XBTS and LOADALL286. Also reorganised CMPXCHG instruction into early−486 and Pentium forms. Thanks to Thobias Jones for the information.
- Fixed two more stupid bugs in ELF, which were causing ‘ld’ to continue to seg−fault in a lot of non−trivial cases.
- Fixed a seg−fault in the label manager.
- Stopped FBLD and FBSTP from _requiring_ the TWORD keyword, which is the only option for BCD loads/stores in any case.
- Ensured FLDCW, FSTCW and FSTSW can cope with the WORD keyword, if anyone bothers to provide it. Previously they complained unless no keyword at all was present.
- Some forms of FDIV/FDIVR and FSUB/FSUBR were still inverted: a vestige of a bug that I thought had been fixed in 0.92. This was fixed, hopefully for good this time...
- Another minor phase error (insofar as a phase error can _ever_ be minor) fixed, this one occurring in code of the form

```
rol ax,forward_reference
forward_reference equ 1
```

- The number supplied to TIMES is now sanity−checked for positivity, and also may be greater than 64K (which previously didn't work on 16−bit systems).
- Added Watcom C makefiles, and misc/pmw.bat, donated by Dominik Behr.
- Added the INCBIN pseudo−opcode.
- Due to the advent of the preprocessor, the [INCLUDE] and [INC] directives have become obsolete. They are still supported in this version, with a warning, but won't be in the next.
- Fixed a bug in OBJ format, which caused incorrect object records to be output when absolute labels were made global.
- Updates to RDOFF subdirectory, and changes to outrdf.c.

C.3.5 Version 0.93 released January 1997

This release went out in a great hurry after semi−crippling bugs were found in 0.92.

- Really *did* fix the stack overflows this time. *blush*
- Had problems with EA instruction sizes changing between passes, when an offset contained a forward reference and so 4 bytes were allocated for the offset in pass one; by pass two the symbol had been defined and happened to be a small absolute value, so only 1 byte got allocated, causing instruction size mismatch between passes and hence incorrect address calculations. Fixed.
- Stupid bug in the revised ELF section generation fixed (associated string−table section for .symtab was hard−coded as 7, even when this didn't fit with the real section table). Was causing 'ld' to seg−fault under Linux.
- Included a new Borland C makefile, Makefile.bc2, donated by Fox Cutter <lmb@comtch.iea.com>.

C.3.6 Version 0.92 released January 1997

- The FDIVP/FDIVRP and FSUBP/FSUBRP pairs had been inverted: this was fixed. This also affected the LCC driver.
- Fixed a bug regarding 32−bit effective addresses of the form `[other_register+ESP]`.
- Documentary changes, notably documentation of the fact that Borland Win32 compilers use 'obj' rather than 'win32' object format.
- Fixed the COMENT record in OBJ files, which was formatted incorrectly.
- Fixed a bug causing segfaults in large RDF files.
- OBJ format now strips initial periods from segment and group definitions, in order to avoid complications with the local label syntax.
- Fixed a bug in disassembling far calls and jumps in NDISASM.
- Added support for user−defined sections in COFF and ELF files.
- Compiled the DOS binaries with a sensible amount of stack, to prevent stack overflows on any arithmetic expression containing parentheses.
- Fixed a bug in handling of files that do not terminate in a newline.

C.3.7 Version 0.91 released November 1996

- Loads of bug fixes.
- Support for RDF added.
- Support for DBG debugging format added.

- Support for 32−bit extensions to Microsoft OBJ format added.
- Revised for Borland C: some variable names changed, makefile added.
- LCC support revised to actually work.
- JMP/CALL NEAR/FAR notation added.
- ‘a16’, ‘o16’, ‘a32’ and ‘o32’ prefixes added.
- Range checking on short jumps implemented.
- MMX instruction support added.
- Negative floating point constant support added.
- Memory handling improved to bypass 64K barrier under DOS.
- `$` prefix to force treatment of reserved words as identifiers added.
- Default−size mechanism for object formats added.
- Compile−time configurability added.
- `#`, `@`, ~ and c{?} are now valid characters in labels.
- `-e` and `-k` options in NDISASM added.

C.3.8 Version 0.90 released October 1996

First release version. First support for object file output. Other changes from previous version (0.3x) too numerous to document.

Index

! operator, unary 34
!= operator 50
$$ token 33, 86
$
  Here token 33
  prefix 26, 29, 89
% operator 33
%! 60
%$ and %$$ prefixes 55
%% operator 33, 43
%+ 39
%? 39
%?? 39
%[39
& operator 33
&& operator 50
* operator 33
+ modifier 44
+ operator
  binary 33
  unary 34
− operator
  binary 33
  unary 34
..@ symbol prefix 36, 44
/ operator 33
// operator 33
< operator 50
<< operator 33
<= operator 50
<> operator 50
= operator 50
== operator 50
> operator 50
>= operator 50
>> operator 33
? MASM syntax 27
^ operator 33
^^ operator 50
| operator 33
|| operator 50
~ operator 34
%0 parameter count 46
%+1 and %-1 syntax 48
16−bit mode, versus 32−bit mode 67
64−bit displacement 114
64−bit immediate 113
-a option 21, 120
A16 26
a16 111
A32 26
a32 111
A64 26
a64 111
a86 14, 24, 25
ABS 29
ABSOLUTE 69, 76
addition 33
addressing, mixed−size 110
address−size prefixes 26
algebra 29
ALIGN 64, 66, 73, 76
  smart 66
ALIGNB 64
alignment
  in bin sections 74
  in elf sections 85
  in obj sections 76
  in win32 sections 79
  of elf common variables 87
ALIGNMODE 66
__ALIGNMODE__ 66
ALINK 91
alink.sourceforge.net 91
all 23
alloc 85
alternate register names 66
alt.lang.asm 14
altreg 66
ambiguity 24
a.out
  BSD version 88
  Linux version 88
aout 88
aoutb 88, 106
%arg 57
arg 98, 105
as86 14, 88
assembler directives 67
assembly−time options 21
%assign 40
ASSUME 25
AT 64
Autoconf 16

autoexec.bat 15
auto-sync 120
-b 119
bin 18, 73
  multisection 74
binary 29
binary files 27
bit shift 33
BITS 67, 73
__BITS__ 61
bitwise AND 33
bitwise OR 33
bitwise XOR 33
block IFs 56
boot loader 73
boot sector 116
Borland
  Pascal 99
  Win32 compilers 75
braces
  after % sign 48
  around macro parameters 42
BSD 106
.bss 85, 88, 89
bugs 117
bugtracker 117
BYTE 116
C calling convention 96, 103
C symbol names 94
c16.mac 98, 101
c32.mac 105
CALL FAR 34
case sensitivity 24, 37, 38, 40, 42, 51, 77
changing sections 68
character constant 27, 31
character strings 30
circular references 37
CLASS 76
%clear 60
coff 85
colon 26
.COM 73, 93
comma 45
command-line 17, 73
commas in macro parameters 44
.comment 85
COMMON 70, 75
  elf extensions to 87
  obj extensions to 78
Common Object File Format 85
common variables 70
  alignment in elf 87
  element size 78
comp.lang.asm.x86 14, 15
comp.os.msdos.programmer 94
concatenating macro parameters 47
concatenating strings 41
condition codes as macro parameters 48
conditional assembly 49
conditional jumps 116
conditional-return macro 48
configure 16
constants 29
context stack 54, 56
context-local labels 55
context-local single-line macros 55
counting macro parameters 46
CPU 71
CPUID 31
creating contexts 54
critical expression 27, 35, 40, 69
-D option 20
-d option 20
daily development snapshots 15
.data 85, 88, 89
_DATA 96
data 87, 89
data structure 98, 105
__DATE__ 62
__DATE_NUM__ 62
DB 27, 31, 32
dbg 90
DD 27, 31, 32
debug information 19
debug information format 19
decimal 29
declaring structures 62
DEFAULT 68
default 87
default macro parameters 45
default name 73
default-WRT mechanism 78
%define 20, 37
defining sections 68
%defstr 41
%deftok 41
%depend 54
design goals 24
DevPac 27, 36
disabling listing expansion 48
division 33
DJGPP 85, 103
djlink 91
DLL symbols
  exporting 77
  importing 77

DO 27, 31, 32
DOS 15, 20
DOS archive
DOS source archive 15
DQ 27, 31, 32
.drectve 79
DT 27, 31, 32
DUP 25, 28
DW 27, 31, 32
DWORD 27
DY 27, 31
-E option 21
-e option 21, 121
effective addresses 26, 28
element size, in common variables 78
ELF 85
    shared libraries 86
    16-bit code and 88
elf, debug formats and 88
elf32 85
elf64 85
%elif 49, 50, 51
%elifctx 50
%elifdef 50
%elifempty 52
%elifid 52
%elifidn 51
%elifidni 51
%elifmacro 50
%elifn 49, 51
%elifnctx 50
%elifndef 50
%elifnempty 52
%elifnid 52
%elifnidn 51
%elifnidni 51
%elifnmacro 50
%elifnnum 52
%elifnstr 52
%elifntoken 52
%elifnum 52
%elifstr 52
%eliftoken 52
%else 49
endproc 98, 105
%endrep 52
ENDSTRUC 62, 69
environment 23
EQU 27, 28
%error 59
error 23
error messages 20
error reporting format 19
escape sequences 30
EVEN 64
exact matches 49
.EXE 75, 91
EXE2BIN 93
EXE_begin 92
exebin.mac 92
exec 85
Executable and Linkable Format 85
EXE_end 92
EXE_stack 92
%exitrep 53
EXPORT 77
export 89
exporting symbols 70
expressions 21, 33
extension 17, 73
EXTERN 70
    obj extensions to 78
    rdf extensions to 89
extracting substrings 42
-F option 19
-f option 18, 73
far call 25
far common variables 78
far pointer 34
FARCODE 99, 101
%fatal 59
__FILE__ 61
FLAT 76
flat memory model 103
flat-form binary 73
FLOAT 71
__FLOAT__ 72
__float128h__ 32
__float128l__ 32
__float16__ 32
__float32__ 32
__float64__ 32
__float8__ 32
__float80e__ 32
__float80m__ 32
__FLOAT_DAZ__ 72
float-denorm 22
floating-point
    constants 32, 71
    packed BCD constants 33
floating-point 25, 26, 27, 32
float-overflow 22
__FLOAT_ROUND__ 72
float-toolong 23
float-underflow 23
follows= 74

format−specific directives 67
frame pointer 96, 100, 103
FreeBSD 88, 106
FreeLink 91
ftp.simtel.net 91
function 87, 89
functions
    C calling convention 96, 103
    Pascal calling convention 100
−g option 19
gas 14
gcc 14
GLOBAL 70
    aoutb extensions to 87
    elf extensions to 87
    rdf extensions to 89
global offset table 106
_GLOBAL_OFFSET_TABLE_ 86
gnu−elf−extensions 22
..got 86
GOT relocations 107
GOT 86, 106
..gotoff 86
GOTOFF relocations 107
..gotpc 86
GOTPC relocations 107
..gottpoff 87
graphics 27
greedy macro parameters 44
GROUP 76
groups 34
−h 119
hexadecimal 29
hidden 87
hybrid syntaxes 24
−I option 20
−i option 20, 120
%iassign 40
%idefine 37
%idefstr 41
%ideftok 41
IEND 64
%if 49, 50
%ifctx 50, 56
%ifdef 49
%ifempty 52
%ifid 51
%ifidn 51
%ifidni 51
%ifmacro 50
%ifn 49, 51
%ifnctx 50
%ifndef 50
%ifnempty 52
%ifnid 52
%ifnidn 51
%ifnidni 51
%ifnmacro 50
%ifnnum 52
%ifnstr 52
%ifntoken 52
%ifnum 51
%ifstr 51
%iftoken 52
%imacro 42
IMPORT 77
import library 77
importing symbols 70
INCBIN 27, 31
%include 20, 53
include search path 20
including other files 53
inefficient code 116
infinite loop 33
__Infinity__ 32
infinity 32
informational section 79
INSTALL 16
installing 15
instances of structures 64
instruction list 122
intel hex 74
Intel number formats 32
internal 87
ISTRUC 64
iterating over macro parameters 46
ith 74
%ixdefine 38
Jcc NEAR 116
JMP DWORD 110
jumps, mixed−size 110
−k 121
−l option 18
label prefix 36
last 45
.lbss 85
ld86 88
.ldata 85
LIBRARY 89
license 14
%line 60
__LINE__ 61
linker, free 91

Linux
  a.out 88
  as86 88
  ELF 85
listing file 18
little−endian 31
%local 58
local labels 35
logical AND 50
logical negation 34
logical OR 50
logical XOR 50
.lrodata 85
−M option 18
Mach, object file format 85
Mach−O 85
macho 85
macho32 85
macho64 85
MacOS X 85
%macro 42
macro indirection 39
macro library 20
macro parameters range 44
macro processor 37
macro−defaults 22
macro−local labels 43
macro−params 22
macros 28
macro−selfref 22
make 16
makefile dependencies 18
makefiles 15, 16
man pages 16
map files 74
MASM 14
MASM 24, 28, 75
−MD option 18
memory models 25, 95
memory operand 27
memory references 24, 28
−MF option 18
−MG option 18
Microsoft OMF 75
minifloat 32
Minix 88
misc subdirectory 92, 98, 105
mixed−language program 94
mixed−size addressing 110
mixed−size instruction 110
MMX registers
ModR/M byte
MODULE 89
modulo operators 33
motorola s−records 74
−MP option 19
−MQ option 19
MS−DOS 73
MS−DOS device drivers 94
−MT option 19
multi−line macros 22, 42
multipass optimization 21
multiple section names 73
multiplication 33
multipush macro 47
multisection 74
__NaN__ 32
NaN 32
NASM version 60
nasm version history 182
nasm version id 60
nasm version string 61
nasm.1 16
__NASMDEFSEG 75
nasm−devel 15
NASMENV 23
nasm.exe 15
nasm −hf 18
__NASM_MAJOR__ 60
__NASM_MINOR__ 60
nasm.out 18
___NASM_PATCHLEVEL__ 60
__NASM_SNAPSHOT__ 60
__NASM_SUBMINOR__ 60
__NASM_VER__ 61
__NASM_VERSION_ID__ 60
nasm−XXX−dos.zip 15
nasm−XXX.tar.gz 16
nasm−XXX−win32.zip 15
nasm−XXX.zip 15
ndisasm 119
ndisasm.1 16
ndisasm.exe 15
near call 25
near common variables 78
NetBSD 88, 106
new releases 15
noalloc 85
nobits 74, 85
noexec 85
.nolist 48
‘nowait’ 25
nowrite 85
number−overflow 22
numeric constants 27, 29
−O option 21

-o option 17, 119
O16 26
o16 111
O32 26
o32 112
O64 26
.OBJ 91
obj 75
object 87, 89
octal 29
OF_DBG 90
OF_DEFAULT 18
OFFSET 24
OMF 75
omitted parameters 45
one's complement 34
OpenBSD 88, 106
operands 26
operand-size prefixes 26
operating system 73
    writing 110
operators 33
ORG 73, 93, 94, 116
orphan-labels 22, 26
OS/2 75, 76
osabi 85
other preprocessor directives 59
out of range, jumps 116
output file format 18
output formats 73
__OUTPUT_FORMAT__ 61
overlapping segments 34
OVERLAY 76
overloading
    multi-line macros 43
    single-line macros 38
-P option 20
-p option 20, 53
paradox 35
PASCAL 101
Pascal calling convention 100
__PASS__ 62
passes, assembly
PATH 15
%pathsearch 20, 53
period 35
Perl 15
perverse 20
PharLap 76
PIC 86, 88, 106
..plt 86
PLT relocations 86, 108, 109
plt relocations 109
%pop 54
position-independent code 86, 88, 106
--postfix 23
precedence 33
pre-defining macros 20, 38
preferred 34
--prefix 23
pre-including files 20
preprocess-only mode 21
preprocessor 21, 28, 34, 37
preprocessor expressions 21
preprocessor loops 52
preprocessor variables 40
primitive directives 67
PRIVATE 75
proc 89, 98, 105
procedure linkage table 86, 108, 109
processor mode 67
progbits 74, 85
program entry point 78, 91
program origin 73
protected 87
pseudo-instructions 27
PUBLIC 70, 75
pure binary 73
%push 54
__QNaN__ 32
quick start 24
QWORD 27
-r 119
rdf 88
rdoff subdirectory 16, 88, 89
redirecting errors 20
REL 29, 68
relational operators 50
release candidates 15
Relocatable Dynamic Object File Format 88
relocations, PIC-specific 86
removing contexts 54
renaming contexts 55
%rep 28, 52
repeating 28, 52
%repl 55
reporting bugs 117
RESB 25, 27
RESD 27
RESO 27
RESQ 27
REST 27
RESW 27
RESY 27
.rodata 85
%rotate 46

rotating macro parameters 46
-s option 20, 120
searching for include files 53
__SECT__ 68, 69
SECTION 68
    elf extensions to 85
    win32 extensions to 79
section alignment
    in bin 74
    in elf 85
    in obj 76
    in win32 79
section, bin extensions to 73
SEG 34, 75
SEGMENT 68
    elf extensions to 75
segment address 34
segment alignment
    in bin 74
    in obj 76
segment names, Borland Pascal 101
segment override 25, 26
segments 34
    groups of 76
separator character 23
shared libraries 88, 106
shared library 87
shift command 46
SIB byte
signed division 33
signed modulo 33
single-line macros 37
size, of symbols 87
smartalign 66
__SNaN__ 32
snapshots, daily development 15
Solaris x86 85
-soname 109
sound 27
source code 15
source-listing file 18
square brackets 24, 28
srec 74
STACK 75
stack relative preprocessor directives 57
%stacksize 58
standard macro packages 66
standard macros 60
standardized section names 68, 79, 85, 88, 89
..start 78, 91
start= 74
stderr 20
stdout 20
%strcat 41
STRICT 34
string constant 27
string constants 31
string length 41
string manipulation in macros 41
strings 30
%strlen 41
STRUC 62, 69, 98, 105
stub preprocessor 21
%substr 42
subtraction 33
suppressible warning 22
suppressing preprocessing 21
switching between sections 68
..sym 86
symbol sizes, specifying 87
symbol types, specifying 87
symbols
    exporting from DLLs 77
    importing from DLLs 77
synchronisation 120
.SYS 73, 94
-t 22
TASM 14, 22
tasm 24, 75
.tbss 85
TBYTE 25
.tdata 85
test subdirectory 91
testing
    arbitrary numeric expressions 50
    context stack 50
    exact text identity 51
    multi-line macro existence 50
    single-line macro existence 49
    token types 51
.text 85, 88, 89
_TEXT 96
thread local storage 87
__TIME__ 62
__TIME_NUM__ 62
TIMES 27, 28, 35, 116, 117
TLINK 93
tls 85, 87
..tlsie 87
trailing colon 26
TWORD 25, 27
type, of symbols 87
-U option 21
-u option 21, 119
unary operators 34
%undef 21, 40

undefining macros 21
underscore, in C symbols 94
Unicode 31
uninitialized 27
uninitialized storage 25
Unix 16
SCO 85
source archive 16
System V 85
UnixWare 85
%unmacro 49
unrolled loops 28
unsigned division 33
unsigned modulo 33
UPPERCASE 24, 77
%use 54, 66
__USE_*__ 62
USE16 68, 76
USE32 68, 76
user 23
user−defined errors 59
user−level assembler directives 60
user−level directives 67
__UTC_DATE__ 62
__UTC_DATE_NUM__ 62
__UTC_TIME__ 62
__UTC_TIME_NUM__ 62
UTF−16 31
UTF−32 31
UTF−8 31
__utf16__ 31
__utf32__ 31
−v option 23
VAL 91
valid characters 26
variable types 24
version 23
version number of NASM 60
vfollows= 74
Visual C++ 79
vstart= 74
−W option 22
−w option 22
%warning 59
warnings 22
[warning *warning−name] 23
[warning +warning−name] 23
[warning −warning−name] 23
website 15
win64 81, 113
Win64 75, 79, 103
Windows 91
Windows 95
Windows NT
write 85
writing operating systems 110
WRT 34, 75, 86, 87, 88
WRT ..got 107
WRT ..gotoff 107
WRT ..gotpc 107
WRT ..plt 109
WRT ..sym 108
WWW page
www.cpan.org 15
www.delorie.com 91
www.pcorner.com 91
−X option 19
x2ftp.oulu.fi 91
%xdefine 38
−y option 23
−Z option 20