

# **NASM – The Netwide Assembler**

---

*version 2.16.03*



© 1996-2024 The NASM Development Team — All Rights Reserved

This document is redistributable under the license given in the section "License".

# Contents

---

Chapter 1: Introduction . . . . .	19
1.1 What Is NASM?. . . . .	19
1.1.1 License . . . . .	19
Chapter 2: Running NASM . . . . .	21
2.1 NASM Command-Line Syntax . . . . .	21
2.1.1 The -o Option: Output File Name . . . . .	21
2.1.2 The -f Option: Output File Format . . . . .	22
2.1.3 The -l Option: Generating a Listing File . . . . .	22
2.1.4 The -L Option: Additional or Modified Listing Info . . . . .	22
2.1.5 The -M Option: Generate Makefile Dependencies . . . . .	23
2.1.6 The -MG Option: Generate Makefile Dependencies . . . . .	23
2.1.7 The -MF Option: Set Makefile Dependency File . . . . .	23
2.1.8 The -MD Option: Assemble and Generate Dependencies . . . . .	23
2.1.9 The -MT Option: Dependency Target Name . . . . .	23
2.1.10 The -MQ Option: Dependency Target Name (Quoted) . . . . .	23
2.1.11 The -MP Option: Emit Phony Makefile Targets . . . . .	23
2.1.12 The -MW Option: Watcom make quoting style. . . . .	23
2.1.13 The -F Option: Debug Information Format . . . . .	24
2.1.14 The -g Option: Enabling Debug Information. . . . .	24
2.1.15 The -x Option: Selecting an Error Reporting Format. . . . .	24
2.1.16 The -z Option: Send Errors to a File . . . . .	24
2.1.17 The -s Option: Send Errors to stdout . . . . .	24
2.1.18 The -i Option: Include File Search Directories. . . . .	25
2.1.19 The -p Option: Pre-Include a File . . . . .	25
2.1.20 The -d Option: Pre-Define a Macro . . . . .	25
2.1.21 The -u Option: Undefine a Macro . . . . .	25
2.1.22 The -E Option: Preprocess Only . . . . .	26
2.1.23 The -a Option: Suppress Preprocessing . . . . .	26
2.1.24 The -o Option: Multipass Optimization . . . . .	26
2.1.25 The -t Option: TASM Compatibility Mode . . . . .	26
2.1.26 The -w and -W Options: Enable or Disable Assembly Warnings. . . . .	27
2.1.27 The -v Option: Display Version Info . . . . .	27
2.1.28 The --(g l)prefix, --(g l)postfix Options. . . . .	27

2.1.29 The --pragma Option . . . . .	27
2.1.30 The --before Option . . . . .	28
2.1.31 The --limit-x Option. . . . .	28
2.1.32 The --keep-all Option . . . . .	28
2.1.33 The --no-line Option. . . . .	28
2.1.34 The --reproducible Option. . . . .	28
2.1.35 The NASMENV Environment Variable . . . . .	28
2.2 Quick Start for MASM Users . . . . .	29
2.2.1 NASM Is Case-Sensitive . . . . .	29
2.2.2 NASM Requires Square Brackets For Memory References. . . . .	29
2.2.3 NASM Doesn't Store Variable Types. . . . .	30
2.2.4 NASM Doesn't ASSUME. . . . .	30
2.2.5 NASM Doesn't Support Memory Models . . . . .	30
2.2.6 Floating-Point Differences . . . . .	30
2.2.7 Other Differences. . . . .	30
2.2.8 MASM compatibility package . . . . .	30
Chapter 3: The NASM Language . . . . .	31
3.1 Layout of a NASM Source Line . . . . .	31
3.2 Pseudo-Instructions. . . . .	32
3.2.1 DX: Declaring Initialized Data . . . . .	32
3.2.2 RESB and Friends: Declaring Uninitialized Data . . . . .	33
3.2.3 INCBIN: Including External Binary Files . . . . .	33
3.2.4 EQU: Defining Constants. . . . .	34
3.2.5 TIMES: Repeating Instructions or Data . . . . .	34
3.3 Effective Addresses . . . . .	34
3.4 Constants . . . . .	35
3.4.1 Numeric Constants . . . . .	36
3.4.2 Character Strings . . . . .	36
3.4.3 Character Constants . . . . .	37
3.4.4 String Constants . . . . .	37
3.4.5 Unicode Strings . . . . .	37
3.4.6 Floating-Point Constants . . . . .	37
3.4.7 Packed BCD Constants . . . . .	39
3.5 Expressions . . . . .	39
3.5.1 ? ... :: Conditional Operator . . . . .	39
3.5.2     : Boolean OR Operator . . . . .	39

3.5.3 : ^: Boolean XOR Operator . . . . .	39
3.5.4 : &&: Boolean AND Operator . . . . .	39
3.5.5 : Comparison Operators . . . . .	39
3.5.6  : Bitwise OR Operator . . . . .	40
3.5.7 ^: Bitwise XOR Operator . . . . .	40
3.5.8 &: Bitwise AND Operator . . . . .	40
3.5.9 Bit Shift Operators . . . . .	40
3.5.10 + and -: Addition and Subtraction Operators . . . . .	40
3.5.11 Multiplication, Division and Modulo . . . . .	40
3.5.12 Unary Operators . . . . .	40
3.6 SEG and WRT . . . . .	41
3.7 STRICT: Inhibiting Optimization . . . . .	41
3.8 Critical Expressions . . . . .	42
3.9 Local Labels . . . . .	42
Chapter 4: The NASM Preprocessor . . . . .	45
4.1 Preprocessor Expansions . . . . .	45
4.1.1 Continuation Line Collapsing . . . . .	45
4.1.2 Comment Removal . . . . .	45
4.1.3 %line directives . . . . .	45
4.1.4 Conditionals, Loops and Multi-Line Macro Definitions . . . . .	45
4.1.5 Directives processing . . . . .	46
4.1.6 Inline expansions and other directives . . . . .	46
4.1.7 Multi-Line Macro Expansion . . . . .	46
4.1.8 Detokenization . . . . .	46
4.2 Single-Line Macros . . . . .	46
4.2.1 The Normal Way: %define . . . . .	46
4.2.2 Resolving %define: %xdefine . . . . .	48
4.2.3 Macro Indirection: %[...] . . . . .	49
4.2.4 Concatenating Single Line Macro Tokens: %+ . . . . .	49
4.2.5 The Macro Name Itself: %? and %?? . . . . .	50
4.2.6 The Single-Line Macro Name: %*? and %*?? . . . . .	50
4.2.7 Undefined Single-Line Macros: %undef . . . . .	50
4.2.8 Preprocessor Variables: %assign . . . . .	51
4.2.9 Defining Strings: %defstr . . . . .	51
4.2.10 Defining Tokens: %deftok . . . . .	51
4.2.11 Defining Aliases: %defalias . . . . .	52

4.2.12 Conditional Comma Operator: % , . . . . .	52
4.3 String Manipulation in Macros. . . . .	52
4.3.1 Concatenating Strings: %strcat . . . . .	52
4.3.2 String Length: %strlen . . . . .	53
4.3.3 Extracting Substrings: %substr . . . . .	53
4.4 Preprocessor Functions . . . . .	53
4.4.1 %abs() Function . . . . .	53
4.4.2 %cond() Function. . . . .	54
4.4.3 %count() Function . . . . .	54
4.4.4 %eval() Function. . . . .	54
4.4.5 %hex() Function . . . . .	54
4.4.6 %is() Family Functions. . . . .	54
4.4.7 %map() Function . . . . .	54
4.4.8 %num() Function . . . . .	55
4.4.9 %sel() Function . . . . .	56
4.4.10 %str() Function. . . . .	56
4.4.11 %strcat() Function . . . . .	56
4.4.12 %strlen() Function . . . . .	56
4.4.13 %substr() Function . . . . .	56
4.4.14 %tok() function . . . . .	56
4.5 Multi-Line Macros: %macro . . . . .	57
4.5.1 Overloading Multi-Line Macros . . . . .	57
4.5.2 Macro-Local Labels . . . . .	58
4.5.3 Greedy Macro Parameters . . . . .	58
4.5.4 Macro Parameters Range . . . . .	59
4.5.5 Default Macro Parameters . . . . .	60
4.5.6 %0: Macro Parameter Counter. . . . .	60
4.5.7 %00: Label Preceding Macro. . . . .	61
4.5.8 %rotate: Rotating Macro Parameters . . . . .	61
4.5.9 Concatenating Macro Parameters . . . . .	62
4.5.10 Condition Codes as Macro Parameters . . . . .	62
4.5.11 Disabling Listing Expansion . . . . .	63
4.5.12 undefining Multi-Line Macros: %unmacro. . . . .	63
4.6 Conditional Assembly . . . . .	63
4.6.1 %ifdef: Testing Single-Line Macro Existence . . . . .	64
4.6.2 %ifmacro: Testing Multi-Line Macro Existence . . . . .	64

4.6.3 %ifctx: Testing the Context Stack . . . . .	64
4.6.4 %if: Testing Arbitrary Numeric Expressions. . . . .	65
4.6.5 %ifidn and %ifidni: Testing Exact Text Identity . . . . .	65
4.6.6 %ifid, %ifnum, %ifstr: Testing Token Types . . . . .	65
4.6.7 %iftoken: Test for a Single Token. . . . .	66
4.6.8 %ifempty: Test for Empty Expansion . . . . .	66
4.6.9 %ifenv: Test If Environment Variable Exists. . . . .	66
4.7 Preprocessor Loops: %rep . . . . .	66
4.8 Source Files and Dependencies . . . . .	67
4.8.1 %include: Including Other Files. . . . .	67
4.8.2 %pathsearch: Search the Include Path . . . . .	68
4.8.3 %depend: Add Dependent Files . . . . .	68
4.8.4 %use: Include Standard Macro Package. . . . .	68
4.9 The Context Stack . . . . .	68
4.9.1 %push and %pop: Creating and Removing Contexts . . . . .	69
4.9.2 Context-Local Labels . . . . .	69
4.9.3 Context-Local Single-Line Macros . . . . .	69
4.9.4 Context Fall-Through Lookup ( <i>deprecated</i> ). . . . .	69
4.9.5 %repl: Renaming a Context. . . . .	70
4.9.6 Example Use of the Context Stack: Block IFs . . . . .	70
4.10 Stack Relative Preprocessor Directives. . . . .	72
4.10.1 %arg Directive . . . . .	72
4.10.2 %stacksize Directive . . . . .	72
4.10.3 %local Directive. . . . .	73
4.11 Reporting User-Defined Errors: %error, %warning, %fatal . . . . .	73
4.12 %pragma: Setting Options. . . . .	74
4.12.1 Preprocessor Pragmas . . . . .	74
4.13 Other Preprocessor Directives . . . . .	75
4.13.1 %line Directive . . . . .	75
4.13.2 %!variable: Read an Environment Variable. . . . .	75
4.13.3 %clear: Clear All Macro Definitions . . . . .	75
Chapter 5: Standard Macros . . . . .	77
5.1 NASM Version Macros . . . . .	77
5.1.1 __?NASM_VERSION_ID?: NASM Version ID . . . . .	77
5.1.2 __?NASM_VER?: NASM Version String . . . . .	77
5.2 __?FILE?__ and __?LINE?__: File Name and Line Number. . . . .	77

5.3	<code>__?BITS?__</code> : Current Code Generation Mode . . . . .	78
5.4	<code>__?OUTPUT_FORMAT?__</code> : Current Output Format . . . . .	78
5.5	<code>__?DEBUG_FORMAT?__</code> : Current Debug Format . . . . .	78
5.6	Assembly Date and Time Macros . . . . .	78
5.7	<code>__?USE_package?__</code> : Package Include Test . . . . .	79
5.8	<code>__?PASS?__</code> : Assembly Pass . . . . .	79
5.9	Structure Data Types . . . . .	79
5.9.1	STRUC and ENDSTRUC: Declaring Structure Data Types . . . . .	79
5.9.2	ISTRUC, AT and IEND: Declaring Instances of Structures . . . . .	80
5.10	Alignment Control . . . . .	81
5.10.1	ALIGN and ALIGNB: Code and Data Alignment . . . . .	81
5.10.2	SECTALIGN: Section Alignment. . . . .	82
Chapter 6:	Standard Macro Packages . . . . .	83
6.1	altreg: Alternate Register Names . . . . .	83
6.2	smartalign: Smart ALIGN Macro . . . . .	83
6.3	fp: Floating-point macros . . . . .	84
6.4	ifunc: Integer functions . . . . .	84
6.4.1	Integer logarithms . . . . .	84
6.5	masm: MASM compatibility . . . . .	84
Chapter 7:	Assembler Directives . . . . .	87
7.1	BITS: Target Processor Mode . . . . .	87
7.1.1	USE16 & USE32: Aliases for BITS . . . . .	88
7.2	DEFAULT: Change the assembler defaults . . . . .	88
7.2.1	REL & ABS: RIP-relative addressing . . . . .	88
7.2.2	BND & NOBND: BND prefix . . . . .	88
7.3	SECTION or SEGMENT: Changing and Defining Sections . . . . .	88
7.3.1	The <code>__?SECT?__</code> Macro . . . . .	88
7.4	ABSOLUTE: Defining Absolute Labels . . . . .	89
7.5	EXTERN: Importing Symbols from Other Modules . . . . .	90
7.6	REQUIRED: Unconditionally Importing Symbols from Other Modules. . . . .	90
7.7	GLOBAL: Exporting Symbols to Other Modules . . . . .	90
7.8	COMMON: Defining Common Data Areas . . . . .	91
7.9	STATIC: Local Symbols within Modules . . . . .	91
7.10	(G L)PREFIX, (G L)POSTFIX: Mangling Symbols . . . . .	91
7.11	CPU: Defining CPU Dependencies . . . . .	92
7.12	FLOAT: Handling of floating-point constants . . . . .	93



7.13 [WARNING]: Enable or disable warnings. . . . .	93
Chapter 8: Output Formats. . . . .	95
8.1 bin: Flat-Form Binary Output . . . . .	95
8.1.1 ORG: Binary File Program Origin. . . . .	95
8.1.2 bin Extensions to the SECTION Directive, bin extensions to} . . . . .	95
8.1.3 Multisection Support for the bin Format . . . . .	96
8.1.4 Map Files . . . . .	96
8.2 ith: Intel Hex Output . . . . .	96
8.3 srec: Motorola S-Records Output . . . . .	96
8.4 obj: Microsoft OMF Object Files . . . . .	97
8.4.1 obj Extensions to the SEGMENT Directive . . . . .	97
8.4.2 GROUP: Defining Groups of Segments . . . . .	98
8.4.3 UPPERCASE: Disabling Case Sensitivity in Output . . . . .	99
8.4.4 IMPORT: Importing DLL Symbols . . . . .	99
8.4.5 EXPORT: Exporting DLL Symbols. . . . .	99
8.4.6 .start: Defining the Program Entry Point . . . . .	100
8.4.7 obj Extensions to the EXTERN Directive . . . . .	100
8.4.8 obj Extensions to the COMMON Directive . . . . .	100
8.4.9 Embedded File Dependency Information. . . . .	101
8.5 win32: Microsoft Win32 Object Files . . . . .	101
8.5.1 win32 Extensions to the SECTION Directive . . . . .	101
8.5.2 win32: Safe Structured Exception Handling. . . . .	102
8.5.3 Debugging formats for Windows . . . . .	103
8.6 win64: Microsoft Win64 Object Files . . . . .	103
8.6.1 win64: Writing Position-Independent Code . . . . .	104
8.6.2 win64: Structured Exception Handling . . . . .	105
8.7 coff: Common Object File Format. . . . .	107
8.8 macho32 and macho64: Mach Object File Format . . . . .	107
8.8.1 macho extensions to the SECTION Directive . . . . .	107
8.8.2 Thread Local Storage in Mach-O: macho special symbols and WRT . . . . .	108
8.8.3 macho specific directive subsections_via_symbols. . . . .	108
8.8.4 macho specific directive no_dead_strip. . . . .	108
8.8.5 macho specific extensions to the GLOBAL Directive: private_extern. . . . .	108
8.9 elf32, elf64, elfx32: Executable and Linkable Format Object Files . . . . .	108
8.9.1 ELF specific directive osabi. . . . .	109
8.9.2 ELF extensions to the SECTION Directive . . . . .	109

8.9.3 Position-Independent Code: ELF Special Symbols and WRT . . . . .	.110
8.9.4 Thread Local Storage in ELF: elf Special Symbols and WRT . . . . .	.110
8.9.5 elf Extensions to the GLOBAL Directive . . . . .	.111
8.9.6 elf Extensions to the EXTERN Directive . . . . .	.111
8.9.7 elf Extensions to the COMMON Directive . . . . .	.111
8.9.8 16-bit code and ELF . . . . .	.112
8.9.9 Debug formats and ELF . . . . .	.112
8.10 aout: Linux a.out Object Files . . . . .	.112
8.11 aoutb: NetBSD/FreeBSD/OpenBSD a.out Object Files . . . . .	.112
8.12 as86: Minix/Linux as86 Object Files. . . . .	.112
8.13 dbg: Debugging Format. . . . .	.112
Chapter 9: Writing 16-bit Code (DOS, Windows 3/3.1) . . . . .	.115
9.1 Producing .EXE Files. . . . .	.115
9.1.1 Using the obj Format To Generate .EXE Files . . . . .	.115
9.1.2 Using the bin Format To Generate .EXE Files . . . . .	.116
9.2 Producing .COM Files. . . . .	.117
9.2.1 Using the bin Format To Generate .COM Files . . . . .	.117
9.2.2 Using the obj Format To Generate .COM Files . . . . .	.117
9.3 Producing .SYS Files. . . . .	.118
9.4 Interfacing to 16-bit C Programs. . . . .	.118
9.4.1 External Symbol Names . . . . .	.118
9.4.2 Memory Models. . . . .	.119
9.4.3 Function Definitions and Function Calls . . . . .	.119
9.4.4 Accessing Data Items . . . . .	.121
9.4.5 c16.mac: Helper Macros for the 16-bit C Interface. . . . .	.122
9.5 Interfacing to Borland Pascal Programs . . . . .	.123
9.5.1 The Pascal Calling Convention . . . . .	.123
9.5.2 Borland Pascal Segment Name Restrictions . . . . .	.124
9.5.3 Using c16.mac With Pascal Programs . . . . .	.124
Chapter 10: Writing 32-bit Code (Unix, Win32, DJGPP) . . . . .	.127
10.1 Interfacing to 32-bit C Programs . . . . .	.127
10.1.1 External Symbol Names . . . . .	.127
10.1.2 Function Definitions and Function Calls. . . . .	.127
10.1.3 Accessing Data Items . . . . .	.128
10.1.4 c32.mac: Helper Macros for the 32-bit C Interface . . . . .	.129
10.2 Writing NetBSD/FreeBSD/OpenBSD and Linux/ELF Shared Libraries . . . . .	.130

10.2.1 Obtaining the Address of the GOT . . . . .	.130
10.2.2 Finding Your Local Data Items. . . . .	.131
10.2.3 Finding External and Common Data Items . . . . .	.131
10.2.4 Exporting Symbols to the Library User . . . . .	.131
10.2.5 Calling Procedures Outside the Library . . . . .	.132
10.2.6 Generating the Library File . . . . .	.132
Chapter 11: Mixing 16- and 32-bit Code . . . . .	.135
11.1 Mixed-Size Jumps . . . . .	.135
11.2 Addressing Between Different-Size Segments . . . . .	.135
11.3 Other Mixed-Size Instructions . . . . .	.136
Chapter 12: Writing 64-bit Code (Unix, Win64) . . . . .	.137
12.1 Register Names in 64-bit Mode . . . . .	.137
12.2 Immediates and Displacements in 64-bit Mode . . . . .	.137
12.3 Interfacing to 64-bit C Programs (Unix). . . . .	.138
12.4 Interfacing to 64-bit C Programs (Win64) . . . . .	.139
Chapter 13: Troubleshooting. . . . .	.141
13.1 Common Problems . . . . .	.141
13.1.1 NASM Generates Inefficient Code . . . . .	.141
13.1.2 My Jumps are Out of Range . . . . .	.141
13.1.3 ORG Doesn't Work . . . . .	.141
13.1.4 TIMES Doesn't Work . . . . .	.142
Appendix A: List of Warning Classes . . . . .	.143
Appendix B: Ndisasm . . . . .	.153
B.1 Introduction. . . . .	.153
B.2 Running NDISASM. . . . .	.153
B.2.1 Specifying the Input Origin . . . . .	.153
B.2.2 Code Following Data: Synchronization . . . . .	.153
B.2.3 Mixed Code and Data: Automatic (Intelligent) Synchronization . . . . .	.154
B.2.4 Other Options . . . . .	.154
Appendix C: NASM Version History . . . . .	.157
C.1 NASM 2 Series . . . . .	.157
C.1.1 Version 2.16.03 . . . . .	.157
C.1.2 Version 2.16.02 . . . . .	.157
C.1.3 Version 2.16.01 . . . . .	.158
C.1.4 Version 2.16 . . . . .	.158
C.1.5 Version 2.15.05 . . . . .	.159

C.1.6 Version 2.15.04 . . . . .	.159
C.1.7 Version 2.15.03 . . . . .	.160
C.1.8 Version 2.15.02 . . . . .	.160
C.1.9 Version 2.15.01 . . . . .	.160
C.1.10 Version 2.15 . . . . .	.161
C.1.11 Version 2.14.03 . . . . .	.162
C.1.12 Version 2.14.02 . . . . .	.162
C.1.13 Version 2.14.01 . . . . .	.162
C.1.14 Version 2.14 . . . . .	.162
C.1.15 Version 2.13.03 . . . . .	.163
C.1.16 Version 2.13.02 . . . . .	.163
C.1.17 Version 2.13.01 . . . . .	.164
C.1.18 Version 2.13 . . . . .	.164
C.1.19 Version 2.12.02 . . . . .	.165
C.1.20 Version 2.12.01 . . . . .	.165
C.1.21 Version 2.12 . . . . .	.166
C.1.22 Version 2.11.09 . . . . .	.166
C.1.23 Version 2.11.08 . . . . .	.166
C.1.24 Version 2.11.07 . . . . .	.166
C.1.25 Version 2.11.06 . . . . .	.167
C.1.26 Version 2.11.05 . . . . .	.167
C.1.27 Version 2.11.04 . . . . .	.167
C.1.28 Version 2.11.03 . . . . .	.167
C.1.29 Version 2.11.02 . . . . .	.167
C.1.30 Version 2.11.01 . . . . .	.167
C.1.31 Version 2.11 . . . . .	.167
C.1.32 Version 2.10.09 . . . . .	.168
C.1.33 Version 2.10.08 . . . . .	.168
C.1.34 Version 2.10.07 . . . . .	.168
C.1.35 Version 2.10.06 . . . . .	.169
C.1.36 Version 2.10.05 . . . . .	.169
C.1.37 Version 2.10.04 . . . . .	.169
C.1.38 Version 2.10.03 . . . . .	.169
C.1.39 Version 2.10.02 . . . . .	.169
C.1.40 Version 2.10.01 . . . . .	.169
C.1.41 Version 2.10 . . . . .	.169

C.1.42 Version 2.09.10 . . . . .	.170
C.1.43 Version 2.09.09 . . . . .	.170
C.1.44 Version 2.09.08 . . . . .	.170
C.1.45 Version 2.09.07 . . . . .	.170
C.1.46 Version 2.09.06 . . . . .	.170
C.1.47 Version 2.09.05 . . . . .	.170
C.1.48 Version 2.09.04 . . . . .	.170
C.1.49 Version 2.09.03 . . . . .	.170
C.1.50 Version 2.09.02 . . . . .	.170
C.1.51 Version 2.09.01 . . . . .	.171
C.1.52 Version 2.09 . . . . .	.171
C.1.53 Version 2.08.02 . . . . .	.172
C.1.54 Version 2.08.01 . . . . .	.172
C.1.55 Version 2.08 . . . . .	.172
C.1.56 Version 2.07 . . . . .	.172
C.1.57 Version 2.06 . . . . .	.173
C.1.58 Version 2.05.01 . . . . .	.173
C.1.59 Version 2.05 . . . . .	.173
C.1.60 Version 2.04 . . . . .	.174
C.1.61 Version 2.03.01 . . . . .	.175
C.1.62 Version 2.03 . . . . .	.175
C.1.63 Version 2.02 . . . . .	.175
C.1.64 Version 2.01 . . . . .	.176
C.1.65 Version 2.00 . . . . .	.176
C.2 NASM 0.98 Series . . . . .	.177
C.2.1 Version 0.98.39 . . . . .	.177
C.2.2 Version 0.98.38 . . . . .	.177
C.2.3 Version 0.98.37 . . . . .	.177
C.2.4 Version 0.98.36 . . . . .	.178
C.2.5 Version 0.98.35 . . . . .	.178
C.2.6 Version 0.98.34 . . . . .	.178
C.2.7 Version 0.98.33 . . . . .	.178
C.2.8 Version 0.98.32 . . . . .	.179
C.2.9 Version 0.98.31 . . . . .	.179
C.2.10 Version 0.98.30 . . . . .	.179
C.2.11 Version 0.98.28 . . . . .	.180

C.2.12 Version 0.98.26 . . . . .	.180
C.2.13 Version 0.98.25alt . . . . .	.180
C.2.14 Version 0.98.25 . . . . .	.180
C.2.15 Version 0.98.24p1 . . . . .	.180
C.2.16 Version 0.98.24 . . . . .	.180
C.2.17 Version 0.98.23 . . . . .	.180
C.2.18 Version 0.98.22 . . . . .	.180
C.2.19 Version 0.98.21 . . . . .	.180
C.2.20 Version 0.98.20 . . . . .	.180
C.2.21 Version 0.98.19 . . . . .	.180
C.2.22 Version 0.98.18 . . . . .	.180
C.2.23 Version 0.98.17 . . . . .	.180
C.2.24 Version 0.98.16 . . . . .	.180
C.2.25 Version 0.98.15 . . . . .	.181
C.2.26 Version 0.98.14 . . . . .	.181
C.2.27 Version 0.98.13 . . . . .	.181
C.2.28 Version 0.98.12 . . . . .	.181
C.2.29 Version 0.98.11 . . . . .	.181
C.2.30 Version 0.98.10 . . . . .	.181
C.2.31 Version 0.98.09 . . . . .	.181
C.2.32 Version 0.98.08 . . . . .	.181
C.2.33 Version 0.98.09b with John Coffman patches released 28-Oct-2001. . . . .	.181
C.2.34 Version 0.98.07 released 01/28/01. . . . .	.182
C.2.35 Version 0.98.06f released 01/18/01 . . . . .	.182
C.2.36 Version 0.98.06e released 01/09/01 . . . . .	.182
C.2.37 Version 0.98p1 . . . . .	.182
C.2.38 Version 0.98bf (bug-fixed) . . . . .	.183
C.2.39 Version 0.98.03 with John Coffman's changes released 27-Jul-2000. . . . .	.183
C.2.40 Version 0.98.03 . . . . .	.183
C.2.41 Version 0.98 . . . . .	.186
C.2.42 Version 0.98p9 . . . . .	.186
C.2.43 Version 0.98p8 . . . . .	.186
C.2.44 Version 0.98p7 . . . . .	.187
C.2.45 Version 0.98p6 . . . . .	.187
C.2.46 Version 0.98p3.7 . . . . .	.187
C.2.47 Version 0.98p3.6 . . . . .	.187

C.2.48 Version 0.98p3.5 . . . . .	.188
C.2.49 Version 0.98p3.4 . . . . .	.188
C.2.50 Version 0.98p3.3 . . . . .	.188
C.2.51 Version 0.98p3.2 . . . . .	.188
C.2.52 Version 0.98p3-hpa . . . . .	.189
C.2.53 Version 0.98 pre-release 3 . . . . .	.189
C.2.54 Version 0.98 pre-release 2 . . . . .	.189
C.2.55 Version 0.98 pre-release 1 . . . . .	.189
C.3 NASM 0.9 Series . . . . .	.190
C.3.1 Version 0.97 released December 1997 . . . . .	.190
C.3.2 Version 0.96 released November 1997 . . . . .	.191
C.3.3 Version 0.95 released July 1997 . . . . .	.193
C.3.4 Version 0.94 released April 1997 . . . . .	.194
C.3.5 Version 0.93 released January 1997 . . . . .	.195
C.3.6 Version 0.92 released January 1997 . . . . .	.195
C.3.7 Version 0.91 released November 1996 . . . . .	.195
C.3.8 Version 0.90 released October 1996 . . . . .	.196
Appendix D: Building NASM from Source. . . . .	.197
D.1 Building from a Source Archive . . . . .	.197
D.2 Building from the git Repository . . . . .	.197
D.3 Building the documentation . . . . .	.197
Appendix E: Contact Information . . . . .	.199
E.1 Website . . . . .	.199
E.1.1 User Forums . . . . .	.199
E.1.2 Development Community . . . . .	.199
E.2 Reporting Bugs . . . . .	.199
Appendix F: Instruction List . . . . .	.201
F.1 Introduction. . . . .	.201
F.1.1 Special instructions (pseudo-ops) . . . . .	.201
F.1.2 Conventional instructions . . . . .	.201
F.1.3 Katmai Streaming SIMD instructions (SSE — a.k.a. KNI, XMM, MMX2). . . . .	.223
F.1.4 Introduced in Deschutes but necessary for SSE support . . . . .	.224
F.1.5 XSAVE group (AVX and extended state) . . . . .	.224
F.1.6 Generic memory operations . . . . .	.224
F.1.7 New MMX instructions introduced in Katmai . . . . .	.224
F.1.8 AMD Enhanced 3DNow! (Athlon) instructions. . . . .	.225

F.1.9 Willamette SSE2 Cacheability Instructions . . . . .	.225
F.1.10 Willamette MMX instructions (SSE2 SIMD Integer Instructions) . . . . .	.225
F.1.11 Willamette Streaming SIMD instructions (SSE2) . . . . .	.226
F.1.12 Prescott New Instructions (SSE3) . . . . .	.228
F.1.13 VMX/SVM Instructions . . . . .	.228
F.1.14 Extended Page Tables VMX instructions . . . . .	.228
F.1.15 SEV-SNP AMD instructions. . . . .	.228
F.1.16 Tejas New Instructions (SSSE3) . . . . .	.228
F.1.17 AMD SSE4A . . . . .	.229
F.1.18 New instructions in Barcelona. . . . .	.229
F.1.19 Penryn New Instructions (SSE4.1) . . . . .	.229
F.1.20 Nehalem New Instructions (SSE4.2) . . . . .	.230
F.1.21 Intel SMX . . . . .	.230
F.1.22 Geode (Cyrix) 3DNow! additions. . . . .	.230
F.1.23 Intel new instructions in ??? . . . . .	.230
F.1.24 Intel AES instructions . . . . .	.230
F.1.25 Intel AVX AES instructions . . . . .	.231
F.1.26 Intel instruction extension based on pub number 319433-030 dated October 2017 . . .	.231
F.1.27 Intel AVX instructions . . . . .	.231
F.1.28 Intel Carry-Less Multiplication instructions (CLMUL) . . . . .	.241
F.1.29 Intel AVX Carry-Less Multiplication instructions (CLMUL) . . . . .	.241
F.1.30 Intel Fused Multiply-Add instructions (FMA) . . . . .	.242
F.1.31 Intel post-32 nm processor instructions. . . . .	.245
F.1.32 Supervisor Mode Access Prevention (SMAP). . . . .	.245
F.1.33 VIA (Centaur) security instructions . . . . .	.245
F.1.34 AMD Lightweight Profiling (LWP) instructions . . . . .	.245
F.1.35 AMD XOP and FMA4 instructions (SSE5) . . . . .	.245
F.1.36 Intel AVX2 instructions . . . . .	.248
F.1.37 Intel Transactional Synchronization Extensions (TSX). . . . .	.250
F.1.38 Intel BMI1 and BMI2 instructions, AMD TBM instructions . . . . .	.251
F.1.39 Intel Memory Protection Extensions (MPX) . . . . .	.251
F.1.40 Intel SHA acceleration instructions . . . . .	.252
F.1.41 SM3 . . . . .	.252
F.1.42 SM4 . . . . .	.252
F.1.43 AVX no exception conversions. . . . .	.252
F.1.44 AVX Vector Neural Network Instructions. . . . .	.252



F.1.45 AVX Integer Fused Multiply-Add . . . . .	.253
F.1.46 AVX-512 mask register instructions . . . . .	.253
F.1.47 AVX-512 mask register instructions (aliases requiring explicit size support) . . . . .	.254
F.1.48 AVX-512 instructions . . . . .	.255
F.1.49 Intel memory protection keys for userspace (PKU aka PKEYs) . . . . .	.286
F.1.50 Read Processor ID . . . . .	.286
F.1.51 New memory instructions . . . . .	.286
F.1.52 Processor trace write . . . . .	.286
F.1.53 Instructions from the Intel Instruction Set Extensions, . . . . .	.287
F.1.54 doc 319433-034 May 2018 . . . . .	.287
F.1.55 Galois field operations (GFNI) . . . . .	.287
F.1.56 AVX512 Vector Bit Manipulation Instructions 2 . . . . .	.287
F.1.57 AVX512 VNNI . . . . .	.288
F.1.58 AVX512 Bit Algorithms . . . . .	.288
F.1.59 AVX512 4-iteration Multiply-Add . . . . .	.288
F.1.60 AVX512 4-iteration Dot Product . . . . .	.289
F.1.61 Intel Software Guard Extensions (SGX) . . . . .	.289
F.1.62 Intel Control-Flow Enforcement Technology (CET) . . . . .	.289
F.1.63 Instructions from ISE doc 319433-040, June 2020 . . . . .	.289
F.1.64 AVX512 Bfloat16 instructions . . . . .	.289
F.1.65 AVX512 mask intersect instructions . . . . .	.289
F.1.66 Intel Advanced Matrix Extensions (AMX) . . . . .	.289
F.1.67 Intel AVX512-FP16 instructions . . . . .	.290
F.1.68 RAO-INT weakly ordered atomic operations . . . . .	.293
F.1.69 User interrupts . . . . .	.294
F.1.70 Compare, exchange and add conditional . . . . .	.294
F.1.71 Flexible Return and Exception Delivery . . . . .	.294
F.1.72 WRMSRNS and MSRLIST instructions . . . . .	.294
F.1.73 History reset . . . . .	.294
F.1.74 Systematic names for the hinting nop instructions . . . . .	.294



# 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, Mach-O, 16-bit and 32-bit .obj (OMF) format, COFF (including its Win32 and Win64 variants.) It can also output plain binary files, Intel hex and Motorola S-Record formats. Its syntax is designed to be simple and easy to understand, similar to the syntax in the Intel Software Developer Manual with minimal complexity. It supports all currently known x86 architectural extensions, and has strong support for macros.

### 1.1.1 License

NASM is under the so-called 2-clause BSD license, also known as the simplified BSD license:

Copyright 1996-2024 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.



## 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
```

The option `--help` is an alias for the `-h` option.

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: 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`, `elfx32`, `ieee`, `macho32` and `macho64`) it will substitute `.o`. For `dbg`, `ith` and `srec`, it will use `.dbg`, `.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 -o driver.sys
```

Note that this is a small o, and is different from a capital O , which is used to specify the number of optimization passes required. See section 2.1.24.

### 2.1.2 The -f Option: Output File Format

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

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

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

### 2.1.3 The -l Option: Generating a Listing File

If you supply the -l option to NASM, followed (with the usual optional space) by a file name, NASM will generate a 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 4.5.11) 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 -L Option: Additional or Modified Listing Info

Use this option to specify listing output details.

Supported options are:

- -Lb show builtin macro packages (standard and %use)
- -Ld show byte and repeat counts in decimal, not hex
- -Le show the preprocessed input
- -Lf ignore .nolist and force listing output
- -Lm show multi-line macro calls with expanded parameters
- -Lp output a list file in every pass, in case of errors
- -Ls show all single-line macro definitions
- -Lw flush the output after every line (very slow, mainly useful to debug NASM crashes)
- -L+ enable *all* listing options except -Lw (very verbose)

These options can be enabled or disabled at runtime using the `%pragma list options` directive:

```
%pragma list options [+|-]flags...
```

For example, to turn on the d and m flags but disable the s flag:

```
%pragma list options +dm -s
```

For forward compatibility reasons, an undefined flag will be ignored. Thus, a new flag introduced in a newer version of NASM can be specified without breaking older versions. Listing flags will always be a single alphanumeric character and are case sensitive.

### 2.1.5 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
```

### 2.1.6 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 nonexistent file is encountered, it is assumed to be a generated file and is added to the dependency list without a prefix.

### 2.1.7 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
```

### 2.1.8 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
```

If the argument after -MD is an option rather than a filename, then the output filename is the first applicable one of:

- the filename set in the -MF option;
- the output filename from the -o option with .d appended;
- the input filename with the extension set to .d.

### 2.1.9 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.

### 2.1.10 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. The default output (if no -MT or -MQ option is specified) is automatically quoted.

### 2.1.11 The -MP Option: Emit Phony Makefile 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.

### 2.1.12 The -MW Option: Watcom make quoting style

This option causes NASM to attempt to quote dependencies according to Watcom make conventions rather than POSIX make conventions (also used by most other make variants.) This quotes # as \$# rather than \#, uses & rather than \ for continuation lines, and encloses filenames containing whitespace in double quotes.

### 2.1.13 The -F Option: 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 2.1.14, 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 -h`. Not all output formats currently support debugging output.

This should not be confused with the -f dbg output format option, see section 8.13.

### 2.1.14 The -g Option: Enabling Debug Information.

This option can be used to generate debugging information in the specified format. See section 2.1.13. 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*.

### 2.1.15 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 8.5.

### 2.1.16 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 2.1.22.

### 2.1.17 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 2.1.16.



### 2.1.18 The -i Option: Include File Search Directories

When NASM sees the `%include` or `%pathsearch` directive in a source file (see section 4.8.1, section 4.8.2 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).

Prior NASM 2.14 a path provided in the option has been considered as a verbatim copy and providing a path separator been up to a caller. One could implicitly concatenate a search path together with a filename. Still this was rather a trick than something useful. Now the trailing path separator is made to always present, thus `-ifoo` will be considered as the `-ifoo/` directory.

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 2.1.35).

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

### 2.1.19 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.

`--include` option is also accepted.

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

### 2.1.20 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 -dF00=100
```

as an alternative to placing the directive

```
%define F00 100
```

at the start of the file. You can miss off the macro value, as well: the option `-dF00` is equivalent to coding `%define F00`. 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`.

### 2.1.21 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 -dF00=100 -uF00
```

would result in `F00` *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`.

### 2.1.22 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 compatibility 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 2.1.16.

### 2.1.23 The -a Option: Suppress Preprocessing

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.

### 2.1.24 The -o Option: Multipass Optimization

Using the -o option, you can tell NASM to carry out different levels of optimization. Multiple flags can be specified after the -o options, some of which can be combined in a single option, e.g. -oxv.

- -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 compatibility 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.
- -ov: At the end of assembly, print the number of passes actually executed.

The -ox mode is recommended for most uses, and is the default since NASM 2.09. *Any other mode will generate worse quality output.* Use -o0 or -o1 only if you need the finer programmer-level control of output and `strict` is not suitable for your use case.

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.

### 2.1.25 The -t Option: 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.

- unprefix forms of some directives supported (arg, elif, else, endif, if, ifdef, ifdifi, ifndef, include, local)

### 2.1.26 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 label-orphan; you can enable warnings of this class by the command-line option -w+label-orphan and disable it by -w-label-orphan.

Since version 2.15, NASM has group aliases for all prefixed warnings, so they can be used to enable or disable all warnings in the group. For example, -w+float enables all warnings with names starting with float-.\*.

Since version 2.00, NASM has also supported the gcc-like syntax -wwarning-class and -wno-warning-class instead of -w+warning-class and -w-warning-class, respectively; both syntaxes work identically.

The option -w+error or -werror can be used to treat warnings as errors. This can be controlled on a per warning class basis (-w+error=warning-class or -werror=warning-class); if no warning-class is specified NASM treats it as -w+error=all; the same applies to -w-error or -wno-error, of course.

In addition, you can control warnings in the source code itself, using the [WARNING] directive. See section 7.13.

See appendix A for the complete list of warning classes.

### 2.1.27 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.

For command-line compatibility with Yasm, the form --v is also accepted for this option starting in NASM version 2.11.05.

### 2.1.28 The --(g|l)prefix, --(g|l)postfix Options.

The --(g)prefix options prepend the given argument to all extern, common, static, and global symbols, and the --lprefix option prepends to all other symbols. Similarly, --(g)postfix and --lpostfix options append the argument in the exactly same way as the --xxprefix options does.

Running this:

```
nasm -f macho --gprefix _
```

is equivalent to place the directive with %pragma macho gprefix \_ at the start of the file (section 7.10). It will prepend the underscore to all global and external variables, as C requires it in some, but not all, system calling conventions.

### 2.1.29 The --pragma Option

NASM accepts an argument as %pragma option, which is like placing a %pragma preprocess statement at the beginning of the source. Running this:

```
nasm -f macho --pragma "macho gprefix _"
```

is equivalent to the example in section 2.1.28. See section 4.12.

### 2.1.30 The `--before` Option

A preprocess statement can be accepted with this option. The example shown in section 2.1.29 is the same as running this:

```
nasm -f macho --before "%pragma macho gprefix _"
```

### 2.1.31 The `--limit-x` Option

This option allows user to setup various maximum values after which NASM will terminate with a fatal error rather than consume arbitrary amount of compute time. Each limit can be set to a positive number or unlimited.

- `--limit-passes`: Number of maximum allowed passes. Default is unlimited.
- `--limit-stalled-passes`: Maximum number of allowed unfinished passes. Default is 1000.
- `--limit-macro-levels`: Define maximum depth of macro expansion (in preprocess). Default is 10000
- `--limit-macro-tokens`: Maximum number of tokens processed during single-line macro expansion. Default is 10000000.
- `--limit-mmacros`: Maximum number of multi-line macros processed before returning to the top-level input. Default is 100000.
- `--limit-rep`: Maximum number of allowed preprocessor loop, defined under `%rep`. Default is 1000000.
- `--limit-eval`: This number sets the boundary condition of allowed expression length. Default is 8192 on most systems.
- `--limit-lines`: Total number of source lines allowed to be processed. Default is 2000000000.

For example, set the maximum line count to 1000:

```
nasm --limit-lines 1000
```

Limits can also be set via the directive `%pragma limit`, for example:

```
%pragma limit lines 1000
```

### 2.1.32 The `--keep-all` Option

This option prevents NASM from deleting any output files even if an error happens.

### 2.1.33 The `--no-line` Option

If this option is given, all `%line` directives in the source code are ignored. This can be useful for debugging already preprocessed code. See section 4.13.1.

### 2.1.34 The `--reproducible` Option

If this option is given, NASM will not emit information that is inherently dependent on the NASM version or different from run to run (such as timestamps) into the output file.

### 2.1.35 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 `F00`. If you're assembling to DOS or OS/2 `.OBJ` files, you can invoke the `UPPERCASE` directive (documented in section 8.4) 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 `%define 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 `x87` 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`.

Historically, 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.

As of NASM 2.15, the MASM syntax is also supported.

In addition to all of this, macros and directives work completely differently to MASM. See chapter 4 and chapter 7 for further details.

### 2.2.8 MASM compatibility package

See section 6.5.

## 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 7) 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 `lods b` 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`, `REPNE/REPZ`, `XACQUIRE/XRELEASE` or `BND/NOBND`, 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 11. 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 `LDSB`, which has no operands and yet can require a segment override, there is no clean syntactic way to proceed apart from `es lods b`.

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`, `DY` and `DZ`; their uninitialized counterparts `RESB`, `RESW`, `RESQ`, `RESB`, `RESW`, `RESQ`, `RESB`, `RESW` and `RESQ`; the `INCBIN` command, the `EQU` command, and the `TIMES` prefix.

In this documentation, the notation "`dx`" and "`RESx`" is used to indicate all the `DB` and `RESB` type directives, respectively.

### 3.2.1 `dx`: Declaring Initialized Data

`DB`, `DW`, `DD`, `DQ`, `DT`, `DO`, `DY` and `DZ` (collectively "`dx`" in this documentation) 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`, `DY` and `DZ` do not accept integer numeric constants as operands.

Starting in NASM 2.15, a the following MASM-like features have been implemented:

- A `?` argument to declare uninitialized storage:
 

```
db      ?              ; uninitialized
```
- A superset of the `DUP` syntax. The NASM version of this has the following syntax specification; capital letters indicate literal keywords:

```

dx      := DB | DW | DD | DQ | DT | DO | DY | DZ
type    := BYTE | WORD | DWORD | QWORD | TWORD | OWORD | YWORD | ZWORD
atom    := expression | string | float | '?'
parlist := '(' value [',' value ...] ')'
duplist := expression DUP [type] ['%'] parlist
list    := duplist | '%' parlist | type ['%'] parlist
value   := [type] atom | list

stmt    := dx value [',' value ...]

```

Note that a *list* needs to be prefixed with a `%` sign unless prefixed by either `DUP` or a *type* in order to avoid confusing it with a parenthesis starting an expression. The following expressions are all valid:

```

db 33
db (44)          ; Integer expression
; db (44,55)      ; Invalid - error
db %(44,55)
db %('XX','YY')
db ('AA')        ; Integer expression - outputs single byte
db %('BB')       ; List, containing a string
db ?

```



```

db 6 dup (33)
db 6 dup (33, 34)
db 6 dup (33, 34), 35
db 7 dup (99)
db 7 dup dword (?, word ?, ?)
dw byte (?, 44)
dw 3 dup (0xcc, 4 dup byte ('PQR'), ?), 0xabcd
dd 16 dup (0xaaaa, ?, 0xbbbbbbb)
dd 64 dup (?)

```

The use of \$ (current address) in a dx statement is undefined in the current version of NASM, *except in the following cases*:

- For the first expression in the statement, either a DUP or a data item.
- An expression of the form "value - \$", which is converted to a self-relative relocation.

Future versions of NASM is likely to produce a different result or issue an error this case.

There is no such restriction on using \$\$ or section-relative symbols.

### 3.2.2 RESB and Friends: Declaring Uninitialized Data

RESB, RESW, RESD, RESQ, REST, RESO, RESY and RESZ 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. The operand to a RESB-type pseudo-instruction *would* be a *critical expression* (see section 3.8), except that for legacy compatibility reasons forward references are permitted, however *the code will be extremely fragile and this should be considered a severe programming error*. A warning will be issued; code generating this warning should be remedied as quickly as possible (see the forward class in appendix A.)

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
zmmvals:     resz    32          ; 32 ZMM registers

```

Since NASM 2.15, the MASM syntax of using ? and DUP in the dx directives is also supported. Thus, the above example could also be written:

```

buffer:      db      64 dup (?)   ; reserve 64 bytes
wordvar:     dw      ?           ; reserve a word
realarray    dq      10 dup (?)  ; array of ten reals
ymmval:      dy      ?           ; one YMM register
zmmvals:     dz      32 dup (?)  ; 32 ZMM registers

```

### 3.2.3 INCBIN: Including External Binary Files

INCBIN includes binary file data 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 11.2). 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. `[nosplit eax*1]` also has the same effect. In another way, a split EA form `[0, eax*2]` can be used, too. However, `NOSPLIT` in `[nosplit eax+eax]` will be ignored because user's intention here is considered as `[eax+eax]`.

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 7.2). The keyword `ABS` overrides `REL`.

A new form of split effective address syntax is also supported. This is mainly intended for mib operands as used by MPX instructions, but can be used for any memory reference. The basic concept of this form is splitting base and index.

```
mov eax,[ebx+8,ecx*4] ; ebx=base, ecx=index, 4=scale, 8=disp
```

For mib operands, there are several ways of writing effective address depending on the tools. NASM supports all currently possible ways of mib syntax:

```
; bndstx
; next 5 lines are parsed same
; base=rax, index=rbx, scale=1, displacement=3
bndstx [rax+0x3,rbx], bnd0 ; NASM - split EA
bndstx [rbx*1+rax+0x3], bnd0 ; GAS - '*1' indicates an index reg
bndstx [rax+rbx+3], bnd0 ; GAS - without hints
bndstx [rax+0x3], bnd0, rbx ; ICC-1
bndstx [rax+0x3], rbx, bnd0 ; ICC-2
```

When broadcasting decorator is used, the `opsiz` keyword should match the size of each element.

```
VDIVPS zmm4, zmm5, dword [rbx]{1to16} ; single-precision float
VDIVPS zmm4, zmm5, zword [rbx] ; packed 512 bit memory
```

## 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        ; octal 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 `dcb a`. 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?__`, `__?utf16le?__`, `__?utf16be?__`, `__?utf32?__`, `__?utf32le?__` and `__?utf32be?__` allows definition of Unicode strings. They take a string in UTF-8 format and converts it to UTF-16 or UTF-32, respectively. Unless the `be` forms are specified, the output is littleendian.

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
```

The UTF operators 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?__`, `__?bfloat16?__`, `__?float32?__`,

`__?float64?__`, `__?float80m?__`, `__?float80e?__`, `__?float128l?__`, and `__?float128h?__`. See also section 6.3.

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 hexadecimal 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
dq    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    3.141592653589793238462 ; 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 `bfloat16` format is effectively a compressed version of the 32-bit single precision format, with a reduced mantissa. It is effectively the same as truncating the 32-bit format to the upper 16 bits, except for rounding. There is no `dx` directive that corresponds to `bfloat16` as it obviously has the same size as the IEEE standard 16-bit half precision format, see however section 6.3.

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?__(3.141592653589793238462)
```

... 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
```

The `%use fp` standard macro package contains a set of convenience macros. See section 6.3.

### 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.

A *boolean* value is true if nonzero and false if zero. The operators which return a boolean value always return 1 for true and 0 for false.

### 3.5.1 ? ... :: Conditional Operator

The syntax of this operator, similar to the C conditional operator, is:

*boolean ? trueval : falseval*

This operator evaluates to *trueval* if *boolean* is true, otherwise to *falseval*.

Note that NASM allows `?` characters in symbol names. Therefore, it is highly advisable to always put spaces around the `?` and `:` characters.

### 3.5.2 : ||: Boolean OR Operator

The `||` operator gives a boolean OR: it evaluates to 1 if both sides of the expression are nonzero, otherwise 0.

### 3.5.3 : ^^: Boolean XOR Operator

The `^^` operator gives a boolean XOR: it evaluates to 1 if any one side of the expression is nonzero, otherwise 0.

### 3.5.4 : &&: Boolean AND Operator

The `&&` operator gives a boolean AND: it evaluates to 1 if both sides of the expression is nonzero, otherwise 0.

### 3.5.5 : Comparison Operators

NASM supports the following comparison operators:

- = or == compare for equality.
- != or <> compare for inequality.
- < compares signed less than.
- <= compares signed less than or equal.
- > compares signed greater than.
- >= compares signed greater than or equal.

These operators evaluate to 0 for false or 1 for true.

- <=> does a signed comparison, and evaluates to -1 for less than, 0 for equal, and 1 for greater than.

At this time, NASM does not provide unsigned comparison operators.

### 3.5.6 | : Bitwise OR Operator

The | operator gives a bitwise OR, exactly as performed by the or machine instruction.

### 3.5.7 ^: Bitwise XOR Operator

^ provides the bitwise XOR operation.

### 3.5.8 &: Bitwise AND Operator

& provides the bitwise AND operation.

### 3.5.9 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 an *unsigned* (logical) bit-shift to the right; the bits shifted in from the left are set to zero.

<<< gives a bit-shift to the left, exactly equivalent to the << operator; it is included for completeness. >>> gives an *signed* (arithmetic) bit-shift to the right; the bits shifted in from the left are filled with copies of the most significant (sign) bit.

### 3.5.10 + and -: Addition and Subtraction Operators

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

### 3.5.11 Multiplication, Division and Modulo

\* 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.

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.

NASM, like ANSI C, provides no guarantees about the sensible operation of the signed modulo operator. On most systems it will match the signed division operator, such that:

$$b * (a // b) + (a \% b) = a \quad (b \neq 0)$$

### 3.5.12 Unary Operators

The highest-priority operators in NASM's expression grammar are those which only apply to one argument. These are:

- - negates (2's complement) its operand.
- + does nothing; it's provided for symmetry with -.



- `~` computes the bitwise negation (1's complement) of its operand.
- `!` is the boolean negation operator. It evaluates to 1 if the argument is 0, otherwise 0.
- `SEG` provides the segment address of its operand (explained in more detail in section 3.6).
- A set of additional operators with leading and trailing double underscores are used to implement the integer functions of the `ifunc` macro package, see section 6.4.

### 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 evaluates to 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 2.1.24), NASM will use size specifiers (`BYTE`, `WORD`, `DWORD`, `QWORD`, `TWORD`, `OWORD`, `YWORD` or `ZWORD`), 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:

```
label11 ; some code

.loop
    ; some more code

    jne     .loop
    ret

label12 ; 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 `label11.loop`, and the second defines a symbol called `label12.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 8.4.6), `..imagebase` is used to find out the offset from a base address of the current image in the `win64` output format (see section 8.6.1). 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. As a result, some care needs to be taken when using the % arithmetic operator to avoid it being confused with a preprocessor directive; it is recommended that it always be surrounded by whitespace.

The NASM preprocessor borrows concepts from both the C preprocessor and the macro facilities of many other assemblers.

### 4.1 Preprocessor Expansions

The input to the preprocessor is expanded in the following ways in the order specified here.

#### 4.1.1 Continuation Line Collapsing

The preprocessor first 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.

#### 4.1.2 Comment Removal

After concatenation, comments are removed. Comments begin with the character ; unless contained inside a quoted string or a handful of other special contexts.

Note that this is applied *after* continuation lines are collapsed. This means that

```
    add al, '\ '    ; Add the ASCII code for \
    mov [ecx], al   ; Save the character
```

will probably not do what you expect, as the second line will be considered part of the preceding comment. Although this behavior is sometimes confusing, it is both the behavior of NASM since the very first version as well as the behavior of the C preprocessor.

#### 4.1.3 %line directives

In this step, %line directives are processed. See section 4.13.1.

#### 4.1.4 Conditionals, Loops and Multi-Line Macro Definitions

In this step, the following preprocessor directives are processed:

- Multi-line macro definitions, specified by the %macro and %imacro directives. The body of a multi-line macro is stored and is not further expanded at this time. See section 4.5.
- Conditional assembly, specified by the %if family of preprocessor directives. Disabled part of the source code are discarded and are not further expanded. See section 4.6.
- Preprocessor loops, specified by the %rep preprocessor directive. A preprocessor loop is very similar to a multi-line macro and as such the body is stored and is not further expanded at this time. See section 4.7.

These constructs are required to be balanced, so that the ending of a block can be detected, but no further processing is done at this time; stored blocks will be inserted at this step when they are expanded (see below.)

It is specific to each directive to what extent inline expansions and detokenization are performed for the arguments of the directives.

#### 4.1.5 Directives processing

Remaining preprocessor directives are processed. It is specific to each directive to what extent the above expansions or the ones specified in section 4.1.8 are performed on their arguments.

It is specific to each directive to what extent inline expansions and detokenization are performed for the arguments of the directives.

#### 4.1.6 Inline expansions and other directives

In this step, the following expansions are performed on each line:

- Single-line macros are expanded. See section 4.2.
- Preprocessor functions are expanded. See section 4.4.
- If this line is the result of multi-line macro expansions (see below), the parameters to that macro are expanded at this time. See section 4.5.
- Macro indirection, using the `%[]` construct, is expanded. See section 4.2.3.
- Token concatenation using either the `%+` operator (see section 4.2.4) or implicitly (see section 4.2.3 and section 4.5.9.)
- Macro-local labels are converted into unique strings, see section 4.5.2.

#### 4.1.7 Multi-Line Macro Expansion

In this step, multi-line macros are expanded into new lines of source, like the typical macro feature of many other assemblers. See section 4.5.

After expansion, the newly injected lines of source are processed starting with the step defined in section 4.1.4.

#### 4.1.8 Detokenization

In this step, the final line of source code is produced. It performs the following operations:

- Environment variables specified using the `%!` construct are expanded. See section 4.9.2.
- Context-local labels are expanded into unique strings. See section 4.9.2.
- All tokens are converted to their text representation. Unlike the C preprocessor, the NASM preprocessor does not insert whitespace between adjacent tokens unless present in the source code. See section 4.5.9.

The resulting line of text either is sent to the assembler, or, if running in preprocessor-only mode, to the output file (see section 2.1.22); if necessary prefixed by a newly inserted `%line` directive.

### 4.2 Single-Line Macros

Single-line macros are expanded inline, much like macros in the C preprocessor.

#### 4.2.1 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`.

Note that single-line macro argument list cannot be preceded by whitespace. Otherwise it will be treated as an expansion. For example:

```
%define foo (a,b)           ; no arguments, (a,b) is the expansion
%define bar(a,b)            ; two arguments, empty expansion
```

Macros defined with `%define` are case sensitive: after `%define foo bar`, only `foo` will expand to `bar`: `Foo` or `F00` will not. By using `%ifndef` instead of `%define` (the 'i' stands for 'insensitive') you can define all the case variants of a macro at once, so that `%ifndef foo bar` would cause `foo`, `Foo`, `F00`, `f00` 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 10.1 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 4.2.8).

The following additional features were added in NASM 2.15:

It is possible to define an empty string instead of an argument name if the argument is never used. For example:

```
%define ereg(foo,) e %+ foo
mov     eax,ereg(dx,cx)
```

A single pair of parentheses is a subcase of a single, unused argument:

```
%define myreg() eax
mov edx,myreg()
```

This is similar to the behavior of the C preprocessor.

- If declared with an `=`, NASM will expand the argument and then evaluate it as a numeric expression. The name of the argument may optionally be followed by `/` followed by a numeric radix character (b, y, o, q, d, t, h or x) and/or the letters u (unsigned) or s (signed), in which the number is formatted accordingly, with a radix prefix if a radix letter is specified. For the case of hexadecimal, if the radix letter is in upper case, alphabetic hex digits will be in upper case.
- If declared with an `&`, NASM will expand the argument and then turn into a quoted string; if the argument already *is* a quoted string, it will be quoted again.
- If declared with `&&`, NASM will expand the argument and then turn it into a quoted string, but if the argument already is a quoted string, it will *not* be re-quoted.
- If declared with a `+`, it is a greedy or variadic parameter; it will include any subsequent commas and parameters.
- If declared with an `!`, NASM will not strip whitespace and braces (potentially useful in conjunction with `&` or `&&`.)

For example:

```
%define xyzzy(=expr,&val,=hex/x) expr, str, hex
%define plugh(x) xyzzy(x,x,x)
db plugh(13+5), '\0' ; Expands to: db 18, "13+5", 0x12, '\0'
```

You can pre-define single-line macros using the `-d` option on the NASM command line: see section 2.1.20.

## 4.2.2 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 its 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.

`%xdefine` and `%ixdefine` supports argument expansion exactly the same way that `%define` and `%idefine` does.

### 4.2.3 Macro Indirection: %[...]

The `%[...]` construct can be used to expand macros in contexts where macro expansion would otherwise not occur, including in the names of 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 5.3) 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 4.5.9 for details.

### 4.2.4 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).

### 4.2.5 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:

```
%imacro Foo 0
    mov %?,%??
%endmacro
```

```
    foo
    F00
```

will expand to:

```
    mov foo,Foo
    mov F00,Foo
```

These tokens can be used for single-line macros *if defined outside any multi-line macros*. See below.

### 4.2.6 The Single-Line Macro Name: %\*? and %\*??

If the tokens %? and %?? are used inside a multi-line macro, they are expanded before any directives are processed. As a result,

```
%imacro Foo 0
    %define Bar _%?
    mov BAR,bAr
%endmacro
```

```
    foo
    mov eax,bar
```

will expand to:

```
    mov _foo,_foo
    mov eax,_foo
```

which may or may not be what you expected. The tokens %\*? and %\*?? behave like %? and %?? but are only expanded inside single-line macros. Thus:

```
%imacro Foo 0
    %define Bar _%*?
    mov BAR,bAr
%endmacro
```

```
    foo
    mov eax,bar
```

will expand to:

```
    mov _BAR,_bAr
    mov eax,_bar
```

The %\*? 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:

```
%define pause $%*? ; Hide the PAUSE instruction
```

%\*? and %\*?? were introduced in NASM 2.15.04.

### 4.2.7 Undefined 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 2.1.21.

#### 4.2.8 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 `%ifdef` 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 4.7 for an example of this. Another use for `%assign` is given in section 9.4 and section 10.1.

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).

See also the `%eval()` preprocessor function, section 4.4.4.

#### 4.2.9 Defining Strings: `%defstr`

`%defstr`, and its case-insensitive counterpart `%ifdefstr`, 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 4.13.2):

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

See also the `%str()` preprocessor function, section 4.4.10.

#### 4.2.10 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
```

See also the `%tok()` preprocessor function, section 4.4.14.

#### 4.2.11 Defining Aliases: `%defalias`

`%defalias`, and its case-insensitive counterpart `%idefalias`, define an alias to a macro, i.e. equivalent of a symbolic link.

When used with various macro defining and undefining directives, it affects the aliased macro. This functionality is intended for being able to rename macros while retaining the legacy names.

When an alias is defined, but the aliased macro is then undefined, the aliases can legitimately point to nonexistent macros.

The alias can be undefined using the `%undefalias` directive. *All* aliases can be undefined using the `%clear defalias` directive. This includes backwards compatibility aliases defined by NASM itself.

To disable aliases without undefining them, use the `%aliases off` directive.

To check whether an alias is defined, regardless of the existence of the aliased macro, use `%ifdefalias`.

For example:

```
%defalias OLD NEW
; OLD and NEW both undefined
%define NEW 123
; OLD and NEW both 123
%undef OLD
; OLD and NEW both undefined
%define OLD 456
; OLD and NEW both 456
%undefalias OLD
; OLD undefined, NEW defined to 456
```

#### 4.2.12 Conditional Comma Operator: `%,`

As of version 2.15, NASM has a conditional comma operator `%`, that expands to a comma *unless* followed by a null expansion, which allows suppressing the comma before an empty argument. This is especially useful with greedy single-line macros.

For example, all the expressions below are valid:

```
%define greedy(a,b,c+) a + 66 %, b * 3 %, c

db greedy(1,2)           ; db 1 + 66, 2 * 3
db greedy(1,2,3)          ; db 1 + 66, 2 * 3, 3
db greedy(1,2,3,4)         ; db 1 + 66, 2 * 3, 3, 4
db greedy(1,2,3,4,5)       ; db 1 + 66, 2 * 3, 3, 4, 5
```

### 4.3 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.

These directives are also available as preprocessor functions, see section 4.4.

#### 4.3.1 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.

The corresponding preprocessor function is `%strcat()`, see section 4.4.11.

### 4.3.2 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.

The corresponding preprocessor function is `%strlen()`, see section 4.4.12.

### 4.3.3 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 4.3.2), 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 preceded 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.

The corresponding preprocessor function is `%substr()`, see section 4.4.13, however please note that the default value for the length parameter, if omitted, is -1 rather than 1 for `%substr()`.

## 4.4 Preprocessor Functions

Preprocessor functions are, fundamentally, a kind of built-in single-line macros. They expand to a string depending on its arguments, and can be used in any context where single-line macro expansion would be performed. Preprocessor functions were introduced in NASM 2.16.

### 4.4.1 `%abs()` Function

The `%abs()` function evaluates its first argument as an expression, and then emits the absolute value. This will always be emitted as a single token containing a decimal number; no minus sign will be emitted even if the input value is the maximum negative number.

#### 4.4.2 %cond() Function

The %cond() function evaluates its first argument as an expression, then expands to its second argument if true (nonzero), and the third, if present, if false (zero). This is in effect a specialized version of the %sel() function; %cond(x,y,z) is equivalent to %sel(2-!(x),y,z).

```
%define a 1
%xdefine astr %cond(a,"true","false") ; %define astr "true"
```

The argument not selected is never expanded.

#### 4.4.3 %count() Function

The %count() function expands to the number of arguments passed to the macro. Note that just as for single-line macros, %count() treats an empty argument list as a single empty argument.

```
%xdefine empty %count() ; %define empty 1
%xdefine one %count(1) ; %define one 1
%xdefine two %count(5,q) ; %define two 2
%define list a,b,46
%xdefine lc1 %count(list) ; %define lc 1 (just one argument)
%xdefine lc2 %count(%[list]) ; %define lc 3 (indirection expands)
```

#### 4.4.4 %eval() Function

The %eval() function evaluates its argument as a numeric expression and expands to the result as an integer constant in much the same way the %assign directive would, see section 4.2.8. Unlike %assign, %eval() supports more than one argument; if more than one argument is specified, it is expanded to a comma-separated list of values.

```
%assign a 2
%assign b 3
%defstr what %eval(a+b,a*b) ; equivalent to %define what "5,6"
```

The expressions passed to %eval() are critical expressions, see section 3.8.

#### 4.4.5 %hex() Function

Equivalent to %eval(), except that the results generated are given as unsigned hexadecimal, with a 0x prefix.

#### 4.4.6 %is() Family Functions

Each %if family directive (see section 4.6) has an equivalent %is() family function, that expands to 1 if the equivalent %if directive would process as true, and 0 if the equivalent %if directive would process as false.

```
; Instead of !%isidn() could have used %isnidn()
%if %isdef(foo) && !%isidn(foo,bar)
    db "foo is defined, but not as 'bar'"
%endif
```

Note that, being functions, the arguments (before expansion) will always need to have balanced parentheses so that the end of the argument list can be defined. This means that the syntax of e.g. %istoken() and %isidn() is somewhat stricter than their corresponding %if directives; it may be necessary to escape the argument to the conditional using {}:

```
; Instead of !%isidn() could have used %isnidn()
%if %isdef(foo) && !%isidn({foo,})
    db "foo is defined, but not as ' '"
%endif
```

#### 4.4.7 %map() Function

The %map() function takes as its first parameter the name of a single-line macro, followed by up to two optional colon-separated subparameters:

- The first subparameter, if present, should be a list of macro parameters enclosed in parentheses. Note that `()` represents a one-argument list containing an empty parameter; omit the parentheses to specify no parameters.
- The second subparameter, if present, represent the number of group size for additional parameters to the macro (default 1).

Further parameters, if any, are then passed as additional parameters to the given macro for expansion, in sets given by the specified group size, and the results turned into a comma-separated list. If no additional parameters are given, `%map()` expands to nothing.

For example:

```
%define alpha(&x)      x
%define alpha(&x,y)    y dup (x)
%define alpha(s,&x,y)  y dup (x,s)
; 0 fixed + 1 grouped parameters per call, calls alpha(&x)
    db %map(alpha,foo,bar,baz,quux)
; 0 fixed + 2 grouped parameters per call, calls alpha(&x,y)
    db %map(alpha::2,foo,bar,baz,quux)
; 1 fixed + 2 grouped parameters per call, calls alpha(s,&x,y)
    db %map(alpha:("!!"):2,foo,bar,baz,quux)
```

... expands to:

```
db 'foo','bar','baz','quux'
db bar dup ('foo'),quux dup ('baz')
db bar dup ('foo','!!'),quux dup ('baz','!!')
```

As a more complex example, a macro that joins quoted strings together with a user-specified delimiter string:

```
%define join(sep)      ''      ; handle the case of zero strings
%define _join(sep,str)  sep,str ; helper macro
%define join(sep,s1,sn+) %strcat(s1, %map(_join:(sep) %, sn))

    db join(':')
    db join(':', 'a')
    db join(':', 'a', 'b')
    db join(':', 'a', 'b', 'c')
    db join(':', 'a', 'b', 'c', 'd')
```

... expands to:

```
db ''
db 'a'
db 'a:b'
db 'a:b:c'
db 'a:b:c:d'
```

#### 4.4.8 %num() Function

The `%num()` function evaluates its arguments as expressions, and then produces a quoted string encoding the first argument as an *unsigned* 64-bit integer.

The second argument is the desired number of digits (max 255, default -1).

The third argument is the encoding base (from 2 to 64, default 10); if the base is given as -2, -8, -10, or -16, then `0b`, `0q`, `0d` or `0x` is prepended, respectively; all other negative values are disallowed.

Only the first argument is required.

If the number of digits is negative, NASM will add additional digits if needed; if positive the string is truncated to the number of digits specified. 0 is treated as -1, except that the input number 0 always generates an empty string (thus, the first digit will never be zero), even if the base given is negative.

The full 64-symbol set used is, in order:

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ@\_

If a *signed* number needs to be converted to a string, use `%abs()`, `%cond()`, and `%strcat()` to format the signed number string to your specific output requirements.

#### 4.4.9 `%sel()` Function

The `%sel()` function evaluates its first argument as an expression, then expands to its second argument if 1, the third argument if 2, and so on. If the value is less than 1 or larger than the number of arguments minus one, then the `%sel()` function expands to nothing.

```
%define b 2
%xdefine bstr %sel(b,"one","two","three") ; %define bstr "two"
```

The arguments not selected are never expanded.

#### 4.4.10 `%str()` Function

The `%str()` function converts its argument, including any commas, to a quoted string, similar to the way the `%defstr` directive would, see section 4.2.9.

Being a function, the argument will need to have balanced parentheses or be escaped using `{}`.

```
; The following lines are all equivalent
%define test 'TEST'
%defstr test TEST
%xdefine test %str(TEST)
```

#### 4.4.11 `%strcat()` Function

The `%strcat()` function concatenates a list of quoted strings, in the same way the `%strcat` directive would, see section 4.3.1.

```
; The following lines are all equivalent
%define alpha 'Alpha: 12" screen'
%strcat alpha "Alpha: ", '12" screen'
%xdefine alpha %strcat("Alpha: ", '12" screen')
```

#### 4.4.12 `%strlen()` Function

The `%strlen()` function expands to the length of a quoted string, in the same way the `%strlen` directive would, see section 4.3.2.

```
; The following lines are all equivalent
%define charcnt 9
%strlen charcnt 'my string'
%xdefine charcnt %strlen('my string')
```

#### 4.4.13 `%substr()` Function

The `%substr()` function extracts a substring of a quoted string, in the same way the `%substr` directive would, see section 4.3.3. Note that unlike the `%substr` directive, commas are required between all parameters, is required after the string argument, and that the default for the length argument, if omitted, is -1 (i.e. the remainder of the string) rather than 1.

```
; The following lines are all equivalent
%define mychar 'xyzw'
%substr mychar 'xyzw' 2, -1
%xdefine mychar %substr('xyzw', 2, 3)
%xdefine mychar %substr('xyzw', 2, -1)
%xdefine mychar %substr('xyzw', 2)
```

#### 4.4.14 `%tok()` function

The `%tok()` function converts a quoted string into a sequence of tokens, in the same way the `%deftok` directive would, see section 4.2.10.



```
; The following lines are all equivalent
%define test TEST
%def tok test 'TEST'
%define test %tok('TEST')
```

## 4.5 Multi-Line Macros: %macro

Multi-line macros much like the type of macro seen in MASM and TASM, and expand to a new set of lines of source code. 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
```

The behavior with regards to empty arguments at the end of multi-line macros before NASM 2.15 was often very strange. For backwards compatibility, NASM attempts to recognize cases where the legacy behavior would give unexpected results, and issues a warning, but largely tries to match the legacy behavior. This can be disabled with the %pragma (see section 4.12.1):

```
%pragma preproc sane_empty_expansion
```

### 4.5.1 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 2.1.26).

#### 4.5.2 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.

These labels are really macro-local *tokens*, and can be used for other purposes where a token unique to each macro invocation is desired, e.g. to name single-line macros without using the context feature (section 4.9.2).

#### 4.5.3 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 7.3.1 for a better way to write the above macro.

#### 4.5.4 Macro Parameters Range

NASM 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 but 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}` which gives you the last argument passed to a macro.

#### 4.5.5 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 4.5.3) 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 2.1.26 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 4.5.6) 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 4.5.8.

#### 4.5.6 %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 4.7) in order to iterate through all the parameters of a macro. Examples are given in section 4.5.8.

### 4.5.7 %00: Label Preceding Macro

%00 will return the label preceding the macro invocation, if any. The label must be on the same line as the macro invocation, may be a local label (see section 3.9), and need not end in a colon.

If %00 is present anywhere in the macro body, the label itself will not be emitted by NASM. You can, of course, put %00: explicitly at the beginning of your macro.

### 4.5.8 %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.

### 4.5.9 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 4.5.2) and context-local labels (section 4.9.2). 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 4.2.3), behaves the same way as macro parameters for the purpose of concatenation.

See also the %+ operator, section 4.2.4.

### 4.5.10 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 4.5.2 can be replaced by a general conditional-return macro like this:

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

```
%%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.

#### 4.5.11 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
```

#### 4.5.12 Undefined 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.

A case-insensitive macro needs to be removed with the `%unimacro` directive.

### 4.6 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`.

#### 4.6.1 `%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`.

#### 4.6.2 `%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`.

#### 4.6.3 `%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 4.9. For a sample use of `%ifctx`, see section 4.9.6.



#### 4.6.4 %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 4.7 for a detailed example.

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

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

#### 4.6.5 %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`.

#### 4.6.6 %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. `$` and `$$` are *not* considered identifiers by `%ifid`.

`%ifnum` works similarly, but tests for the token being an integer numeric constant (not an expression!) possibly preceded by `+` or `-`; `%ifstr` tests for it being a quoted string.

For example, the `writefile` macro defined in section 4.5.3 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

%endmacro
```

```

%endif
        mov     bx,%1
        mov     ah,0x40
        int     0x21

%endmacro

```

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

```

writelfile [file], strpointer, length
writelfile [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`.

#### 4.6.7 `%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.

#### 4.6.8 `%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.

#### 4.6.9 `%ifenv`: Test If Environment Variable Exists

The conditional assembly construct `%ifenv` assembles the subsequent code if and only if the environment variable referenced by the `%!variable` directive exists.

The usual `%elifenv`, `%ifnenv`, and `%elifnenv` variants are also provided.

Just as for `%!variable` the argument should be written as a string if it contains characters that would not be legal in an identifier. See section 4.13.2.

### 4.7 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.

Note the maximum repeat count is limited to the value specified by the `--limit-rep` option or `%pragma limit rep`, see section 2.1.31.

## 4.8 Source Files and Dependencies

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

### 4.8.1 `%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 2.1.19).

#### 4.8.2 `%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"`.

#### 4.8.3 `%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 2.1.5) 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.

#### 4.8.4 `%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 6.

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 5.7.

### 4.9 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.

### 4.9.1 %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.

### 4.9.2 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.

### 4.9.3 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.

### 4.9.4 Context Fall-Through Lookup (*deprecated*)

Context fall-through lookup (automatic searching of outer contexts) is a feature that was added in NASM version 0.98.03. Unfortunately, this feature is unintuitive and can result in buggy code that would have otherwise been prevented by NASM's error reporting. As a result, this feature has been *deprecated*. NASM version 2.09 will issue a warning when usage of this *deprecated* feature is detected.

Starting with NASM version 2.10, usage of this *deprecated* feature will simply result in an *expression syntax error*.

An example usage of this *deprecated* feature follows:

```
%macro ctxthru 0
%push ctx1
    %assign %$external 1
    %push ctx2
        %assign %$internal 1
        mov eax, %$external
        mov eax, %$internal
    %pop
%pop
%endmacro
```

As demonstrated, %\$external is being defined in the ctx1 context and referenced within the ctx2 context. With context fall-through lookup, referencing an undefined context-local macro like this implicitly searches through all outer contexts until a match is made or isn't found in any context. As a result, %\$external referenced within the ctx2 context would implicitly use %\$external as defined in ctx1. Most people would expect NASM to issue an error in this situation because %\$external was never defined within ctx2 and also isn't qualified with the proper context depth, %\$\$external.

Here is a revision of the above example with proper context depth:

```
%macro ctxthru 0
%push ctx1
    %assign %$external 1
    %push ctx2
        %assign %$internal 1
        mov eax, %$$external
        mov eax, %$internal
    %pop
%pop
%endmacro
```

As demonstrated, %\$external is still being defined in the ctx1 context and referenced within the ctx2 context. However, the reference to %\$external within ctx2 has been fully qualified with the proper context depth, %\$\$external, and thus is no longer ambiguous, unintuitive or erroneous.

#### 4.9.5 %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.

#### 4.9.6 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 4.9.2, 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`.

## 4.10 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 4.10.1)
- `%stacksize` (see section 4.10.2)
- `%local` (see section 4.10.3)

### 4.10.1 `%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 9.4.5), 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 9.4.5 and adds the value in `i` to the value pointed to by `j_ptr` and returns the sum in the `ax` register. See section 4.9.1 for an explanation of `push` and `pop` and the use of context stacks.

### 4.10.2 `%stacksize` Directive

The `%stacksize` directive is used in conjunction with the `%arg` (see section 4.10.1) and the `%local` (see section 4.10.3) 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 4.10.3).

### 4.10.3 `%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 4.10.2 and is also compatible with the `%arg` directive (see section 4.10.1). 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.

## 4.11 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:

```
%ifndef 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:

```
%ifndef 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
```

## 4.12 %pragma: Setting Options

The `%pragma` directive controls a number of options in NASM. Pragas are intended to remain backwards compatible, and therefore an unknown `%pragma` directive is not an error.

The various pragmas are documented with the options they affect.

The general structure of a NASM pragma is:

`%pragma namespace directive [arguments...]`

Currently defined namespaces are:

- `ignore`: this `%pragma` is unconditionally ignored.
- `preproc`: preprocessor, see section 4.12.1.
- `limit`: resource limits, see section 2.1.31.
- `asm`: the parser and assembler proper. Currently no such pragmas are defined.
- `list`: listing options, see section 2.1.4.
- `file`: general file handling options. Currently no such pragmas are defined.
- `input`: input file handling options. Currently no such pragmas are defined.
- `output`: output format options.
- `debug`: debug format options.

In addition, the name of any output or debug format, and sometimes groups thereof, also constitute `%pragma` namespaces. The namespaces `output` and `debug` simply refer to *any* output or debug format, respectively.

For example, to prepend an underscore to global symbols regardless of the output format (see section 7.10):

```
%pragma output gprefix _
```

... whereas to prepend an underscore to global symbols only when the output is either `win32` or `win64`:

```
%pragma win gprefix _
```

### 4.12.1 Preprocessor Pragas

The only preprocessor `%pragma` defined in NASM 2.15 is:

- `%pragma preproc sane_empty_expansion`: disables legacy compatibility handling of braceless empty arguments to multi-line macros. See section 4.5 and section 2.1.26.

## 4.13 Other Preprocessor Directives

### 4.13.1 %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 used directly by programmers, but 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. It may be a quoted string, in which case any additional argument after the quoted string will be ignored.

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

If the command line option `--no-line` is given, all `%line` directives are ignored. This may be useful for debugging preprocessed code. See section 2.1.33.

Starting in NASM 2.15, `%line` directives are processed before any other processing takes place.

For compatibility with the output from some other preprocessors, including many C preprocessors, a `#` character followed by whitespace *at the very beginning of a line* is also treated as a `%line` directive, except that double quotes surrounding the filename are treated like NASM backquotes, with `\`-escaped sequences decoded.

### 4.13.2 %!variable: Read an Environment Variable.

The `%!variable` 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 `F00`, and you want the contents of `F00` to be embedded in your program as a quoted string. You could do that as follows:

```
%defstr F00          %!F00
```

See section 4.2.9 for notes on the `%defstr` directive.

If the name of the environment variable contains non-identifier characters, you can use string quotes to surround the name of the variable, for example:

```
%defstr C_colon      %!'C:'
```

### 4.13.3 %clear: Clear All Macro Definitions

The directive `%clear` clears all definitions of a certain type, *including the ones defined by NASM itself*. This can be useful when preprocessing non-NASM code, or to drop backwards compatibility aliases.

The syntax is:

```
%clear [global|context] type...
```

... where `context` indicates that this applies to context-local macros only; the default is `global`.

`type` can be one or more of:

- define single-line macros

- `defalias` single-line macro aliases (useful to remove backwards compatibility aliases)
- `alldefine` same as `define` `defalias`
- `macro` multi-line macros
- `all` same as `alldefine` `macro` (default)

In NASM 2.14 and earlier, only the single syntax `%clear` was supported, which is equivalent to `%clear global all`.

## Chapter 5: 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, see section 4.13.3.

Most user-level directives (see chapter 7) are implemented as macros which invoke primitive directives; these are described in chapter 7. The rest of the standard macro set is described here.

For compatibility with NASM versions before NASM 2.15, most standard macros of the form `__?foo?` have aliases of form `__foo__` (see section 4.2.11). These can be removed with the directive `%clear defalias`.

### 5.1 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*.

#### 5.1.1 `__?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.

#### 5.1.2 `__?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"
```

### 5.2 `__?FILE?` and `__?LINE?`: File Name and Line Number

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

These macros could be used, for example, to communicate debugging information to a macro, since invoking `__?LINE?` inside a macro definition (either single-line or multi-line) will return the line

number of the macro *call*, rather than *definition*. So to determine where in a piece of code a crash is occurring, for example, one could write a routine *stillhere*, which is passed a line number in EAX and outputs something like `line 155: still here`. You could then write a macro:

```
%macro notdeadyet 0

    push    eax
    mov     eax, __?LINE?__
    call    stillhere
    pop     eax

%endmacro
```

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

### 5.3 `__?BITS?__`: Current Code Generation 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.

### 5.4 `__?OUTPUT_FORMAT?__`: Current Output Format

The `__?OUTPUT_FORMAT?__` standard macro holds the current output format name, as given by the `-f` option or NASM's default. Type `nasm -h` for a list.

```
%ifidn __?OUTPUT_FORMAT?__, win32
    %define NEWLINE 13, 10
%elifidn __?OUTPUT_FORMAT?__, elf32
    %define NEWLINE 10
%endif
```

### 5.5 `__?DEBUG_FORMAT?__`: Current Debug Format

If debugging information generation is enabled, The `__?DEBUG_FORMAT?__` standard macro holds the current debug format name as specified by the `-F` or `-g` option or the output format default. Type `nasm -f output y` for a list.

`__?DEBUG_FORMAT?__` is not defined if debugging is not enabled, or if the debug format specified is `NULL`.

### 5.6 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
```

## 5.7 \_\_?USE\_package?\_\_: Package Include Test

When a standard macro package (see chapter 6) is included with the %use directive (see section 4.8.4), 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 6.1), then the macro \_\_?USE\_ALTREG?\_\_ is defined.

## 5.8 \_\_?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 2.1.5) 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.*

## 5.9 Structure Data Types

### 5.9.1 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

```

## 5.9.2 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
```

## 5.10 Alignment Control

### 5.10.1 ALIGN and ALIGNB: Code and 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.

Both ALIGN and ALIGNB do call SECTALIGN macro implicitly. See section 5.10.2 for details.

See also the `smartalign` standard macro package, section 6.2.

### 5.10.2 SECTALIGN: Section Alignment

The SECTALIGN macro provides a way to modify alignment attribute of output file section. Unlike the `align=` attribute (which is allowed at section definition only) the SECTALIGN macro may be used at any time.

For example the directive

```
SECTALIGN 16
```

sets the section alignment requirements to 16 bytes. Once increased it can not be decreased, the magnitude may grow only.

Note that ALIGN (see section 5.10.1) calls the SECTALIGN macro implicitly so the active section alignment requirements may be updated. This is by default behaviour, if for some reason you want the ALIGN do not call SECTALIGN at all use the directive

```
SECTALIGN OFF
```

It is still possible to turn in on again by

```
SECTALIGN ON
```

Note that SECTALIGN <ON|OFF> affects only the ALIGN/ALIGNB directives, not an explicit SECTALIGN directive.

## Chapter 6: Standard Macro Packages

---

The `%use` directive (see section 4.8.4) 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 4.8.1), but the included contents is provided by NASM itself.

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

As of version 2.15, NASM has `%ifusable` and `%ifusing` directives to help the user understand whether an individual package available in this version of NASM (`%ifusable`) or a particular package already loaded (`%ifusing`).

### 6.1 `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 12.1.

### 6.2 `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 5.10.1). 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` (or set it to `nojmp`). 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.

## 6.3 fp: Floating-point macros

This package contains the following floating-point convenience macros:

```
%define Inf          __?Infinity?__
%define NaN          __?QNaN?__
%define QNaN         __?QNaN?__
%define SNaN         __?SNaN?__

%define float8(x)     __?float8?__(x)
%define float16(x)    __?float16?__(x)
%define bfloat16(x)   __?bfloat16?__(x)
%define float32(x)    __?float32?__(x)
%define float64(x)    __?float64?__(x)
%define float80m(x)   __?float80m?__(x)
%define float80e(x)   __?float80e?__(x)
%define float128l(x)  __?float128l?__(x)
%define float128h(x)  __?float128h?__(x)
```

It also defines the a multi-line macro `bf16` that can be used in a similar way to the `dx` directives for the other floating-point numbers:

```
bf16 -3.1415, NaN, 2000.0, +Inf
```

## 6.4 ifunc: Integer functions

This package contains a set of macros which implement integer functions. These are actually implemented as special operators, but are most conveniently accessed via this macro package.

The macros provided are:

### 6.4.1 Integer logarithms

These functions calculate the integer logarithm base 2 of their argument, considered as an unsigned integer. The only differences between the functions is their respective behavior if the argument provided is not a power of two.

The function `ilog2e()` (alias `ilog2()`) generates an error if the argument is not a power of two.

The function `ilog2f()` rounds the argument down to the nearest power of two; if the argument is zero it returns zero.

The function `ilog2c()` rounds the argument up to the nearest power of two.

The functions `ilog2fw()` (alias `ilog2w()`) and `ilog2cw()` generate a warning if the argument is not a power of two, but otherwise behaves like `ilog2f()` and `ilog2c()`, respectively.

## 6.5 masm: MASM compatibility

Since version 2.15, NASM has a MASM compatibility package with minimal functionality, as intended to be used primarily with machine-generated code. It does not include any "programmer-friendly" shortcuts, nor does it in any way support `ASSUME`, symbol typing, or MASM-style structures.

To enable the package, use the directive:

```
%use masm
```

Currently, the MASM compatibility package emulates:

- The `FLAT` and `OFFSET` keywords are recognized and ignored.
- The `PTR` keyword signifies a memory reference, as if the argument had been put in square brackets:

```
mov eax,[foo]           ; memory reference
mov eax,dword ptr foo   ; memory reference
mov eax,dowrd ptr flat:foo ; memory reference
```

```

    mov eax,offset foo          ; address
    mov eax,foo                 ; address (ambiguous syntax in MASM)

```

- The SEGMENT ... ENDS syntax:

```

segname SEGMENT
...
segname ENDS

```

- The PROC ... ENDP syntax:

```

procname PROC [FAR]
...
procname ENDP

```

PROC will also define RET as a macro expanding to either RETF if FAR is specified and RETN otherwise. Any keyword after PROC other than FAR is ignored.

- The TBYTE keyword as an alias for TWORD (see section 2.2.7).
- The END directive is ignored.
- In 64-bit mode relative addressing is the default (DEFAULT REL, see section 7.2.1).

In addition, NASM now natively supports, regardless of whether this package is used or not:

- ? and DUP syntax for the DB etc data declaration directives (see section 3.2.1).
- displacement[base+index] syntax for memory operations, instead of [base+index+displacement].
- seg:[addr] instead of [seg:addr] syntax.
- A pure offset can be given to LEA without square brackets:

```

    lea rax,[foo]               ; standard syntax
    lea rax,foo                 ; also accepted

```



## Chapter 7: 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 8.

### 7.1 BITS: 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.

The `BITS` directive can also be used to generate code for a different mode than the standard one for the output format.

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 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 necessary, e.g. `BITS32` will *not* work!

### 7.1.1 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.

## 7.2 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 occasionally obnoxious, as the explicit form is pretty much the only one one wishes to use.

Currently, DEFAULT can set REL & ABS and BND & NOBND.

### 7.2.1 REL & ABS: RIP-relative addressing

This sets whether 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.

### 7.2.2 BND & NOBND: BND prefix

If DEFAULT BND is set, all bnd-prefix available instructions following this directive are prefixed with bnd. To override it, NOBND prefix can be used.

```
DEFAULT BND
    call foo          ; BND will be prefixed
    nobnd call foo    ; BND will NOT be prefixed
```

DEFAULT NOBND can disable DEFAULT BND and then BND prefix will be added only when explicitly specified in code.

DEFAULT BND is expected to be the normal configuration for writing MPX-enabled code.

## 7.3 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 8.1.3), 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.

### 7.3.1 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 4.5.3 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.

## 7.4 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.

## 7.5 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.

If a variable is declared both GLOBAL and EXTERN, or if it is declared as EXTERN and then defined, it will be treated as GLOBAL. If a variable is declared both as COMMON and EXTERN, it will be treated as COMMON.

## 7.6 REQUIRED: Unconditionally Importing Symbols from Other Modules

The REQUIRED keyword is similar to EXTERN one. The difference is that the EXTERN keyword as of version 2.15 does not generate unknown symbols as that prevents using common header files, as it might cause the linker to pull in a bunch of unnecessary modules.

If the old behavior is required, use REQUIRED keyword instead.

## 7.7 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.

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.

## 7.8 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.

## 7.9 STATIC: Local Symbols within Modules

Opposite to EXTERN and GLOBAL, STATIC is local symbol, but should be named according to the global mangling rules (named by analogy with the C keyword static as applied to functions or global variables).

```
static foo  
foo: ; codes
```

Unlike GLOBAL, STATIC does not allow object formats to accept private extensions mentioned in section 7.7.

## 7.10 (G|L)PREFIX, (G|L)POSTFIX: Mangling Symbols

PREFIX, GPREFIX, LPREFIX, POSTFIX, GPOSTFIX, and LPOSTFIX directives can prepend or append a string to a certain type of symbols, normally to fit specific ABI conventions

- PREFIX|GPREFIX: Prepend the argument to all EXTERN, COMMON, STATIC, and GLOBAL symbols.
- LPREFIX: Prepend the argument to all other symbols such as local labels and backend defined symbols.
- POSTFIX|GPOSTFIX: Append the argument to all EXTERN, COMMON, STATIC, and GLOBAL symbols.
- LPOSTFIX: Append the argument to all other symbols such as local labels and backend defined symbols.

These are macros implemented as pragmas, and using %pragma syntax can be restricted to specific backends (see section 4.12):

```
%pragma macho lprefix L_
```

Command line options are also available. See also section 2.1.28.

One example which supports many ABIs:

```
; The most common conventions
#pragma output gprefix _
#pragma output lprefix L_
; ELF uses a different convention
#pragma elf    gprefix          ; empty
#pragma elf    lprefix .L
```

Some toolchains are aware of a particular prefix for its own optimization options, such as dead code elimination. For instance, the Mach-O binary format has a linker convention that uses a simplistic naming scheme to chunk up sections into smaller subsections, each of which may be eliminated. When the `subsections_via_symbols` directive (section 8.8.4) is declared, each symbol is the start of a separate block. The subsection is, then, defined to include sections before the one that starts with a 'L'. `LPREFIX` is useful here to mark all local symbols with the 'L' prefix to be excluded to the meta section. It converts local symbols compatible with the particular toolchain. Note that local symbols declared with `STATIC` (section 7.9) are excluded from the symbol mangling and also not marked as global.

## 7.11 CPU: Defining CPU Dependencies

The CPU directive restricts assembly to those instructions which are available on the specified CPU. At the moment, it is primarily used to enforce unavailable *encodings* of instructions, such as 5-byte jumps on the 8080.

(If someone would volunteer to work through the database and add proper annotations to each instruction, this could be greatly improved. Please contact the developers to volunteer, see appendix E.)

Current CPU keywords 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
- CPU DEFAULT – All available instructions
- CPU ALL – All available instructions *and flags*

All options are case insensitive.

In addition, optional flags can be specified to modify the instruction selections. These can be combined with a CPU declaration or specified alone. They can be prefixed by + (add flag, default), - (remove flag) or \* (set flag to default); these prefixes are "sticky", so:

```
cpu -foo, bar
```

means remove both the `foo` and `bar` options.

If prefixed with `no`, it inverts the meaning of the flag, but this is not sticky, so:

```
cpu nofoo, bar
```

means remove the `foo` flag but add the `bar` flag.

Currently available flags are:

- `EVEX` – Enable generation of EVEX (AVX-512) encoded instructions without an explicit `{evex}` prefix. Default on.
- `VEX` – Enable generation of VEX (AVX) or XOP encoded instructions without an explicit `{vex}` prefix. Default on.
- `LATEVEX` – Enable generation of VEX (AVX) encoding of instructions where the VEX instructions forms were introduced *after* the corresponding EVEX (AVX-512) instruction forms without requiring an explicit `{vex}` prefix. This is implicit if the `EVEX` flag is disabled and the `VEX` flag is enabled. Default off.

## 7.12 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 bracketed 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.

## 7.13 [WARNING]: Enable or disable warnings

The `[WARNING]` directive can be used to enable or disable classes of warnings in the same way as the `-w` option, see appendix A for more details about warning classes.

- `[warning +warning-class]` enables warnings for *warning-class*.
- `[warning -warning-class]` disables warnings for *warning-class*.
- `[warning *warning-class]` restores *warning-class* to the original value, either the default value or as specified on the command line.
- `[warning push]` saves the current warning state on a stack.

- `[warning pop]` restores the current warning state from the stack.

The `[WARNING]` directive also accepts the `all`, `error` and `error=warning-class` specifiers, see section 2.1.26.

No "user form" (without the brackets) currently exists.

## Chapter 8: 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.

### 8.1 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 8.1.3.

Using the `bin` format puts NASM by default into 16-bit mode (see section 7.1). 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`.

It is extremely important to understand that the binary output format is simply nothing other than *a linker built into the NASM executable*. As such, NASM behaves just as it does when producing any other output format: notably the list file reflects the code output *before* relocation, and the addresses in the list file are addresses relative to the start of the current output section.

#### 8.1.1 ORG: Binary File Program Origin

The `bin` format provides an additional directive to the list given in chapter 7: `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 13.1.3 for further comments.

#### 8.1.2 bin Extensions to the SECTION Directive, bin extensions to }

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.

### 8.1.3 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.

### 8.1.4 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.

## 8.2 `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 further relocation or linking. It is usually used with ROM programmers and similar utilities.

From a programmer point of view, this behaves identically to the `.bin` format; the only difference is the encoding of the output. All extensions supported by the `bin` file format is also supported by the `ith` format.

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

## 8.3 `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.



From a programmer point of view, this behaves identically to the .bin format; the only difference is the encoding of the output. 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.

## 8.4 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
```

### 8.4.1 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.

## 8.4.2 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.

#### 8.4.3 **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.

#### 8.4.4 **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  WSASStartup 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
```

#### 8.4.5 **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 optimization 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

```

#### 8.4.6 `..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.

#### 8.4.7 `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 8.4.8.

#### 8.4.8 `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 8.4.7). 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
```

#### 8.4.9 Embedded File Dependency Information

Since NASM 2.13.02, `obj` files contain embedded dependency file information. To suppress the generation of dependencies, use

```
%pragma obj nodepend
```

### 8.5 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 8.4).

`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.

#### 8.5.1 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 `.directve` 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.

- `comdat=`, followed by a number ("selection"), colon (acting as a separator) and a name, marks the section as a "COMDAT section". It allows Microsoft linkers to perform function-level linking, to deal with multiply defined symbols, to eliminate dead code/data. The "selection" number should be one of the `IMAGE_COMDAT_SELECT_*` constants from COFF format specification; this value controls if the linker allows multiply defined symbols and how it handles them. The name is the "COMDAT symbol" – basically a new name for the section. So even though you have one section given by the main name (e.g. `.text`), it can actually consist of hundreds of COMDAT sections having their own name (and alignment). When the "selection" is `IMAGE_COMDAT_SELECT_ASSOCIATIVE` (5), the following name is the "COMDAT symbol" of the associated COMDAT section; this way you can link a piece of code or data only when another piece of code or data gets actually linked.

So, when linking a NASM-compiled file with some C code, the source may be structured as follows. Note that the default `.text` section is handled in a special way and it doesn't work well with `comdat`; you may want to append a `$` character and an arbitrary suffix to the section name. It will get linked into the `.text` section anyway – see the info on Grouped Sections.

```
section .text$1 align=16 comdat=1:FirstFnc
...                               ; Code linked only if referenced from C

section .text$1 align=16 comdat=1:SecondFnc
...                               ; Code linked only if referenced from C

section .rdata align=32 comdat=5:FirstFnc
...                               ; Data linked only if the related code (FirstFnc) is linked
```

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
```

The win64 format also adds:

```
section .pdata   rdata align=4
section .xdata   rdata align=8
```

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

## 8.5.2 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 .directive 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.

### 8.5.3 Debugging formats for Windows

The win32 and win64 formats support the Microsoft CodeView debugging format. Currently CodeView version 8 format is supported (cv8), but newer versions of the CodeView debugger should be able to handle this format as well.

## 8.6 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 8.5) 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 8.5), in NASM, with regard to this exception.

### 8.6.1 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 a 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, [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, [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

```



### 8.6.2 win64: Structured Exception Handling

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

```

The keyword is that up to `magic_point` original `rsq` value remains in chosen volatile register and no non-volatile register, except for `rsq`, is modified. While past `magic_point` `rsq` 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,
        disp->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.

## 8.7 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.

## 8.8 macho32 and macho64: Mach Object File Format

The macho32 and macho64 output formats produce 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.

### 8.8.1 macho extensions to the SECTION Directive

The macho output format specifies section names in the format "*segment,section*". No spaces are allowed around the comma. The following flags can also be specified:

- data – this section contains initialized data items
- code – this section contains code exclusively
- mixed – this section contains both code and data
- bss – this section is uninitialized and filled with zero
- zerofill – same as bss
- no\_dead\_strip – inhibit dead code stripping for this section
- live\_support – set the live support flag for this section
- strip\_static\_syms – strip static symbols for this section
- debug – this section contains debugging information
- align=*alignment* – specify section alignment

The default is data, unless the section name is \_\_text or \_\_bss in which case the default is text or bss, respectively.

For compatibility with other Unix platforms, the following standard names are also supported:

```
.text    = __TEXT,__text    text
.rodata  = __DATA,__const   data
.data    = __DATA,__data    data
.bss     = __DATA,__bss     bss
```

If the `.rodata` section contains no relocations, it is instead put into the `__TEXT,__const` section unless this section has already been specified explicitly. However, it is probably better to specify `__TEXT,__const` and `__DATA,__const` explicitly as appropriate.

### 8.8.2 Thread Local Storage in Mach-O: macho special symbols and WRT

Mach-O defines the following special symbols that can be used on the right-hand side of the `WRT` operator:

- `__tvp` is used to specify access to thread-local storage.
- `__gotpcrel` is used to specify references to the Global Offset Table. The GOT is supported in the macho64 format only.

### 8.8.3 macho specific directive subsections\_via\_symbols

The directive `subsections_via_symbols` sets the `MH_SUBSECTIONS_VIA_SYMBOLS` flag in the Mach-O header, that effectively separates a block (or a subsection) based on a symbol. It is often used for eliminating dead codes by a linker.

This directive takes no arguments.

This is a macro implemented as a `%pragma`. It can also be specified in its `%pragma` form, in which case it will not affect non-Mach-O builds of the same source code:

```
%pragma macho subsections_via_symbols
```

### 8.8.4 macho specific directive no\_dead\_strip

The directive `no_dead_strip` sets the Mach-O `SH_NO_DEAD_STRIP` section flag on the section containing a specific symbol. This directive takes a list of symbols as its arguments.

This is a macro implemented as a `%pragma`. It can also be specified in its `%pragma` form, in which case it will not affect non-Mach-O builds of the same source code:

```
%pragma macho no_dead_strip symbol...
```

### 8.8.5 macho specific extensions to the GLOBAL Directive: private\_extern

The directive extension to `GLOBAL` marks the symbol with limited global scope. For example, you can specify the global symbol with this extension:

```
global foo:private_extern
foo:
    ; codes
```

Using `with static linker` will clear the `private extern` attribute. But linker option like `-keep_private_externs` can avoid it.

## 8.9 elf32, elf64, elfx32: Executable and Linkable Format Object Files

The `elf32`, `elf64` and `elfx32` 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`.

The `elfx32` format is used for the x32 ABI, which is a 32-bit ABI with the CPU in 64-bit mode.

### 8.9.1 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.

### 8.9.2 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.
- `note` indicates that this section contains ELF notes. The content of ELF notes are specified using normal assembly instructions; it is up to the programmer to ensure these are valid ELF notes.
- `preinit_array` indicates that this section contains function addresses to be called before any other initialization has happened.
- `init_array` indicates that this section contains function addresses to be called during initialization.
- `fini_array` indicates that this section contains function pointers to be called during termination.
- `align=`, used with a trailing number as in `obj`, gives the alignment requirements of the section.
- `byte`, `word`, `dword`, `qword`, `tword`, `oword`, `yword`, or `zword` with an optional \*multiplier specify the fundamental data item size for a section which contains either fixed-sized data structures or strings; it also sets a default alignment. This is generally used with the `strings` and `merge` attributes (see below.) For example `byte*4` defines a unit size of 4 bytes, with a default alignment of 1; `dword` also defines a unit size of 4 bytes, but with a default alignment of 4. The `align=` attribute, if specified, overrides this default alignment.
- `pointer` is equivalent to `dword` for `elf32` or `elfx32`, and `qword` for `elf64`.
- `strings` indicate that this section contains exclusively null-terminated strings. By default these are assumed to be byte strings, but a size specifier can be used to override that.
- `merge` indicates that duplicate data elements in this section should be merged with data elements from other object files. Data elements can be either fixed-sized objects or null-terminated strings (with the `strings` attribute.) A size specifier is required unless `strings` is specified, in which case the size defaults to `byte`.
- `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 .preinit_array	preinit_array	alloc	noexec	nowrite	pointer	
section .init_array	init_array	alloc	noexec	nowrite	pointer	
section .fini_array	fini_array	alloc	noexec	nowrite	pointer	
section .note	note	noalloc	noexec	nowrite	align=4	
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.)

### 8.9.3 Position-Independent Code: ELF Special Symbols and WRT

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 10.2.

### 8.9.4 Thread Local Storage in ELF: `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 or ELF32 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]
```

### 8.9.5 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
```

Since version 2.15, it is possible to specify symbols binding. The keywords are: `weak` to generate weak symbol or `strong`. The default is `strong`.

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 10.2.4.

### 8.9.6 elf Extensions to the EXTERN Directive

Since version 2.15 it is possible to specify keyword `weak` to generate weak external reference. Example:

```
extern weak_ref:weak
```

### 8.9.7 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.

### 8.9.8 16-bit code and ELF

Older versions of the ELF32 specification did not provide relocations for 8- and 16-bit values. It is now part of the formal specification, and any new enough linker should support them.

ELF has currently no support for segmented programming.

### 8.9.9 Debug formats and ELF

ELF provides 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.

## 8.10 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 8.9.) 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.

## 8.11 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 8.9.3 for full documentation of this feature.

aoutb also supports the same extensions to the GLOBAL directive as elf does: see section 8.9.5 for documentation of this.

## 8.12 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.

## 8.13 dbg: Debugging 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 elf32 -o elfprog.i elfprog.asm  
nasm -a -f dbg elfprog.i
```

This preprocesses `elfprog.asm` into `elfprog.i`, keeping the `elf32` object format selected in order to make sure ELF 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.

`dbg` accepts and logs any `%pragma`, but the specific `%pragma`:

```
%pragma dbg maxdump <size>
```

where `<size>` is either a number or `unlimited`, can be used to control the maximum size for dumping the full contents of a `rawdata` output object.



## Chapter 9: 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.

### 9.1 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.

#### 9.1.1 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](http://x2ftp.oulu.fi). An LZH archiver can be found at [ftp.simtel.net](http://ftp.simtel.net). There is another 'free' linker (though this one doesn't come with sources) called FREELINK, available from [www.pcorner.com](http://www.pcorner.com). A third, `djlink`, written by DJ Delorie, is available at [www.delorie.com](http://www.delorie.com). A fourth linker, ALINK, written by Anthony A.J. Williams, is available at [alink.sourceforge.net](http://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 8.4.6). 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.

### 9.1.2 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.

## 9.2 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.

### 9.2.1 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.

### 9.2.2 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.

## 9.3 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`.

## 9.4 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.

### 9.4.1 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 2.1.28.

## 9.4.2 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`.

## 9.4.3 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 9.5.1). 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.

#### 9.4.4 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 9.4.1.) 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.

### 9.4.5 `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 4.10 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.

## 9.5 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.

### 9.5.1 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 `RET` 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);
```

## 9.5.2 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.

## 9.5.3 Using c16.mac With Pascal Programs

The c16.mac macro package, described in section 9.4.5, 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 9.4.5: 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 10: 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.

### 10.1 Interfacing to 32-bit C Programs

A lot of the discussion in section 9.4, 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.

#### 10.1.1 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 9.4.1, will still work. For ELF, though, the leading underscore should not be used.

See also section 2.1.28.

#### 10.1.2 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);
```

### 10.1.3 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 10.1.1.) 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.

#### 10.1.4 `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.

## 10.2 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 10.2.4 for a caveat).

### 10.2.1 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
```

## 10.2.2 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.

## 10.2.3 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.

## 10.2.4 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.

## 10.2.5 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`.

## 10.2.6 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 11: 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 initialization 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.

### 11.1 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.

### 11.2 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 11.1, 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 11.1, 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.

## 11.3 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`, `MOVSB`, `MOVSW`, `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 12: 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 7.2).

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 C/C++ 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>`.

All known 64-bit platforms except some embedded platforms require that the stack is 16-byte aligned at the entry to a function. In order to enforce that, the stack pointer (RSP) needs to be aligned on an odd multiple of 8 bytes before the CALL instruction.

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.

### 12.1 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, respectively:

```
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 defining 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 6.1) can be used for this purpose.

### 12.2 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.

If optimization is enabled and NASM can determine at assembly time that a shorter instruction will suffice, the shorter instruction will be emitted unless of course `STRICT QWORD` or `STRICT DWORD` is specified (see section 3.7):

```

mov rax,1             ; Assembles as "mov eax,1" (5 bytes)
mov rax,strict qword 1 ; Full 10-byte instruction
mov rax,strict dword 1 ; 7-byte instruction
mov rax,symbol        ; 10 bytes, not known at assembly time
lea rax,[rel symbol]   ; 7 bytes, usually preferred by the ABI

```

Note that `lea rax,[rel symbol]` is position-independent, whereas `mov rax,symbol` is not. Most ABIs prefer or even require position-independent code in 64-bit mode. However, the `MOV` instruction is able to reference a symbol anywhere in the 64-bit address space, whereas `LEA` is only able to access a symbol within within 2 GB of the instruction itself (see below.)

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 `ABS 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:0, 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.

## 12.3 Interfacing to 64-bit C Programs (Unix)

On Unix, the 64-bit ABI as well as the x32 ABI (32-bit ABI with the CPU in 64-bit mode) is defined by the documents at:

<http://www.nasm.us/abi/unix64>

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`, which is 80 bits (`TWORD`) on most platforms (Android is one exception; there `long double` is 64 bits and treated the same as `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.

## 12.4 Interfacing to 64-bit C Programs (Win64)

The Win64 ABI is described by the document at:

<http://www.nasm.us/abi/win64>

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 13: Troubleshooting

---

This chapter describes some of the common problems that users have been known to encounter with NASM, and answers them. If you think you have found a bug in NASM, please see section E.2.

## 13.1 Common Problems

### 13.1.1 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 2.1.24).

### 13.1.2 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 2.1.24.

### 13.1.3 `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.

### 13.1.4 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.

## Appendix A: List of Warning Classes

---

These are the warning classes currently defined by NASM for the purpose of enabling, disabling and promoting to error. See section 2.1.26 and section 7.13.

- `all`: all possible warnings  
`all` is a group alias for *all* warning classes. Thus, `-w+all` enables all available warnings, and `-w-all` disables warnings entirely (since NASM 2.13).
- `bad-pragma`: malformed `%pragma`  
`bad-pragma` is a backwards compatibility alias for `pragma-bad`.
- `bnd`: invalid `BND` prefix  
`bnd` is a backwards compatibility alias for `prefix-bnd`.
- `db-empty`: no operand for data declaration  
`db-empty` warns about a `db` declaration with no operands, producing no output. This is permitted, but often indicative of an error. See section 3.2.1.  
Enabled by default.
- `ea`: all `ea-` warnings  
`ea` is a group alias for all warning classes prefixed by `ea-`; currently `ea-absolute` and `ea-dispsize`.
- `ea-absolute`: absolute address cannot be RIP-relative  
`ea-absolute` warns that an address that is inherently absolute cannot be generated with RIP-relative encoding using `REL`, see section 7.2.1.  
Enabled by default.
- `ea-dispsize`: displacement size ignored on absolute address  
`ea-dispsize` warns that NASM does not support generating displacements for inherently absolute addresses that do not match the address size of the instruction.  
Enabled by default.
- `environment`: nonexistent environment variable  
`environment` is a backwards compatibility alias for `pp-environment`.
- `float`: all `float-` warnings  
`float` is a group alias for all warning classes prefixed by `float-`; currently `float-denorm`, `float-overflow`, `float-toolong`, `float-underflow`.
- `float-denorm`: floating point denormal  
`float-denorm` warns about denormal floating point constants.  
Disabled by default.
- `float-overflow`: floating point overflow  
`float-overflow` warns about floating point underflow.  
Enabled by default.
- `float-toolong`: too many digits in floating-point number

`float-too-long` warns about too many digits in floating-point numbers.

Enabled by default.

- `float-underflow`: floating point underflow

`float-underflow` warns about floating point underflow (a nonzero constant rounded to zero.)

Disabled by default.

- `forward`: forward reference may have unpredictable results

`forward` warns that a forward reference is used which may have unpredictable results, notably in a `RESB`-type pseudo-instruction. These would be *critical expressions* (see section 3.8) but are permitted in a handful of cases for compatibility with older versions of NASM. This warning should be treated as a severe programming error as the code could break at any time for any number of reasons.

Enabled by default.

- `hle`: invalid HLE prefix

`hle` is a backwards compatibility alias for `prefix-hle`.

- `label`: all `label-` warnings

`label` is a group alias for all warning classes prefixed by `label-`; currently `label-orphan`, `label-redef`, `label-redef-late`.

- `label-orphan`: labels alone on lines without trailing :

`label-orphan` warns about source lines which contain no instruction but define a label without a trailing colon. This is most likely indicative of a typo, but is technically correct NASM syntax (see section 3.1.)

Enabled by default.

- `label-redef`: label redefined to an identical value

`label-redef` warns if a label is defined more than once, but the value is identical. It is an unconditional error to define the same label more than once to *different* values.

Disabled by default.

- `label-redef-late`: label (re)defined during code generation

`label-redef-late` the value of a label changed during the final, code-generation pass. This may be the result of strange use of the preprocessor. This is very likely to produce incorrect code and may end up being an unconditional error in a future version of NASM.

Enabled and promoted to error by default.

- `lock`: `LOCK` prefix on unlockable instruction

`lock` is a backwards compatibility alias for `prefix-lock-error`.

- `macro-def-case-single`: single-line macro defined both case sensitive and insensitive

`macro-def-case-single` is a backwards compatibility alias for `pp-macro-def-case-single`.

- `macro-def-greedy-single`: single-line macro

`macro-def-greedy-single` is a backwards compatibility alias for `pp-macro-def-greedy-single`.

- `macro-def-param-single`: single-line macro defined with and without parameters

`macro-def-param-single` is a backwards compatibility alias for `pp-macro-def-param-single`.

- `macro-defaults`: macros with more default than optional parameters



`macro-defaults` is a backwards compatibility alias for `pp-macro-defaults`.

- `macro-params-legacy`: improperly calling multi-line macro for legacy support

`macro-params-legacy` is a backwards compatibility alias for `pp-macro-params-legacy`.

- `macro-params-multi`: multi-line macro calls with wrong parameter count

`macro-params-multi` is a backwards compatibility alias for `pp-macro-params-multi`.

- `macro-params-single`: single-line macro calls with wrong parameter count

`macro-params-single` is a backwards compatibility alias for `pp-macro-params-single`.

- `negative-rep`: negative %rep count

`negative-rep` is a backwards compatibility alias for `pp-rep-negative`.

- `not-my-pragma`: %pragma not applicable to this compilation

`not-my-pragma` is a backwards compatibility alias for `pragma-na`.

- `number-overflow`: numeric constant does not fit

`number-overflow` covers warnings about numeric constants which don't fit in 64 bits.

Enabled by default.

- `obsolete`: all obsolete- warnings

`obsolete` is a group alias for all warning classes prefixed by `obsolete-`; currently `obsolete-nop`, `obsolete-removed`, `obsolete-valid`.

- `obsolete-nop`: instruction obsolete and is a noop on the target CPU

`obsolete-nop` warns for an instruction which has been removed from the architecture, but has been architecturally defined to be a noop for future CPUs.

Enabled by default.

- `obsolete-removed`: instruction obsolete and removed on the target CPU

`obsolete-removed` warns for an instruction which has been removed from the architecture, and is no longer included in the CPU definition given in the `[CPU]` directive, for example `POP CS`, the opcode for which, `0Fh`, instead is an opcode prefix on CPUs newer than the first generation 8086.

Enabled by default.

- `obsolete-valid`: instruction obsolete but valid on the target CPU

`obsolete-valid` warns for an instruction which has been removed from the architecture, but is still valid on the specific CPU given in the CPU directive. Code using these instructions is most likely not forward compatible.

Enabled by default.

- `orphan-labels`: labels alone on lines without trailing :

`orphan-labels` is a backwards compatibility alias for `label-orphan`.

- `other`: any warning not specifically mentioned above

`other` specifies any warning not included in any specific warning class.

Enabled by default.

- `phase`: phase error during stabilization

phase warns about symbols having changed values during the second-to-last assembly pass. This is not inherently fatal, but may be a source of bugs.

Disabled by default.

- `pp`: all `pp`- warnings

`pp` is a group alias for all warning classes prefixed by `pp-`; currently `pp-else-elif`, `pp-else-else`, `pp-empty-braces`, `pp-environment`, `pp-macro-def-case-single`, `pp-macro-def-greedy-single`, `pp-macro-def-param-single`, `pp-macro-defaults`, `pp-macro-params-legacy`, `pp-macro-params-multi`, `pp-macro-params-single`, `pp-macro-redef-multi`, `pp-open-braces`, `pp-open-brackets`, `pp-open-string`, `pp-rep-negative`, `pp-sel-range`, `pp-trailing`.

- `pp-else`: all `pp-else`- warnings

`pp-else` is a group alias for all warning classes prefixed by `pp-else-`; currently `pp-else-elif` and `pp-else-else`.

- `pp-else-elif`: `%elif` after `%else`

`pp-else-elif` warns that an `%elif`-type directive was encountered after `%else` has already been encountered. As a result, the content of the `%elif` will never be expanded.

Enabled by default.

- `pp-else-else`: `%else` after `%else`

`pp-else-else` warns that a second `%else` clause was found for the same `%if` statement. The content of this `%else` clause will never be expanded.

Enabled by default.

- `pp-empty-braces`: empty `%{}` construct

`pp-empty-braces` warns that an empty `%{}` was encountered. This expands to a single `%` character, which is normally the `%` arithmetic operator.

Enabled by default.

- `pp-environment`: nonexistent environment variable

`pp-environment` warns if a nonexistent environment variable is accessed using the `%!preprocessor` construct (see section 4.13.2.) Such environment variables are treated as empty (with this warning issued) starting in NASM 2.15; earlier versions of NASM would treat this as an error.

Enabled by default.

- `pp-macro`: all `pp-macro`- warnings

`pp-macro` is a group alias for all warning classes prefixed by `pp-macro-`; currently `pp-macro-def-case-single`, `pp-macro-def-greedy-single`, `pp-macro-def-param-single`, `pp-macro-defaults`, `pp-macro-params-legacy`, `pp-macro-params-multi`, `pp-macro-params-single`, `pp-macro-redef-multi`.

- `pp-macro-def`: all `pp-macro-def`- warnings

`pp-macro-def` is a group alias for all warning classes prefixed by `pp-macro-def-`; currently `pp-macro-def-case-single`, `pp-macro-def-greedy-single`, `pp-macro-def-param-single`.

- `pp-macro-def-case-single`: single-line macro defined both case sensitive and insensitive

`pp-macro-def-case-single` warns when a single-line macro is defined both case sensitive and case insensitive. The new macro definition will override (shadow) the original one, although the original macro is not deleted, and will be re-exposed if the new macro is deleted with `%undef`, or, if the original macro is the case insensitive one, the macro call is done with a different case.

Enabled by default.

- `pp-macro-def-greedy-single`: single-line macro

`pp-macro-def-greedy-single` definition shadows greedy macro warns when a single-line macro is defined which would match a previously existing greedy definition. The new macro definition will override (shadow) the original one, although the original macro is not deleted, and will be re-exposed if the new macro is deleted with `%undef`, and will be invoked if called with a parameter count that does not match the new definition.

Enabled by default.

- `pp-macro-def-param-single`: single-line macro defined with and without parameters

`pp-macro-def-param-single` warns if the same single-line macro is defined with and without parameters. The new macro definition will override (shadow) the original one, although the original macro is not deleted, and will be re-exposed if the new macro is deleted with `%undef`.

Enabled and promoted to error by default.

- `pp-macro-defaults`: macros with more default than optional parameters

`pp-macro-defaults` warns when a macro has more default parameters than optional parameters. See section 4.5.5 for why might want to disable this warning.

Enabled by default.

- `pp-macro-params`: all `pp-macro-params-` warnings

`pp-macro-params` is a group alias for all warning classes prefixed by `pp-macro-params-`; currently `pp-macro-params-legacy`, `pp-macro-params-multi`, `pp-macro-params-single`.

- `pp-macro-params-legacy`: improperly calling multi-line macro for legacy support

`pp-macro-params-legacy` warns about multi-line macros being invoked with the wrong number of parameters, but for bug-compatibility with NASM versions older than 2.15, NASM tried to fix up the parameters to match the legacy behavior and call the macro anyway. This can happen in certain cases where there are empty arguments without braces, sometimes as a result of macro expansion.

The legacy behavior is quite strange and highly context-dependent, and can be disabled with:

```
%pragma preproc sane_empty_expansion true
```

It is highly recommended to use this option in new code.

Enabled by default.

- `pp-macro-params-multi`: multi-line macro calls with wrong parameter count

`pp-macro-params-multi` warns about multi-line macros being invoked with the wrong number of parameters. See section 4.5.1 for an example of why you might want to disable this warning.

Enabled by default.

- `pp-macro-params-single`: single-line macro calls with wrong parameter count

`pp-macro-params-single` warns about single-line macros being invoked with the wrong number of parameters.

Enabled by default.

- `pp-macro-redef-multi`: redefining multi-line macro

`pp-macro-redef-multi` warns that a multi-line macro is being redefined, without first removing the old definition with `%unmacro`.

Enabled by default.

- **pp-open: all pp-open- warnings**  
pp-open is a group alias for all warning classes prefixed by pp-open-; currently pp-open-braces, pp-open-brackets, pp-open-string.
- **pp-open-braces: unterminated %{...}**  
pp-open-braces warns that a preprocessor parameter enclosed in braces %{...} lacks the terminating } character.  
Enabled by default.
- **pp-open-brackets: unterminated %[...]**  
pp-open-brackets warns that a preprocessor %[...] construct lacks the terminating ] character.  
Enabled by default.
- **pp-open-string: unterminated string**  
pp-open-string warns that a quoted string without a closing quotation mark was encountered during preprocessing.  
Enabled by default.
- **pp-rep-negative: regative %rep count**  
pp-rep-negative warns about a negative count given to the %rep preprocessor directive.  
Enabled by default.
- **pp-sel-range: %sel() argument out of range**  
pp-sel-range warns that the %sel() preprocessor function was passed a value less than 1 or larger than the number of available arguments.  
Enabled by default.
- **pp-trailing: trailing garbage ignored**  
pp-trailing warns that the preprocessor encountered additional text where no such text was expected. This can sometimes be the result of an incorrectly written expression, or arguments that are inadvertently separated.  
Enabled by default.
- **pragma: all pragma- warnings**  
pragma is a group alias for all warning classes prefixed by pragma-; currently pragma-bad, pragma-empty, pragma-na, pragma-unknown.
- **pragma-bad: malformed %pragma**  
pragma-bad warns about a malformed or otherwise unparsable %pragma directive.  
Disabled by default.
- **pragma-empty: empty %pragma directive**  
pragma-empty warns about a %pragma directive containing nothing. This is treated identically to %pragma ignore except for this optional warning.  
Disabled by default.
- **pragma-na: %pragma not applicable to this compilation**  
pragma-na warns about a %pragma directive which is not applicable to this particular assembly session. This is not yet implemented.

Disabled by default.

- `pragma-unknown`: unknown `%pragma` facility or directive

`pragma-unknown` warns about an unknown `%pragma` directive. This is not yet implemented for most cases.

Disabled by default.

- `prefix`: all `prefix-` warnings

`prefix` is a group alias for all warning classes prefixed by `prefix-`; currently `prefix-bnd`, `prefix-hle`, `prefix-lock-error`, `prefix-lock-xchg`, `prefix-opsize`, `prefix-seg`.

- `prefix-bnd`: invalid BND prefix

`prefix-bnd` warns about ineffective use of the BND prefix when the `JMP` instruction is converted to the `SHORT` form. This should be extremely rare since the short `JMP` only is applicable to jumps inside the same module, but if it is legitimate, it may be necessary to use `bnd jmp dword`.

Enabled by default.

- `prefix-hle`: invalid HLE prefix

`prefix-hle` warns about invalid use of the HLE `XACQUIRE` or `XRELEASE` prefixes.

Enabled by default.

- `prefix-lock`: all `prefix-lock-` warnings

`prefix-lock` is a group alias for all warning classes prefixed by `prefix-lock-`; currently `prefix-lock-error` and `prefix-lock-xchg`.

- `prefix-lock-error`: LOCK prefix on unlockable instruction

`prefix-lock-error` warns about LOCK prefixes on unlockable instructions.

Enabled by default.

- `prefix-lock-xchg`: superfluous LOCK prefix on XCHG instruction

`prefix-lock-xchg` warns about a LOCK prefix added to an XCHG instruction. The XCHG instruction is *always* locking, and so this prefix is not necessary; however, NASM will generate it if explicitly provided by the user, so this warning indicates that suboptimal code is being generated.

Enabled by default.

- `prefix-opsize`: invalid operand size prefix

`prefix-opsize` warns that an operand prefix (`o16`, `o32`, `o64`, `osp`) invalid for the specified instruction has been specified. The operand prefix will be ignored by the assembler.

Enabled by default.

- `prefix-seg`: segment prefix ignored in 64-bit mode

`prefix-seg` warns that an `es`, `cs`, `ss` or `ds` segment override prefix has no effect in 64-bit mode. The prefix will still be generated as requested.

Enabled by default.

- `ptr`: non-NASM keyword used in other assemblers

`ptr` warns about keywords used in other assemblers that might indicate a mistake in the source code. Currently only the MASM `PTR` keyword is recognized. If (limited) MASM compatibility is desired, the `%use masm` macro package is available, see section 6.5; however, carefully note the caveats listed.

Enabled by default.

- `regsize`: register size specification ignored

`regsize` warns about a register with implicit size (such as `EAX`, which is always 32 bits) been given an explicit size specification which is inconsistent with the size of the named register, e.g. `WORD EAX`. `DWORD EAX` or `WORD AX` are permitted, and do not trigger this warning. Some registers which *do not* imply a specific size, such as `K0`, may need this specification unless the instruction itself implies the instruction size:

```
KMOVW K0, [foo]      ; Permitted, KMOVW implies 16 bits
KMOV  WORD K0, [foo] ; Permitted, WORD K0 specifies instruction size
KMOV  K0, WORD [foo] ; Permitted, WORD [foo] specifies instruction size
KMOV  K0, [foo]      ; Not permitted, instruction size ambiguous
```

Enabled by default.

- `reloc`: all `reloc`- warnings

`reloc` is a group alias for all warning classes prefixed by `reloc-`; currently `reloc-abs-byte`, `reloc-abs-dword`, `reloc-abs-qword`, `reloc-abs-word`, `reloc-rel-byte`, `reloc-rel-dword`, `reloc-rel-qword`, `reloc-rel-word`.

- `reloc-abs`: all `reloc-abs`- warnings

`reloc-abs` is a group alias for all warning classes prefixed by `reloc-abs-`; currently `reloc-abs-byte`, `reloc-abs-dword`, `reloc-abs-qword`, `reloc-abs-word`.

- `reloc-abs-byte`: 8-bit absolute section-crossing relocation

`reloc-abs-byte` warns that an 8-bit absolute relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- `reloc-abs-dword`: 32-bit absolute section-crossing relocation

`reloc-abs-dword` warns that a 32-bit absolute relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- `reloc-abs-qword`: 64-bit absolute section-crossing relocation

`reloc-abs-qword` warns that a 64-bit absolute relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- `reloc-abs-word`: 16-bit absolute section-crossing relocation

`reloc-abs-word` warns that a 16-bit absolute relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- `reloc-rel`: all `reloc-rel`- warnings

`reloc-rel` is a group alias for all warning classes prefixed by `reloc-rel-`; currently `reloc-rel-byte`, `reloc-rel-dword`, `reloc-rel-qword`, `reloc-rel-word`.

- `reloc-rel-byte`: 8-bit relative section-crossing relocation

reloc-rel-byte warns that an 8-bit relative relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- reloc-rel-dword: 32-bit relative section-crossing relocation

reloc-rel-dword warns that a 32-bit relative relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- reloc-rel-qword: 64-bit relative section-crossing relocation

reloc-rel-qword warns that an 64-bit relative relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- reloc-rel-word: 16-bit relative section-crossing relocation

reloc-rel-word warns that a 16-bit relative relocation that could not be resolved at assembly time was generated in the output format. This is usually normal, but may not be handled by all possible target environments

Disabled by default.

- unknown-pragma: unknown %pragma facility or directive

unknown-pragma is a backwards compatibility alias for pragma-unknown.

- unknown-warning: unknown warning in -w/-W or warning directive

unknown-warning warns about a -w or -W option or a [WARNING] directive that contains an unknown warning name or is otherwise not possible to process.

Disabled by default.

- user:%warning directives

user controls output of %warning directives (see section 4.11).

Enabled by default.

- warn-stack-empty: warning stack empty

warn-stack-empty a [WARNING POP] directive was executed when the warning stack is empty. This is treated as a [WARNING \*all] directive.

Enabled by default.

- zeroing: RESX in initialized section becomes zero

zeroing a RESX directive was used in a section which contains initialized data, and the output format does not support this. Instead, this will be replaced with explicit zero content, which may produce a large output file.

Enabled by default.

- zext-reloc: relocation zero-extended to match output format

zext-reloc warns that a relocation has been zero-extended due to limitations in the output format.

Enabled by default.





# Appendix B: Ndisasm

---

The Netwide Disassembler, NDISASM

## B.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.

## B.2 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.

### B.2.1 Specifying the Input 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.

### B.2.2 Code Following Data: Synchronization

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 ‘synchronization’ point, or indeed as many synchronization 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.

### B.2.3 Mixed Code and Data: Automatic (Intelligent) Synchronization

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.

### B.2.4 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.



# Appendix C: NASM Version History

---

## C.1 NASM 2 Series

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

### C.1.1 Version 2.16.03

*This is a source build machinery and documentation update only. There are no functionality changes.*

- Fix building from git in a separate directory from the source.
- Remove some irrelevant files from the source distribution.
- Make the documentation stronger that -00 or -01 are probably not what the user wants. See section 2.1.24.
- Fix configure --enable-lto build option.
- Update the included RPM .spec file.

### C.1.2 Version 2.16.02

- Fix building from the source distribution in a separate directory from the source.
- Fix a number of issues when building from source, mostly involving configure or dependency generation.

In particular, more aggressively avoid cross-compilation problems on Unix/Linux systems automatically invoking WINE. We could end up invoking WINE even when we didn't want to, making configure think it was running native when in fact cross-compiling.

- Hopefully fix compiling with the latest versions of MSVC/nmake.
- Windows host: add embedded manifest file. Without a manifest, Windows applications force a fixed PATH\_MAX limit to any pathname; this is unnecessary.
- Add support VEX-encoded SM4-NI instructions.
- Add support for VEX-encoded SM3-NI instructions.
- Add support for VEX-encoded SHA512-NI instructions.
- PTWRITE opcode corrected (F3 prefix required.)
- Disassembler: the SMAP instructions are NP; notably the prefixed versions of CLAC are ERETU/ERETS.
- Add support for Flexible Return and Exception Delivery (FRED): the LKGS, ERETS and ERETU instructions.
- Fix external references to segments in the obj (OMF) and possibly other output formats.
- Always support up to 8 characters, i.e. 64 bits, in a string-to-numeric conversion.
- Preprocessor: add %map( ) function to expand a macro from a list of arguments, see section 4.4.7.
- Preprocessor: allow the user to specify the desired radix for an evaluated parameter. It doesn't make any direct difference, but can be nice for debugging or turning into strings. See the = modifier in section 4.2.1.
- Update documentation: \_\_USE\_package\_\_ is now \_\_?USE\_package?\_\_.

- Documentation: correct a minor problem in the expression grammar for `dx` statements, see section 3.2.1.
- Preprocessor: correctly handle empty `%rep` blocks.
- Preprocessor: add options for a base prefix to `%num()`, see section 4.4.8.
- Preprocessor: add a `%hex()` function, equivalent to `%eval()` except that it produces hexadecimal values that are nevertheless valid NASM numeric constants, see section 4.4.5.
- Preprocessor: fix the parameter number in error messages (should be 1-based, like `%num` references to multi-line macro arguments.)
- Documentation: be more clear than the `bin` format is simply a linker built into NASM. See section 8.1.
- Adjust the `LOCK` prefix warning for `XCHG`.  
`LOCK XCHG reg, mem` would issue a warning for being unlockable, which is incorrect. In this case the `reg, mem` encoding is simply an alias for the `mem, reg` encoding. However, `XCHG` is *always* locked, so create a new warning (`-w+prefix-lock-xchg`) to explicitly flag a user-specified `LOCK XCHG`; default off. Future versions of NASM may remove the `LOCK` prefix when optimization is enabled.
- Fix broken dependency-list generation.
- Add optional warnings for specific relocation types (`-w+reloc-*`, see appendix A), default off.  
 Some target environments may have specific restrictions on what kinds of relocations are possible or allowed.
- Error out on certain bad syntax in `dx` statements, such as `db 1 2`. See section 3.2.1.

### C.1.3 Version 2.16.01

*This is a documentation update release only. There are no functionality changes.*

- Fix the creation of the table of contents in the HTML version of the documentation.

### C.1.4 Version 2.16

- Support for the `rdi` format has been discontinued and all the `RDOFF` utilities has been removed.
- The `--reproducible` option now leaves the filename field in the COFF object format blank. This was always rather useless since it is only 18 characters long; as such debug formats have to carry their own filename information anyway.
- Fix handling of MASM-syntax reserved memory (e.g. `dw ?`) when used in structure definitions.
- The preprocessor now supports functions, which can be less verbose and more convenient than the equivalent code implemented using directives. See section 4.4.
- Fix the handling of `%00` in the preprocessor.
- Fix incorrect handling of path names affecting error messages, dependency generation, and debug format output.
- Support for the `RDOFF` output format and the `RDOFF` tools have been removed. The `RDOFF` tools had already been broken since at least NASM 2.14. For flat code the ELF output format recommended; for segmented code the `obj` (OMF) output format.
- New facility: preprocessor functions. Preprocessor functions, which are expanded similarly to single-line macros, can greatly simplify code that in the past would have required a lengthy list of directives and intermediate macros. See section 4.4.
- Single-line macros can now declare parameters (using a `&&` prefix) that creates a quoted string, but does *not* quote an already quoted string. See section 4.2.1.

- Instruction table updated per public information available as of November 2022.
- All warnings in the preprocessor have now been assigned warning classes. See appendix A.
- Fix the invalid use of RELA-type relocations instead of REL-type relocations when generating DWARF debug information for the `elf32` output format.
- Fix the handling at in `istruc` when the structure contains local labels. See section 5.9.2.
- When assembling with `--reproducible`, don't encode the filename in the COFF header for the `coff`, `win32` or `win64` output formats. The COFF header only has space for an 18-character filename, which makes this field rather useless in the first place. Debug output data, if enabled, is not affected.
- Fix incorrect size calculation when using MASM syntax for non-byte reservations (e.g. `dw ?.`)
- Allow forcing an instruction in 64-bit mode to have a (possibly redundant) REX prefix, using the syntax `{rex}` as a prefix.
- Add a `{vex}` prefix to enforce VEX (AVX) encoding of an instruction, either using the 2- or 3-byte VEX prefixes.
- The CPU directive has been augmented to allow control of generation of VEX (AVX) versus EVEX (AVX-512) instruction formats, see section 7.11.
- Some recent instructions that previously have been only available using EVEX encodings are now also encodable using VEX (AVX) encodings. For backwards compatibility these encodings are not enabled by default, but can be generated either via an explicit `{vex}` prefix or by specifying either CPU `LATEVEX` or CPU `NOEVEX`; see section 7.11.
- Document the already existing `%unimacro` directive. See section 4.5.12.
- Fix a code range generation bug in the DWARF debug format (incorrect information in the `DW_AT_high_pc` field) for the ELF output formats. This bug happened to cancel out with a bug in older versions of the GNU binutils linker, but breaks with other linkers and updated or other linkers that expect the spec to be followed.
- Fix segment symbols with addends, e.g. `jmp _TEXT+10h:0` in output formats that support segment relocations, e.g. the `obj` format.
- Fix various crashes and hangs on invalid input.

### C.1.5 Version 2.15.05

- Fix `%ifid $` and `%ifid $$` incorrectly being treated as true. See section 4.6.6.
- Add `--reproducible` option to suppress NASM version numbers and timestamps in output files. See section 2.1.34.

### C.1.6 Version 2.15.04

- More sensible handling of the case where one single-line macro definition will shadow another. A warning will be issued, but the additional definition will be allowed. For the existing error case where both a parameterless and parametered macro are created, that warning is promoted to an error by default.
- Add special preprocessor tokens `%*?` and `%*??` that expand like `%?` and `%??` in single-line macros only. See section 4.2.6.
- Correct the encoding of the `ENQCMD5` and `TILELOADT1` instructions.
- Fix case where the COFF backend (the `coff`, `win32` and `win64` output formats) would add padding bytes in the middle of a section if a `SECTION/SEGMENT` directive was provided which repeated an `ALIGN=` attribute. This neither matched legacy behavior, other backends, or user expectations.

- Fix SSE instructions not being recognized with an explicit memory operation size (e.g. `movsd qword [eax],xmm0`).
- The `-L+` option no longer enables `-Lw`, which is mainly useful to debug NASM crashes. See section 2.1.4.
- Document long-standing hazards in the use of `$` in `dx` statements, see section 3.2.1.
- The NASM-only RDOFF output format backend, which has been broken since at least NASM 2.14, has been disabled. The RDOFF tools are scheduled to be removed from the NASM distribution in NASM 2.16. If you have a concrete use case for RDOFF, please file a NASM bug report at <https://bugs.nasm.us/> as soon as possible.

### C.1.7 Version 2.15.03

- Add instructions from the Intel Instruction Set Extensions and Future Features Programming Reference, June 2020. This includes AVX512 `bfloat16`, AVX512 mask intersect, and Intel Advanced Matrix Extensions (AMX).
- Support for `bfloat16` floating-point constants. See section 3.4.6 and section 6.3.
- Properly display warnings in preprocess-only mode.
- Fix copy-and-paste of examples from the PDF documentation.
- Debug information now properly reflect the line numbers of macro invocations (unless declared `.nolist`).
- Fix excessive alignment of sections in the `coff/win32/win64` output formats when the user-specified alignment is less than the default alignment for the section or section type.
- Fix explicit token pasting (`%+`, section 4.2.4) for the cases where one or more parts result from empty token expansion, resulting in `%+` tokens at the beginning or end, or multiple ones in a row.
- Fix macro label capture (`%00`, section 4.5.7).
- Much better documentation for the MASM compatibility package, `%use masm` (see section 6.5).
- Fix LEA without square brackets, for MASM compatibility.
- Portability fixes.

### C.1.8 Version 2.15.02

- Fix miscompilation when building with `c lang`.
- Add `db-empty` warning class, see section 2.1.26.
- Fix the dependencies in the MSVC NMAKE makefile (`Mkfiles/msvc.mak`).
- Some documentation improvements and cleanups.
- Fix the handling of macro parameter ranges (`%{:}`), including with brace-enclosed original arguments.

### C.1.9 Version 2.15.01

- Fix building the documentation from the release archive. For 2.15, the user has to do `make warnings` manually in the main directory in order to be able to build the documentation, which means Perl needs to be installed on the system.
- Add instructions for Intel Control Flow Enforcement Technology (CET).



### C.1.10 Version 2.15

- The comparison and booleanizing operators can now be used in any expression context, not just `%if`. See section 3.5.
- New operator `? ... :`. See section 3.5.1.
- Signed shift operators `<<<` and `>>>`. See section 3.5.9.
- The MASM `DUP` syntax for data definitions is now supported, in a somewhat enhanced form. See section 3.2.1.
- Warn for strange legacy behavior regarding empty arguments in multi-line macro expansion, but try to match legacy behavior in most cases. Legacy behavior can be disabled with the directive `%pragma preproc sane_empty_expansion`, see section 4.5 and section 4.12.1.
- A much more sensible limit to expression evaluation depth. The previously defined limit would rarely trigger before NASM died with a stack overrun error on most systems. See section 2.1.31.
- The state of warnings can now be saved and restored via the `[WARNING PUSH]` and `[WARNING POP]` directives. See section 7.13.
- The `sectalign on|off` switch does not affect an explicit directive. See section 5.10.2.
- Added `configure` option to enable building with profiling (`--enable-profiling`).
- Attempt to support of long path names, up to 32767 of UTF-16 characters, on Windows.
- Fixed 'mismatch in operand sizes' error in the `MOVDDUP`, `CMPXCHG8B` and `CMPXCHG16B` instructions.
- Improved error messages in the string transformation routine.
- Removed obsolete `gnu-elf-extensions` warning about 8- and 16-bit relocation generation. See section 8.9.8
- Added group aliases for all prefixed warnings. See section 2.1.26.
- Allowed building with MSVC versions older than 1700.
- Added implicitly sized versions of the `k...` instructions, which allows the `k...` instructions to be specified without a size suffix as long as the operands are sized.
- Added `-L` option for additional listing information. See section 2.1.4.
- Added some warnings for obsolete instructions for a specified CPU.
- Deprecated `-hf` and `-y` options. Use `-h` instead.
- Made DWARF as the default debug format for ELF.
- Added `%pragma list options...` to set or clear listing options (see `opt -L`).
- Allowed immediate syntax for `LEA` instruction (ignore operand size completely).
- Added limited functionality MASM compatibility package. See section 6.5.
- Add single-line macros aliases using `%defalias` or `%idefalias`. These behave like a kind of "symbolic links" for single-line macros. See section 4.2.11 and `clear`.
- Added support for `stringify`, `nostrip`, `evaluating`, and `greedy` single-line macro arguments. See section 4.2.1.
- Unused single-line macro arguments no longer need to have a specified name. See section 4.2.1.
- Added conditional comma operator `%,`. See section 4.2.12.

- Changed private namespace from `__foo__` to `__?foo?__`, so a user namespace starting from underscore is now clean from symbols. For backwards compatibility, the previous names are defined as aliases; see section 4.2.11, section 4.13.3 and chapter 5.
- Added support of ELF weak symbols and external references. See section 8.9.5.
- Changed the behavior of the `EXTERN` keyword and introduced `REQUIRED` keyword. See section 7.6.
- Added `%ifusable` and `%ifusing` directives. See chapter 6.
- Made various performance improvements and stability fixes in macro preprocessor engine.
- Improved NASM error handling and cleaned up error messages.
- Many, many bug fixes.

#### C.1.11 Version 2.14.03

- Suppress nuisance "label changed during code generation" messages after a real error.
- Add support for the `merge` and `strings` attributes on ELF sections. See section 8.9.2.
- Add support for the `note`, `preinit_array`, `init_array`, and `fini_array` sections type in ELF. See section 8.9.2.
- Handle more than 32,633 sections in ELF.

#### C.1.12 Version 2.14.02

- Fix crash due to multiple errors or warnings during the code generation pass if a list file is specified.

#### C.1.13 Version 2.14.01

- Create all system-defined macros before processing command-line given preprocessing directives (`-p`, `-d`, `-u`, `--pragma`, `--before`).
- If debugging is enabled, define a `__DEBUG_FORMAT__` predefined macro. See section 5.5.
- Fix an assert for the case in the `obj` format when a `SEG` operator refers to an `EXTERN` symbol declared further down in the code.
- Fix a corner case in the floating-point code where a binary, octal or hexadecimal floating-point having at least 32, 11, or 8 mantissa digits could produce slightly incorrect results under very specific conditions.
- Support `-MD` without a filename, for `gcc` compatibility. `-MF` can be used to set the dependencies output filename. See section 2.1.8.
- Fix `-E` in combination with `-MD`. See section 2.1.22.
- Fix missing errors on redefined labels; would cause convergence failure instead which is very slow and not easy to debug.
- Duplicate definitions of the same label *with the same value* is now explicitly permitted (2.14 would allow it in some circumstances.)
- Add the option `--no-line` to ignore `%line` directives in the source. See section 2.1.33 and section 4.13.1.

#### C.1.14 Version 2.14

- Changed `-I` option semantics by adding a trailing path separator unconditionally.
- Fixed null dereference in corrupted invalid single line macros.
- Fixed division by zero which may happen if source code is malformed.

- Fixed out of bound access in processing of malformed segment override.
- Fixed out of bound access in certain EQU parsing.
- Fixed buffer underflow in float parsing.
- Added SGX (Intel Software Guard Extensions) instructions.
- Added +n syntax for multiple contiguous registers.
- Fixed subsections\_via\_symbols for macho object format.
- Added the --gprefix, --gpostfix, --lprefix, and --lpostfix command line options, to allow command line base symbol renaming. See section 2.1.28.
- Allow label renaming to be specified by %pragma in addition to from the command line. See section 7.10.
- Supported generic %pragma namespaces, output and debug. See section 4.12.
- Added the --pragma command line option to inject a %pragma directive. See section 2.1.29.
- Added the --before command line option to accept preprocess statement before input. See section 2.1.30.
- Added AVX512 VBMI2 (Additional Bit Manipulation), VNNI (Vector Neural Network), BITALG (Bit Algorithm), and GFNI (Galois Field New Instruction) instructions.
- Added the STATIC directive for local symbols that should be renamed using global-symbol rules. See section 7.9.
- Allow a symbol to be defined as EXTERN and then later overridden as GLOBAL or COMMON. Furthermore, a symbol declared EXTERN and then defined will be treated as GLOBAL. See section 7.5.
- The GLOBAL directive no longer is required to precede the definition of the symbol.
- Support private\_extern as macho specific extension to the GLOBAL directive. See section 8.8.5.
- Updated UD0 encoding to match with the specification
- Added the --limit-x command line option to set execution limits. See section 2.1.31.
- Updated the Codeview version number to be aligned with MASM.
- Added the --keep-all command line option to preserve output files. See section 2.1.32.
- Added the --include command line option, an alias to -P (section 2.1.19).
- Added the --help command line option as an alias to -h (section 3.1).
- Added -w, -D, and -Q suffix aliases for RET instructions so the operand sizes of these instructions can be encoded without using o16, o32 or o64.

### C.1.15 Version 2.13.03

- Added AVX and AVX512 VAES\* and VPCLMULQDQ instructions.
- Fixed missing dwarf record in x32 ELF output format.

### C.1.16 Version 2.13.02

- Fix false positive in testing of numeric overflows.
- Fix generation of PEXTRW instruction.
- Fix smartalign package which could trigger an error during optimization if the alignment code expanded too much due to optimization of the previous code.

- Fix a case where negative value in `TIMES` directive causes panic instead of an error.
- Always finalize `.debug_abbrev` section with a null in `dwarf` output format.
- Support debug flag in section attributes for `macho` output format. See section 8.8.1.
- Support up to 16 characters in section names for `macho` output format.
- Fix missing update of global `BITS` setting if `SECTION` directive specified a bit size using output format-specific extensions (e.g. `USE32` for the `obj` output format.)
- Fix the incorrect generation of VEX-encoded instruction when static mode decorators are specified on scalar instructions, losing the decorators as they require EVEX encoding.
- Option `-MW` to quote dependency outputs according to Watcom Make conventions instead of POSIX Make conventions. See section 2.1.12.
- The `obj` output format now contains embedded dependency file information, unless disabled with `%pragma obj nodepend`. See section 8.4.9.
- Fix generation of dependency lists.
- Fix a number of null pointer reference and memory allocation errors.
- Always generate symbol-relative relocations for the `macho64` output format; at least some versions of the XCode/LLVM linker fails for section-relative relocations.

#### C.1.17 Version 2.13.01

- Fix incorrect output for some types of `FAR` or `SEG` references in the `obj` output format, and possibly other 16-bit output formats.
- Fix the address in the list file for an instruction containing a `TIMES` directive.
- Fix error with `TIMES` used together with an instruction which can vary in size, e.g. `JMP`.
- Fix breakage on some uses of the `DZ` pseudo-op.

#### C.1.18 Version 2.13

- Support the official forms of the `UD0` and `UD1` instructions.
- Allow self-segment-relative expressions in immediates and displacements, even when combined with an external or otherwise out-of-segment special symbol, e.g.:
 

```
extern foo
mov eax,[foo - $ + ebx]           ; Now legal
```
- Handle a 64-bit origin in `NDISASM`.
- NASM can now generate sparse output files for relevant output formats, if the underlying operating system supports them.
- The `macho` object format now supports the `subsections_via_symbols` and `no_dead_strip` directives, see section 8.8.4.
- The `macho` object format now supports the `no_dead_strip`, `live_support` and `strip_static_syms` section flags, see section 8.8.1.
- The `macho` object format now supports the `dwarf` debugging format, as required by newer toolchains.
- All warnings can now be suppressed if desired; warnings not otherwise part of any warning class are now considered its own warning class called `other` (e.g. `-w-other`). Furthermore, `warning-as-error` can now be controlled on a per warning class basis, using the syntax `-w+error=warning-class` and its equivalent for all other warning control options. See section 2.1.26 for the command-line options and warning classes and section 7.13 for the `[WARNING]` directive.

- Fix a number of bugs related to AVX-512 decorators.
- Significant improvements to building NASM with Microsoft Visual Studio via `Mkfiles/msvc.mak`. It is now possible to build the full Windows installer binary as long as the necessary prerequisites are installed; see `Mkfiles/README`
- To build NASM with custom modifications (table changes) or from the git tree now requires Perl 5.8 at the very minimum, quite possibly a higher version (Perl 5.24.1 tested.) There is no requirement to have Perl on your system at all if all you want to do is build unmodified NASM from source archives.
- Fix the `{z}` decorator on AVX-512 `vmovdq*` instructions.
- Add new warnings for certain dangerous constructs which never ought to have been allowed. In particular, the `RESx` family of instructions should have been taking a critical expression all along.
- Fix the EVEX (AVX-512) versions of the `VPBROADCAST`, `VPEXTR`, and `VPINSR` instructions.
- Support contracted forms of additional instructions. As a general rule, if an instruction has a non-destructive source immediately after a destination register that isn't used as an input, NASM supports omitting that source register, using the destination register as that value. This among other things makes it easier to convert SSE code to the equivalent AVX code:
 

```
addps xmm1, xmm0           ; SSE instruction
vaddps ymm1, ymm1, ymm0    ; AVX official long form
vaddps ymm1, ymm0          ; AVX contracted form
```
- Fix Codeview malformed compiler version record.
- Add the `CLWB` and `PCOMMIT` instructions. Note that the `PCOMMIT` instruction has been deprecated and will never be included in a shipping product; it is included for completeness only.
- Add the `%pragma` preprocessor directive for soft-error directives.
- Add the `RDPID` instruction.

### C.1.19 Version 2.12.02

- Fix preprocessor errors, especially `%error` and `%warning`, inside `%if` statements.
- Fix relative relocations in 32-bit Mach-O.
- More Codeview debug format fixes.
- If the MASM `PTR` keyword is encountered, issue a warning. This is much more likely to indicate a MASM-ism encountered in NASM than it is a valid label. This warning can be suppressed with `-w-ptr`, the `[warning]` directive (see section 2.1.26) or by the macro definition `%define ptr $%?` (see section 4.2.5).
- When an error or a warning comes from the expansion of a multi-line macro, display the file and line numbers for the expanded macros. Macros defined with `.nolist` do not get displayed.
- Add macros `ilog2fw()` and `ilog2cw()` to the `ifunc` macro package. See section 6.4.1.

### C.1.20 Version 2.12.01

- Portability fixes for some platforms.
- Fix error when not specifying a list file.
- Correct the handling of macro-local labels in the Codeview debugging format.
- Add `CLZERO`, `MONITORX` and `MWAITX` instructions.

### C.1.21 Version 2.12

- Major fixes to the macho backend (section 8.8); earlier versions would produce invalid symbols and relocations on a regular basis.
- Support for thread-local storage in Mach-O.
- Support for arbitrary sections in Mach-O.
- Fix wrong negative size treated as a big positive value passed into backend causing NASM to crash.
- Fix handling of zero-extending unsigned relocations, we have been printing wrong message and forgot to assign segment with predefined value before passing it into output format.
- Fix potential write of oversized (with size greater than allowed in output format) relative relocations.
- Portability fixes for building NASM with the LLVM compiler.
- Add support of Codeview version 8 (cv8) debug format for win32 and win64 formats in the COFF backend, see section 8.5.3.
- Allow 64-bit outputs in 16/32-bit only backends. Unsigned 64-bit relocations are zero-extended from 32-bits with a warning (suppressible via `-w-zext-reloc`); signed 64-bit relocations are an error.
- Line numbers in list files now correspond to the lines in the source files, instead of simply being sequential.
- There is now an official 64-bit (x64 a.k.a. x86-64) build for Windows.

### C.1.22 Version 2.11.09

- Fix potential stack overwrite in macho32 backend.
- Fix relocation records in macho64 backend.
- Fix symbol lookup computation in macho64 backend.
- Adjust `.symtab` and `.rela.text` sections alignments to 8 bytes in elf64 backed.
- Fix section length computation in bin backend which leded in incorrect relocation records.

### C.1.23 Version 2.11.08

- Fix section length computation in bin backend which leded in incorrect relocation records.
- Add a warning for numeric preprocessor definitions passed via command line which might have unexpected results otherwise.
- Add ability to specify a module name record in rdoft linker with `-mn` option.
- Increase label length capacity up to 256 bytes in rdoft backend for FreePascal sake, which tends to generate very long labels for procedures.
- Fix segmentation failure when rip addressing is used in macho64 backend.
- Fix access on out of memory when handling strings with a single grave. We have sixed similar problem in previous release but not all cases were covered.
- Fix NULL dereference in disassembled on BND instruction.

### C.1.24 Version 2.11.07

- Fix 256 bit VMOVNTPS instruction.
- Fix `-MD` option handling, which was rather broken in previous release changing command line api.
- Fix access to uninitialized space when handling strings with a single grave.

- Fix nil dereference in handling memory reference parsing.

#### C.1.25 Version 2.11.06

- Update AVX512 instructions based on the Extension Reference (319433-021 Sept 2014).
- Fix the behavior of -MF and -MD options (Bugzilla 3392280)
- Updated Win32 Makefile to fix issue with build

#### C.1.26 Version 2.11.05

- Add --v as an alias for -v (see section 2.1.27), for command-line compatibility with Yasm.
- Fix a bug introduced in 2.11.03 whereby certain instructions would contain multiple REX prefixes, and thus be corrupt.

#### C.1.27 Version 2.11.04

- Removed an invalid error checking code. Sometimes a memref only with a displacement can also set an evex flag. For example:

```
vmovdqu32 [0xabcd]{k1}, zmm0
```

- Fixed a bug in disassembler that EVEX.L'L vector length was not matched when EVEX.b was set because it was simply considered as EVEC.RC. Separated EVEX.L'L case from EVEX.RC which is ignored in matching.

#### C.1.28 Version 2.11.03

- Fix a bug there REX prefixes were missing on instructions inside a TIMES statement.

#### C.1.29 Version 2.11.02

- Add the XSAVEC, XSAVES and XRSTORS family instructions.
- Add the CLFLUSHOPT instruction.

#### C.1.30 Version 2.11.01

- Allow instructions which implicitly uses XMM0 (VBLENDVPD, VBLENDVPS, PBLENDVB and SHA256RND\$2) to be specified without an explicit xmm0 on the assembly line. In other words, the following two lines produce the same output:

```
vblendvpd xmm2,xmm1,xmm0      ; Last operand is fixed xmm0
vblendvpd xmm2,xmm1           ; Implicit xmm0 omitted
```

- In the ELF backends, don't crash the assembler if section align is specified without a value.

#### C.1.31 Version 2.11

- Add support for the Intel AVX-512 instruction set:
- 16 new, 512-bit SIMD registers. Total 32 (ZMM0 ~ ZMM31)
- 8 new opmask registers (K0 ~ K7). One of 7 registers (K1 ~ K7) can be used as an opmask for conditional execution.
- A new EVEX encoding prefix. EVEX is based on VEX and provides more capabilities: opmasks, broadcasting, embedded rounding and compressed displacements.
- opmask
 

```
VDIVPD zmm0{k1}{z}, zmm1, zmm3 ; conditional vector operation
                                ; using opmask k1.
                                ; {z} is for zero-masking
```
- broadcasting
 

```
VDIVPS zmm4, zmm5, [rbx]{1to16} ; load single-precision float and
```

```

; replicate it 16 times. 32 * 16 = 512
- embedded rounding
  VCVTSI2SD xmm6, xmm7, {rz-sae}, rax ; round toward zero. note that it
; is used as if a separate operand.
; it comes after the last SIMD operand

```

- Add support for ZWORD (512 bits), DZ and RESZ.
- Add support for the MPX and SHA instruction sets.
- Better handling of section redefinition.
- Generate manpages when running 'make dist'.
- Handle all token chains in mmacro params range.
- Support split [base,index] effective address:

```
mov eax,[eax+8,ecx*4] ; eax=base, ecx=index, 4=scale, 8=disp
```

This is expected to be most useful for the MPX instructions.

- Support BND prefix for branch instructions (for MPX).
- The DEFAULT directive can now take BND and NOBND options to indicate whether all relevant branches should be getting BND prefixes. This is expected to be the normal for use in MPX code.
- Add {evex}, {vex3} and {vex2} instruction prefixes to have NASM encode the corresponding instruction, if possible, with an EVEX, 3-byte VEX, or 2-byte VEX prefix, respectively.
- Support for section names longer than 8 bytes in Win32/Win64 COFF.
- The NOSPLIT directive by itself no longer forces a single register to become an index register, unless it has an explicit multiplier.

```
mov eax,[nosplit eax] ; eax as base register
mov eax,[nosplit eax*1] ; eax as index register

```

### C.1.32 Version 2.10.09

- Pregenerate man pages.

### C.1.33 Version 2.10.08

- Fix VMOVNTDQA, MOVNTDQA and MOVLPD instructions.
- Fix collision for VGATHERQPS, VPGATHERQD instructions.
- Fix VPMOVSXBQ, VGATHERQPD, VSPLW instructions.
- Add a bunch of AMD TBM instructions.
- Fix potential stack overwrite in numbers conversion.
- Allow byte size in PREFETCHT<sub>x</sub> instructions.
- Make manual pages up to date.
- Make F3 and F2 SSE prefixes to override 66.
- Support of AMD SVM instructions in 32 bit mode.
- Fix near offsets code generation for JMP, CALL instructions in long mode.
- Fix preprocessor parse regression when id is expanding to a whitespace.

### C.1.34 Version 2.10.07

- Fix line continuation parsing being broken in previous version.



### C.1.35 Version 2.10.06

- Always quote the dependency source names when using the automatic dependency generation options.
- If no dependency target name is specified via the -MT or -MQ options, quote the default output name.
- Fix assembly of shift operations in CPU 8086 mode.
- Fix incorrect generation of explicit immediate byte for shift by 1 under certain circumstances.
- Fix assembly of the VPCMPGTQ instruction.
- Fix RIP-relative relocations in the macho64 backend.

### C.1.36 Version 2.10.05

- Add the CLAC and STAC instructions.

### C.1.37 Version 2.10.04

- Add back the inadvertently deleted 256-bit version of the VORPD instruction.
- Correct disassembly of instructions starting with byte 82 hex.
- Fix corner cases in token pasting, for example:

```
%define N 1e%++%+ 5
      dd N, 1e+5
```

### C.1.38 Version 2.10.03

- Correct the assembly of the instruction:

```
XRELEASE MOV [absolute],AL
```

Previous versions would incorrectly generate F3 A2 for this instruction and issue a warning; correct behavior is to emit F3 88 05.

### C.1.39 Version 2.10.02

- Add the ifunc macro package with integer functions, currently only integer logarithms. See section 6.4.
- Add the RDSEED, ADCX and ADOX instructions.

### C.1.40 Version 2.10.01

- Add missing VPMOVMskb instruction with reg32, ymmreg operands.

### C.1.41 Version 2.10

- When optimization is enabled, `mov r64, imm` now optimizes to the shortest form possible between:

```
mov r32,imm32          ; 5 bytes
mov r64,imm32          ; 7 bytes
mov r64,imm64          ; 10 bytes
```

To force a specific form, use the `STRICT` keyword, see section 3.7.

- Add support for the Intel AVX2 instruction set.
- Add support for Bit Manipulation Instructions 1 and 2.
- Add support for Intel Transactional Synchronization Extensions (TSX).
- Add support for x32 ELF (32-bit ELF with the CPU in 64-bit mode.) See section 8.9.
- Add support for bigendian UTF-16 and UTF-32. See section 3.4.5.

#### **C.1.42 Version 2.09.10**

- Fix up NSIS script to protect uninstaller against registry keys absence or corruption. It brings in a few additional questions to a user during deinstallation procedure but still it is better than unpredictable file removal.

#### **C.1.43 Version 2.09.09**

- Fix initialization of section attributes of `bin` output format.
- Fix `mach64` output format bug that crashes NASM due to NULL symbols.

#### **C.1.44 Version 2.09.08**

- Fix `__OUTPUT_FORMAT__` assignment when output driver alias is used. For example when `-f elf` is used `__OUTPUT_FORMAT__` must be set to `elf`, if `-f elf32` is used `__OUTPUT_FORMAT__` must be assigned accordingly, i.e. to `elf32`. The rule applies to all output driver aliases. See section 5.4.

#### **C.1.45 Version 2.09.07**

- Fix attempts to close same file several times when `-a` option is used.
- Fixes for `VEXTRACTF128`, `VMASKMOVPS` encoding.

#### **C.1.46 Version 2.09.06**

- Fix missed section attribute initialization in `bin` output target.

#### **C.1.47 Version 2.09.05**

- Fix arguments encoding for `VPEXTRW` instruction.
- Remove invalid form of `VPEXTRW` instruction.
- Add `VLDQQU` as alias for `VLDQQU` to match specification.

#### **C.1.48 Version 2.09.04**

- Fix incorrect labels offset for VEX instructions.
- Eliminate bogus warning on implicit operand size override.
- `%if` term could not handle 64 bit numbers.
- The COFF backend was limiting relocations number to 16 bits even if in real there were a way more relocations.

#### **C.1.49 Version 2.09.03**

- Print `%macro` name inside `%rep` blocks on error.
- Fix preprocessor expansion behaviour. It happened sometime too early and sometime simply wrong. Move behaviour back to the origins (down to NASM 2.05.01).
- Fix uninitialized data dereference on OMF output format.
- Issue warning on unterminated `%{` construct.
- Fix for documentation typo.

#### **C.1.50 Version 2.09.02**

- Fix reversed tokens when `%def tok` produces more than one output token.
- Fix segmentation fault on disassembling some VEX instructions.
- Missing `%endif` did not always cause error.

- Fix typo in documentation.
- Compound context local preprocessor single line macro identifiers were not expanded early enough and as result lead to unresolved symbols.

#### **C.1.51 Version 2.09.01**

- Fix NULL dereference on missed `%deftok` second parameter.
- Fix NULL dereference on invalid `%substr` parameters.

#### **C.1.52 Version 2.09**

- Fixed assignment the magnitude of `%rep` counter. It is limited to 62 bits now.
- Fixed NULL dereference if argument of `%strlen` resolves to whitespace. For example if nonexistent macro parameter is used.
- `%ifenv`, `%elifenv`, `%ifnenv`, and `%elifnenv` directives introduced. See section 4.6.9.
- Fixed NULL dereference if environment variable is missed.
- Updates of new AVX v7 Intel instructions.
- `PUSH imm32` is now officially documented.
- Fix for encoding the LFS, LGS and LSS in 64-bit mode.
- Fixes for compatibility with OpenWatcom compiler and DOS 8.3 file format limitation.
- Macros parameters range expansion introduced. See section 4.5.4.
- Backward compatibility on expanding of local single macros restored.
- 8 bit relocations for `elf` and `bin` output formats are introduced.
- Short intersegment jumps are permitted now.
- An alignment more than 64 bytes are allowed for `win32`, `win64` output formats.
- `SECTALIGN` directive introduced. See section 5.10.2.
- `nojmp` option introduced in `smartalign` package. See section 6.2.
- Short aliases `win`, `elf` and `macho` for output formats are introduced. Each stands for `win32`, `elf32` and `macho32` accordingly.
- Faster handling of missing directives implemented.
- Various small improvements in documentation.
- No hang anymore if unable to open `malloc.log` file.
- The environments without `vsnprintf` function are able to build `nasm` again.
- AMD LWP instructions updated.
- Tighten EA checks. We warn a user if there overflow in EA addressing.
- Make `-ox` the default optimization level. For the legacy behavior, specify `-o0` explicitly. See section 2.1.24.
- Environment variables read with `%!`  or tested with `%ifenv` can now contain non-identifier characters if surrounded by quotes. See section 4.13.2.
- Add a new standard macro package `%use fp` for floating-point convenience macros. See section 6.3.

### C.1.53 Version 2.08.02

- Fix crash under certain circumstances when using the %+ operator.

### C.1.54 Version 2.08.01

- Fix the %use statement, which was broken in 2.08.

### C.1.55 Version 2.08

- A number of enhancements/fixes in macros area.
- Support for converting strings to tokens. See section 4.2.10.
- 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.1.56 Version 2.07

- NASM is now under the 2-clause BSD license. See section 1.1.1.
- 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 8.2 and section 8.3.
- 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 8.8.
- Fix assert failure on certain operations that involve strings with high-bit bytes.

### C.1.57 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 4.2.3.
- `%pop` can now take an argument, see section 4.9.1.
- 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 8.9.4.
- 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 `016 FSAVE` to work correctly.
- Support for structures with a non-zero base offset. See section 5.9.1.
- Correctly handle preprocessor token concatenation (see section 4.5.9) 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 8.9.2.

### C.1.58 Version 2.05.01

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

### C.1.59 Version 2.05

- Fix redundant `REX.W` prefix on `JMP reg64`.
- Make the behaviour of `-o0` match NASM 0.98 legacy behavior. See section 2.1.24.
- `-w-user` can be used to suppress the output of `%warning` directives. See section 2.1.26.
- 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.1.60 Version 2.04

- Sanitize macro handling 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 4.8.4.
- Excess default parameters to %macro now issues a warning by default. See section 4.5.
- Fix %ifn and %elifn.
- Fix nested %else clauses.
- Correct the handling of nested %reps.
- New %unmacro directive to undeclare a multi-line macro. See section 4.5.12.
- Builtin macro \_\_PASS\_\_ which expands to the current assembly pass. See section 5.8.
- \_\_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.1.61 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.1.62 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.
- SAFESSEH support for Win32, IMAGEREL for Win64 (SEH).
- %? and %?? to refer to the name of a macro itself. In particular, %ifdef 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 %ifdefstr to stringize macro definitions before creation.
- Fix forward references used in EQU statements.

### C.1.63 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.1.64 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.1.65 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 2.1.26.
- Add `-w+error` to treat warnings as errors. See section 2.1.26.
- Add `-w+all` and `-w-all` to enable or disable all suppressible warnings. See section 2.1.26.

## 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.
- -02 and -03 are no longer aliases for -010 and -015. 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 handling 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 READMExx 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 inadvertently 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**

- Rdoft 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 insensitive "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 backward 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 immediate 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"
- 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"
- 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...
- James Seter – -postfix, -prefix command line switches.
- Yuri Zaporozhets – 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 %ifdef 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\_macro 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" macro enters into the "if" context, so %\$arg1 is not valid anymore inside "if". Of course it could be worked around by using explicitly %\$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 PMOVMASKB 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/rdsr.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\_CDec1\_\_, 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,<segment>' 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,<segment>' 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 disassembled 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 COMMENT 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.

## Appendix D: Building NASM from Source

---

The source code for NASM is available from our website, <http://www.nasm.us/>, see section E.1.

### D.1 Building from a Source Archive

The source archives available on the web site should be capable of building on a number of platforms. This is the recommended method for building NASM to support platforms for which executables are not available.

On a system which has Unix shell (sh), run:

```
sh configure
make everything
```

A number of options can be passed to `configure`; see `sh configure --help`.

A set of Makefiles for some other environments are also available; please see the file `Mkfiles/README`.

To build the installer for the Windows platform, you will need the *Nullsoft Scriptable Installer*, NSIS, installed.

To build the documentation, you will need a set of additional tools. The documentation is not likely to be able to build on non-Unix systems.

### D.2 Building from the git Repository

The NASM development tree is kept in a source code repository using the `git` distributed source control system. The link is available on the website. This is recommended only to participate in the development of NASM or to assist with testing the development code.

To build NASM from the `git` repository you will need a Perl interpreter and, if building on a Unix system, GNU `autoconf` installed on your system.

To build on a Unix system, run:

```
sh autogen.sh
```

to create the `configure` script and then build as listed above.

### D.3 Building the documentation

To build the documentation, you will need a Perl interpreter, a Postscript to PDF converter such as Ghostscript, and suitable fonts installed on your system. The recommended (and default) fonts are Adobe's Source Sans and Source Code fonts, which are freely available under the SIL Open Font License.



# Appendix E: Contact Information

---

## E.1 Website

NASM has a website at <http://www.nasm.us/>.

New releases, release candidates, and daily development snapshots of NASM are available from the official web site in source form as well as binaries for a number of common platforms.

### E.1.1 User Forums

Users of NASM may find the Forums on the website useful. These are, however, not frequented much by the developers of NASM, so they are not suitable for reporting bugs.

### E.1.2 Development Community

The development of NASM is coordinated primarily through the `nasm-devel` mailing list. If you wish to participate in development of NASM, please join this mailing list. Subscription links and archives of past posts are available on the website.

## E.2 Reporting Bugs

To report bugs in NASM, please use the bug tracker at <http://www.nasm.us/> (click on "Bug Tracker"), or if that fails then through one of the contacts in section E.1.

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 13.1, and don't bother reporting the bug if it's listed there.

If you do report a bug, *please* make sure your bug report includes the following information:

- What operating system you're running NASM under. Linux, FreeBSD, NetBSD, MacOS X, Win16, Win32, Win64, MS-DOS, OS/2, VMS, whatever.
- If you compiled your own executable from a source archive, compiled your own executable from `git`, used the standard distribution binaries from the website, or got an executable from somewhere else (e.g. a Linux distribution.) 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 F: Instruction List

---

## F.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.

### F.1.1 Special instructions (pseudo-ops)

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

### F.1.2 Conventional instructions

AAA		8086, NO LONG
AAD		8086, NO LONG
AAD	imm	8086, NO LONG
AAM		8086, NO LONG
AAM	imm	8086, NO LONG
AAS		8086, NO LONG
ADC	mem, reg8	8086, LOCK
ADC	reg8, reg8	8086
ADC	mem, reg16	8086, LOCK
ADC	reg16, reg16	8086
ADC	mem, reg32	386, LOCK
ADC	reg32, reg32	386
ADC	mem, reg64	X86_64, LONG, LOCK
ADC	reg64, reg64	X86_64, LONG
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	X86_64, LONG
ADC	reg64, reg64	X86_64, LONG
ADC	rm16, imm8	8086, LOCK
ADC	rm32, imm8	386, LOCK
ADC	rm64, imm8	X86_64, LONG, LOCK
ADC	reg_al, imm	8086
ADC	reg_ax, sbytedword	8086, ND
ADC	reg_ax, imm	8086
ADC	reg_eax, sbytedword	386, ND
ADC	reg_eax, imm	386
ADC	reg_rax, sbytedword	X86_64, LONG, ND
ADC	reg_rax, imm	X86_64, LONG
ADC	rm8, imm	8086, LOCK

ADC	rm16, sbyteword	8086, LOCK, ND
ADC	rm16, imm	8086, LOCK
ADC	rm32, sbyteword	386, LOCK, ND
ADC	rm32, imm	386, LOCK
ADC	rm64, sbyteword	X86_64, LONG, LOCK, ND
ADC	rm64, imm	X86_64, LONG, LOCK
ADC	mem, imm8	8086, LOCK, ND
ADC	mem, sbyteword16	8086, LOCK, ND
ADC	mem, imm16	8086, LOCK
ADC	mem, sbyteword32	386, LOCK, ND
ADC	mem, imm32	386, LOCK
ADC	rm8, imm	8086, LOCK, ND, NOLONG
ADD	mem, reg8	8086, LOCK
ADD	reg8, reg8	8086
ADD	mem, reg16	8086, LOCK
ADD	reg16, reg16	8086
ADD	mem, reg32	386, LOCK
ADD	reg32, reg32	386
ADD	mem, reg64	X86_64, LONG, LOCK
ADD	reg64, reg64	X86_64, LONG
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	X86_64, LONG
ADD	reg64, reg64	X86_64, LONG
ADD	rm16, imm8	8086, LOCK
ADD	rm32, imm8	386, LOCK
ADD	rm64, imm8	X86_64, LONG, LOCK
ADD	reg_al, imm	8086
ADD	reg_ax, sbyteword	8086, ND
ADD	reg_ax, imm	8086
ADD	reg_eax, sbyteword	386, ND
ADD	reg_eax, imm	386
ADD	reg_rax, sbyteword	X86_64, LONG, ND
ADD	reg_rax, imm	X86_64, LONG
ADD	rm8, imm	8086, LOCK
ADD	rm16, sbyteword	8086, LOCK, ND
ADD	rm16, imm	8086, LOCK
ADD	rm32, sbyteword	386, LOCK, ND
ADD	rm32, imm	386, LOCK
ADD	rm64, sbyteword	X86_64, LONG, LOCK, ND
ADD	rm64, imm	X86_64, LONG, LOCK
ADD	mem, imm8	8086, LOCK
ADD	mem, sbyteword16	8086, LOCK, ND
ADD	mem, imm16	8086, LOCK
ADD	mem, sbyteword32	386, LOCK, ND
ADD	mem, imm32	386, LOCK
ADD	rm8, imm	8086, LOCK, ND, NOLONG
AND	mem, reg8	8086, LOCK
AND	reg8, reg8	8086
AND	mem, reg16	8086, LOCK
AND	reg16, reg16	8086
AND	mem, reg32	386, LOCK
AND	reg32, reg32	386
AND	mem, reg64	X86_64, LONG, LOCK
AND	reg64, reg64	X86_64, LONG
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	X86_64, LONG
AND	reg64, reg64	X86_64, LONG

AND	rm16, imm8	8086, LOCK
AND	rm32, imm8	386, LOCK
AND	rm64, imm8	X86_64, LONG, LOCK
AND	reg_al, imm	8086
AND	reg_ax, sbyteword	8086, ND
AND	reg_ax, imm	8086
AND	reg_eax, sbytedword	386, ND
AND	reg_eax, imm	386
AND	reg_rax, sbytedword	X86_64, LONG, ND
AND	reg_rax, imm	X86_64, LONG
AND	rm8, imm	8086, LOCK
AND	rm16, sbyteword	8086, LOCK, ND
AND	rm16, imm	8086, LOCK
AND	rm32, sbytedword	386, LOCK, ND
AND	rm32, imm	386, LOCK
AND	rm64, sbytedword	X86_64, LONG, LOCK, ND
AND	rm64, imm	X86_64, LONG, LOCK
AND	mem, imm8	8086, LOCK
AND	mem, sbyteword16	8086, LOCK, ND
AND	mem, imm16	8086, LOCK
AND	mem, sbytedword32	386, LOCK, ND
AND	mem, imm32	386, LOCK
AND	rm8, imm	8086, LOCK, ND, NOLONG
ARPL	mem, reg16	286, PROT, NOLONG
ARPL	reg16, reg16	286, PROT, NOLONG
BB0_RESET		PENT, CYRIX, ND, OBSOLETE
BB1_RESET		PENT, CYRIX, ND, OBSOLETE
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	X86_64, LONG
BSF	reg64, reg64	X86_64, LONG
BSR	reg16, mem	386
BSR	reg16, reg16	386
BSR	reg32, mem	386
BSR	reg32, reg32	386
BSR	reg64, mem	X86_64, LONG
BSR	reg64, reg64	X86_64, LONG
BSWAP	reg32	486
BSWAP	reg64	X86_64, LONG
BT	mem, reg16	386
BT	reg16, reg16	386
BT	mem, reg32	386
BT	reg32, reg32	386
BT	mem, reg64	X86_64, LONG
BT	reg64, reg64	X86_64, LONG
BT	rm16, imm8	386
BT	rm32, imm8	386
BT	rm64, imm8	X86_64, LONG
BTC	mem, reg16	386, LOCK
BTC	reg16, reg16	386
BTC	mem, reg32	386, LOCK
BTC	reg32, reg32	386
BTC	mem, reg64	X86_64, LONG, LOCK
BTC	reg64, reg64	X86_64, LONG
BTC	rm16, imm8	386, LOCK
BTC	rm32, imm8	386, LOCK
BTC	rm64, imm8	X86_64, LONG, LOCK
BTR	mem, reg16	386, LOCK
BTR	reg16, reg16	386
BTR	mem, reg32	386, LOCK
BTR	reg32, reg32	386
BTR	mem, reg64	X86_64, LONG, LOCK
BTR	reg64, reg64	X86_64, LONG

BTR	rm16, imm8	386, LOCK
BTR	rm32, imm8	386, LOCK
BTR	rm64, imm8	X86_64, LONG, LOCK
BTS	mem, reg16	386, LOCK
BTS	reg16, reg16	386
BTS	mem, reg32	386, LOCK
BTS	reg32, reg32	386
BTS	mem, reg64	X86_64, LONG, LOCK
BTS	reg64, reg64	X86_64, LONG
BTS	rm16, imm8	386, LOCK
BTS	rm32, imm8	386, LOCK
BTS	rm64, imm8	X86_64, LONG, LOCK
CALL	imm	8086, BND
CALL	imm near	8086, ND, BND
CALL	imm far	8086, ND, NOLONG
CALL	imm16	8086, NOLONG, BND
CALL	imm16 near	8086, ND, NOLONG, BND
CALL	imm16 far	8086, ND, NOLONG
CALL	imm32	386, NOLONG, BND
CALL	imm32 near	386, ND, NOLONG, BND
CALL	imm32 far	386, ND, NOLONG
CALL	imm64	X86_64, LONG, BND
CALL	imm64 near	X86_64, LONG, ND, BND
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	X86_64, LONG
CALL	mem16 far	8086
CALL	mem32 far	386
CALL	mem64 far	X86_64, LONG
CALL	mem near	8086, ND, BND
CALL	rm16 near	8086, NOLONG, ND, BND
CALL	rm32 near	386, NOLONG, ND, BND
CALL	rm64 near	X86_64, LONG, ND, BND
CALL	mem	8086, BND
CALL	rm16	8086, NOLONG, BND
CALL	rm32	386, NOLONG, BND
CALL	rm64	X86_64, LONG, BND
CBW		8086
CDQ		386
CDQE		X86_64, LONG
CLC		8086
CLD		8086
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	X86_64, LONG
CMP	reg64, reg64	X86_64, LONG
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	X86_64, LONG
CMP	reg64, reg64	X86_64, LONG
CMP	rm16, imm8	8086
CMP	rm32, imm8	386

CMP	rm64, imm8	X86_64, LONG
CMP	reg_al, imm	8086
CMP	reg_ax, sbyteword	8086, ND
CMP	reg_ax, imm	8086
CMP	reg_eax, sbytedword	386, ND
CMP	reg_eax, imm	386
CMP	reg_rax, sbytedword	X86_64, LONG, ND
CMP	reg_rax, imm	X86_64, LONG
CMP	rm8, imm	8086
CMP	rm16, sbyteword	8086, ND
CMP	rm16, imm	8086
CMP	rm32, sbytedword	386, ND
CMP	rm32, imm	386
CMP	rm64, sbytedword	X86_64, LONG, ND
CMP	rm64, imm	X86_64, LONG
CMP	mem, imm8	8086
CMP	mem, sbyteword16	8086, ND
CMP	mem, imm16	8086
CMP	mem, sbytedword32	386, ND
CMP	mem, imm32	386
CMP	rm8, imm	8086, ND, NOLONG
CMPSB		8086
CMPSD		386
CMPSQ		X86_64, LONG
CMPSW		8086
CMPXCHG	mem, reg8	PENT, LOCK
CMPXCHG	reg8, reg8	PENT
CMPXCHG	mem, reg16	PENT, LOCK
CMPXCHG	reg16, reg16	PENT
CMPXCHG	mem, reg32	PENT, LOCK
CMPXCHG	reg32, reg32	PENT
CMPXCHG	mem, reg64	X86_64, LONG, LOCK
CMPXCHG	reg64, reg64	X86_64, LONG
CMPXCHG486	mem, reg8	486, UNDOC, ND, LOCK, OBSOLETE
CMPXCHG486	reg8, reg8	486, UNDOC, ND, OBSOLETE
CMPXCHG486	mem, reg16	486, UNDOC, ND, LOCK, OBSOLETE
CMPXCHG486	reg16, reg16	486, UNDOC, ND, OBSOLETE
CMPXCHG486	mem, reg32	486, UNDOC, ND, LOCK, OBSOLETE
CMPXCHG486	reg32, reg32	486, UNDOC, ND, OBSOLETE
CMPXCHG8B	mem64	PENT, LOCK
CMPXCHG16B	mem128	X86_64, LONG, LOCK
CPUID		PENT
CPU_READ		PENT, CYRIX
CPU_WRITE		PENT, CYRIX
CQO		X86_64, LONG
CWD		8086
CWDE		386
DAA		8086, NOLONG
DAS		8086, NOLONG
DEC	reg16	8086, NOLONG
DEC	reg32	386, NOLONG
DEC	rm8	8086, LOCK
DEC	rm16	8086, LOCK
DEC	rm32	386, LOCK
DEC	rm64	X86_64, LONG, LOCK
DIV	rm8	8086
DIV	rm16	8086
DIV	rm32	386
DIV	rm64	X86_64, LONG
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, OBSOLETE
IBTS	reg16, reg16	386, UNDOC, ND, OBSOLETE
IBTS	mem, reg32	386, SD, UNDOC, ND, OBSOLETE
IBTS	reg32, reg32	386, UNDOC, ND, OBSOLETE
ICEBP		386, ND
IDIV	rm8	8086
IDIV	rm16	8086
IDIV	rm32	386
IDIV	rm64	X86_64, LONG
IMUL	rm8	8086
IMUL	rm16	8086
IMUL	rm32	386
IMUL	rm64	X86_64, LONG
IMUL	reg16, mem	386
IMUL	reg16, reg16	386
IMUL	reg32, mem	386
IMUL	reg32, reg32	386
IMUL	reg64, mem	X86_64, LONG
IMUL	reg64, reg64	X86_64, LONG
IMUL	reg16, mem, imm8	186
IMUL	reg16, mem, sbytedword	186, ND
IMUL	reg16, mem, imm16	186
IMUL	reg16, mem, imm	186, ND
IMUL	reg16, reg16, imm8	186
IMUL	reg16, reg16, sbytedword	186, ND
IMUL	reg16, reg16, imm16	186
IMUL	reg16, reg16, imm	186, ND
IMUL	reg32, mem, imm8	386
IMUL	reg32, mem, sbytedword	386, ND
IMUL	reg32, mem, imm32	386
IMUL	reg32, mem, imm	386, ND
IMUL	reg32, reg32, imm8	386
IMUL	reg32, reg32, sbytedword	386, ND
IMUL	reg32, reg32, imm32	386
IMUL	reg32, reg32, imm	386, ND
IMUL	reg64, mem, imm8	X86_64, LONG
IMUL	reg64, mem, sbytedword	X86_64, LONG, ND
IMUL	reg64, mem, imm32	X86_64, LONG
IMUL	reg64, mem, imm	X86_64, LONG, ND
IMUL	reg64, reg64, imm8	X86_64, LONG
IMUL	reg64, reg64, sbytedword	X86_64, LONG, ND
IMUL	reg64, reg64, imm32	X86_64, LONG
IMUL	reg64, reg64, imm	X86_64, LONG, ND
IMUL	reg16, imm8	186
IMUL	reg16, sbytedword	186, ND
IMUL	reg16, imm16	186
IMUL	reg16, imm	186, ND
IMUL	reg32, imm8	386
IMUL	reg32, sbytedword	386, ND
IMUL	reg32, imm32	386
IMUL	reg32, imm	386, ND
IMUL	reg64, imm8	X86_64, LONG

IMUL	reg64,sbytedword	X86_64, LONG, ND
IMUL	reg64,imm32	X86_64, LONG
IMUL	reg64,imm	X86_64, LONG, 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, LOCK
INC	rm16	8086, LOCK
INC	rm32	386, LOCK
INC	rm64	X86_64, LONG, LOCK
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
INVPCID	reg32,mem128	INVPCID, PRIV, NOLONG
INVPCID	reg64,mem128	INVPCID, PRIV, LONG
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	X86_64, LONG, AMD
INVLPGA		X86_64, AMD
IRET		8086
IRETD		386
IRETQ		X86_64, LONG
IRETW		8086
JCXZ	imm	8086, NOLONG
JECXZ	imm	386
JRCXZ	imm	X86_64, LONG
JMP	imm short	8086
JMP	imm	8086, ND
JMP	imm	8086, BND
JMP	imm near	8086, ND, BND
JMP	imm far	8086, ND, NOLONG
JMP	imm16	8086, NOLONG, BND
JMP	imm16 near	8086, ND, NOLONG, BND
JMP	imm16 far	8086, ND, NOLONG
JMP	imm32	386, NOLONG, BND
JMP	imm32 near	386, ND, NOLONG, BND
JMP	imm32 far	386, ND, NOLONG
JMP	imm64	X86_64, LONG, BND
JMP	imm64 near	X86_64, LONG, ND, BND
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	X86_64, LONG
JMP	mem16 far	8086
JMP	mem32 far	386
JMP	mem64 far	X86_64, LONG
JMP	mem near	8086, ND, BND
JMP	rm16 near	8086, NOLONG, ND, BND
JMP	rm32 near	386, NOLONG, ND, BND
JMP	rm64 near	X86_64, LONG, ND, BND
JMP	mem	8086, BND

JMP	rm16	8086, NOLONG, BND
JMP	rm32	386, NOLONG, BND
JMP	rm64	X86_64, LONG, BND
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	X86_64, LONG, PROT, ND
LAR	reg32, mem	386, PROT, SW
LAR	reg32, reg16	386, PROT
LAR	reg32, reg32	386, PROT
LAR	reg32, reg64	X86_64, LONG, PROT, ND
LAR	reg64, mem	X86_64, LONG, PROT, SW
LAR	reg64, reg16	X86_64, LONG, PROT
LAR	reg64, reg32	X86_64, LONG, PROT
LAR	reg64, reg64	X86_64, LONG, PROT
LDS	reg16, mem	8086, NOLONG
LDS	reg32, mem	386, NOLONG
LEA	reg16, mem	8086, ANYSIZE
LEA	reg32, mem	386, ANYSIZE
LEA	reg64, mem	X86_64, LONG, ANYSIZE
LEA	reg16, imm	8086, ND, ANYSIZE
LEA	reg32, imm	386, ND, ANYSIZE
LEA	reg64, imm	X86_64, LONG, ND, ANYSIZE
LEAVE		186
LES	reg16, mem	8086, NOLONG
LES	reg32, mem	386, NOLONG
LFENCE		X86_64, LONG, AMD
LFS	reg16, mem	386
LFS	reg32, mem	386
LFS	reg64, mem	X86_64, LONG
LGDT	mem	286, PRIV
LGS	reg16, mem	386
LGS	reg32, mem	386
LGS	reg64, mem	X86_64, LONG
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, ND, OBSOLETE
LOADALL286		286, UNDOC, ND, OBSOLETE
LODSB		8086
LODSD		386
LODSQ		X86_64, LONG
LODSW		8086
LOOP	imm	8086
LOOP	imm, reg_cx	8086, NOLONG
LOOP	imm, reg_ecx	386
LOOP	imm, reg_rcx	X86_64, LONG
LOOPE	imm	8086
LOOPE	imm, reg_cx	8086, NOLONG
LOOPE	imm, reg_ecx	386
LOOPE	imm, reg_rcx	X86_64, LONG
LOOPNE	imm	8086
LOOPNE	imm, reg_cx	8086, NOLONG
LOOPNE	imm, reg_ecx	386
LOOPNE	imm, reg_rcx	X86_64, LONG
LOOPNZ	imm	8086
LOOPNZ	imm, reg_cx	8086, NOLONG

LOOPNZ	imm, reg_ecx	386
LOOPNZ	imm, reg_rcx	X86_64, LONG
LOOPZ	imm	8086
LOOPZ	imm, reg_cx	8086, NOLONG
LOOPZ	imm, reg_ecx	386
LOOPZ	imm, reg_rcx	X86_64, LONG
LSL	reg16, mem	286, PROT, SW
LSL	reg16, reg16	286, PROT
LSL	reg16, reg32	386, PROT
LSL	reg16, reg64	X86_64, LONG, PROT, ND
LSL	reg32, mem	386, PROT, SW
LSL	reg32, reg16	386, PROT
LSL	reg32, reg32	386, PROT
LSL	reg32, reg64	X86_64, LONG, PROT, ND
LSL	reg64, mem	X86_64, LONG, PROT, SW
LSL	reg64, reg16	X86_64, LONG, PROT
LSL	reg64, reg32	X86_64, LONG, PROT
LSL	reg64, reg64	X86_64, LONG, PROT
LSS	reg16, mem	386
LSS	reg32, mem	386
LSS	reg64, mem	X86_64, LONG
LTR	mem	286, PROT, PRIV
LTR	mem16	286, PROT, PRIV
LTR	reg16	286, PROT, PRIV
MFENCE		X86_64, LONG, AMD
MONITOR		PRESCOTT
MONITOR	reg_eax, reg_ecx, reg_edx	PRESCOTT, NOLONG, ND
MONITOR	reg_rax, reg_ecx, reg_edx	X86_64, LONG, ND
MONITORX		AMD
MONITORX	reg_rax, reg_ecx, reg_edx	X86_64, LONG, AMD, ND
MONITORX	reg_eax, reg_ecx, reg_edx	AMD, ND
MONITORX	reg_ax, reg_ecx, reg_edx	AMD, ND
MOV	mem, reg_sreg	8086, SW
MOV	reg16, reg_sreg	8086
MOV	reg32, reg_sreg	386
MOV	reg64, reg_sreg	X86_64, LONG, OPT, ND
MOV	rm64, reg_sreg	X86_64, LONG
MOV	reg_sreg, mem	8086, SW
MOV	reg_sreg, reg16	8086, OPT, ND
MOV	reg_sreg, reg32	386, OPT, ND
MOV	reg_sreg, reg64	X86_64, LONG, OPT, ND
MOV	reg_sreg, reg16	8086
MOV	reg_sreg, reg32	386
MOV	reg_sreg, rm64	X86_64, LONG
MOV	reg_al, mem_offs	8086
MOV	reg_ax, mem_offs	8086
MOV	reg_eax, mem_offs	386
MOV	reg_rax, mem_offs	X86_64, LONG
MOV	mem_offs, reg_al	8086, NOHLE
MOV	mem_offs, reg_ax	8086, NOHLE
MOV	mem_offs, reg_eax	386, NOHLE
MOV	mem_offs, reg_rax	X86_64, LONG, NOHLE
MOV	reg32, reg_creg	386, PRIV, NOLONG
MOV	reg64, reg_creg	X86_64, LONG, PRIV
MOV	reg_creg, reg32	386, PRIV, NOLONG
MOV	reg_creg, reg64	X86_64, LONG, PRIV
MOV	reg32, reg_dreg	386, PRIV, NOLONG
MOV	reg64, reg_dreg	X86_64, LONG, PRIV
MOV	reg_dreg, reg32	386, PRIV, NOLONG
MOV	reg_dreg, reg64	X86_64, LONG, 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	X86_64, LONG
MOV	reg64, reg64	X86_64, LONG
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	X86_64, LONG
MOV	reg64, reg64	X86_64, LONG
MOV	reg8, imm	8086
MOV	reg16, imm	8086
MOV	reg32, imm	386
MOV	reg64, udword	X86_64, LONG, OPT, ND
MOV	reg64, sdword	X86_64, LONG, OPT, ND
MOV	reg64, imm	X86_64, LONG
MOV	rm8, imm	8086
MOV	rm16, imm	8086
MOV	rm32, imm	386
MOV	rm64, imm	X86_64, LONG
MOV	rm64, imm32	X86_64, LONG
MOV	mem, imm8	8086
MOV	mem, imm16	8086
MOV	mem, imm32	386
MOVD	mmxreg, rm32	PENT, MMX, SD
MOVD	rm32, mmxreg	PENT, MMX, SD
MOVD	mmxreg, rm64	X86_64, LONG, MMX, SX, ND
MOVD	rm64, mmxreg	X86_64, LONG, MMX, SX, ND
MOVQ	mmxreg, mmxrm	PENT, MMX
MOVQ	mmxrm, mmxreg	PENT, MMX
MOVQ	mmxreg, rm64	X86_64, LONG, MMX
MOVQ	rm64, mmxreg	X86_64, LONG, MMX
MOVSB		8086
MOVSD		386
MOVSQ		X86_64, LONG
MOVSW		8086
MOVSX	reg16, mem	386
MOVSX	reg16, reg8	386
MOVSX	reg32, rm8	386
MOVSX	reg32, rm16	386
MOVSX	reg64, rm8	X86_64, LONG
MOVSX	reg64, rm16	X86_64, LONG
MOVSLD	reg64, rm32	X86_64, LONG
MOVSLD	reg64, rm32	X86_64, LONG, ND
MOVZX	reg16, mem	386
MOVZX	reg16, reg8	386
MOVZX	reg32, rm8	386
MOVZX	reg32, rm16	386
MOVZX	reg64, rm8	X86_64, LONG
MOVZX	reg64, rm16	X86_64, LONG
MUL	rm8	8086
MUL	rm16	8086
MUL	rm32	386
MUL	rm64	X86_64, LONG
MWAIT		PRESCOTT
MWAIT	reg_eax, reg_ecx	PRESCOTT, ND
MWAITX		AMD
MWAITX	reg_eax, reg_ecx	AMD, ND
NEG	rm8	8086, LOCK
NEG	rm16	8086, LOCK
NEG	rm32	386, LOCK
NEG	rm64	X86_64, LONG, LOCK
NOP		8086
NOP	rm16	P6
NOP	rm32	P6
NOP	rm64	X86_64, LONG

NOT	rm8	8086, LOCK
NOT	rm16	8086, LOCK
NOT	rm32	386, LOCK
NOT	rm64	X86_64, LONG, LOCK
OR	mem, reg8	8086, LOCK
OR	reg8, reg8	8086
OR	mem, reg16	8086, LOCK
OR	reg16, reg16	8086
OR	mem, reg32	386, LOCK
OR	reg32, reg32	386
OR	mem, reg64	X86_64, LONG, LOCK
OR	reg64, reg64	X86_64, LONG
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	X86_64, LONG
OR	reg64, reg64	X86_64, LONG
OR	rm16, imm8	8086, LOCK
OR	rm32, imm8	386, LOCK
OR	rm64, imm8	X86_64, LONG, LOCK
OR	reg_al, imm	8086
OR	reg_ax, sbytedword	8086, ND
OR	reg_ax, imm	8086
OR	reg_eax, sbytedword	386, ND
OR	reg_eax, imm	386
OR	reg_rax, sbytedword	X86_64, LONG, ND
OR	reg_rax, imm	X86_64, LONG
OR	rm8, imm	8086, LOCK
OR	rm16, sbytedword	8086, LOCK, ND
OR	rm16, imm	8086, LOCK
OR	rm32, sbytedword	386, LOCK, ND
OR	rm32, imm	386, LOCK
OR	rm64, sbytedword	X86_64, LONG, LOCK, ND
OR	rm64, imm	X86_64, LONG, LOCK
OR	mem, imm8	8086, LOCK
OR	mem, sbytedword16	8086, LOCK, ND
OR	mem, imm16	8086, LOCK
OR	mem, sbytedword32	386, LOCK, ND
OR	mem, imm32	386, LOCK
OR	rm8, imm	8086, LOCK, ND, NOLONG
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
PADDSD	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
PMULHRCW	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	X86_64, LONG
POP	rm16	8086
POP	rm32	386, NOLONG
POP	rm64	X86_64, LONG
POP	reg_es	8086, NOLONG
POP	reg_cs	8086, UNDOC, ND, OBSOLETE
POP	reg_ss	8086, NOLONG
POP	reg_ds	8086, NOLONG
POP	reg_fs	386
POP	reg_gs	386
POPA		186, NOLONG
POPAD		386, NOLONG
POPAW		186, NOLONG
POPF		8086
POPFD		386, NOLONG
POPFQ		X86_64, LONG
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	X86_64, LONG
PUSH	rm16	8086
PUSH	rm32	386, NOLONG
PUSH	rm64	X86_64, LONG
PUSH	reg_es	8086, NOLONG
PUSH	reg_cs	8086, NOLONG
PUSH	reg_ss	8086, NOLONG
PUSH	reg_ds	8086, NOLONG
PUSH	reg_fs	386
PUSH	reg_gs	386
PUSH	imm8	186
PUSH	sbytedword16	186, AR0, SIZE, ND
PUSH	imm16	186, AR0, SIZE
PUSH	sbytedword32	386, NOLONG, AR0, SIZE, ND
PUSH	imm32	386, NOLONG, AR0, SIZE
PUSH	sbytedword32	386, NOLONG, SD, ND
PUSH	imm32	386, NOLONG, SD
PUSH	sbytedword64	X86_64, LONG, AR0, SIZE, ND
PUSH	imm64	X86_64, LONG, AR0, SIZE
PUSH	sbytedword32	X86_64, LONG, AR0, SIZE, ND
PUSH	imm32	X86_64, LONG, AR0, SIZE
PUSHA		186, NOLONG
PUSHAD		386, NOLONG
PUSHAW		186, NOLONG
PUSHF		8086
PUSHFD		386, NOLONG
PUSHFQ		X86_64, LONG
PUSHFW		8086
PXOR	mmxreg,mmxrm	PENT, MMX
RCL	rm8,unity	8086
RCL	rm8,reg_cl	8086
RCL	rm8,imm8	186
RCL	rm16,unity	8086
RCL	rm16,reg_cl	8086
RCL	rm16,imm8	186
RCL	rm32,unity	386
RCL	rm32,reg_cl	386
RCL	rm32,imm8	386
RCL	rm64,unity	X86_64, LONG
RCL	rm64,reg_cl	X86_64, LONG
RCL	rm64,imm8	X86_64, LONG
RCR	rm8,unity	8086
RCR	rm8,reg_cl	8086



RCR	rm8,imm8	186
RCR	rm16,unity	8086
RCR	rm16,reg_cl	8086
RCR	rm16,imm8	186
RCR	rm32,unity	386
RCR	rm32,reg_cl	386
RCR	rm32,imm8	386
RCR	rm64,unity	X86_64, LONG
RCR	rm64,reg_cl	X86_64, LONG
RCR	rm64,imm8	X86_64, LONG
RDSHR	rm32	P6, CYRIX, SMM
RDMSR		PENT, PRIV
RDPMC		P6
RDTSR		PENT
RET		X86_64
RET	imm	8086, BND
RETF		8086, SW, BND
RETF	imm	8086
RETN		8086, SW
RETN	imm	8086, BND
RETW		8086, SW, BND
RETW	imm	8086, BND
RETFW		8086, SW, BND
RETFW	imm	8086
RETNW		8086, SW
RETNW	imm	8086, BND
RETD		8086, BND, NOLONG
RETD	imm	8086, SW, BND, NOLONG
RETFD		8086
RETFD	imm	8086, SW
RETND		8086, BND, NOLONG
RETND	imm	8086, SW, BND, NOLONG
RETQ		X86_64, LONG, BND
RETQ	imm	X86_64, LONG, SW, BND
RETFQ		X86_64, LONG
RETFQ	imm	X86_64, LONG, SW
RETNQ		X86_64, LONG, BND
RETNQ	imm	X86_64, LONG, SW, BND
ROL	rm8,unity	8086
ROL	rm8,reg_cl	8086
ROL	rm8,imm8	186
ROL	rm16,unity	8086
ROL	rm16,reg_cl	8086
ROL	rm16,imm8	186
ROL	rm32,unity	386
ROL	rm32,reg_cl	386
ROL	rm32,imm8	386
ROL	rm64,unity	X86_64, LONG
ROL	rm64,reg_cl	X86_64, LONG
ROL	rm64,imm8	X86_64, LONG
ROR	rm8,unity	8086
ROR	rm8,reg_cl	8086
ROR	rm8,imm8	186
ROR	rm16,unity	8086
ROR	rm16,reg_cl	8086
ROR	rm16,imm8	186
ROR	rm32,unity	386
ROR	rm32,reg_cl	386
ROR	rm32,imm8	386
ROR	rm64,unity	X86_64, LONG
ROR	rm64,reg_cl	X86_64, LONG
ROR	rm64,imm8	X86_64, LONG
RDM		P6, CYRIX, ND
RSDC	reg_sreg, mem80	486, CYRIX, SMM
RSLDT	mem80	486, CYRIX, SMM
RSM		PENT, SMM

RSTS	mem80	486, CYRIX, SMM
SAHF		8086
SAL	rm8, unity	8086, ND
SAL	rm8, reg_cl	8086, ND
SAL	rm8, imm8	186, ND
SAL	rm16, unity	8086, ND
SAL	rm16, reg_cl	8086, ND
SAL	rm16, imm8	186, ND
SAL	rm32, unity	386, ND
SAL	rm32, reg_cl	386, ND
SAL	rm32, imm8	386, ND
SAL	rm64, unity	X86_64, LONG, ND
SAL	rm64, reg_cl	X86_64, LONG, ND
SAL	rm64, imm8	X86_64, LONG, ND
SALC		8086, UNDOC
SAR	rm8, unity	8086
SAR	rm8, reg_cl	8086
SAR	rm8, imm8	186
SAR	rm16, unity	8086
SAR	rm16, reg_cl	8086
SAR	rm16, imm8	186
SAR	rm32, unity	386
SAR	rm32, reg_cl	386
SAR	rm32, imm8	386
SAR	rm64, unity	X86_64, LONG
SAR	rm64, reg_cl	X86_64, LONG
SAR	rm64, imm8	X86_64, LONG
SBB	mem, reg8	8086, LOCK
SBB	reg8, reg8	8086
SBB	mem, reg16	8086, LOCK
SBB	reg16, reg16	8086
SBB	mem, reg32	386, LOCK
SBB	reg32, reg32	386
SBB	mem, reg64	X86_64, LONG, LOCK
SBB	reg64, reg64	X86_64, LONG
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	X86_64, LONG
SBB	reg64, reg64	X86_64, LONG
SBB	rm16, imm8	8086, LOCK
SBB	rm32, imm8	386, LOCK
SBB	rm64, imm8	X86_64, LONG, LOCK
SBB	reg_al, imm	8086
SBB	reg_ax, sbyte word	8086, ND
SBB	reg_ax, imm	8086
SBB	reg_eax, sbyte word	386, ND
SBB	reg_eax, imm	386
SBB	reg_rax, sbyte word	X86_64, LONG, ND
SBB	reg_rax, imm	X86_64, LONG
SBB	rm8, imm	8086, LOCK
SBB	rm16, sbyte word	8086, LOCK, ND
SBB	rm16, imm	8086, LOCK
SBB	rm32, sbyte word	386, LOCK, ND
SBB	rm32, imm	386, LOCK
SBB	rm64, sbyte word	X86_64, LONG, LOCK, ND
SBB	rm64, imm	X86_64, LONG, LOCK
SBB	mem, imm8	8086, LOCK
SBB	mem, sbyte word16	8086, LOCK, ND
SBB	mem, imm16	8086, LOCK
SBB	mem, sbyte word32	386, LOCK, ND
SBB	mem, imm32	386, LOCK
SBB	rm8, imm	8086, LOCK, ND, NOLONG
SCASB		8086

SCASD		386
SCASQ		X86_64, LONG
SCASW		8086
SFENCE		X86_64, LONG, AMD
SGDT	mem	286
SHL	rm8, unity	8086
SHL	rm8, reg_cl	8086
SHL	rm8, imm8	186
SHL	rm16, unity	8086
SHL	rm16, reg_cl	8086
SHL	rm16, imm8	186
SHL	rm32, unity	386
SHL	rm32, reg_cl	386
SHL	rm32, imm8	386
SHL	rm64, unity	X86_64, LONG
SHL	rm64, reg_cl	X86_64, LONG
SHL	rm64, imm8	X86_64, LONG
SHLD	mem, reg16, imm	386
SHLD	reg16, reg16, imm	386
SHLD	mem, reg32, imm	386
SHLD	reg32, reg32, imm	386
SHLD	mem, reg64, imm	X86_64, LONG
SHLD	reg64, reg64, imm	X86_64, LONG
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	X86_64, LONG
SHLD	reg64, reg64, reg_cl	X86_64, LONG
SHR	rm8, unity	8086
SHR	rm8, reg_cl	8086
SHR	rm8, imm8	186
SHR	rm16, unity	8086
SHR	rm16, reg_cl	8086
SHR	rm16, imm8	186
SHR	rm32, unity	386
SHR	rm32, reg_cl	386
SHR	rm32, imm8	386
SHR	rm64, unity	X86_64, LONG
SHR	rm64, reg_cl	X86_64, LONG
SHR	rm64, imm8	X86_64, LONG
SHRD	mem, reg16, imm	386
SHRD	reg16, reg16, imm	386
SHRD	mem, reg32, imm	386
SHRD	reg32, reg32, imm	386
SHRD	mem, reg64, imm	X86_64, LONG
SHRD	reg64, reg64, imm	X86_64, LONG
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	X86_64, LONG
SHRD	reg64, reg64, reg_cl	X86_64, LONG
SIDT	mem	286
SLDT	mem	286
SLDT	mem16	286
SLDT	reg16	286
SLDT	reg32	386
SLDT	reg64	X86_64, LONG, ND
SLDT	reg64	X86_64, LONG
SKINIT		X86_64, LONG
SMI		386, UNDOC
SMINT		P6, CYRIX, ND
SMINTOLD		486, CYRIX, ND, OBSOLETE
SMSW	mem	286
SMSW	mem16	286
SMSW	reg16	286

SMSW	reg32	386
SMSW	reg64	X86_64, LONG
STC		8086
STD		8086
STI		8086
STOSB		8086
STOSD		386
STOSQ		X86_64, LONG
STOSW		8086
STR	mem	286, PROT
STR	mem16	286, PROT
STR	reg16	286, PROT
STR	reg32	386, PROT
STR	reg64	X86_64, LONG
SUB	mem, reg8	8086, LOCK
SUB	reg8, reg8	8086
SUB	mem, reg16	8086, LOCK
SUB	reg16, reg16	8086
SUB	mem, reg32	386, LOCK
SUB	reg32, reg32	386
SUB	mem, reg64	X86_64, LONG, LOCK
SUB	reg64, reg64	X86_64, LONG
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	X86_64, LONG
SUB	reg64, reg64	X86_64, LONG
SUB	rm16, imm8	8086, LOCK
SUB	rm32, imm8	386, LOCK
SUB	rm64, imm8	X86_64, LONG, LOCK
SUB	reg_al, imm	8086
SUB	reg_ax, sbyteword	8086, ND
SUB	reg_ax, imm	8086
SUB	reg_eax, sbytedword	386, ND
SUB	reg_eax, imm	386
SUB	reg_rax, sbytedword	X86_64, LONG, ND
SUB	reg_rax, imm	X86_64, LONG
SUB	rm8, imm	8086, LOCK
SUB	rm16, sbyteword	8086, LOCK, ND
SUB	rm16, imm	8086, LOCK
SUB	rm32, sbytedword	386, LOCK, ND
SUB	rm32, imm	386, LOCK
SUB	rm64, sbytedword	X86_64, LONG, LOCK, ND
SUB	rm64, imm	X86_64, LONG, LOCK
SUB	mem, imm8	8086, LOCK
SUB	mem, sbyteword16	8086, LOCK, ND
SUB	mem, imm16	8086, LOCK
SUB	mem, sbytedword32	386, LOCK, ND
SUB	mem, imm32	386, LOCK
SUB	rm8, imm	8086, LOCK, ND, NO LONG
SVDC	mem80, reg_sreg	486, CYRIX, SMM
SVLDT	mem80	486, CYRIX, SMM, ND
SVTS	mem80	486, CYRIX, SMM
SWAPGS		X86_64, LONG
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	X86_64, LONG
TEST	reg64, reg64	X86_64, LONG
TEST	reg8, mem	8086
TEST	reg16, mem	8086
TEST	reg32, mem	386
TEST	reg64, mem	X86_64, LONG
TEST	reg_al, imm	8086
TEST	reg_ax, imm	8086
TEST	reg_eax, imm	386
TEST	reg_rax, imm	X86_64, LONG
TEST	rm8, imm	8086
TEST	rm16, imm	8086
TEST	rm32, imm	386
TEST	rm64, imm	X86_64, LONG
TEST	mem, imm8	8086
TEST	mem, imm16	8086
TEST	mem, imm32	386
UD0		186, OBSOLETE
UD0	reg16, rm16	186
UD0	reg32, rm32	186
UD0	reg64, rm64	186
UD1	reg16, rm16	186
UD1	reg32, rm32	186
UD1	reg64, rm64	186
UD1		186, ND
UD2B		186, ND
UD2B	reg16, rm16	186, ND
UD2B	reg32, rm32	186, ND
UD2B	reg64, rm64	186, 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, CYRIX, SMM
WRMSR		PENT, PRIV
XADD	mem, reg8	486, LOCK
XADD	reg8, reg8	486
XADD	mem, reg16	486, LOCK
XADD	reg16, reg16	486
XADD	mem, reg32	486, LOCK
XADD	reg32, reg32	486
XADD	mem, reg64	X86_64, LONG, LOCK
XADD	reg64, reg64	X86_64, LONG
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	X86_64, LONG
XCHG	reg16, reg_ax	8086
XCHG	reg32na, reg_eax	386
XCHG	reg64, reg_rax	X86_64, LONG
XCHG	reg_eax, reg_eax	386, NOLONG
XCHG	reg8, reg8	8086
XCHG	reg16, reg16	8086
XCHG	reg32, reg32	386
XCHG	reg64, reg64	X86_64, LONG
XCHG	mem, reg8	8086, LOCK
XCHG	mem, reg16	8086, LOCK
XCHG	mem, reg32	386, LOCK
XCHG	mem, reg64	X86_64, LONG, LOCK
XCHG	reg8, mem	8086, LOCK1
XCHG	reg16, mem	8086, LOCK1
XCHG	reg32, mem	386, LOCK1
XCHG	reg64, mem	X86_64, LONG, LOCK1
XLATB		8086
XLAT		8086
XOR	mem, reg8	8086, LOCK
XOR	reg8, reg8	8086
XOR	mem, reg16	8086, LOCK
XOR	reg16, reg16	8086
XOR	mem, reg32	386, LOCK
XOR	reg32, reg32	386
XOR	mem, reg64	X86_64, LONG, LOCK
XOR	reg64, reg64	X86_64, LONG
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	X86_64, LONG
XOR	reg64, reg64	X86_64, LONG
XOR	rm16, imm8	8086, LOCK
XOR	rm32, imm8	386, LOCK
XOR	rm64, imm8	X86_64, LONG, LOCK
XOR	reg_al, imm	8086
XOR	reg_ax, sbyteword	8086, ND
XOR	reg_ax, imm	8086
XOR	reg_eax, sbytedword	386, ND
XOR	reg_eax, imm	386
XOR	reg_rax, sbytedword	X86_64, LONG, ND
XOR	reg_rax, imm	X86_64, LONG
XOR	rm8, imm	8086, LOCK
XOR	rm16, sbyteword	8086, LOCK, ND
XOR	rm16, imm	8086, LOCK
XOR	rm32, sbytedword	386, LOCK, ND
XOR	rm32, imm	386, LOCK
XOR	rm64, sbytedword	X86_64, LONG, LOCK, ND
XOR	rm64, imm	X86_64, LONG, LOCK
XOR	mem, imm8	8086, LOCK
XOR	mem, sbyteword16	8086, LOCK, ND
XOR	mem, imm16	8086, LOCK
XOR	mem, sbytedword32	386, LOCK, ND
XOR	mem, imm32	386, LOCK
XOR	rm8, imm	8086, LOCK, ND, NOLONG
CMOVcc	reg16, mem	P6
CMOVcc	reg16, reg16	P6
CMOVcc	reg32, mem	P6
CMOVcc	reg32, reg32	P6
CMOVcc	reg64, mem	X86_64, LONG
CMOVcc	reg64, reg64	X86_64, LONG
Jcc	imm near	386, BND
Jcc	imm16 near	386, NOLONG, BND
Jcc	imm32 near	386, NOLONG, BND

Jcc	imm64 near	X86_64, LONG, BND
Jcc	imm short	8086, ND, BND
Jcc	imm	8086, ND, BND
Jcc	imm	386, ND, BND
Jcc	imm	8086, ND, BND
Jcc	imm	8086, BND
SETcc	mem	386
SETcc	reg8	386

### F.1.3 Katmai Streaming SIMD instructions (SSE -- a.k.a. KNI, XMM, MMX2)

ADDPS	xmmreg, xmmrm128	KATMAI, SSE
ADDSS	xmmreg, xmmrm32	KATMAI, SSE
ANDNPS	xmmreg, xmmrm128	KATMAI, SSE
ANDPS	xmmreg, xmmrm128	KATMAI, SSE
CMPEQPS	xmmreg, xmmrm128	KATMAI, SSE
CMPEQSS	xmmreg, xmmrm32	KATMAI, SSE
CMPLEPS	xmmreg, xmmrm128	KATMAI, SSE
CMPLESS	xmmreg, xmmrm32	KATMAI, SSE
CMPLTPS	xmmreg, xmmrm128	KATMAI, SSE
CMPLTSS	xmmreg, xmmrm32	KATMAI, SSE
CMPNEQPS	xmmreg, xmmrm128	KATMAI, SSE
CMPNEQSS	xmmreg, xmmrm32	KATMAI, SSE
CMPNLEPS	xmmreg, xmmrm128	KATMAI, SSE
CMPNLESS	xmmreg, xmmrm32	KATMAI, SSE
CMPNLTPS	xmmreg, xmmrm128	KATMAI, SSE
CMPNLTSS	xmmreg, xmmrm32	KATMAI, SSE
CMPORDPS	xmmreg, xmmrm128	KATMAI, SSE
CMPORDSS	xmmreg, xmmrm32	KATMAI, SSE
CMPUNORDPS	xmmreg, xmmrm128	KATMAI, SSE
CMPUNORDSS	xmmreg, xmmrm32	KATMAI, SSE
CMPPS	xmmreg, xmmrm128, imm8	KATMAI, SSE
CMPSS	xmmreg, xmmrm32, imm8	KATMAI, SSE
COMISS	xmmreg, xmmrm32	KATMAI, SSE
CVTPI2PS	xmmreg, mmxrm64	KATMAI, SSE, MMX
CVTPS2PI	mmxreg, xmmrm64	KATMAI, SSE, MMX
CVTSI2SS	xmmreg, mem	KATMAI, SSE, SD, AR1, ND
CVTSI2SS	xmmreg, rm32	KATMAI, SSE, SD, AR1
CVTSI2SS	xmmreg, rm64	X86_64, LONG, SSE, AR1
CVTSS2SI	reg32, xmmreg	KATMAI, SSE, SD, AR1
CVTSS2SI	reg32, mem	KATMAI, SSE, SD, AR1
CVTSS2SI	reg64, xmmreg	X86_64, LONG, SSE, SD, AR1
CVTSS2SI	reg64, mem	X86_64, LONG, SSE, SD, AR1
CVTTPS2PI	mmxreg, xmmrm	KATMAI, SSE, MMX
CVTTSS2SI	reg32, xmmrm	KATMAI, SSE, SD, AR1
CVTTSS2SI	reg64, xmmrm	X86_64, LONG, SSE, SD, AR1
DIVPS	xmmreg, xmmrm128	KATMAI, SSE
DIVSS	xmmreg, xmmrm32	KATMAI, SSE
LDMXCSR	mem32	KATMAI, SSE
MAXPS	xmmreg, xmmrm128	KATMAI, SSE
MAXSS	xmmreg, xmmrm32	KATMAI, SSE
MINPS	xmmreg, xmmrm128	KATMAI, SSE
MINSS	xmmreg, xmmrm32	KATMAI, SSE
MOVAPS	xmmreg, xmmrm128	KATMAI, SSE
MOVAPS	xmmrm128, xmmreg	KATMAI, SSE
MOVHPS	xmmreg, mem64	KATMAI, SSE
MOVHPS	mem64, xmmreg	KATMAI, SSE
MOVLHPS	xmmreg, xmmreg	KATMAI, SSE
MOVLPS	xmmreg, mem64	KATMAI, SSE
MOVLPS	mem64, xmmreg	KATMAI, SSE
MOVHLPS	xmmreg, xmmreg	KATMAI, SSE
MOVMSKPS	reg32, xmmreg	KATMAI, SSE
MOVMSKPS	reg64, xmmreg	X86_64, LONG, SSE
MOVNTPS	mem128, xmmreg	KATMAI, SSE
MOVSS	xmmreg, xmmrm32	KATMAI, SSE
MOVSS	xmmrm32, xmmreg	KATMAI, SSE
MOVUPS	xmmreg, xmmrm128	KATMAI, SSE

MOVUPS	xmmrm128, xmmreg	KATMAI, SSE
MULPS	xmmreg, xmmrm128	KATMAI, SSE
MULSS	xmmreg, xmmrm32	KATMAI, SSE
ORPS	xmmreg, xmmrm128	KATMAI, SSE
RCPPS	xmmreg, xmmrm128	KATMAI, SSE
RCPSS	xmmreg, xmmrm32	KATMAI, SSE
RSQRTPS	xmmreg, xmmrm128	KATMAI, SSE
RSQRTSS	xmmreg, xmmrm32	KATMAI, SSE
SHUFPS	xmmreg, xmmrm128, imm8	KATMAI, SSE
SQRTPS	xmmreg, xmmrm128	KATMAI, SSE
SQRTSS	xmmreg, xmmrm32	KATMAI, SSE
STMXCSR	mem32	KATMAI, SSE
SUBPS	xmmreg, xmmrm128	KATMAI, SSE
SUBSS	xmmreg, xmmrm32	KATMAI, SSE
UCOMISS	xmmreg, xmmrm32	KATMAI, SSE
UNPCKHPS	xmmreg, xmmrm128	KATMAI, SSE
UNPCKLPS	xmmreg, xmmrm128	KATMAI, SSE
XORPS	xmmreg, xmmrm128	KATMAI, SSE

#### F.1.4 Introduced in Deschutes but necessary for SSE support

FXRSTOR	mem	P6, SSE, FPU
FXRSTOR64	mem	X86_64, LONG, SSE, FPU
FXSAVE	mem	P6, SSE, FPU
FXSAVE64	mem	X86_64, LONG, SSE, FPU

#### F.1.5 XSAVE group (AVX and extended state)

XGETBV		NEHALEM
XSETBV		NEHALEM, PRIV
XSAVE	mem	NEHALEM
XSAVE64	mem	LONG, NEHALEM
XSAVEC	mem	
XSAVEC64	mem	LONG
XSAVEOPT	mem	
XSAVEOPT64	mem	LONG
XSAVES	mem	
XSAVES64	mem	LONG
XRSTOR	mem	NEHALEM
XRSTOR64	mem	LONG, NEHALEM
XRSTORS	mem	
XRSTORS64	mem	LONG

#### F.1.6 Generic memory operations

PREFETCHNTA	mem8	KATMAI
PREFETCHT0	mem8	KATMAI
PREFETCHT1	mem8	KATMAI
PREFETCHT2	mem8	KATMAI
PREFETCHIT0	mem8	PREFETCHI
PREFETCHIT1	mem8	PREFETCHI
SFENCE		KATMAI

#### F.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
PMOVBK	reg32, mmxreg	KATMAI, MMX



PMULHUW	mmxreg, mmxrm	KATMAI, MMX
PSADBW	mmxreg, mmxrm	KATMAI, MMX
PSHUFW	mmxreg, mmxrm, imm	KATMAI, MMX

### F.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

### F.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	X86_64, LONG
MOVNTPD	mem, xmmreg	WILLAMETTE, SSE2, SO
LFENCE		WILLAMETTE, SSE2
MFENCE		WILLAMETTE, SSE2

### F.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, xmmrm128	WILLAMETTE, SSE2, SO
MOVDQA	xmmrm128, xmmreg	WILLAMETTE, SSE2, SO
MOVDQU	xmmreg, xmmrm128	WILLAMETTE, SSE2, SO
MOVDQU	xmmrm128, xmmreg	WILLAMETTE, SSE2, SO
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	X86_64, LONG, SSE2
MOVQ	rm64, xmmreg	X86_64, LONG, 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
PADD	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
PCMPQB	xmmreg, xmmrm	WILLAMETTE, SSE2, SO
PCMPQW	xmmreg, xmmrm	WILLAMETTE, SSE2, SO
PCMPQD	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
PEXTRW	reg64, xmmreg, imm	X86_64, LONG, SSE2, ND
PINSRW	xmmreg, reg16, imm	WILLAMETTE, SSE2
PINSRW	xmmreg, reg32, imm	WILLAMETTE, SSE2, ND
PINSRW	xmmreg, reg64, imm	X86_64, LONG, SSE2, ND

PINSRW	xmmreg, mem, imm	WILLAMETTE, SSE2
PINSRW	xmmreg, mem16, imm	WILLAMETTE, SSE2
PMADDWD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMAXS	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMAXUB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMINSW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMINUB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMOVMASKB	reg32, xmmreg	WILLAMETTE, SSE2
PMULHUW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMULHW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMULLW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PMULUDQ	mmxreg, mmxrm	WILLAMETTE, SSE2, S0
PMULUDQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
POR	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSADB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSHUFD	xmmreg, xmmreg, imm	WILLAMETTE, SSE2
PSHUFD	xmmreg, mem, imm	WILLAMETTE, SSE2
PSHUFHW	xmmreg, xmmreg, imm	WILLAMETTE, SSE2
PSHUFHW	xmmreg, mem, imm	WILLAMETTE, SSE2
PSHUFLW	xmmreg, xmmreg, imm	WILLAMETTE, SSE2
PSHUFLW	xmmreg, mem, imm	WILLAMETTE, SSE2
PSLDDQ	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSLLW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSLLW	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSLLD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSLLD	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSLLQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSLLQ	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRAW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSRAW	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRAD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSRAD	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRLDQ	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRLW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSRLW	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRLD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSRLD	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSRLQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSRLQ	xmmreg, imm	WILLAMETTE, SSE2, AR1
PSUBB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBQ	mmxreg, mmxrm	WILLAMETTE, SSE2, S0
PSUBQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBSB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBSW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBUSB	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PSUBUSW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKHBW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKHWD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKHDQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKHQDQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKLBW	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKLWD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKLDQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PUNPCKLQDQ	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
PXOR	xmmreg, xmmrm	WILLAMETTE, SSE2, S0

### F.1.11 Willamette Streaming SIMD instructions (SSE2)

ADDPD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
ADDSD	xmmreg, xmmrm	WILLAMETTE, SSE2
ANDNPD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
ANDPD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
CMPEQPD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0
CMPEQSD	xmmreg, xmmrm	WILLAMETTE, SSE2
CMPLPD	xmmreg, xmmrm	WILLAMETTE, SSE2, S0

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, xmmrm128, imm8	WILLAMETTE, SSE2
CMPSD	xmmreg, xmmrm128, imm8	WILLAMETTE, SSE2
COMISD	xmmreg, xmmrm64	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	X86_64, LONG, SSE2, AR1
CVTSD2SI	reg64, mem	X86_64, LONG, 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	X86_64, LONG, 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	X86_64, LONG, SSE2, AR1
CVTTSD2SI	reg64, mem	X86_64, LONG, 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, xmmrm128	WILLAMETTE, SSE2
MOVAPD	xmmrm128, xmmreg	WILLAMETTE, SSE2
MOVHPD	mem64, xmmreg	WILLAMETTE, SSE2
MOVHPD	xmmreg, mem64	WILLAMETTE, SSE2
MOVLPD	mem64, xmmreg	WILLAMETTE, SSE2
MOVLPD	xmmreg, mem64	WILLAMETTE, SSE2
MOVMSKPD	reg32, xmmreg	WILLAMETTE, SSE2
MOVMSKPD	reg64, xmmreg	X86_64, LONG, SSE2
MOVSD	xmmreg, xmmrm64	WILLAMETTE, SSE2
MOVSD	xmmrm64, xmmreg	WILLAMETTE, SSE2
MOVUPD	xmmreg, xmmrm128	WILLAMETTE, SSE2
MOVUPD	xmmrm128, xmmreg	WILLAMETTE, SSE2
MULPD	xmmreg, xmmrm128	WILLAMETTE, SSE2, SO
MULSD	xmmreg, xmmrm64	WILLAMETTE, SSE2
ORPD	xmmreg, xmmrm128	WILLAMETTE, SSE2, SO
SHUFPD	xmmreg, xmmrm128, imm8	WILLAMETTE, SSE2
SQRTPD	xmmreg, xmmrm128	WILLAMETTE, SSE2, SO
SQRTSD	xmmreg, xmmrm64	WILLAMETTE, SSE2
SUBPD	xmmreg, xmmrm128	WILLAMETTE, SSE2, SO
SUBSD	xmmreg, xmmrm64	WILLAMETTE, SSE2
UCOMISD	xmmreg, xmmrm64	WILLAMETTE, SSE2

UNPCKHPD	xmmreg, xmmrm128	WILLAMETTE, SSE2
UNPCKLPD	xmmreg, xmmrm128	WILLAMETTE, SSE2
XORPD	xmmreg, xmmrm128	WILLAMETTE, SSE2

### F.1.12 Prescott New Instructions (SSE3)

ADDSUBPD	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
ADDSUBPS	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
HADDPD	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
HADDPS	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
HSUBPD	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
HSUBPS	xmmreg, xmmrm128	PRESCOTT, SSE3, SO
LDDQU	xmmreg, mem128	PRESCOTT, SSE3, SO
MOVDDUP	xmmreg, xmmrm64	PRESCOTT, SSE3
MOVSHDUP	xmmreg, xmmrm128	PRESCOTT, SSE3
MOVSLDUP	xmmreg, xmmrm128	PRESCOTT, SSE3

### F.1.13 VMX/SVM Instructions

CLGI		VMX, AMD
STGI		VMX, AMD
VMCALL		VMX
VMCLEAR	mem	VMX
VMFUNC		VMX
VMLAUNCH		VMX
VMLOAD		VMX, AMD
VMMCALL		VMX, AMD
VMPTRLD	mem	VMX
VMPTRST	mem	VMX
VMREAD	rm32, reg32	VMX, NOLONG, SD
VMREAD	rm64, reg64	X86_64, LONG, VMX
VMRESUME		VMX
VMRUN		VMX, AMD
VMSAVE		VMX, AMD
VMWRITE	reg32, rm32	VMX, NOLONG, SD
VMWRITE	reg64, rm64	X86_64, LONG, VMX
VMXOFF		VMX
VMXON	mem	VMX

### F.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

### F.1.15 SEV-SNP AMD instructions

PVALIDATE		VMX, AMD
RMPADJUST		VMX, AMD
VMGEXIT		VMX, AMD
VMGEXIT		VMX, AMD

### F.1.16 Tejas New Instructions (SSSE3)

PABSB	mmxreg, mmxrm	SSSE3, MMX
PABSB	xmmreg, xmmrm128	SSSE3
PABSW	mmxreg, mmxrm	SSSE3, MMX
PABSW	xmmreg, xmmrm128	SSSE3
PABSD	mmxreg, mmxrm	SSSE3, MMX
PABSD	xmmreg, xmmrm128	SSSE3
PALIGNR	mmxreg, mmxrm, imm	SSSE3, MMX
PALIGNR	xmmreg, xmmrm, imm	SSSE3
PHADDW	mmxreg, mmxrm	SSSE3, MMX
PHADDW	xmmreg, xmmrm128	SSSE3
PHADD	mmxreg, mmxrm	SSSE3, MMX
PHADD	xmmreg, xmmrm128	SSSE3
PHADDSW	mmxreg, mmxrm	SSSE3, MMX

PHADDSW	xmmreg, xmmrm128	SSSE3
PHSUBW	mmxreg, mmxrm	SSSE3, MMX
PHSUBW	xmmreg, xmmrm128	SSSE3
PHSUBD	mmxreg, mmxrm	SSSE3, MMX
PHSUBD	xmmreg, xmmrm128	SSSE3
PHSUBSW	mmxreg, mmxrm	SSSE3, MMX
PHSUBSW	xmmreg, xmmrm128	SSSE3
PMADDUBSW	mmxreg, mmxrm	SSSE3, MMX
PMADDUBSW	xmmreg, xmmrm128	SSSE3
PMULHRWSW	mmxreg, mmxrm	SSSE3, MMX
PMULHRWSW	xmmreg, xmmrm128	SSSE3
PSHUFB	mmxreg, mmxrm	SSSE3, MMX
PSHUFB	xmmreg, xmmrm128	SSSE3
PSIGNB	mmxreg, mmxrm	SSSE3, MMX
PSIGNB	xmmreg, xmmrm128	SSSE3
PSIGNW	mmxreg, mmxrm	SSSE3, MMX
PSIGNW	xmmreg, xmmrm128	SSSE3
PSIGND	mmxreg, mmxrm	SSSE3, MMX
PSIGND	xmmreg, xmmrm128	SSSE3

### F.1.17 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	mem64, xmmreg	SSE4A, AMD
MOVNTSS	mem32, xmmreg	SSE4A, AMD, SD

### F.1.18 New instructions in Barcelona

LZCNT	reg16, rm16	P6, AMD
LZCNT	reg32, rm32	P6, AMD
LZCNT	reg64, rm64	X86_64, LONG, AMD

### F.1.19 Penryn New Instructions (SSE4.1)

BLENDPD	xmmreg, xmmrm128, imm8	SSE41
BLENDPS	xmmreg, xmmrm128, imm8	SSE41
BLENDVPD	xmmreg, xmmrm128, xmm0	SSE41
BLENDVPD	xmmreg, xmmrm128	SSE41
BLENDVPS	xmmreg, xmmrm128, xmm0	SSE41
BLENDVPS	xmmreg, xmmrm128	SSE41
DPPD	xmmreg, xmmrm128, imm8	SSE41
DPPS	xmmreg, xmmrm128, imm8	SSE41
EXTRACTPS	rm32, xmmreg, imm8	SSE41
EXTRACTPS	reg64, xmmreg, imm8	SSE41, X86_64, LONG
INSERTPS	xmmreg, xmmrm32, imm8	SSE41
MOVNTDQA	xmmreg, mem128	SSE41
MPSADBW	xmmreg, xmmrm128, imm8	SSE41
PACKUSDW	xmmreg, xmmrm128	SSE41
PBLENDVB	xmmreg, xmmrm, xmm0	SSE41
PBLENDVB	xmmreg, xmmrm128	SSE41
PBLENDW	xmmreg, xmmrm128, imm8	SSE41
PCMPEQQ	xmmreg, xmmrm128	SSE41
PEXTRB	reg32, xmmreg, imm8	SSE41
PEXTRB	mem8, xmmreg, imm8	SSE41
PEXTRB	reg64, xmmreg, imm8	SSE41, X86_64, LONG
PEXTRD	rm32, xmmreg, imm8	SSE41
PEXTRQ	rm64, xmmreg, imm8	SSE41, X86_64, LONG
PEXTRW	reg32, xmmreg, imm8	SSE41
PEXTRW	mem16, xmmreg, imm8	SSE41
PEXTRW	reg64, xmmreg, imm8	SSE41, X86_64, LONG
PHMINPOSUW	xmmreg, xmmrm128	SSE41
PINSRB	xmmreg, mem, imm8	SSE41
PINSRB	xmmreg, rm8, imm8	SSE41
PINSRB	xmmreg, reg32, imm8	SSE41
PINSRD	xmmreg, rm32, imm8	SSE41

PINSRQ	xmmreg, rm64, imm8	SSE41, X86_64, LONG
PMASXB	xmmreg, xmmrm128	SSE41
PMASXD	xmmreg, xmmrm128	SSE41
PMASUD	xmmreg, xmmrm128	SSE41
PMASUW	xmmreg, xmmrm128	SSE41
PMINSB	xmmreg, xmmrm128	SSE41
PMINSD	xmmreg, xmmrm128	SSE41
PMINUD	xmmreg, xmmrm128	SSE41
PMINUW	xmmreg, xmmrm128	SSE41
PMOVSXBW	xmmreg, xmmrm64	SSE41
PMOVSXBD	xmmreg, xmmrm32	SSE41, SD
PMOVSXBQ	xmmreg, xmmrm16	SSE41, SW
PMOVSWD	xmmreg, xmmrm64	SSE41
PMOVSWQ	xmmreg, xmmrm32	SSE41, SD
PMOVXSDQ	xmmreg, xmmrm64	SSE41
PMOVZXBW	xmmreg, xmmrm64	SSE41
PMOVZXBQ	xmmreg, xmmrm32	SSE41, SD
PMOVZXBQ	xmmreg, xmmrm16	SSE41, SW
PMOVZWD	xmmreg, xmmrm64	SSE41
PMOVZXWQ	xmmreg, xmmrm32	SSE41, SD
PMOVZXDQ	xmmreg, xmmrm64	SSE41
PMULDQ	xmmreg, xmmrm128	SSE41
PMULLD	xmmreg, xmmrm128	SSE41
PTEST	xmmreg, xmmrm128	SSE41
ROUNDPD	xmmreg, xmmrm128, imm8	SSE41
ROUNDPS	xmmreg, xmmrm128, imm8	SSE41
ROUNDSD	xmmreg, xmmrm64, imm8	SSE41
ROUNDSS	xmmreg, xmmrm32, imm8	SSE41

### F.1.20 Nehalem New Instructions (SSE4.2)

CRC32	reg32, rm8	SSE42
CRC32	reg32, rm16	SSE42
CRC32	reg32, rm32	SSE42
CRC32	reg64, rm8	SSE42, X86_64, LONG
CRC32	reg64, rm64	SSE42, X86_64, LONG
PCPESTRI	xmmreg, xmmrm128, imm8	SSE42
PCPESTRM	xmmreg, xmmrm128, imm8	SSE42
PCMPISTRI	xmmreg, xmmrm128, imm8	SSE42
PCMPISTRM	xmmreg, xmmrm128, imm8	SSE42
PCMPGTQ	xmmreg, xmmrm128	SSE42
POPCNT	reg16, rm16	NEHALEM, SW
POPCNT	reg32, rm32	NEHALEM, SD
POPCNT	reg64, rm64	NEHALEM, LONG

### F.1.21 Intel SMX

GETSEC	KATMAI
--------	--------

### F.1.22 Geode (Cyrix) 3DNow! additions

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

### F.1.23 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

### F.1.24 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

### F.1.25 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

### F.1.26 Intel instruction extension based on pub number 319433-030 dated October 2017

VAESENC	ymmreg, ymmreg*, ymmrm256	VAES
VAESENCLAST	ymmreg, ymmreg*, ymmrm256	VAES
VAESDEC	ymmreg, ymmreg*, ymmrm256	VAES
VAESDECLAST	ymmreg, ymmreg*, ymmrm256	VAES
VAESENC	xmmreg, xmmreg*, xmmrm128	AVX512VL, VAES
VAESENC	ymmreg, ymmreg*, ymmrm256	AVX512VL, VAES
VAESENCLAST	xmmreg, xmmreg*, xmmrm128	AVX512VL, VAES
VAESENCLAST	ymmreg, ymmreg*, ymmrm256	AVX512VL, VAES
VAESDEC	xmmreg, xmmreg*, xmmrm128	AVX512VL, VAES
VAESDEC	ymmreg, ymmreg*, ymmrm256	AVX512VL, VAES
VAESDECLAST	xmmreg, xmmreg*, xmmrm128	AVX512VL, VAES
VAESDECLAST	ymmreg, ymmreg*, ymmrm256	AVX512VL, VAES
VAESENC	zmmreg, zmmreg*, zmmrm512	AVX512, VAES
VAESENCLAST	zmmreg, zmmreg*, zmmrm512	AVX512, VAES
VAESDEC	zmmreg, zmmreg*, zmmrm512	AVX512, VAES
VAESDECLAST	zmmreg, zmmreg*, zmmrm512	AVX512, VAES

### F.1.27 Intel AVX instructions

VADDPD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VADDPD	ymmreg, ymmreg*, ymmrm256	AVX, SANDYBRIDGE
VADDPDPS	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VADDPDPS	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
VBLENDVDP	xmmreg, xmmreg*, xmmrm128, xmmreg	AVX, SANDYBRIDGE
VBLENDVDP	ymmreg, ymmreg*, ymmrm256, ymmreg	AVX, SANDYBRIDGE
VBLENDVPS	xmmreg, xmmreg*, xmmrm128, xmmreg	AVX, SANDYBRIDGE
VBLENDVPS	ymmreg, ymmreg*, ymmrm256, ymmreg	AVX, SANDYBRIDGE
VBROADCASTSS	xmmreg, mem32	AVX, SANDYBRIDGE
VBROADCASTSS	ymmreg, mem32	AVX, SANDYBRIDGE
VBROADCASTSD	ymmreg, mem64	AVX, SANDYBRIDGE
VBROADCASTF128	ymmreg, mem128	AVX, SANDYBRIDGE
VCMPSEQ_OSPD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VCMPSEQ_OSPD	ymmreg, ymmreg*, ymmrm256	AVX, SANDYBRIDGE
VCMPQPD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VCMPQPD	ymmreg, ymmreg*, ymmrm256	AVX, SANDYBRIDGE





[illegible]





VCMPORD_SSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPSEQ_USSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPNGE_UQSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPNGT_UQSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPFALSE_OSSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPNEQ_OSSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPGE_OQSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPGT_OQSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPTRUE_USSS	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VCMPSS	xmmreg, xmmreg*, xmmrm64, 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
VCVTS2SI	reg32, xmmrm64	AVX, SANDYBRIDGE
VCVTS2SI	reg64, xmmrm64	AVX, SANDYBRIDGE, LONG
VCVTS2SS	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
VCVTSD2SI	reg32, xmmrm64	AVX, SANDYBRIDGE
VCVTSD2SI	reg64, xmmrm64	AVX, SANDYBRIDGE, LONG
VCVTSS2SI	reg32, xmmrm32	AVX, SANDYBRIDGE
VCVTSS2SI	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
VEEXTRACTF128	xmmrm128, ymmreg, imm8	AVX, SANDYBRIDGE
VEEXTRACTPS	rm32, xmmreg, imm8	AVX, SANDYBRIDGE
VHADDPD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VHADDPD	ymmreg, ymmreg*, ymmrm256	AVX, SANDYBRIDGE
VHADDPD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VHADDPD	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
VLDQQU	xmmreg, mem128	AVX, SANDYBRIDGE
VLDQQU	ymmreg, mem256	AVX, SANDYBRIDGE
VLDQQU	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, ymmreg, ymmreg	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
VMOVD	xmmreg, rm32	AVX, SANDYBRIDGE
VMOVD	rm32, xmmreg	AVX, SANDYBRIDGE
VMOVQ	xmmreg, xmmrm64	AVX, SANDYBRIDGE
VMOVQ	xmmrm64, xmmreg	AVX, SANDYBRIDGE
VMOVQ	xmmreg, rm64	AVX, SANDYBRIDGE, LONG
VMOVQ	rm64, xmmreg	AVX, SANDYBRIDGE, LONG
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, ymmrm256	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	mem256, 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, mem32	AVX, SANDYBRIDGE
VMOVSS	xmmreg, xmmreg*, xmmreg	AVX, SANDYBRIDGE
VMOVSS	mem32, 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
VPADD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPADDQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPADDQB	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPADDSDW	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
VPCPESTRI	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPCPESTRM	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPCPISTRI	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPCPISTRM	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPCPEQB	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPEQW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPEQD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPEQQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPGTB	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPGTW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPGTD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCPGTQ	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
VPERMILPS	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPERMILPS	ymmreg, ymmreg*, ymmrm256	AVX, SANDYBRIDGE
VPERMILPS	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPERMILPS	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	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
VPEXTRD	rm64, xmmreg, imm8	AVX, SANDYBRIDGE, LONG
VPHADDW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPHADDW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPHADDW	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
VPMINSW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPMINSW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPMINUB	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPMINUW	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPMINUD	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPMOVMASKB	reg64, xmmreg	AVX, SANDYBRIDGE, LONG
VPMOVMASKB	reg32, xmmreg	AVX, SANDYBRIDGE

VPMOVSXBW	xmmreg, xmmrm64	AVX, SANDYBRIDGE
VPMOVXBD	xmmreg, xmmrm32	AVX, SANDYBRIDGE
VPMOVXBDQ	xmmreg, xmmrm16	AVX, SANDYBRIDGE
VPMOVXBDQ	xmmreg, xmmrm64	AVX, SANDYBRIDGE
VPMOVXWQ	xmmreg, xmmrm32	AVX, SANDYBRIDGE
VPMOVXDQ	xmmreg, xmmrm64	AVX, SANDYBRIDGE
VPMOVZXBW	xmmreg, xmmrm64	AVX, SANDYBRIDGE
VPMOVZXBW	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
VPMULHRW	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
VPSADBQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPSHUFQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPSHUFQ	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPSHUFHW	xmmreg, xmmrm128, imm8	AVX, SANDYBRIDGE
VPSHUFHW	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



VRCPS	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
VSHUFDP	xmmreg, xmmreg*, xmmrm128, imm8	AVX, SANDYBRIDGE
VSHUFDP	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
VSQRTPSD	xmmreg, xmmreg*, xmmrm64	AVX, SANDYBRIDGE
VSQRTPSS	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

## F.1.28 Intel Carry-Less Multiplication instructions (CLMUL)

PCLMULLQLDQ	xmmreg, xmmrm128	SSE, WESTMERE
PCLMULHQLDQ	xmmreg, xmmrm128	SSE, WESTMERE
PCLMULLQHQQDQ	xmmreg, xmmrm128	SSE, WESTMERE
PCLMULHQHQQDQ	xmmreg, xmmrm128	SSE, WESTMERE
PCLMULQDQ	xmmreg, xmmrm128, imm8	SSE, WESTMERE

## F.1.29 Intel AVX Carry-Less Multiplication instructions (CLMUL)

VPCLMULLQLDQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCLMULHQLDQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCLMULLQHQQDQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCLMULHQHQQDQ	xmmreg, xmmreg*, xmmrm128	AVX, SANDYBRIDGE
VPCLMULQDQ	xmmreg, xmmreg*, xmmrm128, imm8	AVX, SANDYBRIDGE
VPCLMULLQLDQ	ymmreg, ymmreg*, ymmrm256	VPCLMULQDQ
VPCLMULHQLDQ	ymmreg, ymmreg*, ymmrm256	VPCLMULQDQ
VPCLMULLQHQQDQ	ymmreg, ymmreg*, ymmrm256	VPCLMULQDQ
VPCLMULHQHQQDQ	ymmreg, ymmreg*, ymmrm256	VPCLMULQDQ

VPCLMULQDQ	ymmreg, ymmreg*, ymmrm256, imm8	VPCLMULQDQ
VPCLMULLQLQDQ	xmmreg, xmmreg*, xmmrm128	AVX512VL, VPCLMULQDQ
VPCLMULHQLQDQ	xmmreg, xmmreg*, xmmrm128	AVX512VL, VPCLMULQDQ
VPCLMULLQHQQDQ	xmmreg, xmmreg*, xmmrm128	AVX512VL, VPCLMULQDQ
VPCLMULHQQHQQDQ	xmmreg, xmmreg*, xmmrm128	AVX512VL, VPCLMULQDQ
VPCLMULQDQ	xmmreg, xmmreg*, xmmrm128, imm8	AVX512VL, VPCLMULQDQ
VPCLMULLQLQDQ	ymmreg, ymmreg*, ymmrm256	AVX512VL, VPCLMULQDQ
VPCLMULHQLQDQ	ymmreg, ymmreg*, ymmrm256	AVX512VL, VPCLMULQDQ
VPCLMULLQHQQDQ	ymmreg, ymmreg*, ymmrm256	AVX512VL, VPCLMULQDQ
VPCLMULHQQHQQDQ	ymmreg, ymmreg*, ymmrm256	AVX512VL, VPCLMULQDQ
VPCLMULQDQ	ymmreg, ymmreg*, ymmrm256, imm8	AVX512VL, VPCLMULQDQ
VPCLMULLQLQDQ	zmmreg, zmmreg*, zmmrm512	AVX512, VPCLMULQDQ
VPCLMULHQLQDQ	zmmreg, zmmreg*, zmmrm512	AVX512, VPCLMULQDQ
VPCLMULLQHQQDQ	zmmreg, zmmreg*, zmmrm512	AVX512, VPCLMULQDQ
VPCLMULHQQHQQDQ	zmmreg, zmmreg*, zmmrm512	AVX512, VPCLMULQDQ
VPCLMULQDQ	zmmreg, zmmreg*, zmmrm512, imm8	AVX512, VPCLMULQDQ

### F.1.30 Intel Fused Multiply-Add instructions (FMA)

VFMADD132PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD132PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD132PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD132PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADD312PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD312PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD312PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD312PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADD213PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD213PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD213PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD213PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADD123PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD123PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD123PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD123PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADD231PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD231PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD231PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD231PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADD321PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADD321PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADD321PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADD321PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB132PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB132PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB132PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB132PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB312PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB312PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB312PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB312PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB213PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB213PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB213PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB213PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB123PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB123PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB123PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB123PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB231PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB231PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB231PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB231PD	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB321PS	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB321PS	ymmreg, ymmreg, ymmrm256	FMA
VFMADDSUB321PD	xmmreg, xmmreg, xmmrm128	FMA
VFMADDSUB321PD	ymmreg, ymmreg, ymmrm256	FMA

VFMSUB132PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB132PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB132PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB132PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB312PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB312PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB312PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB312PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB213PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB213PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB213PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB213PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB123PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB123PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB123PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB123PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB231PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB231PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB231PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB231PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB321PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB321PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUB321PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUB321PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD132PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD132PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD132PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD132PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD312PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD312PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD312PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD312PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD213PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD213PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD213PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD213PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD231PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD231PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD231PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD231PD	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD321PS	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD321PS	ymmreg, ymmreg, ymmrm256	FMA
VFMSUBADD321PD	xmmreg, xmmreg, xmmrm128	FMA
VFMSUBADD321PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD132PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD132PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD132PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD132PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD312PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD312PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD312PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD312PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD213PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD213PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD213PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD213PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD231PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD231PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD231PD	xmmreg, xmmreg, xmmrm128	FMA

VFNMADD231PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD321PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD321PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMADD321PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMADD321PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB132PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB132PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB132PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB132PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB312PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB312PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB312PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB312PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB213PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB213PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB213PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB213PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB123PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB123PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB123PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB123PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB231PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB231PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB231PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB231PD	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB321PS	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB321PS	ymmreg, ymmreg, ymmrm256	FMA
VFNMSUB321PD	xmmreg, xmmreg, xmmrm128	FMA
VFNMSUB321PD	ymmreg, ymmreg, ymmrm256	FMA
VFMAADD132SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD132SD	xmmreg, xmmreg, xmmrm64	FMA
VFMAADD312SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD312SD	xmmreg, xmmreg, xmmrm64	FMA
VFMAADD213SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD213SD	xmmreg, xmmreg, xmmrm64	FMA
VFMAADD123SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD123SD	xmmreg, xmmreg, xmmrm64	FMA
VFMAADD231SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD231SD	xmmreg, xmmreg, xmmrm64	FMA
VFMAADD321SS	xmmreg, xmmreg, xmmrm32	FMA
VFMAADD321SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB132SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB132SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB312SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB312SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB213SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB213SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB123SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB123SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB231SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB231SD	xmmreg, xmmreg, xmmrm64	FMA
VFMSUB321SS	xmmreg, xmmreg, xmmrm32	FMA
VFMSUB321SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD132SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD132SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD312SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD312SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD213SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD213SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD123SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD123SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD231SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD231SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMADD321SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMADD321SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMSUB132SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB132SD	xmmreg, xmmreg, xmmrm64	FMA

VFNMSUB312SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB312SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMSUB213SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB213SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMSUB123SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB123SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMSUB231SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB231SD	xmmreg, xmmreg, xmmrm64	FMA
VFNMSUB321SS	xmmreg, xmmreg, xmmrm32	FMA
VFNMSUB321SD	xmmreg, xmmreg, xmmrm64	FMA

### F.1.31 Intel post-32 nm processor instructions

RDFSBASE	reg32	LONG
RDFSBASE	reg64	LONG
RDGSBASE	reg32	LONG
RDGSBASE	reg64	LONG
RDRAND	reg16	
RDRAND	reg32	
RDRAND	reg64	LONG
WRFSBASE	reg32	LONG
WRFSBASE	reg64	LONG
WRGSBASE	reg32	LONG
WRGSBASE	reg64	LONG
VCVTPH2PS	ymmreg, xmmrm128	AVX
VCVTPH2PS	xmmreg, xmmrm64	AVX
VCVTPS2PH	xmmrm128, ymmreg, imm8	AVX
VCVTPS2PH	xmmrm64, xmmreg, imm8	AVX
ADCX	reg32, rm32	
ADCX	reg64, rm64	LONG
ADOX	reg32, rm32	
ADOX	reg64, rm64	LONG
RDSEED	reg16	
RDSEED	reg32	
RDSEED	reg64	LONG

### F.1.32 Supervisor Mode Access Prevention (SMAP)

CLAC	SMAP, PRIV
STAC	SMAP, PRIV

### F.1.33 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

### F.1.34 AMD Lightweight Profiling (LWP) instructions

LLWPCB	reg32	AMD, 386
LLWPCB	reg64	AMD, X86_64, LONG
SLWPCB	reg32	AMD, 386
SLWPCB	reg64	AMD, X86_64, LONG
LWPVAL	reg32, rm32, imm32	AMD, 386
LWPVAL	reg64, rm32, imm32	AMD, X86_64, LONG
LWPINS	reg32, rm32, imm32	AMD, 386
LWPINS	reg64, rm32, imm32	AMD, X86_64, LONG

### F.1.35 AMD XOP and FMA4 instructions (SSE5)

VFMADDPD	xmmreg, xmmreg*, xmmrm128, xmmreg	AMD, SSE5
VFMADDPD	ymmreg, ymmreg*, ymmrm256, ymmreg	AMD, SSE5

[illegible]

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
VPHADDDUBD	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDUBQ	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDUBW	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDUDQ	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDUWD	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDUWQ	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDWD	xmmreg, xmmrm128*	AMD, SSE5
VPHADDDWQ	xmmreg, xmmrm128*	AMD, SSE5
VPHSUBBW	xmmreg, xmmrm128*	AMD, SSE5
VPHSUBDQ	xmmreg, xmmrm128*	AMD, SSE5
VPHSUBWD	xmmreg, xmmrm128*	AMD, SSE5
VPMACSD	xmmreg, xmmreg*, xmmrm128, xmmreg	AMD, SSE5
VPMACSDQH	xmmreg, xmmreg*, xmmrm128, xmmreg	AMD, SSE5
VPMACSDQL	xmmreg, xmmreg*, xmmrm128, xmmreg	AMD, SSE5
VPMACSSD	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

### F.1.36 Intel AVX2 instructions

VMPSADBW	ymmreg, ymmreg*, ymmrm256, imm8	AVX2
VPABSB	ymmreg, ymmrm256	AVX2
VPABSW	ymmreg, ymmrm256	AVX2
VPABSD	ymmreg, ymmrm256	AVX2
VPACKSSWB	ymmreg, ymmreg*, ymmrm256	AVX2
VPACKSSDW	ymmreg, ymmreg*, ymmrm256	AVX2
VPACKUSDW	ymmreg, ymmreg*, ymmrm256	AVX2
VPACKUSWB	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDB	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDW	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDD	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDSB	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDUSB	ymmreg, ymmreg*, ymmrm256	AVX2
VPADDUSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPALIGNR	ymmreg, ymmreg*, ymmrm256, imm8	AVX2
VPAND	ymmreg, ymmreg*, ymmrm256	AVX2
VPANDN	ymmreg, ymmreg*, ymmrm256	AVX2
VPAVGB	ymmreg, ymmreg*, ymmrm256	AVX2
VPAVGW	ymmreg, ymmreg*, ymmrm256	AVX2
VPBLENDVB	ymmreg, ymmreg*, ymmrm256, ymmreg	AVX2
VPBLENDW	ymmreg, ymmreg*, ymmrm256, imm8	AVX2
VPCMPQB	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPQW	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPQD	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPQQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPGTB	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPGTW	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPGTD	ymmreg, ymmreg*, ymmrm256	AVX2
VPCMPGTQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPHADDW	ymmreg, ymmreg*, ymmrm256	AVX2
VPHADD	ymmreg, ymmreg*, ymmrm256	AVX2
VPHADDSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPHSUBW	ymmreg, ymmreg*, ymmrm256	AVX2
VPHSUBD	ymmreg, ymmreg*, ymmrm256	AVX2
VPHSUBSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMADDUBSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMADDWD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMASB	ymmreg, ymmreg*, ymmrm256	AVX2
VPMASW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMASD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMASUB	ymmreg, ymmreg*, ymmrm256	AVX2
VPMAXUW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMAXUD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINSB	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINSD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINUB	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINUW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMINUD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMOVMASKB	reg32, ymmreg	AVX2
VPMOVMASKB	reg64, ymmreg	AVX2
VPMOVSXBW	ymmreg, xmmrm128	AVX2
VPMOVSXBD	ymmreg, mem64	AVX2
VPMOVSXBD	ymmreg, xmmreg	AVX2
VPMOVSXBQ	ymmreg, mem32	AVX2
VPMOVSXBQ	ymmreg, xmmreg	AVX2
VPMOVSXWD	ymmreg, xmmrm128	AVX2
VPMOVSXWQ	ymmreg, mem64	AVX2
VPMOVSXWQ	ymmreg, xmmreg	AVX2



VPMOVSXDQ	ymmreg, xmmrm128	AVX2
VPMOVZXBW	ymmreg, xmmrm128	AVX2
VPMOVZXB	ymmreg, mem64	AVX2
VPMOVZXB	ymmreg, xmmreg	AVX2
VPMOVZXBQ	ymmreg, mem32	AVX2
VPMOVZXBQ	ymmreg, xmmreg	AVX2
VPMOVZXWD	ymmreg, xmmrm128	AVX2
VPMOVZXWQ	ymmreg, mem64	AVX2
VPMOVZXWQ	ymmreg, xmmreg	AVX2
VPMOVZXDQ	ymmreg, xmmrm128	AVX2
VPMULDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULHRSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULHUW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULHW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULLW	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULLD	ymmreg, ymmreg*, ymmrm256	AVX2
VPMULUDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPOR	ymmreg, ymmreg*, ymmrm256	AVX2
VPSADB	ymmreg, ymmreg*, ymmrm256	AVX2
VPSHUF	ymmreg, ymmreg*, ymmrm256	AVX2
VPSHUF	ymmreg, ymmrm256, imm8	AVX2
VPSHUFH	ymmreg, ymmrm256, imm8	AVX2
VPSHUFH	ymmreg, ymmrm256, imm8	AVX2
VPSIGN	ymmreg, ymmreg*, ymmrm256	AVX2
VPSIGN	ymmreg, ymmreg*, ymmrm256	AVX2
VPSIGN	ymmreg, ymmreg*, ymmrm256	AVX2
VPSLLDQ	ymmreg, ymmreg*, imm8	AVX2
VPSLLW	ymmreg, ymmreg*, xmmrm128	AVX2
VPSLLW	ymmreg, ymmreg*, imm8	AVX2
VPSLLD	ymmreg, ymmreg*, xmmrm128	AVX2
VPSLLD	ymmreg, ymmreg*, imm8	AVX2
VPSLLQ	ymmreg, ymmreg*, xmmrm128	AVX2
VPSLLQ	ymmreg, ymmreg*, imm8	AVX2
VPSRAW	ymmreg, ymmreg*, xmmrm128	AVX2
VPSRAW	ymmreg, ymmreg*, imm8	AVX2
VPSRAD	ymmreg, ymmreg*, xmmrm128	AVX2
VPSRAD	ymmreg, ymmreg*, imm8	AVX2
VPSRLDQ	ymmreg, ymmreg*, imm8	AVX2
VPSRLW	ymmreg, ymmreg*, xmmrm128	AVX2
VPSRLW	ymmreg, ymmreg*, imm8	AVX2
VPSRLD	ymmreg, ymmreg*, xmmrm128	AVX2
VPSRLD	ymmreg, ymmreg*, imm8	AVX2
VPSRLQ	ymmreg, ymmreg*, xmmrm128	AVX2
VPSRLQ	ymmreg, ymmreg*, imm8	AVX2
VPSUB	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUB	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBD	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBSB	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBUSB	ymmreg, ymmreg*, ymmrm256	AVX2
VPSUBUSW	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKHBW	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKHWD	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKHDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKHQDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKLBW	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKLWD	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKLDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPUNPCKLDQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPXOR	ymmreg, ymmreg*, ymmrm256	AVX2
VMOVNTDQA	ymmreg, mem256	AVX2
VBROADCASTSS	xmmreg, xmmreg	AVX2
VBROADCASTSS	ymmreg, xmmreg	AVX2
VBROADCASTSD	ymmreg, xmmreg	AVX2
VBROADCASTI128	ymmreg, mem128	AVX2
VPBLEND	xmmreg, xmmreg*, xmmrm128, imm8	AVX2

VPBLEND	ymmreg, ymmreg*, ymmrm256, imm8	AVX2
VPBROADCASTB	xmmreg, mem8	AVX2
VPBROADCASTB	xmmreg, xmmreg	AVX2
VPBROADCASTB	ymmreg, mem8	AVX2
VPBROADCASTB	ymmreg, xmmreg	AVX2
VPBROADCASTW	xmmreg, mem16	AVX2
VPBROADCASTW	xmmreg, xmmreg	AVX2
VPBROADCASTW	ymmreg, mem16	AVX2
VPBROADCASTW	ymmreg, xmmreg	AVX2
VPBROADCASTD	xmmreg, mem32	AVX2
VPBROADCASTD	xmmreg, xmmreg	AVX2
VPBROADCASTD	ymmreg, mem32	AVX2
VPBROADCASTD	ymmreg, xmmreg	AVX2
VPBROADCASTQ	xmmreg, mem64	AVX2
VPBROADCASTQ	xmmreg, xmmreg	AVX2
VPBROADCASTQ	ymmreg, mem64	AVX2
VPBROADCASTQ	ymmreg, xmmreg	AVX2
VPERMD	ymmreg, ymmreg*, ymmrm256	AVX2
VPERMPD	ymmreg, ymmrm256, imm8	AVX2
VPERMPS	ymmreg, ymmreg*, ymmrm256	AVX2
VPERMQ	ymmreg, ymmrm256, imm8	AVX2
VPERM2I128	ymmreg, ymmreg*, ymmrm256, imm8	AVX2
VEEXTRACTI128	xmmrm128, ymmreg, imm8	AVX2
VINSERTI128	ymmreg, ymmreg*, xmmrm128, imm8	AVX2
VPMASKMOVD	xmmreg, xmmreg*, mem128	AVX2
VPMASKMOVD	ymmreg, ymmreg*, mem256	AVX2
VPMASKMOVQ	xmmreg, xmmreg*, mem128	AVX2
VPMASKMOVQ	ymmreg, ymmreg*, mem256	AVX2
VPMASKMOVD	mem128, xmmreg*, xmmreg	AVX2
VPMASKMOVD	mem256, ymmreg*, ymmreg	AVX2
VPMASKMOVQ	mem128, xmmreg*, xmmreg	AVX2
VPMASKMOVQ	mem256, ymmreg*, ymmreg	AVX2
VPSLLVD	xmmreg, xmmreg*, xmmrm128	AVX2
VPSLLVQ	xmmreg, xmmreg*, xmmrm128	AVX2
VPSLLVD	ymmreg, ymmreg*, ymmrm256	AVX2
VPSLLVQ	ymmreg, ymmreg*, ymmrm256	AVX2
VPSRAVD	xmmreg, xmmreg*, xmmrm128	AVX2
VPSRAVD	ymmreg, ymmreg*, ymmrm256	AVX2
VPSRLVD	xmmreg, xmmreg*, xmmrm128	AVX2
VPSRLVQ	xmmreg, xmmreg*, xmmrm128	AVX2
VPSRLVD	ymmreg, ymmreg*, ymmrm256	AVX2
VPSRLVQ	ymmreg, ymmreg*, ymmrm256	AVX2
VGATHERDPD	xmmreg, xmem64, xmmreg	AVX2
VGATHERQPD	xmmreg, xmem64, xmmreg	AVX2
VGATHERDPD	ymmreg, xmem64, ymmreg	AVX2
VGATHERQPD	ymmreg, ymem64, ymmreg	AVX2
VGATHERDPS	xmmreg, xmem32, xmmreg	AVX2
VGATHERQPS	xmmreg, xmem32, xmmreg	AVX2
VGATHERDPS	ymmreg, ymem32, ymmreg	AVX2
VGATHERQPS	xmmreg, ymem32, xmmreg	AVX2
VPGATHERDD	xmmreg, xmem32, xmmreg	AVX2
VPGATHERQD	xmmreg, xmem32, xmmreg	AVX2
VPGATHERDD	ymmreg, ymem32, ymmreg	AVX2
VPGATHERQD	xmmreg, ymem32, xmmreg	AVX2
VPGATHERDQ	xmmreg, xmem64, xmmreg	AVX2
VPGATHERQQ	xmmreg, xmem64, xmmreg	AVX2
VPGATHERDQ	ymmreg, xmem64, ymmreg	AVX2
VPGATHERQQ	ymmreg, ymem64, ymmreg	AVX2

### F.1.37 Intel Transactional Synchronization Extensions (TSX)

XABORT	imm	RTM
XABORT	imm8	RTM
XBEGIN	imm	RTM
XBEGIN	imm   near	RTM, ND
XBEGIN	imm16	RTM, NOLONG
XBEGIN	imm16   near	RTM, NOLONG, ND

XBEGIN	imm32	RTM, NOLONG
XBEGIN	imm32 near	RTM, NOLONG, ND
XBEGIN	imm64	RTM, LONG
XBEGIN	imm64 near	RTM, LONG, ND
XEND		RTM
XTEST		HLE, RTM

### F.1.38 Intel BMI1 and BMI2 instructions, AMD TBM instructions

ANDN	reg32, reg32, rm32	BMI1
ANDN	reg64, reg64, rm64	LONG, BMI1
BEXTR	reg32, rm32, reg32	BMI1
BEXTR	reg64, rm64, reg64	LONG, BMI1
BEXTR	reg32, rm32, imm32	TBM
BEXTR	reg64, rm64, imm32	LONG, TBM
BLCI	reg32, rm32	TBM
BLCI	reg64, rm64	LONG, TBM
BLCIC	reg32, rm32	TBM
BLCIC	reg64, rm64	LONG, TBM
BLSI	reg32, rm32	BMI1
BLSI	reg64, rm64	LONG, BMI1
BLSIC	reg32, rm32	TBM
BLSIC	reg64, rm64	LONG, TBM
BLCFILL	reg32, rm32	TBM
BLCFILL	reg64, rm64	LONG, TBM
BLSFILL	reg32, rm32	TBM
BLSFILL	reg64, rm64	LONG, TBM
BLCMSK	reg32, rm32	TBM
BLCMSK	reg64, rm64	LONG, TBM
BLMSK	reg32, rm32	BMI1
BLMSK	reg64, rm64	LONG, BMI1
BLSR	reg32, rm32	BMI1
BLSR	reg64, rm64	LONG, BMI1
BLCS	reg32, rm32	TBM
BLCS	reg64, rm64	LONG, TBM
BZHI	reg32, rm32, reg32	BMI2
BZHI	reg64, rm64, reg64	LONG, BMI2
MULX	reg32, reg32, rm32	BMI2
MULX	reg64, reg64, rm64	LONG, BMI2
PDEP	reg32, reg32, rm32	BMI2
PDEP	reg64, reg64, rm64	LONG, BMI2
PEXT	reg32, reg32, rm32	BMI2
PEXT	reg64, reg64, rm64	LONG, BMI2
RORX	reg32, rm32, imm8	BMI2
RORX	reg64, rm64, imm8	LONG, BMI2
SARX	reg32, rm32, reg32	BMI2
SARX	reg64, rm64, reg64	LONG, BMI2
SHLX	reg32, rm32, reg32	BMI2
SHLX	reg64, rm64, reg64	LONG, BMI2
SHRX	reg32, rm32, reg32	BMI2
SHRX	reg64, rm64, reg64	LONG, BMI2
TZCNT	reg16, rm16	BMI1
TZCNT	reg32, rm32	BMI1
TZCNT	reg64, rm64	LONG, BMI1
TZMSK	reg32, rm32	TBM
TZMSK	reg64, rm64	LONG, TBM
T1MSKC	reg32, rm32	TBM
T1MSKC	reg64, rm64	LONG, TBM
PREFETCHWT1	mem8	PREFETCHWT1

### F.1.39 Intel Memory Protection Extensions (MPX)

BNDMK	bndreg, mem	MPX, MIB
BNDCL	bndreg, mem	MPX
BNDCL	bndreg, reg32	MPX, NOLONG
BNDCL	bndreg, reg64	MPX, LONG
BNDCL	bndreg, mem	MPX

BNDQU	bndreg, reg32	MPX, NO LONG
BNDQU	bndreg, reg64	MPX, LONG
BNDQCN	bndreg, mem	MPX
BNDQCN	bndreg, reg32	MPX, NO LONG
BNDQCN	bndreg, reg64	MPX, LONG
BNDMOV	bndreg, bndreg	MPX
BNDMOV	bndreg, mem	MPX
BNDMOV	bndreg, bndreg	MPX
BNDMOV	mem, bndreg	MPX
BNDLDX	bndreg, mem	MPX, MIB
BNDLDX	bndreg, mem, reg32	MPX, MIB, NO LONG
BNDLDX	bndreg, mem, reg64	MPX, MIB, LONG
BNDSTX	mem, bndreg	MPX, MIB
BNDSTX	mem, reg32, bndreg	MPX, MIB, NO LONG
BNDSTX	mem, reg64, bndreg	MPX, MIB, LONG
BNDSTX	mem, bndreg, reg32	MPX, MIB, NO LONG
BNDSTX	mem, bndreg, reg64	MPX, MIB, LONG

#### F.1.40 Intel SHA acceleration instructions

SHA1MSG1	xmmreg, xmmrm128	SHA
SHA1MSG2	xmmreg, xmmrm128	SHA
SHA1NEXTE	xmmreg, xmmrm128	SHA
SHA1RNDSD	xmmreg, xmmrm128, imm8	SHA
SHA256MSG1	xmmreg, xmmrm128	SHA
SHA256MSG2	xmmreg, xmmrm128	SHA
SHA256RNDSD	xmmreg, xmmrm128, xmm0	SHA
SHA256RNDSD	xmmreg, xmmrm128	SHA
VSHA512MSG1	ymmreg, ymmreg	SHA512, AVX
VSHA512MSG2	ymmreg, ymmreg	SHA512, AVX
VSHA512RNDSD	ymmreg, ymmreg, xmmreg	SHA512, AVX

#### F.1.41 SM3

VSM3MSG1	xmmreg, xmmreg, xmmreg	SM3, AVX
VSM3MSG2	xmmreg, xmmreg, xmmreg	SM3, AVX
VSM3RNDSD	xmmreg, xmmreg, xmmreg, imm8	SM3, AVX

#### F.1.42 SM4

VSM4KEY4	xmmreg, xmmreg, xmmrm128	SM4, AVX
VSM4KEY4	ymmreg, ymmreg, ymmrm128	SM4, AVX
VSM4RNDSD	xmmreg, xmmreg, xmmrm128	SM4, AVX
VSM4RNDSD	ymmreg, ymmreg, ymmrm128	SM4, AVX

#### F.1.43 AVX no exception conversions

VBCSTNEBF16PS	xmmreg, mem16	AVXNECONVERT, LATEVEX, SW
VBCSTNEBF16PS	ymmreg, mem16	AVXNECONVERT, LATEVEX, SW
VBCSTNESH2PS	xmmreg, mem16	AVXNECONVERT, LATEVEX, SW
VBCSTNESH2PS	ymmreg, mem16	AVXNECONVERT, LATEVEX, SW
VCVTNEEBF162PS	xmmreg, mem128	AVXNECONVERT, LATEVEX, SX
VCVTNEEBF162PS	ymmreg, mem256	AVXNECONVERT, LATEVEX, SY
VCVTNEEPH2PS	xmmreg, mem128	AVXNECONVERT, LATEVEX, SX
VCVTNEEPH2PS	ymmreg, mem256	AVXNECONVERT, LATEVEX, SY
VCVTNEOBF162PS	xmmreg, mem128	AVXNECONVERT, LATEVEX, SX
VCVTNEOBF162PS	ymmreg, mem256	AVXNECONVERT, LATEVEX, SY
VCVTNEOPH2PS	xmmreg, mem128	AVXNECONVERT, LATEVEX, SX
VCVTNEOPH2PS	ymmreg, mem256	AVXNECONVERT, LATEVEX, SY
VCVTNEPS2BF16	xmmreg, xmmrm128	AVXNECONVERT, LATEVEX, SX
VCVTNEPS2BF16	ymmreg, ymmrm256	AVXNECONVERT, LATEVEX, SY

#### F.1.44 AVX Vector Neural Network Instructions

VPDPBSSD	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBSSD	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY
VPDPBSSDS	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBSSDS	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY

VPDPBSUD	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBSUD	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY
VPDPBSUDS	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBSUDS	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY
VPDPBUUD	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBUUD	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY
VPDPBUUDS	xmmreg, xmmreg, xmmrm128	AVXVNNIINT8, LATEVEX, SX
VPDPBUUDS	ymmreg, ymmreg, ymmrm256	AVXVNNIINT8, LATEVEX, SY

#### F.1.45 AVX Integer Fused Multiply-Add

VPMADD52HUQ	xmmreg, xmmreg, xmmrm128	AVXIFMA, LATEVEX, SX
VPMADD52HUQ	ymmreg, ymmreg, ymmrm256	AVXIFMA, LATEVEX, SY
VPMADD52LUQ	xmmreg, xmmreg, xmmrm128	AVXIFMA, LATEVEX, SX
VPMADD52LUQ	ymmreg, ymmreg, ymmrm256	AVXIFMA, LATEVEX, SY

#### F.1.46 AVX-512 mask register instructions

KADDB	kreg, kreg, kreg	
KADDD	kreg, kreg, kreg	
KADDQ	kreg, kreg, kreg	
KADDW	kreg, kreg, kreg	
KANDB	kreg, kreg, kreg	
KANDD	kreg, kreg, kreg	
KANDNB	kreg, kreg, kreg	
KANDND	kreg, kreg, kreg	
KANDNQ	kreg, kreg, kreg	
KANDNW	kreg, kreg, kreg	
KANDQ	kreg, kreg, kreg	
KANDW	kreg, kreg, kreg	
KMOVB	kreg, krm8	
KMOVB	mem8, kreg	
KMOVB	kreg, reg32	
KMOVB	kreg, reg8	ND
KMOVB	reg32, kreg	
KMOVD	kreg, krm32	
KMOVD	mem32, kreg	
KMOVD	kreg, reg32	
KMOVD	reg32, kreg	
KMOVQ	kreg, krm64	
KMOVQ	mem64, kreg	
KMOVQ	kreg, reg64	
KMOVQ	reg64, kreg	
KMOVW	kreg, krm16	
KMOVW	mem16, kreg	
KMOVW	kreg, reg32	
KMOVW	kreg, reg16	ND
KMOVW	reg32, kreg	
KNOTB	kreg, kreg	
KNOTD	kreg, kreg	
KNOTQ	kreg, kreg	
KNOTW	kreg, kreg	
KORB	kreg, kreg, kreg	
KORD	kreg, kreg, kreg	
KORQ	kreg, kreg, kreg	
KORW	kreg, kreg, kreg	
KORTESTB	kreg, kreg	
KORTESTD	kreg, kreg	
KORTESTQ	kreg, kreg	
KORTESTW	kreg, kreg	
KSHIFTLB	kreg, kreg, imm8	
KSHIFTLB	kreg, kreg, imm8	
KSHIFTLQ	kreg, kreg, imm8	
KSHIFTLW	kreg, kreg, imm8	
KSHIFTRB	kreg, kreg, imm8	
KSHIFTRD	kreg, kreg, imm8	
KSHIFTRQ	kreg, kreg, imm8	

KSHIFTRW	kreg, kreg, imm8
KTESTB	kreg, kreg
KTESTD	kreg, kreg
KTESTQ	kreg, kreg
KTESTW	kreg, kreg
KUNPCKBW	kreg, kreg, kreg
KUNPCKDQ	kreg, kreg, kreg
KUNPCKWD	kreg, kreg, kreg
KXNORB	kreg, kreg, kreg
KXNORD	kreg, kreg, kreg
KXNORQ	kreg, kreg, kreg
KXNORW	kreg, kreg, kreg
KXORB	kreg, kreg, kreg
KXORD	kreg, kreg, kreg
KXORQ	kreg, kreg, kreg
KXORW	kreg, kreg, kreg

#### F.1.47 AVX-512 mask register instructions (aliases requiring explicit size support)

KADD	kreg8, kreg, kreg	ND
KADD	kreg32, kreg, kreg	ND
KADD	kreg64, kreg, kreg	ND
KADD	kreg16, kreg, kreg	ND
KAND	kreg8, kreg, kreg	ND
KAND	kreg32, kreg, kreg	ND
KANDN	kreg64, kreg, kreg	ND
KANDN	kreg16, kreg, kreg	ND
KANDN	kreg8, kreg, kreg	ND
KANDN	kreg32, kreg, kreg	ND
KAND	kreg64, kreg, kreg	ND
KAND	kreg16, kreg, kreg	ND
KMOV	kreg8, krm8	ND
KMOV	mem8, kreg8	ND
KMOV	kreg8, reg32	ND, SX
KMOV	kreg8, reg8	ND
KMOV	reg32, kreg8	ND, SX
KMOV	kreg32, krm32	ND
KMOV	mem32, kreg32	ND
KMOV	kreg32, reg32	ND
KMOV	reg32, kreg32	ND
KMOV	kreg64, krm64	ND
KMOV	mem64, kreg64	ND
KMOV	kreg64, reg64	ND
KMOV	reg64, kreg64	ND
KMOV	kreg16, krm16	ND
KMOV	mem16, kreg16	ND
KMOV	kreg16, reg32	ND, SX
KMOV	reg32, kreg16	ND, SX
KMOV	kreg16, reg32	ND, SX
KMOV	kreg16, reg16	ND
KNOT	kreg8, kreg8	ND
KNOT	kreg32, kreg32	ND
KNOT	kreg64, kreg64	ND
KNOT	kreg16, kreg16	ND
KOR	kreg8, kreg, kreg	ND
KOR	kreg32, kreg, kreg	ND
KOR	kreg64, kreg, kreg	ND
KOR	kreg16, kreg, kreg	ND
KORTEST	kreg8, kreg	ND
KORTEST	kreg32, kreg	ND
KORTEST	kreg64, kreg	ND
KORTEST	kreg16, kreg	ND
KSHIFTL	kreg8, kreg, imm8	ND
KSHIFTL	kreg32, kreg, imm8	ND
KSHIFTL	kreg64, kreg, imm8	ND
KSHIFTL	kreg16, kreg, imm8	ND
KSHIFTR	kreg8, kreg, imm8	ND

KSHIFTR	kreg32, kreg, imm8	ND
KSHIFTR	kreg64, kreg, imm8	ND
KSHIFTR	kreg16, kreg, imm8	ND
KTEST	kreg8, kreg	ND
KTEST	kreg32, kreg	ND
KTEST	kreg64, kreg	ND
KTEST	kreg16, kreg	ND
KUNPCK	kreg16, kreg8, kreg8	ND
KUNPCK	kreg64, kreg32, kreg32	ND
KUNPCK	kreg32, kreg16, kreg16	ND
KXNOR	kreg8, kreg, kreg	ND
KXNOR	kreg32, kreg, kreg	ND
KXNOR	kreg64, kreg, kreg	ND
KXNOR	kreg16, kreg, kreg	ND
KXOR	kreg8, kreg, kreg	ND
KXOR	kreg32, kreg, kreg	ND
KXOR	kreg64, kreg, kreg	ND
KXOR	kreg16, kreg, kreg	ND

## F.1.48 AVX-512 instructions

VADDPD	xmmreg mask z, xmmreg*, xmrm128 b64	AVX512VL
VADDPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VADDPD	zmmreg mask z, zmmreg*, zmmrm512 b64 er	AVX512
VADDPS	xmmreg mask z, xmmreg*, xmrm128 b32	AVX512VL
VADDPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VADDPS	zmmreg mask z, zmmreg*, zmmrm512 b32 er	AVX512
VADDSD	xmmreg mask z, xmmreg*, xmrm64 er	AVX512
VADDSS	xmmreg mask z, xmmreg*, xmrm32 er	AVX512
VALIGND	xmmreg mask z, xmmreg*, xmrm128 b32, imm8	AVX512VL
VALIGND	ymmreg mask z, ymmreg*, ymmrm256 b32, imm8	AVX512VL
VALIGND	zmmreg mask z, zmmreg*, zmmrm512 b32, imm8	AVX512
VALIGNQ	xmmreg mask z, xmmreg*, xmrm128 b64, imm8	AVX512VL
VALIGNQ	ymmreg mask z, ymmreg*, ymmrm256 b64, imm8	AVX512VL
VALIGNQ	zmmreg mask z, zmmreg*, zmmrm512 b64, imm8	AVX512
VANDNPD	xmmreg mask z, xmmreg*, xmrm128 b64	AVX512VL/DQ
VANDNPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL/DQ
VANDNPD	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512DQ
VANDNPS	xmmreg mask z, xmmreg*, xmrm128 b32	AVX512VL/DQ
VANDNPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL/DQ
VANDNPS	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512DQ
VANDPD	xmmreg mask z, xmmreg*, xmrm128 b64	AVX512VL/DQ
VANDPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL/DQ
VANDPD	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512DQ
VANDPS	xmmreg mask z, xmmreg*, xmrm128 b32	AVX512VL/DQ
VANDPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL/DQ
VANDPS	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512DQ
VBLENDMPD	xmmreg mask z, xmmreg, xmrm128 b64	AVX512VL
VBLENDMPD	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL
VBLENDMPD	zmmreg mask z, zmmreg, zmmrm512 b64	AVX512
VBLENDMPS	xmmreg mask z, xmmreg, xmrm128 b32	AVX512VL
VBLENDMPS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VBLENDMPS	zmmreg mask z, zmmreg, zmmrm512 b32	AVX512
VBROADCASTF32X2	ymmreg mask z, xmrm64	AVX512VL/DQ
VBROADCASTF32X2	zmmreg mask z, xmrm64	AVX512DQ
VBROADCASTF32X4	ymmreg mask z, mem128	AVX512VL
VBROADCASTF32X4	zmmreg mask z, mem128	AVX512
VBROADCASTF32X8	zmmreg mask z, mem256	AVX512DQ
VBROADCASTF64X2	ymmreg mask z, mem128	AVX512VL/DQ
VBROADCASTF64X2	zmmreg mask z, mem128	AVX512DQ
VBROADCASTF64X4	zmmreg mask z, mem256	AVX512
VBROADCASTI32X2	xmmreg mask z, xmrm64	AVX512VL/DQ
VBROADCASTI32X2	ymmreg mask z, xmrm64	AVX512VL/DQ
VBROADCASTI32X2	zmmreg mask z, xmrm64	AVX512DQ
VBROADCASTI32X4	ymmreg mask z, mem128	AVX512VL
VBROADCASTI32X4	zmmreg mask z, mem128	AVX512
VBROADCASTI32X8	zmmreg mask z, mem256	AVX512DQ

VBROADCASTI64X2	ymmreg mask z,mem128	AVX512VL/DQ
VBROADCASTI64X2	zmmreg mask z,mem128	AVX512DQ
VBROADCASTI64X4	zmmreg mask z,mem256	AVX512
VBROADCASTSD	ymmreg mask z,mem64	AVX512VL
VBROADCASTSD	zmmreg mask z,mem64	AVX512
VBROADCASTSD	ymmreg mask z,xmmreg	AVX512VL
VBROADCASTSD	zmmreg mask z,xmmreg	AVX512
VBROADCASTSS	xmmreg mask z,mem32	AVX512VL
VBROADCASTSS	ymmreg mask z,mem32	AVX512VL
VBROADCASTSS	zmmreg mask z,mem32	AVX512
VBROADCASTSS	xmmreg mask z,xmmreg	AVX512VL
VBROADCASTSS	ymmreg mask z,xmmreg	AVX512VL
VBROADCASTSS	zmmreg mask z,xmmreg	AVX512
VCMPPEQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPPEQPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPPEQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPPEQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPPEQPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPPEQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPPEQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPPEQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPPEQ_OQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPPEQ_OQPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPPEQ_OQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPPEQ_OQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPPEQ_OQPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPPEQ_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPPEQ_OQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPPEQ_OQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPPLTPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPPLTPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPPLTPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPPLTPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPPLTPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPPLTPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPPLTSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPPLTSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPPLT_OSPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPPLT_OSPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPPLT_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPPLT_OSPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPPLT_OSPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPPLT_OSPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPPLT_OSSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPPLT_OSSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPLEPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPLEPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPLEPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPLEPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPLEPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPLEPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPLESD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPLESS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPLE_OSPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPLE_OSPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPLE_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPLE_OSPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPLE_OSPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPLE_OSPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPLE_OSSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPLE_OSSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPUNORDPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPUNORDPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPUNORDPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPUNORDPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPUNORDPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPUNORDPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512



VCMPUNORDSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPUNORDSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPUNORD_QPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPUNORD_QPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPUNORD_QPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPUNORD_QPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPUNORD_QPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPUNORD_QPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPUNORD_QSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPUNORD_QSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNEQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNEQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNEQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNEQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNEQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNEQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNEQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNEQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNEQ_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNEQ_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNEQ_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNEQ_UQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNEQ_UQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNEQ_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNEQ_UQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNEQ_UQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNLTPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNLTPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNLTPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNLTPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNLTPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNLTPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNLTS	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNLTS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNL_USPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNL_USPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNL_USPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNL_USPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNL_USPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNL_USPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNL_USSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNL_USSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNLEPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNLEPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNLEPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNLEPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNLEPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNLEPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNLESD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNLESS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNLE_USPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNLE_USPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNLE_USPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNLE_USPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNLE_USPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNLE_USPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNLE_USSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNLE_USSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPORDPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPORDPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPORDPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPORDPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPORDPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPORDPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPORDSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPORDSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPORD_QPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL

VCMPORD_QPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPORD_QPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPORD_QPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPORD_QPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPORD_QPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPORD_QSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPORD_QSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPSEQ_UQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPSEQ_UQPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPSEQ_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPSEQ_UQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPSEQ_UQPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPSEQ_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPSEQ_UQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPSEQ_UQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNGEPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNGEPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPNGEPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNGEPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNGEPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPNGEPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNGESD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNGESS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNGE_USPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNGE_USPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPNGE_USPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNGE_USPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNGE_USPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPNGE_USPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNGE_USSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNGE_USSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNGTPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNGTPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPNGTPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNGTPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNGTPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPNGTPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNGTSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNGTSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNGT_USPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNGT_USPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPNGT_USPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNGT_USPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNGT_USPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPNGT_USPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNGT_USSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNGT_USSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPFALSEPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPFALSEPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPFALSEPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPFALSEPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPFALSEPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPFALSEPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPFALSESD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPFALSESS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPFALSE_QQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPFALSE_QQPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPFALSE_QQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPFALSE_QQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPFALSE_QQPS	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VCMPFALSE_QQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPFALSE_QQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPFALSE_QQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNEQ_QQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNEQ_QQPD	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VCMPNEQ_QQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNEQ_QQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL

VCMPNEQ_OQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNEQ_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNEQ_OQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNEQ_OQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPGEPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPGEPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPGEPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPGEPs	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPGEPs	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPGEPs	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPGESD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPGESS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPGE_OSPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPGE_OSPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPGE_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPGE_OSPs	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPGE_OSPs	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPGE_OSPs	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPGE_OSSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPGE_OSSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPGTPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPGTPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPGTPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPGTPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPGTPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPGTPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPGTSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPGTSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPGT_OSPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPGT_OSPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPGT_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPGT_OSPs	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPGT_OSPs	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPGT_OSPs	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPGT_OSSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPGT_OSSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPTRUEPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPTRUEPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPTRUEPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPTRUEPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPTRUEPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPTRUEPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPTRUESD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPTRUESS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPTRUE_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPTRUE_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPTRUE_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPTRUE_UQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPTRUE_UQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPTRUE_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPTRUE_UQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPTRUE_UQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPSEQ_OSPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPSEQ_OSPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPSEQ_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPSEQ_OSPs	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPSEQ_OSPs	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPSEQ_OSPs	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPSEQ_OSSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPSEQ_OSSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPLT_OQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPLT_OQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPLT_OQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPLT_OQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPLT_OQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPLT_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPLT_OQSD	kreg mask,xmmreg,xmmrm64 sae AVX512

VCMPLT_OQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPLE_OQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPLE_OQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPLE_OQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPLE_OQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPLE_OQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPLE_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPLE_OQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPLE_OQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPUNORD_SPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPUNORD_SPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPUNORD_SPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPUNORD_SPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPUNORD_SPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPUNORD_SPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPUNORD_SSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPUNORD_SSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNEQ_USPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNEQ_USPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNEQ_USPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNEQ_USPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNEQ_USPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNEQ_USPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNEQ_USSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNEQ_USSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNLT_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNLT_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNLT_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNLT_UQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNLT_UQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNLT_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNLT_UQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNLT_UQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNLE_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNLE_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNLE_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNLE_UQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNLE_UQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNLE_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNLE_UQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNLE_UQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPORD_SPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPORD_SPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPORD_SPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPORD_SPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPORD_SPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPORD_SPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPORD_SSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPORD_SSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPPEQ_USPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPPEQ_USPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPPEQ_USPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPPEQ_USPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPPEQ_USPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPPEQ_USPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPPEQ_USSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPPEQ_USSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNGE_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNGE_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VCMPNGE_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae AVX512
VCMPNGE_UQPS	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VCMPNGE_UQPS	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VCMPNGE_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae AVX512
VCMPNGE_UQSD	kreg mask,xmmreg,xmmrm64 sae AVX512
VCMPNGE_UQSS	kreg mask,xmmreg,xmmrm32 sae AVX512
VCMPNGT_UQPD	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VCMPNGT_UQPD	kreg mask,ymmreg,ymmrm256 b64 AVX512VL

VCMPNGT_UQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNGT_UQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNGT_UQPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPNGT_UQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNGT_UQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNGT_UQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPFALSE_OSPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPFALSE_OSPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPFALSE_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPFALSE_OSPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPFALSE_OSPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPFALSE_OSPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPFALSE_OSSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPFALSE_OSSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPNEQ_OSPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPNEQ_OSPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPNEQ_OSPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPNEQ_OSPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPNEQ_OSPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPNEQ_OSPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPNEQ_OSSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPNEQ_OSSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPGE_OQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPGE_OQPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPGE_OQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPGE_OQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPGE_OQPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPGE_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPGE_OQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPGE_OQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPGT_OQPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPGT_OQPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPGT_OQPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPGT_OQPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPGT_OQPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPGT_OQPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPGT_OQSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPGT_OQSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPTRUE_USPD	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VCMPTRUE_USPD	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VCMPTRUE_USPD	kreg mask,zmmreg,zmmrm512 b64 sae	AVX512
VCMPTRUE_USPS	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VCMPTRUE_USPS	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VCMPTRUE_USPS	kreg mask,zmmreg,zmmrm512 b32 sae	AVX512
VCMPTRUE_USSD	kreg mask,xmmreg,xmmrm64 sae	AVX512
VCMPTRUE_USSS	kreg mask,xmmreg,xmmrm32 sae	AVX512
VCMPPD	kreg mask,xmmreg,xmmrm128 b64,imm8	AVX512VL
VCMPPD	kreg mask,ymmreg,ymmrm256 b64,imm8	AVX512VL
VCMPPD	kreg mask,zmmreg,zmmrm512 b64 sae,imm8	AVX512
VCMPPS	kreg mask,xmmreg,xmmrm128 b32,imm8	AVX512VL
VCMPPS	kreg mask,ymmreg,ymmrm256 b32,imm8	AVX512VL
VCMPPS	kreg mask,zmmreg,zmmrm512 b32 sae,imm8	AVX512
VCMPSD	kreg mask,xmmreg,xmmrm64 sae,imm8	AVX512
VCMPSS	kreg mask,xmmreg,xmmrm32 sae,imm8	AVX512
VCOMISD	xmmreg,xmmrm64 sae	AVX512
VCOMISS	xmmreg,xmmrm32 sae	AVX512
VCOMPRESSPD	mem128 mask,xmmreg	AVX512VL
VCOMPRESSPD	mem256 mask,ymmreg	AVX512VL
VCOMPRESSPD	mem512 mask,zmmreg	AVX512
VCOMPRESSPD	xmmreg mask z,xmmreg	AVX512VL
VCOMPRESSPD	ymmreg mask z,ymmreg	AVX512VL
VCOMPRESSPD	zmmreg mask z,zmmreg	AVX512
VCOMPRESSPS	mem128 mask,xmmreg	AVX512VL
VCOMPRESSPS	mem256 mask,ymmreg	AVX512VL
VCOMPRESSPS	mem512 mask,zmmreg	AVX512
VCOMPRESSPS	xmmreg mask z,xmmreg	AVX512VL
VCOMPRESSPS	ymmreg mask z,ymmreg	AVX512VL

VCOMPRESSPS	zmmreg mask z, zmmreg	AVX512
VCVTDQ2PD	xmmreg mask z, xmmrm64 b32	AVX512VL
VCVTDQ2PD	ymmreg mask z, xmmrm128 b32	AVX512VL
VCVTDQ2PD	zmmreg mask z, ymmrm256 b32 er	AVX512
VCVTDQ2PS	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTDQ2PS	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTDQ2PS	zmmreg mask z, zmmrm512 b32 er	AVX512
VCVTPD2DQ	xmmreg mask z, xmmrm128 b64	AVX512VL
VCVTPD2DQ	xmmreg mask z, ymmrm256 b64	AVX512VL
VCVTPD2DQ	ymmreg mask z, zmmrm512 b64 er	AVX512
VCVTPD2PS	xmmreg mask z, xmmrm128 b64	AVX512VL
VCVTPD2PS	xmmreg mask z, ymmrm256 b64	AVX512VL
VCVTPD2PS	ymmreg mask z, zmmrm512 b64 er	AVX512
VCVTPD2QQ	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTPD2QQ	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTPD2QQ	zmmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTPD2UDQ	xmmreg mask z, xmmrm128 b64	AVX512VL
VCVTPD2UDQ	xmmreg mask z, ymmrm256 b64	AVX512VL
VCVTPD2UDQ	ymmreg mask z, zmmrm512 b64 er	AVX512
VCVTPD2UQQ	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTPD2UQQ	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTPD2UQQ	zmmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTPH2PS	xmmreg mask z, xmmrm64	AVX512VL
VCVTPH2PS	ymmreg mask z, xmmrm128	AVX512VL
VCVTPH2PS	zmmreg mask z, ymmrm256 sae	AVX512
VCVTPS2DQ	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTPS2DQ	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTPS2DQ	zmmreg mask z, zmmrm512 b32 er	AVX512
VCVTPS2PD	xmmreg mask z, xmmrm64 b32	AVX512VL
VCVTPS2PD	ymmreg mask z, xmmrm128 b32	AVX512VL
VCVTPS2PD	zmmreg mask z, ymmrm256 b32 sae	AVX512
VCVTPS2PH	xmmreg mask z, xmmreg, imm8	AVX512VL
VCVTPS2PH	xmmreg mask z, ymmreg, imm8	AVX512VL
VCVTPS2PH	ymmreg mask z, zmmreg sae, imm8	AVX512
VCVTPS2PH	mem64 mask, xmmreg, imm8	AVX512VL
VCVTPS2PH	mem128 mask, ymmreg, imm8	AVX512VL
VCVTPS2PH	mem256 mask, zmmreg sae, imm8	AVX512
VCVTPS2QQ	xmmreg mask z, xmmrm64 b32	AVX512VL/DQ
VCVTPS2QQ	ymmreg mask z, xmmrm128 b32	AVX512VL/DQ
VCVTPS2QQ	zmmreg mask z, ymmrm256 b32 er	AVX512DQ
VCVTPS2UDQ	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTPS2UDQ	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTPS2UDQ	zmmreg mask z, zmmrm512 b32 er	AVX512
VCVTPS2UQQ	xmmreg mask z, xmmrm64 b32	AVX512VL/DQ
VCVTPS2UQQ	ymmreg mask z, xmmrm128 b32	AVX512VL/DQ
VCVTPS2UQQ	zmmreg mask z, ymmrm256 b32 er	AVX512DQ
VCVTQQ2PD	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTQQ2PD	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTQQ2PD	zmmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTQQ2PS	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTQQ2PS	xmmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTQQ2PS	ymmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTSD2SI	reg32, xmmrm64 er	AVX512
VCVTSD2SI	reg64, xmmrm64 er	AVX512
VCVTSD2SS	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VCVTSD2USI	reg32, xmmrm64 er	AVX512
VCVTSD2USI	reg64, xmmrm64 er	AVX512
VCVTSI2SD	xmmreg, xmmreg er, rm32	AVX512
VCVTSI2SD	xmmreg, xmmreg er, rm64	AVX512
VCVTSI2SS	xmmreg, xmmreg er, rm32	AVX512
VCVTSI2SS	xmmreg, xmmreg er, rm64	AVX512
VCVTSS2SD	xmmreg mask z, xmmreg, xmmrm32 sae	AVX512
VCVTSS2SI	reg32, xmmrm32 er	AVX512
VCVTSS2SI	reg64, xmmrm32 er	AVX512
VCVTSS2USI	reg32, xmmrm32 er	AVX512
VCVTSS2USI	reg64, xmmrm32 er	AVX512
VCVTPD2DQ	xmmreg mask z, xmmrm128 b64	AVX512VL

VCVTTDPD2DQ	xmmreg mask z, ymmrm256 b64	AVX512VL
VCVTTDPD2DQ	ymmreg mask z, zmmrm512 b64 sae	AVX512
VCVTTDPD2QQ	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTTDPD2QQ	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTTDPD2QQ	zmmreg mask z, zmmrm512 b64 sae	AVX512DQ
VCVTTDPD2UDQ	xmmreg mask z, xmmrm128 b64	AVX512VL
VCVTTDPD2UDQ	xmmreg mask z, ymmrm256 b64	AVX512VL
VCVTTDPD2UDQ	ymmreg mask z, zmmrm512 b64 sae	AVX512
VCVTTDPD2UQQ	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTTDPD2UQQ	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTTDPD2UQQ	zmmreg mask z, zmmrm512 b64 sae	AVX512DQ
VCVTTPS2DQ	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTTPS2DQ	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTTPS2DQ	zmmreg mask z, zmmrm512 b32 sae	AVX512
VCVTTPS2QQ	xmmreg mask z, xmmrm64 b32	AVX512VL/DQ
VCVTTPS2QQ	ymmreg mask z, xmmrm128 b32	AVX512VL/DQ
VCVTTPS2QQ	zmmreg mask z, ymmrm256 b32 sae	AVX512DQ
VCVTTPS2UDQ	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTTPS2UDQ	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTTPS2UDQ	zmmreg mask z, zmmrm512 b32 sae	AVX512
VCVTTPS2UQQ	xmmreg mask z, xmmrm64 b32	AVX512VL/DQ
VCVTTPS2UQQ	ymmreg mask z, xmmrm128 b32	AVX512VL/DQ
VCVTTPS2UQQ	zmmreg mask z, ymmrm256 b32 sae	AVX512DQ
VCVTSSD2SI	reg32, xmmrm64 sae	AVX512
VCVTSSD2SI	reg64, xmmrm64 sae	AVX512
VCVTSSD2USI	reg32, xmmrm64 sae	AVX512
VCVTSSD2USI	reg64, xmmrm64 sae	AVX512
VCVTSS2SI	reg32, xmmrm32 sae	AVX512
VCVTSS2SI	reg64, xmmrm32 sae	AVX512
VCVTSS2USI	reg32, xmmrm32 sae	AVX512
VCVTSS2USI	reg64, xmmrm32 sae	AVX512
VCVTUDQ2PD	xmmreg mask z, xmmrm64 b32	AVX512VL
VCVTUDQ2PD	ymmreg mask z, xmmrm128 b32	AVX512VL
VCVTUDQ2PD	zmmreg mask z, ymmrm256 b32 er	AVX512
VCVTUDQ2PS	xmmreg mask z, xmmrm128 b32	AVX512VL
VCVTUDQ2PS	ymmreg mask z, ymmrm256 b32	AVX512VL
VCVTUDQ2PS	zmmreg mask z, zmmrm512 b32 er	AVX512
VCVTUQQ2PD	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTUQQ2PD	ymmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTUQQ2PD	zmmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTUQQ2PS	xmmreg mask z, xmmrm128 b64	AVX512VL/DQ
VCVTUQQ2PS	xmmreg mask z, ymmrm256 b64	AVX512VL/DQ
VCVTUQQ2PS	ymmreg mask z, zmmrm512 b64 er	AVX512DQ
VCVTUSI2SD	xmmreg, xmmreg er, rm32	AVX512
VCVTUSI2SD	xmmreg, xmmreg er, rm64	AVX512
VCVTUSI2SS	xmmreg, xmmreg er, rm32	AVX512
VCVTUSI2SS	xmmreg, xmmreg er, rm64	AVX512
VDBPSADBW	xmmreg mask z, xmmreg*, xmmrm128, imm8	AVX512VL/BW
VDBPSADBW	ymmreg mask z, ymmreg*, ymmrm256, imm8	AVX512VL/BW
VDBPSADBW	zmmreg mask z, zmmreg*, zmmrm512, imm8	AVX512BW
VDIVPD	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VDIVPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VDIVPD	zmmreg mask z, zmmreg*, zmmrm512 b64 er	AVX512
VDIVPS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VDIVPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VDIVPS	zmmreg mask z, zmmreg*, zmmrm512 b32 er	AVX512
VDIVSD	xmmreg mask z, xmmreg*, xmmrm64 er	AVX512
VDIVSS	xmmreg mask z, xmmreg*, xmmrm32 er	AVX512
VEXP2PD	zmmreg mask z, zmmrm512 b64 sae	AVX512ER
VEXP2PS	zmmreg mask z, zmmrm512 b32 sae	AVX512ER
VEXPANDPD	xmmreg mask z, mem128	AVX512VL
VEXPANDPD	ymmreg mask z, mem256	AVX512VL
VEXPANDPD	zmmreg mask z, mem512	AVX512
VEXPANDPD	xmmreg mask z, xmmreg	AVX512VL
VEXPANDPD	ymmreg mask z, ymmreg	AVX512VL
VEXPANDPD	zmmreg mask z, zmmreg	AVX512
VEXPANDPS	xmmreg mask z, mem128	AVX512VL

VEXPANDPS	ymmreg mask z,mem256	AVX512VL
VEXPANDPS	zmmreg mask z,mem512	AVX512
VEXPANDPS	xmmreg mask z,xmmreg	AVX512VL
VEXPANDPS	ymmreg mask z,ymmreg	AVX512VL
VEXPANDPS	zmmreg mask z,zmmreg	AVX512
VEXTRACTF32X4	xmmreg mask z,ymmreg,imm8	AVX512VL
VEXTRACTF32X4	xmmreg mask z,zmmreg,imm8	AVX512
VEXTRACTF32X4	mem128 mask,ymmreg,imm8	AVX512VL
VEXTRACTF32X4	mem128 mask,zmmreg,imm8	AVX512
VEXTRACTF32X8	ymmreg mask z,zmmreg,imm8	AVX512DQ
VEXTRACTF32X8	mem256 mask,zmmreg,imm8	AVX512DQ
VEXTRACTF64X2	xmmreg mask z,ymmreg,imm8	AVX512VL/DQ
VEXTRACTF64X2	xmmreg mask z,zmmreg,imm8	AVX512DQ
VEXTRACTF64X2	mem128 mask,ymmreg,imm8	AVX512VL/DQ
VEXTRACTF64X2	mem128 mask,zmmreg,imm8	AVX512DQ
VEXTRACTF64X4	ymmreg mask z,zmmreg,imm8	AVX512
VEXTRACTF64X4	mem256 mask,zmmreg,imm8	AVX512
VEXTRACTI32X4	xmmreg mask z,ymmreg,imm8	AVX512VL
VEXTRACTI32X4	xmmreg mask z,zmmreg,imm8	AVX512
VEXTRACTI32X4	mem128 mask,ymmreg,imm8	AVX512VL
VEXTRACTI32X4	mem128 mask,zmmreg,imm8	AVX512
VEXTRACTI32X8	ymmreg mask z,zmmreg,imm8	AVX512DQ
VEXTRACTI32X8	mem256 mask,zmmreg,imm8	AVX512DQ
VEXTRACTI64X2	xmmreg mask z,ymmreg,imm8	AVX512VL/DQ
VEXTRACTI64X2	xmmreg mask z,zmmreg,imm8	AVX512DQ
VEXTRACTI64X2	mem128 mask,ymmreg,imm8	AVX512VL/DQ
VEXTRACTI64X2	mem128 mask,zmmreg,imm8	AVX512DQ
VEXTRACTI64X4	ymmreg mask z,zmmreg,imm8	AVX512
VEXTRACTI64X4	mem256 mask,zmmreg,imm8	AVX512
VEXTRACTPS	reg32,xmmreg,imm8	AVX512
VEXTRACTPS	reg64,xmmreg,imm8	AVX512
VEXTRACTPS	mem32,xmmreg,imm8	AVX512
VFIXUPIIMPD	xmmreg mask z,xmmreg*,xmrm128 b64,imm8	AVX512VL
VFIXUPIIMPD	ymmreg mask z,ymmreg*,ymrm256 b64,imm8	AVX512VL
VFIXUPIIMPD	zmmreg mask z,zmmreg*,zmmrm512 b64 sae,imm8	AVX512
VFIXUPIIMPS	xmmreg mask z,xmmreg*,xmrm128 b32,imm8	AVX512VL
VFIXUPIIMPS	ymmreg mask z,ymmreg*,ymrm256 b32,imm8	AVX512VL
VFIXUPIIMPS	zmmreg mask z,zmmreg*,zmmrm512 b32 sae,imm8	AVX512
VFIXUPIIMSD	xmmreg mask z,xmmreg*,xmrm64 sae,imm8	AVX512
VFIXUPIIMSS	xmmreg mask z,xmmreg*,xmrm32 sae,imm8	AVX512
VFMADD132PD	xmmreg mask z,xmmreg,xmrm128 b64	AVX512VL
VFMADD132PD	ymmreg mask z,ymmreg,ymrm256 b64	AVX512VL
VFMADD132PD	zmmreg mask z,zmmreg,zmmrm512 b64 er	AVX512
VFMADD132PS	xmmreg mask z,xmmreg,xmrm128 b32	AVX512VL
VFMADD132PS	ymmreg mask z,ymmreg,ymrm256 b32	AVX512VL
VFMADD132PS	zmmreg mask z,zmmreg,zmmrm512 b32 er	AVX512
VFMADD132SD	xmmreg mask z,xmmreg,xmrm64 er	AVX512
VFMADD132SS	xmmreg mask z,xmmreg,xmrm32 er	AVX512
VFMADD213PD	xmmreg mask z,xmmreg,xmrm128 b64	AVX512VL
VFMADD213PD	ymmreg mask z,ymmreg,ymrm256 b64	AVX512VL
VFMADD213PD	zmmreg mask z,zmmreg,zmmrm512 b64 er	AVX512
VFMADD213PS	xmmreg mask z,xmmreg,xmrm128 b32	AVX512VL
VFMADD213PS	ymmreg mask z,ymmreg,ymrm256 b32	AVX512VL
VFMADD213PS	zmmreg mask z,zmmreg,zmmrm512 b32 er	AVX512
VFMADD213SD	xmmreg mask z,xmmreg,xmrm64 er	AVX512
VFMADD213SS	xmmreg mask z,xmmreg,xmrm32 er	AVX512
VFMADD231PD	xmmreg mask z,xmmreg,xmrm128 b64	AVX512VL
VFMADD231PD	ymmreg mask z,ymmreg,ymrm256 b64	AVX512VL
VFMADD231PD	zmmreg mask z,zmmreg,zmmrm512 b64 er	AVX512
VFMADD231PS	xmmreg mask z,xmmreg,xmrm128 b32	AVX512VL
VFMADD231PS	ymmreg mask z,ymmreg,ymrm256 b32	AVX512VL
VFMADD231PS	zmmreg mask z,zmmreg,zmmrm512 b32 er	AVX512
VFMADD231SD	xmmreg mask z,xmmreg,xmrm64 er	AVX512
VFMADD231SS	xmmreg mask z,xmmreg,xmrm32 er	AVX512
VFMADDSUB132PD	xmmreg mask z,xmmreg,xmrm128 b64	AVX512VL
VFMADDSUB132PD	ymmreg mask z,ymmreg,ymrm256 b64	AVX512VL
VFMADDSUB132PD	zmmreg mask z,zmmreg,zmmrm512 b64 er	AVX512





VFNMADD213PD	zmmreg mask z, zmmreg, zmmrm512 b64 er	AVX512
VFNMADD213PS	xmmreg mask z, xmmreg, xmmrm128 b32	AVX512VL
VFNMADD213PS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VFNMADD213PS	zmmreg mask z, zmmreg, zmmrm512 b32 er	AVX512
VFNMADD213SD	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VFNMADD213SS	xmmreg mask z, xmmreg, xmmrm32 er	AVX512
VFNMADD231PD	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL
VFNMADD231PD	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL
VFNMADD231PD	zmmreg mask z, zmmreg, zmmrm512 b64 er	AVX512
VFNMADD231PS	xmmreg mask z, xmmreg, xmmrm128 b32	AVX512VL
VFNMADD231PS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VFNMADD231PS	zmmreg mask z, zmmreg, zmmrm512 b32 er	AVX512
VFNMADD231SD	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VFNMADD231SS	xmmreg mask z, xmmreg, xmmrm32 er	AVX512
VFNMSUB132PD	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL
VFNMSUB132PD	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL
VFNMSUB132PD	zmmreg mask z, zmmreg, zmmrm512 b64 er	AVX512
VFNMSUB132PS	xmmreg mask z, xmmreg, xmmrm128 b32	AVX512VL
VFNMSUB132PS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VFNMSUB132PS	zmmreg mask z, zmmreg, zmmrm512 b32 er	AVX512
VFNMSUB132SD	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VFNMSUB132SS	xmmreg mask z, xmmreg, xmmrm32 er	AVX512
VFNMSUB213PD	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL
VFNMSUB213PD	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL
VFNMSUB213PD	zmmreg mask z, zmmreg, zmmrm512 b64 er	AVX512
VFNMSUB213PS	xmmreg mask z, xmmreg, xmmrm128 b32	AVX512VL
VFNMSUB213PS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VFNMSUB213PS	zmmreg mask z, zmmreg, zmmrm512 b32 er	AVX512
VFNMSUB213SD	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VFNMSUB213SS	xmmreg mask z, xmmreg, xmmrm32 er	AVX512
VFNMSUB231PD	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL
VFNMSUB231PD	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL
VFNMSUB231PD	zmmreg mask z, zmmreg, zmmrm512 b64 er	AVX512
VFNMSUB231PS	xmmreg mask z, xmmreg, xmmrm128 b32	AVX512VL
VFNMSUB231PS	ymmreg mask z, ymmreg, ymmrm256 b32	AVX512VL
VFNMSUB231PS	zmmreg mask z, zmmreg, zmmrm512 b32 er	AVX512
VFNMSUB231SD	xmmreg mask z, xmmreg, xmmrm64 er	AVX512
VFNMSUB231SS	xmmreg mask z, xmmreg, xmmrm32 er	AVX512
VFPCLASSPD	kreg mask, xmmrm128 b64, imm8	AVX512VL/DQ
VFPCLASSPD	kreg mask, ymmrm256 b64, imm8	AVX512VL/DQ
VFPCLASSPD	kreg mask, zmmrm512 b64, imm8	AVX512DQ
VFPCLASSPS	kreg mask, xmmrm128 b32, imm8	AVX512VL/DQ
VFPCLASSPS	kreg mask, ymmrm256 b32, imm8	AVX512VL/DQ
VFPCLASSPS	kreg mask, zmmrm512 b32, imm8	AVX512DQ
VFPCLASSSD	kreg mask, xmmrm64, imm8	AVX512DQ
VFPCLASSSS	kreg mask, xmmrm32, imm8	AVX512DQ
VGATHERDPD	xmmreg mask, xmem64	AVX512VL
VGATHERDPD	ymmreg mask, xmem64	AVX512VL
VGATHERDPD	zmmreg mask, ymem64	AVX512
VGATHERDPS	xmmreg mask, xmem32	AVX512VL
VGATHERDPS	ymmreg mask, ymem32	AVX512VL
VGATHERDPS	zmmreg mask, zmem32	AVX512
VGATHERPF0DPD	ymem64 mask	AVX512PF
VGATHERPF0DPS	zmem32 mask	AVX512PF
VGATHERPF0QPD	zmem64 mask	AVX512PF
VGATHERPF0QPS	zmem32 mask	AVX512PF
VGATHERPF1DPD	ymem64 mask	AVX512PF
VGATHERPF1DPS	zmem32 mask	AVX512PF
VGATHERPF1QPD	zmem64 mask	AVX512PF
VGATHERPF1QPS	zmem32 mask	AVX512PF
VGATHERQPD	xmmreg mask, xmem64	AVX512VL
VGATHERQPD	ymmreg mask, ymem64	AVX512VL
VGATHERQPD	zmmreg mask, zmem64	AVX512
VGATHERQPS	xmmreg mask, xmem32	AVX512VL
VGATHERQPS	xmmreg mask, ymem32	AVX512VL
VGATHERQPS	ymmreg mask, zmem32	AVX512
VGETEXPPD	xmmreg mask z, xmmrm128 b64	AVX512VL

VGETEXPPD	ymmreg mask z, ymmrm256 b64	AVX512VL
VGETEXPPD	zmmreg mask z, zmmrm512 b64 sae	AVX512
VGETEXPPS	xmmreg mask z, xmmrm128 b32	AVX512VL
VGETEXPPS	ymmreg mask z, ymmrm256 b32	AVX512VL
VGETEXPPS	zmmreg mask z, zmmrm512 b32 sae	AVX512
VGETEXPSD	xmmreg mask z, xmmreg, xmmrm64 sae	AVX512
VGETEXPS	xmmreg mask z, xmmreg, xmmrm32 sae	AVX512
VGETMANTPD	xmmreg mask z, xmmrm128 b64, imm8	AVX512VL
VGETMANTPD	ymmreg mask z, ymmrm256 b64, imm8	AVX512VL
VGETMANTPD	zmmreg mask z, zmmrm512 b64 sae, imm8	AVX512
VGETMANTPS	xmmreg mask z, xmmrm128 b32, imm8	AVX512VL
VGETMANTPS	ymmreg mask z, ymmrm256 b32, imm8	AVX512VL
VGETMANTPS	zmmreg mask z, zmmrm512 b32 sae, imm8	AVX512
VGETMANTSD	xmmreg mask z, xmmreg, xmmrm64 sae, imm8	AVX512
VGETMANTSS	xmmreg mask z, xmmreg, xmmrm32 sae, imm8	AVX512
VINSERTF32X4	ymmreg mask z, ymmreg*, xmmrm128, imm8	AVX512VL
VINSERTF32X4	zmmreg mask z, zmmreg*, xmmrm128, imm8	AVX512
VINSERTF32X8	zmmreg mask z, zmmreg*, ymmrm256, imm8	AVX512DQ
VINSERTF64X2	ymmreg mask z, ymmreg*, xmmrm128, imm8	AVX512VL/DQ
VINSERTF64X2	zmmreg mask z, zmmreg*, xmmrm128, imm8	AVX512DQ
VINSERTF64X4	zmmreg mask z, zmmreg*, ymmrm256, imm8	AVX512
VINSERTI32X4	ymmreg mask z, ymmreg*, xmmrm128, imm8	AVX512VL
VINSERTI32X4	zmmreg mask z, zmmreg*, xmmrm128, imm8	AVX512
VINSERTI32X8	zmmreg mask z, zmmreg*, ymmrm256, imm8	AVX512DQ
VINSERTI64X2	ymmreg mask z, ymmreg*, xmmrm128, imm8	AVX512VL/DQ
VINSERTI64X2	zmmreg mask z, zmmreg*, xmmrm128, imm8	AVX512DQ
VINSERTI64X4	zmmreg mask z, zmmreg*, ymmrm256, imm8	AVX512
VINSERTPS	xmmreg, xmmreg*, xmmrm32, imm8	AVX512
VMAXPD	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VMAXPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VMAXPD	zmmreg mask z, zmmreg*, zmmrm512 b64 sae	AVX512
VMAXPS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VMAXPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VMAXPS	zmmreg mask z, zmmreg*, zmmrm512 b32 sae	AVX512
VMAXSD	xmmreg mask z, xmmreg*, xmmrm64 sae	AVX512
VMAXSS	xmmreg mask z, xmmreg*, xmmrm32 sae	AVX512
VMINPD	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VMINPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VMINPD	zmmreg mask z, zmmreg*, zmmrm512 b64 sae	AVX512
VMINPS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VMINPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VMINPS	zmmreg mask z, zmmreg*, zmmrm512 b32 sae	AVX512
VMINSD	xmmreg mask z, xmmreg*, xmmrm64 sae	AVX512
VMINSS	xmmreg mask z, xmmreg*, xmmrm32 sae	AVX512
VMOVAPD	xmmreg mask z, xmmrm128	AVX512VL
VMOVAPD	ymmreg mask z, ymmrm256	AVX512VL
VMOVAPD	zmmreg mask z, zmmrm512	AVX512
VMOVAPD	xmmreg mask z, xmmreg	AVX512VL
VMOVAPD	ymmreg mask z, ymmreg	AVX512VL
VMOVAPD	zmmreg mask z, zmmreg	AVX512
VMOVAPD	mem128 mask, xmmreg	AVX512VL
VMOVAPD	mem256 mask, ymmreg	AVX512VL
VMOVAPD	mem512 mask, zmmreg	AVX512
VMOVAPS	xmmreg mask z, xmmrm128	AVX512VL
VMOVAPS	ymmreg mask z, ymmrm256	AVX512VL
VMOVAPS	zmmreg mask z, zmmrm512	AVX512
VMOVAPS	xmmreg mask z, xmmreg	AVX512VL
VMOVAPS	ymmreg mask z, ymmreg	AVX512VL
VMOVAPS	zmmreg mask z, zmmreg	AVX512
VMOVAPS	mem128 mask, xmmreg	AVX512VL
VMOVAPS	mem256 mask, ymmreg	AVX512VL
VMOVAPS	mem512 mask, zmmreg	AVX512
VMOVD	xmmreg, rm32	AVX512
VMOVD	rm32, xmmreg	AVX512
VMOVDDUP	xmmreg mask z, xmmrm64	AVX512VL
VMOVDDUP	ymmreg mask z, ymmrm256	AVX512VL
VMOVDDUP	zmmreg mask z, zmmrm512	AVX512

VMOVDQA32	xmmreg mask z, xmmrm128	AVX512VL
VMOVDQA32	ymmreg mask z, ymmrm256	AVX512VL
VMOVDQA32	zmmreg mask z, zmmrm512	AVX512
VMOVDQA32	xmmrm128 mask z, xmmreg	AVX512VL
VMOVDQA32	ymmrm256 mask z, ymmreg	AVX512VL
VMOVDQA32	zmmrm512 mask z, zmmreg	AVX512
VMOVDQA64	xmmreg mask z, xmmrm128	AVX512VL
VMOVDQA64	ymmreg mask z, ymmrm256	AVX512VL
VMOVDQA64	zmmreg mask z, zmmrm512	AVX512
VMOVDQA64	xmmrm128 mask z, xmmreg	AVX512VL
VMOVDQA64	ymmrm256 mask z, ymmreg	AVX512VL
VMOVDQA64	zmmrm512 mask z, zmmreg	AVX512
VMOVDQU16	xmmreg mask z, xmmrm128	AVX512VL/BW
VMOVDQU16	ymmreg mask z, ymmrm256	AVX512VL/BW
VMOVDQU16	zmmreg mask z, zmmrm512	AVX512BW
VMOVDQU16	xmmrm128 mask z, xmmreg	AVX512VL/BW
VMOVDQU16	ymmrm256 mask z, ymmreg	AVX512VL/BW
VMOVDQU16	zmmrm512 mask z, zmmreg	AVX512BW
VMOVDQU32	xmmreg mask z, xmmrm128	AVX512VL
VMOVDQU32	ymmreg mask z, ymmrm256	AVX512VL
VMOVDQU32	zmmreg mask z, zmmrm512	AVX512
VMOVDQU32	xmmrm128 mask z, xmmreg	AVX512VL
VMOVDQU32	ymmrm256 mask z, ymmreg	AVX512VL
VMOVDQU32	zmmrm512 mask z, zmmreg	AVX512
VMOVDQU64	xmmreg mask z, xmmrm128	AVX512VL
VMOVDQU64	ymmreg mask z, ymmrm256	AVX512VL
VMOVDQU64	zmmreg mask z, zmmrm512	AVX512
VMOVDQU64	xmmrm128 mask z, xmmreg	AVX512VL
VMOVDQU64	ymmrm256 mask z, ymmreg	AVX512VL
VMOVDQU64	zmmrm512 mask z, zmmreg	AVX512
VMOVDQU8	xmmreg mask z, xmmrm128	AVX512VL/BW
VMOVDQU8	ymmreg mask z, ymmrm256	AVX512VL/BW
VMOVDQU8	zmmreg mask z, zmmrm512	AVX512BW
VMOVDQU8	xmmrm128 mask z, xmmreg	AVX512VL/BW
VMOVDQU8	ymmrm256 mask z, ymmreg	AVX512VL/BW
VMOVDQU8	zmmrm512 mask z, zmmreg	AVX512BW
VMOVHPS	xmmreg, xmmreg*, xmmreg	AVX512
VMOVHPD	xmmreg, xmmreg*, mem64	AVX512
VMOVHPD	mem64, xmmreg	AVX512
VMOVHPS	xmmreg, xmmreg*, mem64	AVX512
VMOVHPS	mem64, xmmreg	AVX512
VMOVLHPS	xmmreg, xmmreg*, xmmreg	AVX512
VMOVLPD	xmmreg, xmmreg*, mem64	AVX512
VMOVLPD	mem64, xmmreg	AVX512
VMOVLPS	xmmreg, xmmreg*, mem64	AVX512
VMOVLPS	mem64, xmmreg	AVX512
VMOVNTDQ	mem128, xmmreg	AVX512VL
VMOVNTDQ	mem256, ymmreg	AVX512VL
VMOVNTDQ	mem512, zmmreg	AVX512
VMOVNTDQA	xmmreg, mem128	AVX512VL
VMOVNTDQA	ymmreg, mem256	AVX512VL
VMOVNTDQA	zmmreg, mem512	AVX512
VMOVNTPD	mem128, xmmreg	AVX512VL
VMOVNTPD	mem256, ymmreg	AVX512VL
VMOVNTPD	mem512, zmmreg	AVX512
VMOVNTPS	mem128, xmmreg	AVX512VL
VMOVNTPS	mem256, ymmreg	AVX512VL
VMOVNTPS	mem512, zmmreg	AVX512
VMOVQ	xmmreg, rm64	AVX512
VMOVQ	rm64, xmmreg	AVX512
VMOVQ	xmmreg, xmmrm64	AVX512
VMOVQ	xmmrm64, xmmreg	AVX512
VMOVSD	xmmreg mask z, mem64	AVX512
VMOVSD	mem64 mask, xmmreg	AVX512
VMOVSD	xmmreg mask z, xmmreg*, xmmreg	AVX512
VMOVSD	xmmreg mask z, xmmreg*, xmmreg	AVX512
VMOVSHDUP	xmmreg mask z, xmmrm128	AVX512VL

VMOVSHDUP	ymmreg mask z, ymmrm256	AVX512VL
VMOVSHDUP	zmmreg mask z, zmmrm512	AVX512
VMOVSLDUP	xmmreg mask z, xmmrm128	AVX512VL
VMOVSLDUP	ymmreg mask z, ymmrm256	AVX512VL
VMOVSLDUP	zmmreg mask z, zmmrm512	AVX512
VMOVSS	xmmreg mask z, mem32	AVX512
VMOVSS	mem32 mask, xmmreg	AVX512
VMOVSS	xmmreg mask z, xmmreg*, xmmreg	AVX512
VMOVSS	xmmreg mask z, xmmreg*, xmmreg	AVX512
VMOVUPD	xmmreg mask z, xmmrm128	AVX512VL
VMOVUPD	ymmreg mask z, ymmrm256	AVX512VL
VMOVUPD	zmmreg mask z, zmmrm512	AVX512
VMOVUPD	xmmreg mask z, xmmreg	AVX512VL
VMOVUPD	ymmreg mask z, ymmreg	AVX512VL
VMOVUPD	zmmreg mask z, zmmreg	AVX512
VMOVUPD	mem128 mask, xmmreg	AVX512VL
VMOVUPD	mem256 mask, ymmreg	AVX512VL
VMOVUPD	mem512 mask, zmmreg	AVX512
VMOVUPS	xmmreg mask z, xmmrm128	AVX512VL
VMOVUPS	ymmreg mask z, ymmrm256	AVX512VL
VMOVUPS	zmmreg mask z, zmmrm512	AVX512
VMOVUPS	xmmreg mask z, xmmreg	AVX512VL
VMOVUPS	ymmreg mask z, ymmreg	AVX512VL
VMOVUPS	zmmreg mask z, zmmreg	AVX512
VMOVUPS	mem128 mask, xmmreg	AVX512VL
VMOVUPS	mem256 mask, ymmreg	AVX512VL
VMOVUPS	mem512 mask, zmmreg	AVX512
VMULPD	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VMULPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VMULPD	zmmreg mask z, zmmreg*, zmmrm512 b64 er	AVX512
VMULPS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VMULPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VMULPS	zmmreg mask z, zmmreg*, zmmrm512 b32 er	AVX512
VMULSD	xmmreg mask z, xmmreg*, xmmrm64 er	AVX512
VMULSS	xmmreg mask z, xmmreg*, xmmrm32 er	AVX512
VORPD	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL/DQ
VORPD	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL/DQ
VORPD	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512DQ
VORPS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL/DQ
VORPS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL/DQ
VORPS	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512DQ
VPABSB	xmmreg mask z, xmmrm128	AVX512VL/BW
VPABSB	ymmreg mask z, ymmrm256	AVX512VL/BW
VPABSB	zmmreg mask z, zmmrm512	AVX512BW
VPABSD	xmmreg mask z, xmmrm128 b32	AVX512VL
VPABSD	ymmreg mask z, ymmrm256 b32	AVX512VL
VPABSD	zmmreg mask z, zmmrm512 b32	AVX512
VPABSQ	xmmreg mask z, xmmrm128 b64	AVX512VL
VPABSQ	ymmreg mask z, ymmrm256 b64	AVX512VL
VPABSQ	zmmreg mask z, zmmrm512 b64	AVX512
VPABSW	xmmreg mask z, xmmrm128	AVX512VL/BW
VPABSW	ymmreg mask z, ymmrm256	AVX512VL/BW
VPABSW	zmmreg mask z, zmmrm512	AVX512BW
VPACKSSDW	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL/BW
VPACKSSDW	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL/BW
VPACKSSDW	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512BW
VPACKSSWB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPACKSSWB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPACKSSWB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPACKUSDW	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL/BW
VPACKUSDW	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL/BW
VPACKUSDW	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512BW
VPACKUSWB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPACKUSWB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPACKUSWB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPADDB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW

VPADDB	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPADD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPADD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPADD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPADDQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPADDQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPADDQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPADDSB	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDSB	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPADDSB	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPADDSW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDSW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPADDSW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPADDUSB	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDUSB	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPADDUSB	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPADDUSW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDUSW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPADDUSW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPADDW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPADDW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPADDW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPALIGNR	xmmreg	mask	z, xmmreg*, xmmrm128, imm8	AVX512VL/BW
VPALIGNR	ymmreg	mask	z, ymmreg*, ymmrm256, imm8	AVX512VL/BW
VPALIGNR	zmmreg	mask	z, zmmreg*, zmmrm512, imm8	AVX512BW
VPANDD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPANDD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPANDD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPANDND	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPANDND	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPANDND	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPANDNQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPANDNQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPANDNQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPANDQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPANDQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPANDQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPAVGB	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPAVGB	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPAVGB	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPAVGW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPAVGW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPAVGW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPBLENDMB	xmmreg	mask	z, xmmreg, xmmrm128	AVX512VL/BW
VPBLENDMB	ymmreg	mask	z, ymmreg, ymmrm256	AVX512VL/BW
VPBLENDMB	zmmreg	mask	z, zmmreg, zmmrm512	AVX512BW
VPBLENDMD	xmmreg	mask	z, xmmreg, xmmrm128	b32 AVX512VL
VPBLENDMD	ymmreg	mask	z, ymmreg, ymmrm256	b32 AVX512VL
VPBLENDMD	zmmreg	mask	z, zmmreg, zmmrm512	b32 AVX512
VPBLENDMQ	xmmreg	mask	z, xmmreg, xmmrm128	b64 AVX512VL
VPBLENDMQ	ymmreg	mask	z, ymmreg, ymmrm256	b64 AVX512VL
VPBLENDMQ	zmmreg	mask	z, zmmreg, zmmrm512	b64 AVX512
VPBLENDMW	xmmreg	mask	z, xmmreg, xmmrm128	AVX512VL/BW
VPBLENDMW	ymmreg	mask	z, ymmreg, ymmrm256	AVX512VL/BW
VPBLENDMW	zmmreg	mask	z, zmmreg, zmmrm512	AVX512BW
VPBROADCASTB	xmmreg	mask	z, xmmrm8	AVX512VL/BW
VPBROADCASTB	ymmreg	mask	z, xmmrm8	AVX512VL/BW
VPBROADCASTB	zmmreg	mask	z, xmmrm8	AVX512BW
VPBROADCASTB	xmmreg	mask	z, reg8	AVX512VL/BW
VPBROADCASTB	xmmreg	mask	z, reg16	AVX512VL/BW
VPBROADCASTB	xmmreg	mask	z, reg32	AVX512VL/BW
VPBROADCASTB	xmmreg	mask	z, reg64	AVX512VL/BW
VPBROADCASTB	ymmreg	mask	z, reg8	AVX512VL/BW
VPBROADCASTB	ymmreg	mask	z, reg16	AVX512VL/BW
VPBROADCASTB	ymmreg	mask	z, reg32	AVX512VL/BW
VPBROADCASTB	ymmreg	mask	z, reg64	AVX512VL/BW
VPBROADCASTB	zmmreg	mask	z, reg8	AVX512BW

VPBROADCASTB	zmmreg mask z, reg16	AVX512BW
VPBROADCASTB	zmmreg mask z, reg32	AVX512BW
VPBROADCASTB	zmmreg mask z, reg64	AVX512BW
VPBROADCASTD	xmmreg mask z, mem32	AVX512VL
VPBROADCASTD	ymmreg mask z, mem32	AVX512VL
VPBROADCASTD	zmmreg mask z, mem32	AVX512
VPBROADCASTD	xmmreg mask z, xmmreg	AVX512VL
VPBROADCASTD	ymmreg mask z, xmmreg	AVX512VL
VPBROADCASTD	zmmreg mask z, xmmreg	AVX512
VPBROADCASTD	xmmreg mask z, reg32	AVX512VL
VPBROADCASTD	ymmreg mask z, reg32	AVX512VL
VPBROADCASTD	zmmreg mask z, reg32	AVX512
VPBROADCASTMB2Q	xmmreg, kreg	AVX512VL/CD
VPBROADCASTMB2Q	ymmreg, kreg	AVX512VL/CD
VPBROADCASTMB2Q	zmmreg, kreg	AVX512CD
VPBROADCASTMW2D	xmmreg, kreg	AVX512VL/CD
VPBROADCASTMW2D	ymmreg, kreg	AVX512VL/CD
VPBROADCASTMW2D	zmmreg, kreg	AVX512CD
VPBROADCASTQ	xmmreg mask z, mem64	AVX512VL
VPBROADCASTQ	ymmreg mask z, mem64	AVX512VL
VPBROADCASTQ	zmmreg mask z, mem64	AVX512
VPBROADCASTQ	xmmreg mask z, xmmreg	AVX512VL
VPBROADCASTQ	ymmreg mask z, xmmreg	AVX512VL
VPBROADCASTQ	zmmreg mask z, xmmreg	AVX512
VPBROADCASTQ	xmmreg mask z, reg64	AVX512VL
VPBROADCASTQ	ymmreg mask z, reg64	AVX512VL
VPBROADCASTQ	zmmreg mask z, reg64	AVX512
VPBROADCASTW	xmmreg mask z, xmmrm16	AVX512VL/BW
VPBROADCASTW	ymmreg mask z, xmmrm16	AVX512VL/BW
VPBROADCASTW	zmmreg mask z, xmmrm16	AVX512BW
VPBROADCASTW	xmmreg mask z, reg16	AVX512VL/BW
VPBROADCASTW	xmmreg mask z, reg32	AVX512VL/BW
VPBROADCASTW	xmmreg mask z, reg64	AVX512VL/BW
VPBROADCASTW	ymmreg mask z, reg16	AVX512VL/BW
VPBROADCASTW	ymmreg mask z, reg32	AVX512VL/BW
VPBROADCASTW	ymmreg mask z, reg64	AVX512VL/BW
VPBROADCASTW	zmmreg mask z, reg16	AVX512BW
VPBROADCASTW	zmmreg mask z, reg32	AVX512BW
VPBROADCASTW	zmmreg mask z, reg64	AVX512BW
VPCMPQEB	kreg mask, xmmreg, xmmrm128	AVX512VL/BW
VPCMPQEB	kreg mask, ymmreg, ymmrm256	AVX512VL/BW
VPCMPQEB	kreg mask, zmmreg, zmmrm512	AVX512BW
VPCMPQED	kreg mask, xmmreg, xmmrm128 b32	AVX512VL
VPCMPQED	kreg mask, ymmreg, ymmrm256 b32	AVX512VL
VPCMPQED	kreg mask, zmmreg, zmmrm512 b32	AVX512
VPCMPQEQ	kreg mask, xmmreg, xmmrm128 b64	AVX512VL
VPCMPQEQ	kreg mask, ymmreg, ymmrm256 b64	AVX512VL
VPCMPQEQ	kreg mask, zmmreg, zmmrm512 b64	AVX512
VPCMPQEW	kreg mask, xmmreg, xmmrm128	AVX512VL/BW
VPCMPQEW	kreg mask, ymmreg, ymmrm256	AVX512VL/BW
VPCMPQEW	kreg mask, zmmreg, zmmrm512	AVX512BW
VPCMPGTB	kreg mask, xmmreg, xmmrm128	AVX512VL/BW
VPCMPGTB	kreg mask, ymmreg, ymmrm256	AVX512VL/BW
VPCMPGTB	kreg mask, zmmreg, zmmrm512	AVX512BW
VPCMPGTD	kreg mask, xmmreg, xmmrm128 b32	AVX512VL
VPCMPGTD	kreg mask, ymmreg, ymmrm256 b32	AVX512VL
VPCMPGTD	kreg mask, zmmreg, zmmrm512 b32	AVX512
VPCMPGTQ	kreg mask, xmmreg, xmmrm128 b64	AVX512VL
VPCMPGTQ	kreg mask, ymmreg, ymmrm256 b64	AVX512VL
VPCMPGTQ	kreg mask, zmmreg, zmmrm512 b64	AVX512
VPCMPGTW	kreg mask, xmmreg, xmmrm128	AVX512VL/BW
VPCMPGTW	kreg mask, ymmreg, ymmrm256	AVX512VL/BW
VPCMPGTW	kreg mask, zmmreg, zmmrm512	AVX512BW
VPCMPQEB	kreg mask, xmmreg, xmmrm128	AVX512VL/BW
VPCMPQEB	kreg mask, ymmreg, ymmrm256	AVX512VL/BW
VPCMPQEB	kreg mask, zmmreg, zmmrm512	AVX512BW
VPCMPQED	kreg mask, xmmreg, xmmrm128 b32	AVX512VL

VPCMP EQD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP EQD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP EQQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP EQQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP EQQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP EQUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP EQUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP EQUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP EQUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMP EQUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP EQUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP EQUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP EQUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP EQUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP EUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP EUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP EUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP EQW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP EQW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP EQW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GEB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GEB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GEB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GED	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMP GED	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP GED	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP GEQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP GEQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP GEQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP GEUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GEUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GEUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GEUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMP GEUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP GEUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP GEUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP GEUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP GEUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP GEUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GEUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GEUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GEW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GEW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GEW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GTB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GTB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GTB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GTD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMP GTD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP GTD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP GTQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP GTQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP GTQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP GTUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GTUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GTUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GTUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMP GTUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMP GTUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMP GTUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMP GTUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMP GTUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMP GTUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GTUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMP GTUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMP GTW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMP GTW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW



VPCMPGTW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLEB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLEB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLEB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLED	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPLED	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPLED	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPLEQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPLEQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPLEQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPLEUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLEUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLEUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLEUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPLEUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPLEUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPLEUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPLEUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPLEUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPLEUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLEUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLEUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLEW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLEW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLEW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLTB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLTB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLTB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLTD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPLTD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPLTD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPLTQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPLTQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPLTQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPLTUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLTUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLTUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLTUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPLTUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPLTUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPLTUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPLTUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPLTUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPLTUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLTUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLTUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPLTW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPLTW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPLTW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNEQB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNEQB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNEQB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNEQD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPNEQD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPNEQD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPNEQQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPNEQQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPNEQQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPNEQUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNEQUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNEQUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNEQUD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPCMPNEQUD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL
VPCMPNEQUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPNEQUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPNEQUQ	kreg mask,ymmreg,ymmrm256 b64	AVX512VL
VPCMPNEQUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512

VPCMPNEQUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNEQUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNEQUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNEQW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNEQW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNEQW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNGTB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNGTB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNGTB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNGTD	kreg mask,xmmreg,xmmrm128	b32 AVX512VL
VPCMPNGTD	kreg mask,ymmreg,ymmrm256	b32 AVX512VL
VPCMPNGTD	kreg mask,zmmreg,zmmrm512	b32 AVX512
VPCMPNGTQ	kreg mask,xmmreg,xmmrm128	b64 AVX512VL
VPCMPNGTQ	kreg mask,ymmreg,ymmrm256	b64 AVX512VL
VPCMPNGTQ	kreg mask,zmmreg,zmmrm512	b64 AVX512
VPCMPNGTUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNGTUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNGTUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNGTUD	kreg mask,xmmreg,xmmrm128	b32 AVX512VL
VPCMPNGTUD	kreg mask,ymmreg,ymmrm256	b32 AVX512VL
VPCMPNGTUD	kreg mask,zmmreg,zmmrm512	b32 AVX512
VPCMPNGTUQ	kreg mask,xmmreg,xmmrm128	b64 AVX512VL
VPCMPNGTUQ	kreg mask,ymmreg,ymmrm256	b64 AVX512VL
VPCMPNGTUQ	kreg mask,zmmreg,zmmrm512	b64 AVX512
VPCMPNGTUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNGTUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNGTUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNGTW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNGTW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNGTW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLEB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLEB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLEB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLED	kreg mask,xmmreg,xmmrm128	b32 AVX512VL
VPCMPNLED	kreg mask,ymmreg,ymmrm256	b32 AVX512VL
VPCMPNLED	kreg mask,zmmreg,zmmrm512	b32 AVX512
VPCMPNLEQ	kreg mask,xmmreg,xmmrm128	b64 AVX512VL
VPCMPNLEQ	kreg mask,ymmreg,ymmrm256	b64 AVX512VL
VPCMPNLEQ	kreg mask,zmmreg,zmmrm512	b64 AVX512
VPCMPNLEUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLEUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLEUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLEUD	kreg mask,xmmreg,xmmrm128	b32 AVX512VL
VPCMPNLEUD	kreg mask,ymmreg,ymmrm256	b32 AVX512VL
VPCMPNLEUD	kreg mask,zmmreg,zmmrm512	b32 AVX512
VPCMPNLEUQ	kreg mask,xmmreg,xmmrm128	b64 AVX512VL
VPCMPNLEUQ	kreg mask,ymmreg,ymmrm256	b64 AVX512VL
VPCMPNLEUQ	kreg mask,zmmreg,zmmrm512	b64 AVX512
VPCMPNLEUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLEUW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLEUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLEW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLEW	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLEW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLTB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLTB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLTB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLTD	kreg mask,xmmreg,xmmrm128	b32 AVX512VL
VPCMPNLTD	kreg mask,ymmreg,ymmrm256	b32 AVX512VL
VPCMPNLTD	kreg mask,zmmreg,zmmrm512	b32 AVX512
VPCMPNLTQ	kreg mask,xmmreg,xmmrm128	b64 AVX512VL
VPCMPNLTQ	kreg mask,ymmreg,ymmrm256	b64 AVX512VL
VPCMPNLTQ	kreg mask,zmmreg,zmmrm512	b64 AVX512
VPCMPNLTUB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLTUB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPCMPNLTUB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLTUD	kreg mask,xmmreg,xmmrm128	b32 AVX512VL

VPCMPNLTUD	kreg mask,ymmreg,ymmr256 b32	AVX512VL
VPCMPNLTUD	kreg mask,zmmreg,zmmrm512 b32	AVX512
VPCMPNLTUQ	kreg mask,xmmreg,xmmrm128 b64	AVX512VL
VPCMPNLTUQ	kreg mask,ymmreg,ymmr256 b64	AVX512VL
VPCMPNLTUQ	kreg mask,zmmreg,zmmrm512 b64	AVX512
VPCMPNLTUW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLTUW	kreg mask,ymmreg,ymmr256	AVX512VL/BW
VPCMPNLTUW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPNLTW	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPCMPNLTW	kreg mask,ymmreg,ymmr256	AVX512VL/BW
VPCMPNLTW	kreg mask,zmmreg,zmmrm512	AVX512BW
VPCMPB	kreg mask,xmmreg,xmmrm128,imm8	AVX512VL/BW
VPCMPB	kreg mask,ymmreg,ymmr256,imm8	AVX512VL/BW
VPCMPB	kreg mask,zmmreg,zmmrm512,imm8	AVX512BW
VPCMPD	kreg mask,xmmreg,xmmrm128 b32,imm8	AVX512VL
VPCMPD	kreg mask,ymmreg,ymmr256 b32,imm8	AVX512VL
VPCMPD	kreg mask,zmmreg,zmmrm512 b32,imm8	AVX512
VPCMPQ	kreg mask,xmmreg,xmmrm128 b64,imm8	AVX512VL
VPCMPQ	kreg mask,ymmreg,ymmr256 b64,imm8	AVX512VL
VPCMPQ	kreg mask,zmmreg,zmmrm512 b64,imm8	AVX512
VPCMPUB	kreg mask,xmmreg,xmmrm128,imm8	AVX512VL/BW
VPCMPUB	kreg mask,ymmreg,ymmr256,imm8	AVX512VL/BW
VPCMPUB	kreg mask,zmmreg,zmmrm512,imm8	AVX512BW
VPCMPUD	kreg mask,xmmreg,xmmrm128 b32,imm8	AVX512VL
VPCMPUD	kreg mask,ymmreg,ymmr256 b32,imm8	AVX512VL
VPCMPUD	kreg mask,zmmreg,zmmrm512 b32,imm8	AVX512
VPCMPUQ	kreg mask,xmmreg,xmmrm128 b64,imm8	AVX512VL
VPCMPUQ	kreg mask,ymmreg,ymmr256 b64,imm8	AVX512VL
VPCMPUQ	kreg mask,zmmreg,zmmrm512 b64,imm8	AVX512
VPCMPUW	kreg mask,xmmreg,xmmrm128,imm8	AVX512VL/BW
VPCMPUW	kreg mask,ymmreg,ymmr256,imm8	AVX512VL/BW
VPCMPUW	kreg mask,zmmreg,zmmrm512,imm8	AVX512BW
VPCMPW	kreg mask,xmmreg,xmmrm128,imm8	AVX512VL/BW
VPCMPW	kreg mask,ymmreg,ymmr256,imm8	AVX512VL/BW
VPCMPW	kreg mask,zmmreg,zmmrm512,imm8	AVX512BW
VPCOMPRESSD	mem128 mask,xmmreg	AVX512VL
VPCOMPRESSD	mem256 mask,ymmreg	AVX512VL
VPCOMPRESSD	mem512 mask,zmmreg	AVX512
VPCOMPRESSD	xmmreg mask z,xmmreg	AVX512VL
VPCOMPRESSD	ymmreg mask z,ymmreg	AVX512VL
VPCOMPRESSD	zmmreg mask z,zmmreg	AVX512
VPCOMPRESSQ	mem128 mask,xmmreg	AVX512VL
VPCOMPRESSQ	mem256 mask,ymmreg	AVX512VL
VPCOMPRESSQ	mem512 mask,zmmreg	AVX512
VPCOMPRESSQ	xmmreg mask z,xmmreg	AVX512VL
VPCOMPRESSQ	ymmreg mask z,ymmreg	AVX512VL
VPCOMPRESSQ	zmmreg mask z,zmmreg	AVX512
VPCONFLICTD	xmmreg mask z,xmmrm128 b32	AVX512VL/CD
VPCONFLICTD	ymmreg mask z,ymmr256 b32	AVX512VL/CD
VPCONFLICTD	zmmreg mask z,zmmrm512 b32	AVX512CD
VPCONFLICTQ	xmmreg mask z,xmmrm128 b64	AVX512VL/CD
VPCONFLICTQ	ymmreg mask z,ymmr256 b64	AVX512VL/CD
VPCONFLICTQ	zmmreg mask z,zmmrm512 b64	AVX512CD
VPERMB	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/VBMI
VPERMB	ymmreg mask z,ymmreg*,ymmr256	AVX512VL/VBMI
VPERMB	zmmreg mask z,zmmreg*,zmmrm512	AVX512VBMI
VPERMD	ymmreg mask z,ymmreg*,ymmr256 b32	AVX512VL
VPERMD	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512
VPERMI2B	xmmreg mask z,xmmreg,xmmrm128	AVX512VL/VBMI
VPERMI2B	ymmreg mask z,ymmreg,ymmr256	AVX512VL/VBMI
VPERMI2B	zmmreg mask z,zmmreg,zmmrm512	AVX512VBMI
VPERMI2D	xmmreg mask z,xmmreg,xmmrm128 b32	AVX512VL
VPERMI2D	ymmreg mask z,ymmreg,ymmr256 b32	AVX512VL
VPERMI2D	zmmreg mask z,zmmreg,zmmrm512 b32	AVX512
VPERMI2PD	xmmreg mask z,xmmreg,xmmrm128 b64	AVX512VL
VPERMI2PD	ymmreg mask z,ymmreg,ymmr256 b64	AVX512VL
VPERMI2PD	zmmreg mask z,zmmreg,zmmrm512 b64	AVX512

VPERMI2PS	xmmreg	mask	z, xmmreg, xmmrm128	b32 AVX512VL
VPERMI2PS	ymmreg	mask	z, ymmreg, ymmrm256	b32 AVX512VL
VPERMI2PS	zmmreg	mask	z, zmmreg, zmmrm512	b32 AVX512
VPERMI2Q	xmmreg	mask	z, xmmreg, xmmrm128	b64 AVX512VL
VPERMI2Q	ymmreg	mask	z, ymmreg, ymmrm256	b64 AVX512VL
VPERMI2Q	zmmreg	mask	z, zmmreg, zmmrm512	b64 AVX512
VPERMI2W	xmmreg	mask	z, xmmreg, xmmrm128	AVX512VL/BW
VPERMI2W	ymmreg	mask	z, ymmreg, ymmrm256	AVX512VL/BW
VPERMI2W	zmmreg	mask	z, zmmreg, zmmrm512	AVX512BW
VPERMILPD	xmmreg	mask	z, xmmrm128	b64, imm8 AVX512VL
VPERMILPD	ymmreg	mask	z, ymmrm256	b64, imm8 AVX512VL
VPERMILPD	zmmreg	mask	z, zmmrm512	b64, imm8 AVX512
VPERMILPD	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPERMILPD	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPERMILPD	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPERMILPS	xmmreg	mask	z, xmmrm128	b32, imm8 AVX512VL
VPERMILPS	ymmreg	mask	z, ymmrm256	b32, imm8 AVX512VL
VPERMILPS	zmmreg	mask	z, zmmrm512	b32, imm8 AVX512
VPERMILPS	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPERMILPS	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPERMILPS	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPERMPD	ymmreg	mask	z, ymmrm256	b64, imm8 AVX512VL
VPERMPD	zmmreg	mask	z, zmmrm512	b64, imm8 AVX512
VPERMPD	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPERMPD	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPERMPS	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPERMPS	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPERMQ	ymmreg	mask	z, ymmrm256	b64, imm8 AVX512VL
VPERMQ	zmmreg	mask	z, zmmrm512	b64, imm8 AVX512
VPERMQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPERMQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPERMT2B	xmmreg	mask	z, xmmreg, xmmrm128	AVX512VL/VBMI
VPERMT2B	ymmreg	mask	z, ymmreg, ymmrm256	AVX512VL/VBMI
VPERMT2B	zmmreg	mask	z, zmmreg, zmmrm512	AVX512VBMI
VPERMT2D	xmmreg	mask	z, xmmreg, xmmrm128	b32 AVX512VL
VPERMT2D	ymmreg	mask	z, ymmreg, ymmrm256	b32 AVX512VL
VPERMT2D	zmmreg	mask	z, zmmreg, zmmrm512	b32 AVX512
VPERMT2PD	xmmreg	mask	z, xmmreg, xmmrm128	b64 AVX512VL
VPERMT2PD	ymmreg	mask	z, ymmreg, ymmrm256	b64 AVX512VL
VPERMT2PD	zmmreg	mask	z, zmmreg, zmmrm512	b64 AVX512
VPERMT2PS	xmmreg	mask	z, xmmreg, xmmrm128	b32 AVX512VL
VPERMT2PS	ymmreg	mask	z, ymmreg, ymmrm256	b32 AVX512VL
VPERMT2PS	zmmreg	mask	z, zmmreg, zmmrm512	b32 AVX512
VPERMT2Q	xmmreg	mask	z, xmmreg, xmmrm128	b64 AVX512VL
VPERMT2Q	ymmreg	mask	z, ymmreg, ymmrm256	b64 AVX512VL
VPERMT2Q	zmmreg	mask	z, zmmreg, zmmrm512	b64 AVX512
VPERMT2W	xmmreg	mask	z, xmmreg, xmmrm128	AVX512VL/BW
VPERMT2W	ymmreg	mask	z, ymmreg, ymmrm256	AVX512VL/BW
VPERMT2W	zmmreg	mask	z, zmmreg, zmmrm512	AVX512BW
VPERMW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPERMW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPERMW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPEXPANDD	xmmreg	mask	z, mem128	AVX512VL
VPEXPANDD	ymmreg	mask	z, mem256	AVX512VL
VPEXPANDD	zmmreg	mask	z, mem512	AVX512
VPEXPANDD	xmmreg	mask	z, xmmreg	AVX512VL
VPEXPANDD	ymmreg	mask	z, ymmreg	AVX512VL
VPEXPANDD	zmmreg	mask	z, zmmreg	AVX512
VPEXPANDQ	xmmreg	mask	z, mem128	AVX512VL
VPEXPANDQ	ymmreg	mask	z, mem256	AVX512VL
VPEXPANDQ	zmmreg	mask	z, mem512	AVX512
VPEXPANDQ	xmmreg	mask	z, xmmreg	AVX512VL
VPEXPANDQ	ymmreg	mask	z, ymmreg	AVX512VL
VPEXPANDQ	zmmreg	mask	z, zmmreg	AVX512
VPEXTRB	reg8, xmmreg		, imm8	AVX512BW
VPEXTRB	reg16, xmmreg		, imm8	AVX512BW
VPEXTRB	reg32, xmmreg		, imm8	AVX512BW

VPEXTRB	reg64, xmmreg, imm8	AVX512BW
VPEXTRB	mem8, xmmreg, imm8	AVX512BW
VPEXTRD	rm32, xmmreg, imm8	AVX512DQ
VPEXTRQ	rm64, xmmreg, imm8	AVX512DQ
VPEXTRW	reg16, xmmreg, imm8	AVX512BW
VPEXTRW	reg32, xmmreg, imm8	AVX512BW
VPEXTRW	reg64, xmmreg, imm8	AVX512BW
VPEXTRW	mem16, xmmreg, imm8	AVX512BW
VPEXTRW	reg16, xmmreg, imm8	AVX512BW
VPEXTRW	reg32, xmmreg, imm8	AVX512BW
VPEXTRW	reg64, xmmreg, imm8	AVX512BW
VPGATHERDD	xmmreg mask, xmem32	AVX512VL
VPGATHERDD	ymmreg mask, ymem32	AVX512VL
VPGATHERDD	zmmreg mask, zmem32	AVX512
VPGATHERDQ	xmmreg mask, xmem64	AVX512VL
VPGATHERDQ	ymmreg mask, xmem64	AVX512VL
VPGATHERDQ	zmmreg mask, ymem64	AVX512
VPGATHERQD	xmmreg mask, xmem32	AVX512VL
VPGATHERQD	xmmreg mask, ymem32	AVX512VL
VPGATHERQD	ymmreg mask, zmem32	AVX512
VPGATHERQQ	xmmreg mask, xmem64	AVX512VL
VPGATHERQQ	ymmreg mask, ymem64	AVX512VL
VPGATHERQQ	zmmreg mask, zmem64	AVX512
VPINSRB	xmmreg, xmmreg*, reg32, imm8	AVX512BW
VPINSRB	xmmreg, xmmreg*, mem8, imm8	AVX512BW
VPINSRD	xmmreg, xmmreg*, rm32, imm8	AVX512DQ
VPINSRQ	xmmreg, xmmreg*, rm64, imm8	AVX512DQ
VPINSRW	xmmreg, xmmreg*, reg32, imm8	AVX512BW
VPINSRW	xmmreg, xmmreg*, mem16, imm8	AVX512BW
VPLZCNTD	xmmreg mask z, xmmrm128 b32	AVX512VL/CD
VPLZCNTD	ymmreg mask z, ymmrm256 b32	AVX512VL/CD
VPLZCNTD	zmmreg mask z, zmmrm512 b32	AVX512CD
VPLZCNTQ	xmmreg mask z, xmmrm128 b64	AVX512VL/CD
VPLZCNTQ	ymmreg mask z, ymmrm256 b64	AVX512VL/CD
VPLZCNTQ	zmmreg mask z, zmmrm512 b64	AVX512CD
VPMADD52HUQ	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL/IFMA
VPMADD52HUQ	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL/IFMA
VPMADD52HUQ	zmmreg mask z, zmmreg, zmmrm512 b64	AVX512IFMA
VPMADD52LUQ	xmmreg mask z, xmmreg, xmmrm128 b64	AVX512VL/IFMA
VPMADD52LUQ	ymmreg mask z, ymmreg, ymmrm256 b64	AVX512VL/IFMA
VPMADD52LUQ	zmmreg mask z, zmmreg, zmmrm512 b64	AVX512IFMA
VPMADDUBSW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMADDUBSW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMADDUBSW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMADDWD	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMADDWD	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMADDWD	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMAXSB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMAXSB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMAXSB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMAXSD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPMAXSD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPMAXSD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPMAXSQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPMAXSQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VPMAXSQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPMAXSW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMAXSW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMAXSW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMAXUB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMAXUB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMAXUB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMAXUD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPMAXUD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPMAXUD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPMAXUQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPMAXUQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL

VPMAXUQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPMAXUW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMAXUW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMAXUW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMINSB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMINSB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMINSB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMINSQ	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPMINSQ	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPMINSQ	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPMINSQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPMINSQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VPMINSQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPMINSW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMINSW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMINSW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMINUB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMINUB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMINUB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMINUD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPMINUD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPMINUD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPMINUQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPMINUQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VPMINUQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPMINUW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPMINUW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPMINUW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPMOVB2M	kreg, xmmreg	AVX512VL/BW
VPMOVB2M	kreg, ymmreg	AVX512VL/BW
VPMOVB2M	kreg, zmmreg	AVX512BW
VPMOVD2M	kreg, xmmreg	AVX512VL/DQ
VPMOVD2M	kreg, ymmreg	AVX512VL/DQ
VPMOVD2M	kreg, zmmreg	AVX512DQ
VPMOVBDB	xmmreg mask z, xmmreg	AVX512VL
VPMOVBDB	xmmreg mask z, ymmreg	AVX512VL
VPMOVBDB	xmmreg mask z, zmmreg	AVX512
VPMOVBDB	mem32 mask, xmmreg	AVX512VL
VPMOVBDB	mem64 mask, ymmreg	AVX512VL
VPMOVBDB	mem128 mask, zmmreg	AVX512
VPMOVDW	xmmreg mask z, xmmreg	AVX512VL
VPMOVDW	xmmreg mask z, ymmreg	AVX512VL
VPMOVDW	ymmreg mask z, zmmreg	AVX512
VPMOVDW	mem64 mask, xmmreg	AVX512VL
VPMOVDW	mem128 mask, ymmreg	AVX512VL
VPMOVDW	mem256 mask, zmmreg	AVX512
VPMOVM2B	xmmreg, kreg	AVX512VL/BW
VPMOVM2B	ymmreg, kreg	AVX512VL/BW
VPMOVM2B	zmmreg, kreg	AVX512BW
VPMOVM2D	xmmreg, kreg	AVX512VL/DQ
VPMOVM2D	ymmreg, kreg	AVX512VL/DQ
VPMOVM2D	zmmreg, kreg	AVX512DQ
VPMOVM2Q	xmmreg, kreg	AVX512VL/DQ
VPMOVM2Q	ymmreg, kreg	AVX512VL/DQ
VPMOVM2Q	zmmreg, kreg	AVX512DQ
VPMOVM2W	xmmreg, kreg	AVX512VL/BW
VPMOVM2W	ymmreg, kreg	AVX512VL/BW
VPMOVM2W	zmmreg, kreg	AVX512BW
VPMOVQ2M	kreg, xmmreg	AVX512VL/DQ
VPMOVQ2M	kreg, ymmreg	AVX512VL/DQ
VPMOVQ2M	kreg, zmmreg	AVX512DQ
VPMOVQB	xmmreg mask z, xmmreg	AVX512VL
VPMOVQB	xmmreg mask z, ymmreg	AVX512VL
VPMOVQB	xmmreg mask z, zmmreg	AVX512
VPMOVQB	mem16 mask, xmmreg	AVX512VL
VPMOVQB	mem32 mask, ymmreg	AVX512VL
VPMOVQB	mem64 mask, zmmreg	AVX512

VPMOVQD	xmmreg mask z, xmmreg	AVX512VL
VPMOVQD	xmmreg mask z, ymmreg	AVX512VL
VPMOVQD	ymmreg mask z, zmmreg	AVX512
VPMOVQD	mem64 mask, xmmreg	AVX512VL
VPMOVQD	mem128 mask, ymmreg	AVX512VL
VPMOVQD	mem256 mask, zmmreg	AVX512
VPMOVQW	xmmreg mask z, xmmreg	AVX512VL
VPMOVQW	xmmreg mask z, ymmreg	AVX512VL
VPMOVQW	xmmreg mask z, zmmreg	AVX512
VPMOVQW	mem32 mask, xmmreg	AVX512VL
VPMOVQW	mem64 mask, ymmreg	AVX512VL
VPMOVQW	mem128 mask, zmmreg	AVX512
VPMOVSDb	xmmreg mask z, xmmreg	AVX512VL
VPMOVSDb	xmmreg mask z, ymmreg	AVX512VL
VPMOVSDb	xmmreg mask z, zmmreg	AVX512
VPMOVSDb	mem32 mask, xmmreg	AVX512VL
VPMOVSDb	mem64 mask, ymmreg	AVX512VL
VPMOVSDb	mem128 mask, zmmreg	AVX512
VPMOVSDW	xmmreg mask z, xmmreg	AVX512VL
VPMOVSDW	xmmreg mask z, ymmreg	AVX512VL
VPMOVSDW	ymmreg mask z, zmmreg	AVX512
VPMOVSDW	mem64 mask, xmmreg	AVX512VL
VPMOVSDW	mem128 mask, ymmreg	AVX512VL
VPMOVSDW	mem256 mask, zmmreg	AVX512
VPMOVSQB	xmmreg mask z, xmmreg	AVX512VL
VPMOVSQB	xmmreg mask z, ymmreg	AVX512VL
VPMOVSQB	xmmreg mask z, zmmreg	AVX512
VPMOVSQB	mem16 mask, xmmreg	AVX512VL
VPMOVSQB	mem32 mask, ymmreg	AVX512VL
VPMOVSQB	mem64 mask, zmmreg	AVX512
VPMOVSQD	xmmreg mask z, xmmreg	AVX512VL
VPMOVSQD	xmmreg mask z, ymmreg	AVX512VL
VPMOVSQD	ymmreg mask z, zmmreg	AVX512
VPMOVSQD	mem64 mask, xmmreg	AVX512VL
VPMOVSQD	mem128 mask, ymmreg	AVX512VL
VPMOVSQD	mem256 mask, zmmreg	AVX512
VPMOVSQW	xmmreg mask z, xmmreg	AVX512VL
VPMOVSQW	xmmreg mask z, ymmreg	AVX512VL
VPMOVSQW	xmmreg mask z, zmmreg	AVX512
VPMOVSQW	mem32 mask, xmmreg	AVX512VL
VPMOVSQW	mem64 mask, ymmreg	AVX512VL
VPMOVSQW	mem128 mask, zmmreg	AVX512
VPMOVSWb	xmmreg mask z, xmmreg	AVX512VL/BW
VPMOVSWb	xmmreg mask z, ymmreg	AVX512VL/BW
VPMOVSWb	ymmreg mask z, zmmreg	AVX512BW
VPMOVSWb	mem64 mask, xmmreg	AVX512VL/BW
VPMOVSWb	mem128 mask, ymmreg	AVX512VL/BW
VPMOVSWb	mem256 mask, zmmreg	AVX512BW
VPMOVSXBD	xmmreg mask z, xmmrm32	AVX512VL
VPMOVSXBD	ymmreg mask z, xmmrm64	AVX512VL
VPMOVSXBD	zmmreg mask z, xmmrm128	AVX512
VPMOVSXBQ	xmmreg mask z, xmmrm16	AVX512VL
VPMOVSXBQ	ymmreg mask z, xmmrm32	AVX512VL
VPMOVSXBQ	zmmreg mask z, xmmrm64	AVX512
VPMOVSXBW	xmmreg mask z, xmmrm64	AVX512VL/BW
VPMOVSXBW	ymmreg mask z, xmmrm128	AVX512VL/BW
VPMOVSXBW	zmmreg mask z, ymmrm256	AVX512BW
VPMOVXSDQ	xmmreg mask z, xmmrm64	AVX512VL
VPMOVXSDQ	ymmreg mask z, xmmrm128	AVX512VL
VPMOVXSDQ	zmmreg mask z, ymmrm256	AVX512
VPMOVSXWD	xmmreg mask z, xmmrm64	AVX512VL
VPMOVSXWD	ymmreg mask z, xmmrm128	AVX512VL
VPMOVSXWD	zmmreg mask z, ymmrm256	AVX512
VPMOVSXWQ	xmmreg mask z, xmmrm32	AVX512VL
VPMOVSXWQ	ymmreg mask z, xmmrm64	AVX512VL
VPMOVSXWQ	zmmreg mask z, xmmrm128	AVX512
VPMOVUSDB	xmmreg mask z, xmmreg	AVX512VL

VPMOVUSDB	xmmreg mask z,ymmreg	AVX512VL
VPMOVUSDB	xmmreg mask z,zmmreg	AVX512
VPMOVUSDB	mem32 mask,xmmreg	AVX512VL
VPMOVUSDB	mem64 mask,ymmreg	AVX512VL
VPMOVUSDB	mem128 mask,zmmreg	AVX512
VPMOVUSDW	xmmreg mask z,xmmreg	AVX512VL
VPMOVUSDW	xmmreg mask z,ymmreg	AVX512VL
VPMOVUSDW	ymmreg mask z,zmmreg	AVX512
VPMOVUSDW	mem64 mask,xmmreg	AVX512VL
VPMOVUSDW	mem128 mask,ymmreg	AVX512VL
VPMOVUSDW	mem256 mask,zmmreg	AVX512
VPMOVUSQB	xmmreg mask z,xmmreg	AVX512VL
VPMOVUSQB	xmmreg mask z,ymmreg	AVX512VL
VPMOVUSQB	xmmreg mask z,zmmreg	AVX512
VPMOVUSQB	mem16 mask,xmmreg	AVX512VL
VPMOVUSQB	mem32 mask,ymmreg	AVX512VL
VPMOVUSQB	mem64 mask,zmmreg	AVX512
VPMOVUSQD	xmmreg mask z,xmmreg	AVX512VL
VPMOVUSQD	xmmreg mask z,ymmreg	AVX512VL
VPMOVUSQD	ymmreg mask z,zmmreg	AVX512
VPMOVUSQD	mem64 mask,xmmreg	AVX512VL
VPMOVUSQD	mem128 mask,ymmreg	AVX512VL
VPMOVUSQD	mem256 mask,zmmreg	AVX512
VPMOVUSQW	xmmreg mask z,xmmreg	AVX512VL
VPMOVUSQW	xmmreg mask z,ymmreg	AVX512VL
VPMOVUSQW	xmmreg mask z,zmmreg	AVX512
VPMOVUSQW	mem32 mask,xmmreg	AVX512VL
VPMOVUSQW	mem64 mask,ymmreg	AVX512VL
VPMOVUSQW	mem128 mask,zmmreg	AVX512
VPMOVUSWB	xmmreg mask z,xmmreg	AVX512VL/BW
VPMOVUSWB	xmmreg mask z,ymmreg	AVX512VL/BW
VPMOVUSWB	ymmreg mask z,zmmreg	AVX512BW
VPMOVUSWB	mem64 mask,xmmreg	AVX512VL/BW
VPMOVUSWB	mem128 mask,ymmreg	AVX512VL/BW
VPMOVUSWB	mem256 mask,zmmreg	AVX512BW
VPMOVW2M	kreg,xmmreg	AVX512VL/BW
VPMOVW2M	kreg,ymmreg	AVX512VL/BW
VPMOVW2M	kreg,zmmreg	AVX512BW
VPMOVWB	xmmreg mask z,xmmreg	AVX512VL/BW
VPMOVWB	xmmreg mask z,ymmreg	AVX512VL/BW
VPMOVWB	ymmreg mask z,zmmreg	AVX512BW
VPMOVWB	mem64 mask,xmmreg	AVX512VL/BW
VPMOVWB	mem128 mask,ymmreg	AVX512VL/BW
VPMOVWB	mem256 mask,zmmreg	AVX512BW
VPMOVZXBDB	xmmreg mask z,xmmrm32	AVX512VL
VPMOVZXBDB	ymmreg mask z,xmmrm64	AVX512VL
VPMOVZXBDB	zmmreg mask z,xmmrm128	AVX512
VPMOVZXBQ	xmmreg mask z,xmmrm16	AVX512VL
VPMOVZXBQ	ymmreg mask z,xmmrm32	AVX512VL
VPMOVZXBQ	zmmreg mask z,xmmrm64	AVX512
VPMOVZXBW	xmmreg mask z,xmmrm64	AVX512VL/BW
VPMOVZXBW	ymmreg mask z,xmmrm128	AVX512VL/BW
VPMOVZXBW	zmmreg mask z,ymmrm256	AVX512BW
VPMOVZXDQ	xmmreg mask z,xmmrm64	AVX512VL
VPMOVZXDQ	ymmreg mask z,xmmrm128	AVX512VL
VPMOVZXDQ	zmmreg mask z,ymmrm256	AVX512
VPMOVZXWD	xmmreg mask z,xmmrm64	AVX512VL
VPMOVZXWD	ymmreg mask z,xmmrm128	AVX512VL
VPMOVZXWD	zmmreg mask z,ymmrm256	AVX512
VPMOVZXWQ	xmmreg mask z,xmmrm32	AVX512VL
VPMOVZXWQ	ymmreg mask z,xmmrm64	AVX512VL
VPMOVZXWQ	zmmreg mask z,xmmrm128	AVX512
VPMULDQ	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VPMULDQ	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VPMULDQ	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512
VPMULHRWS	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPMULHRWS	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW



VPMULHRSW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPMULHUW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPMULHUW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPMULHUW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPMULHW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPMULHW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPMULHW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPMULLD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPMULLD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPMULLD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPMULLQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL/DQ
VPMULLQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL/DQ
VPMULLQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512DQ
VPMULLW	xmmreg	mask	z, xmmreg*, xmmrm128	AVX512VL/BW
VPMULLW	ymmreg	mask	z, ymmreg*, ymmrm256	AVX512VL/BW
VPMULLW	zmmreg	mask	z, zmmreg*, zmmrm512	AVX512BW
VPMULTISHIFTQB	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL/VBMI
VPMULTISHIFTQB	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL/VBMI
VPMULTISHIFTQB	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512VBMI
VPMULUDQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPMULUDQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPMULUDQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPORD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPORD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPORD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPORQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPORQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPORQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPROLD	xmmreg	mask	z, xmmrm128 b32*, imm8	AVX512VL
VPROLD	ymmreg	mask	z, ymmrm256 b32*, imm8	AVX512VL
VPROLD	zmmreg	mask	z, zmmrm512 b32*, imm8	AVX512
VPROLQ	xmmreg	mask	z, xmmrm128 b64*, imm8	AVX512VL
VPROLQ	ymmreg	mask	z, ymmrm256 b64*, imm8	AVX512VL
VPROLQ	zmmreg	mask	z, zmmrm512 b64*, imm8	AVX512
VPROLVD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPROLVD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPROLVD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPROLVQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPROLVQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPROLVQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPRORD	xmmreg	mask	z, xmmrm128 b32*, imm8	AVX512VL
VPRORD	ymmreg	mask	z, ymmrm256 b32*, imm8	AVX512VL
VPRORD	zmmreg	mask	z, zmmrm512 b32*, imm8	AVX512
VPRORQ	xmmreg	mask	z, xmmrm128 b64*, imm8	AVX512VL
VPRORQ	ymmreg	mask	z, ymmrm256 b64*, imm8	AVX512VL
VPRORQ	zmmreg	mask	z, zmmrm512 b64*, imm8	AVX512
VPRORVD	xmmreg	mask	z, xmmreg*, xmmrm128	b32 AVX512VL
VPRORVD	ymmreg	mask	z, ymmreg*, ymmrm256	b32 AVX512VL
VPRORVD	zmmreg	mask	z, zmmreg*, zmmrm512	b32 AVX512
VPRORVQ	xmmreg	mask	z, xmmreg*, xmmrm128	b64 AVX512VL
VPRORVQ	ymmreg	mask	z, ymmreg*, ymmrm256	b64 AVX512VL
VPRORVQ	zmmreg	mask	z, zmmreg*, zmmrm512	b64 AVX512
VPSADBW	xmmreg, xmmreg*		xmmrm128	AVX512VL/BW
VPSADBW	ymmreg, ymmreg*		ymmrm256	AVX512VL/BW
VPSADBW	zmmreg, zmmreg*		zmmrm512	AVX512BW
VPSCATTERDD	xmem32	mask, xmmreg		AVX512VL
VPSCATTERDD	yem32	mask, ymmreg		AVX512VL
VPSCATTERDD	zmem32	mask, zmmreg		AVX512
VPSCATTERDQ	xmem64	mask, xmmreg		AVX512VL
VPSCATTERDQ	xmem64	mask, ymmreg		AVX512VL
VPSCATTERDQ	yem64	mask, zmmreg		AVX512
VPSCATTERQD	xmem32	mask, xmmreg		AVX512VL
VPSCATTERQD	yem32	mask, xmmreg		AVX512VL
VPSCATTERQD	zmem32	mask, ymmreg		AVX512
VPSCATTERQQ	xmem64	mask, xmmreg		AVX512VL
VPSCATTERQQ	yem64	mask, ymmreg		AVX512VL
VPSCATTERQQ	zmem64	mask, zmmreg		AVX512

VPSHUFB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPSHUFB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPSHUFB	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPSHUFD	xmmreg mask z, xmmrm128 b32, imm8	AVX512VL
VPSHUFD	ymmreg mask z, ymmrm256 b32, imm8	AVX512VL
VPSHUFD	zmmreg mask z, zmmrm512 b32, imm8	AVX512
VPSHUFHW	xmmreg mask z, xmmrm128, imm8	AVX512VL/BW
VPSHUFHW	ymmreg mask z, ymmrm256, imm8	AVX512VL/BW
VPSHUFHW	zmmreg mask z, zmmrm512, imm8	AVX512BW
VPSHUFLW	xmmreg mask z, xmmrm128, imm8	AVX512VL/BW
VPSHUFLW	ymmreg mask z, ymmrm256, imm8	AVX512VL/BW
VPSHUFLW	zmmreg mask z, zmmrm512, imm8	AVX512BW
VPSLLD	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL
VPSLLD	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL
VPSLLD	zmmreg mask z, zmmreg*, zmmrm128	AVX512
VPSLLD	xmmreg mask z, xmmrm128 b32*, imm8	AVX512VL
VPSLLD	ymmreg mask z, ymmrm256 b32*, imm8	AVX512VL
VPSLLD	zmmreg mask z, zmmrm512 b32*, imm8	AVX512
VPSLLDQ	xmmreg, xmmrm128*, imm8	AVX512VL/BW
VPSLLDQ	ymmreg, ymmrm256*, imm8	AVX512VL/BW
VPSLLDQ	zmmreg, zmmrm512*, imm8	AVX512BW
VPSLLQ	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL
VPSLLQ	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL
VPSLLQ	zmmreg mask z, zmmreg*, zmmrm128	AVX512
VPSLLQ	xmmreg mask z, xmmrm128 b64*, imm8	AVX512VL
VPSLLQ	ymmreg mask z, ymmrm256 b64*, imm8	AVX512VL
VPSLLQ	zmmreg mask z, zmmrm512 b64*, imm8	AVX512
VPSLLVD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPSLLVD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPSLLVD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPSLLVQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPSLLVQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VPSLLVQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPSLLVW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPSLLVW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPSLLVW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPSLLW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPSLLW	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL/BW
VPSLLW	zmmreg mask z, zmmreg*, zmmrm128	AVX512BW
VPSLLW	xmmreg mask z, xmmrm128*, imm8	AVX512VL/BW
VPSLLW	ymmreg mask z, ymmrm256*, imm8	AVX512VL/BW
VPSLLW	zmmreg mask z, zmmrm512*, imm8	AVX512BW
VPSRAD	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL
VPSRAD	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL
VPSRAD	zmmreg mask z, zmmreg*, zmmrm128	AVX512
VPSRAD	xmmreg mask z, xmmrm128 b32*, imm8	AVX512VL
VPSRAD	ymmreg mask z, ymmrm256 b32*, imm8	AVX512VL
VPSRAD	zmmreg mask z, zmmrm512 b32*, imm8	AVX512
VPSRAQ	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL
VPSRAQ	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL
VPSRAQ	zmmreg mask z, zmmreg*, zmmrm128	AVX512
VPSRAQ	xmmreg mask z, xmmrm128 b64*, imm8	AVX512VL
VPSRAQ	ymmreg mask z, ymmrm256 b64*, imm8	AVX512VL
VPSRAQ	zmmreg mask z, zmmrm512 b64*, imm8	AVX512
VPSRAVD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VL
VPSRAVD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VL
VPSRAVD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512
VPSRAVQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VL
VPSRAVQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VL
VPSRAVQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512
VPSRAVW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPSRAVW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL/BW
VPSRAVW	zmmreg mask z, zmmreg*, zmmrm512	AVX512BW
VPSRAW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL/BW
VPSRAW	ymmreg mask z, ymmreg*, ymmrm128	AVX512VL/BW
VPSRAW	zmmreg mask z, zmmreg*, zmmrm128	AVX512BW
VPSRAW	xmmreg mask z, xmmrm128*, imm8	AVX512VL/BW

VPSRAW	ymmreg mask z,ymmrm256*,imm8	AVX512VL/BW
VPSRAW	zmmreg mask z,zmmrm512*,imm8	AVX512BW
VPSRLD	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL
VPSRLD	ymmreg mask z,ymmreg*,xmmrm128	AVX512VL
VPSRLD	zmmreg mask z,zmmreg*,xmmrm128	AVX512
VPSRLD	xmmreg mask z,xmmrm128 b32*,imm8	AVX512VL
VPSRLD	ymmreg mask z,ymmrm256 b32*,imm8	AVX512VL
VPSRLD	zmmreg mask z,zmmrm512 b32*,imm8	AVX512
VPSRLDQ	xmmreg,xmmrm128*,imm8	AVX512VL/BW
VPSRLDQ	ymmreg,ymmrm256*,imm8	AVX512VL/BW
VPSRLDQ	zmmreg,zmmrm512*,imm8	AVX512BW
VPSRLQ	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL
VPSRLQ	ymmreg mask z,ymmreg*,xmmrm128	AVX512VL
VPSRLQ	zmmreg mask z,zmmreg*,xmmrm128	AVX512
VPSRLQ	xmmreg mask z,xmmrm128 b64*,imm8	AVX512VL
VPSRLQ	ymmreg mask z,ymmrm256 b64*,imm8	AVX512VL
VPSRLQ	zmmreg mask z,zmmrm512 b64*,imm8	AVX512
VPSRLVD	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL
VPSRLVD	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL
VPSRLVD	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512
VPSRLVQ	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VPSRLVQ	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VPSRLVQ	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512
VPSRLVW	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSRLVW	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSRLVW	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSRLW	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSRLW	ymmreg mask z,ymmreg*,xmmrm128	AVX512VL/BW
VPSRLW	zmmreg mask z,zmmreg*,xmmrm128	AVX512BW
VPSRLW	xmmreg mask z,xmmrm128*,imm8	AVX512VL/BW
VPSRLW	ymmreg mask z,ymmrm256*,imm8	AVX512VL/BW
VPSRLW	zmmreg mask z,zmmrm512*,imm8	AVX512BW
VPSUBB	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBB	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBB	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSUBD	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL
VPSUBD	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL
VPSUBD	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512
VPSUBQ	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VPSUBQ	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VPSUBQ	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512
VPSUBSB	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBSB	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBSB	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSUBSW	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBSW	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBSW	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSUBUSB	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBUSB	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBUSB	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSUBUSW	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBUSW	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBUSW	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPSUBW	xmmreg mask z,xmmreg*,xmmrm128	AVX512VL/BW
VPSUBW	ymmreg mask z,ymmreg*,ymmrm256	AVX512VL/BW
VPSUBW	zmmreg mask z,zmmreg*,zmmrm512	AVX512BW
VPTERNLOGD	xmmreg mask z,xmmreg,xmmrm128 b32,imm8	AVX512VL
VPTERNLOGD	ymmreg mask z,ymmreg,ymmrm256 b32,imm8	AVX512VL
VPTERNLOGD	zmmreg mask z,zmmreg,zmmrm512 b32,imm8	AVX512
VPTERNLOGQ	xmmreg mask z,xmmreg,xmmrm128 b64,imm8	AVX512VL
VPTERNLOGQ	ymmreg mask z,ymmreg,ymmrm256 b64,imm8	AVX512VL
VPTERNLOGQ	zmmreg mask z,zmmreg,zmmrm512 b64,imm8	AVX512
VPTESTMB	kreg mask,xmmreg,xmmrm128	AVX512VL/BW
VPTESTMB	kreg mask,ymmreg,ymmrm256	AVX512VL/BW
VPTESTMB	kreg mask,zmmreg,zmmrm512	AVX512BW
VPTESTMD	kreg mask,xmmreg,xmmrm128 b32	AVX512VL
VPTESTMD	kreg mask,ymmreg,ymmrm256 b32	AVX512VL

VPTESTMD	kreg mask,zmmreg,zmmrm512 b32 AVX512
VPTESTMQ	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VPTESTMQ	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VPTESTMQ	kreg mask,zmmreg,zmmrm512 b64 AVX512
VPTESTMW	kreg mask,xmmreg,xmmrm128 AVX512VL/BW
VPTESTMW	kreg mask,ymmreg,ymmrm256 AVX512VL/BW
VPTESTMW	kreg mask,zmmreg,zmmrm512 AVX512BW
VPTESTNMB	kreg mask,xmmreg,xmmrm128 AVX512VL/BW
VPTESTNMB	kreg mask,ymmreg,ymmrm256 AVX512VL/BW
VPTESTNMB	kreg mask,zmmreg,zmmrm512 AVX512BW
VPTESTNMD	kreg mask,xmmreg,xmmrm128 b32 AVX512VL
VPTESTNMD	kreg mask,ymmreg,ymmrm256 b32 AVX512VL
VPTESTNMD	kreg mask,zmmreg,zmmrm512 b32 AVX512
VPTESTNMQ	kreg mask,xmmreg,xmmrm128 b64 AVX512VL
VPTESTNMQ	kreg mask,ymmreg,ymmrm256 b64 AVX512VL
VPTESTNMQ	kreg mask,zmmreg,zmmrm512 b64 AVX512
VPTESTNMW	kreg mask,xmmreg,xmmrm128 AVX512VL/BW
VPTESTNMW	kreg mask,ymmreg,ymmrm256 AVX512VL/BW
VPTESTNMW	kreg mask,zmmreg,zmmrm512 AVX512BW
VPUNPCKHBW	xmmreg mask z,xmmreg*,xmmrm128 AVX512VL/BW
VPUNPCKHBW	ymmreg mask z,ymmreg*,ymmrm256 AVX512VL/BW
VPUNPCKHBW	zmmreg mask z,zmmreg*,zmmrm512 AVX512BW
VPUNPCKHDQ	xmmreg mask z,xmmreg*,xmmrm128 b32 AVX512VL
VPUNPCKHDQ	ymmreg mask z,ymmreg*,ymmrm256 b32 AVX512VL
VPUNPCKHDQ	zmmreg mask z,zmmreg*,zmmrm512 b32 AVX512
VPUNPCKHQDQ	xmmreg mask z,xmmreg*,xmmrm128 b64 AVX512VL
VPUNPCKHQDQ	ymmreg mask z,ymmreg*,ymmrm256 b64 AVX512VL
VPUNPCKHQDQ	zmmreg mask z,zmmreg*,zmmrm512 b64 AVX512
VPUNPCKHWD	xmmreg mask z,xmmreg*,xmmrm128 AVX512VL/BW
VPUNPCKHWD	ymmreg mask z,ymmreg*,ymmrm256 AVX512VL/BW
VPUNPCKHWD	zmmreg mask z,zmmreg*,zmmrm512 AVX512BW
VPUNPCKLBW	xmmreg mask z,xmmreg*,xmmrm128 AVX512VL/BW
VPUNPCKLBW	ymmreg mask z,ymmreg*,ymmrm256 AVX512VL/BW
VPUNPCKLBW	zmmreg mask z,zmmreg*,zmmrm512 AVX512BW
VPUNPCKLDQ	xmmreg mask z,xmmreg*,xmmrm128 b32 AVX512VL
VPUNPCKLDQ	ymmreg mask z,ymmreg*,ymmrm256 b32 AVX512VL
VPUNPCKLDQ	zmmreg mask z,zmmreg*,zmmrm512 b32 AVX512
VPUNPCKLQDQ	xmmreg mask z,xmmreg*,xmmrm128 b64 AVX512VL
VPUNPCKLQDQ	ymmreg mask z,ymmreg*,ymmrm256 b64 AVX512VL
VPUNPCKLQDQ	zmmreg mask z,zmmreg*,zmmrm512 b64 AVX512
VPUNPCKLWD	xmmreg mask z,xmmreg*,xmmrm128 AVX512VL/BW
VPUNPCKLWD	ymmreg mask z,ymmreg*,ymmrm256 AVX512VL/BW
VPUNPCKLWD	zmmreg mask z,zmmreg*,zmmrm512 AVX512BW
VPXORD	xmmreg mask z,xmmreg*,xmmrm128 b32 AVX512VL
VPXORD	ymmreg mask z,ymmreg*,ymmrm256 b32 AVX512VL
VPXORD	zmmreg mask z,zmmreg*,zmmrm512 b32 AVX512
VPXORQ	xmmreg mask z,xmmreg*,xmmrm128 b64 AVX512VL
VPXORQ	ymmreg mask z,ymmreg*,ymmrm256 b64 AVX512VL
VPXORQ	zmmreg mask z,zmmreg*,zmmrm512 b64 AVX512
VRANGEPD	xmmreg mask z,xmmreg*,xmmrm128 b64,imm8 AVX512VL/DQ
VRANGEPD	ymmreg mask z,ymmreg*,ymmrm256 b64,imm8 AVX512VL/DQ
VRANGEPD	zmmreg mask z,zmmreg*,zmmrm512 b64 sae,imm8 AVX512DQ
VRANGEPS	xmmreg mask z,xmmreg*,xmmrm128 b32,imm8 AVX512VL/DQ
VRANGEPS	ymmreg mask z,ymmreg*,ymmrm256 b32,imm8 AVX512VL/DQ
VRANGEPS	zmmreg mask z,zmmreg*,zmmrm512 b32 sae,imm8 AVX512DQ
VRANGESD	xmmreg mask z,xmmreg*,xmmrm64 sae,imm8 AVX512DQ
VRANGESS	xmmreg mask z,xmmreg*,xmmrm32 sae,imm8 AVX512DQ
VRCP14PD	xmmreg mask z,xmmrm128 b64 AVX512VL
VRCP14PD	ymmreg mask z,ymmrm256 b64 AVX512VL
VRCP14PD	zmmreg mask z,zmmrm512 b64 AVX512
VRCP14PS	xmmreg mask z,xmmrm128 b32 AVX512VL
VRCP14PS	ymmreg mask z,ymmrm256 b32 AVX512VL
VRCP14PS	zmmreg mask z,zmmrm512 b32 AVX512
VRCP14SD	xmmreg mask z,xmmreg*,xmmrm64 AVX512
VRCP14SS	xmmreg mask z,xmmreg*,xmmrm32 AVX512
VRCP28PD	zmmreg mask z,zmmrm512 b64 sae AVX512ER
VRCP28PS	zmmreg mask z,zmmrm512 b32 sae AVX512ER

VRCP28SD	xmmreg mask z,xmmreg*,xmrm64 sae AVX512ER
VRCP28SS	xmmreg mask z,xmmreg*,xmrm32 sae AVX512ER
VREDUCEPD	xmmreg mask z,xmrm128 b64,imm8 AVX512VL/DQ
VREDUCEPD	ymmreg mask z,ymrm256 b64,imm8 AVX512VL/DQ
VREDUCEPD	zmmreg mask z,zmrm512 b64 sae,imm8 AVX512DQ
VREDUCEPS	xmmreg mask z,xmrm128 b32,imm8 AVX512VL/DQ
VREDUCEPS	ymmreg mask z,ymrm256 b32,imm8 AVX512VL/DQ
VREDUCEPS	zmmreg mask z,zmrm512 b32 sae,imm8 AVX512DQ
VREDUCESD	xmmreg mask z,xmmreg*,xmrm64 sae,imm8 AVX512DQ
VREDUCESD	ymmreg mask z,xmmreg*,xmrm32 sae,imm8 AVX512DQ
VRNDSCALEPD	xmmreg mask z,xmrm128 b64,imm8 AVX512VL
VRNDSCALEPD	ymmreg mask z,ymrm256 b64,imm8 AVX512VL
VRNDSCALEPD	zmmreg mask z,zmrm512 b64 sae,imm8 AVX512
VRNDSCALEPS	xmmreg mask z,xmrm128 b32,imm8 AVX512VL
VRNDSCALEPS	ymmreg mask z,ymrm256 b32,imm8 AVX512VL
VRNDSCALEPS	zmmreg mask z,zmrm512 b32 sae,imm8 AVX512
VRNDSCALESD	xmmreg mask z,xmmreg*,xmrm64 sae,imm8 AVX512
VRNDSCALESS	xmmreg mask z,xmmreg*,xmrm32 sae,imm8 AVX512
VRSQRT14PD	xmmreg mask z,xmrm128 b64 AVX512VL
VRSQRT14PD	ymmreg mask z,ymrm256 b64 AVX512VL
VRSQRT14PD	zmmreg mask z,zmrm512 b64 AVX512
VRSQRT14PS	xmmreg mask z,xmrm128 b32 AVX512VL
VRSQRT14PS	ymmreg mask z,ymrm256 b32 AVX512VL
VRSQRT14PS	zmmreg mask z,zmrm512 b32 AVX512
VRSQRT14SD	xmmreg mask z,xmmreg*,xmrm64 AVX512
VRSQRT14SS	xmmreg mask z,xmmreg*,xmrm32 AVX512
VRSQRT28PD	zmmreg mask z,zmrm512 b64 sae AVX512ER
VRSQRT28PS	zmmreg mask z,zmrm512 b32 sae AVX512ER
VRSQRT28SD	xmmreg mask z,xmmreg*,xmrm64 sae AVX512ER
VRSQRT28SS	xmmreg mask z,xmmreg*,xmrm32 sae AVX512ER
VSCALEFPD	xmmreg mask z,xmmreg*,xmrm128 b64 AVX512VL
VSCALEFPD	ymmreg mask z,ymmreg*,ymrm256 b64 AVX512VL
VSCALEFPD	zmmreg mask z,zmmreg*,zmrm512 b64 er AVX512
VSCALEFPS	xmmreg mask z,xmmreg*,xmrm128 b32 AVX512VL
VSCALEFPS	ymmreg mask z,ymmreg*,ymrm256 b32 AVX512VL
VSCALEFPS	zmmreg mask z,zmmreg*,zmrm512 b32 er AVX512
VSCALEFSD	xmmreg mask z,xmmreg*,xmrm64 er AVX512
VSCALEFSS	xmmreg mask z,xmmreg*,xmrm32 er AVX512
VSCATTERDPD	xmem64 mask,xmmreg AVX512VL
VSCATTERDPD	xmem64 mask,ymmreg AVX512VL
VSCATTERDPD	yem64 mask,zmmreg AVX512
VSCATTERDPS	xmem32 mask,xmmreg AVX512VL
VSCATTERDPS	yem32 mask,ymmreg AVX512VL
VSCATTERDPS	zem32 mask,zmmreg AVX512
VSCATTERPF0DPD	yem64 mask AVX512PF
VSCATTERPF0DPS	zem32 mask AVX512PF
VSCATTERPF0QPD	zem64 mask AVX512PF
VSCATTERPF0QPS	zem32 mask AVX512PF
VSCATTERPF1DPD	yem64 mask AVX512PF
VSCATTERPF1DPS	zem32 mask AVX512PF
VSCATTERPF1QPD	zem64 mask AVX512PF
VSCATTERPF1QPS	zem32 mask AVX512PF
VSCATTERQPD	xmem64 mask,xmmreg AVX512VL
VSCATTERQPD	yem64 mask,ymmreg AVX512VL
VSCATTERQPD	zem64 mask,zmmreg AVX512
VSCATTERQPS	xmem32 mask,xmmreg AVX512VL
VSCATTERQPS	yem32 mask,xmmreg AVX512VL
VSCATTERQPS	zem32 mask,ymmreg AVX512
VSHUFF32X4	ymmreg mask z,ymmreg*,ymrm256 b32,imm8 AVX512VL
VSHUFF32X4	zmmreg mask z,zmmreg*,zmrm512 b32,imm8 AVX512
VSHUFF64X2	ymmreg mask z,ymmreg*,ymrm256 b64,imm8 AVX512VL
VSHUFF64X2	zmmreg mask z,zmmreg*,zmrm512 b64,imm8 AVX512
VSHUFFI32X4	ymmreg mask z,ymmreg*,ymrm256 b32,imm8 AVX512VL
VSHUFFI32X4	zmmreg mask z,zmmreg*,zmrm512 b32,imm8 AVX512
VSHUFFI64X2	ymmreg mask z,ymmreg*,ymrm256 b64,imm8 AVX512VL
VSHUFFI64X2	zmmreg mask z,zmmreg*,zmrm512 b64,imm8 AVX512
VSHUFPD	xmmreg mask z,xmmreg*,xmrm128 b64,imm8 AVX512VL

VSHUFPD	ymmreg mask z,ymmreg*,ymmrm256 b64,imm8	AVX512VL
VSHUFPD	zmmreg mask z,zmmreg*,zmmrm512 b64,imm8	AVX512
VSHUFPS	xmmreg mask z,xmmreg*,xmmrm128 b32,imm8	AVX512VL
VSHUFPS	ymmreg mask z,ymmreg*,ymmrm256 b32,imm8	AVX512VL
VSHUFPS	zmmreg mask z,zmmreg*,zmmrm512 b32,imm8	AVX512
VSQRTPD	xmmreg mask z,xmmrm128 b64	AVX512VL
VSQRTPD	ymmreg mask z,ymmrm256 b64	AVX512VL
VSQRTPD	zmmreg mask z,zmmrm512 b64 er	AVX512
VSQRTPS	xmmreg mask z,xmmrm128 b32	AVX512VL
VSQRTPS	ymmreg mask z,ymmrm256 b32	AVX512VL
VSQRTPS	zmmreg mask z,zmmrm512 b32 er	AVX512
VSQRTSD	xmmreg mask z,xmmreg*,xmmrm64 er	AVX512
VSQRTSS	xmmreg mask z,xmmreg*,xmmrm32 er	AVX512
VSUBPD	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VSUBPD	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VSUBPD	zmmreg mask z,zmmreg*,zmmrm512 b64 er	AVX512
VSUBPS	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL
VSUBPS	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL
VSUBPS	zmmreg mask z,zmmreg*,zmmrm512 b32 er	AVX512
VSUBSD	xmmreg mask z,xmmreg*,xmmrm64 er	AVX512
VSUBSS	xmmreg mask z,xmmreg*,xmmrm32 er	AVX512
VUCOMISD	xmmreg,xmmrm64 sae	AVX512
VUCOMISS	xmmreg,xmmrm32 sae	AVX512
VUNPCKHPD	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VUNPCKHPD	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VUNPCKHPD	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512
VUNPCKHPS	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL
VUNPCKHPS	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL
VUNPCKHPS	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512
VUNPCKLPD	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL
VUNPCKLPD	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL
VUNPCKLPD	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512
VUNPCKLPS	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL
VUNPCKLPS	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL
VUNPCKLPS	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512
VXORPD	xmmreg mask z,xmmreg*,xmmrm128 b64	AVX512VL/DQ
VXORPD	ymmreg mask z,ymmreg*,ymmrm256 b64	AVX512VL/DQ
VXORPD	zmmreg mask z,zmmreg*,zmmrm512 b64	AVX512DQ
VXORPS	xmmreg mask z,xmmreg*,xmmrm128 b32	AVX512VL/DQ
VXORPS	ymmreg mask z,ymmreg*,ymmrm256 b32	AVX512VL/DQ
VXORPS	zmmreg mask z,zmmreg*,zmmrm512 b32	AVX512DQ

#### F.1.49 Intel memory protection keys for userspace (PKU aka PKEYs)

RDPKRU	LONG
WRPKRU	LONG

#### F.1.50 Read Processor ID

RDPID	reg32	NOLONG
RDPID	reg64	LONG
RDPID	reg32	LONG, UNDOC

#### F.1.51 New memory instructions

CLFLUSHOPT	mem	
CLWB	mem	
PCOMMIT		NEVER, NOP
CLZERO		AMD
CLZERO	reg_ax	AMD, ND, NOLONG
CLZERO	reg_eax	AMD, ND
CLZERO	reg_rax	AMD, ND, LONG

#### F.1.52 Processor trace write

PTWRITE	rm32	
PTWRITE	rm64	LONG

## F.1.53 Instructions from the Intel Instruction Set Extensions,

### F.1.54 doc 319433-034 May 2018

CLDEMOTE	mem	
MOVDIRI	mem32, reg32	SD
MOVDIRI	mem64, reg64	LONG
MOVDIR64B	reg16, mem512	NOLONG
MOVDIR64B	reg32, mem512	
MOVDIR64B	reg64, mem512	LONG
PCONFIG		
TPAUSE	reg32	
TPAUSE	reg32, reg_edx, reg_eax	ND
UMONITOR	reg16	NOLONG
UMONITOR	reg32	
UMONITOR	reg64	LONG
UMWAIT	reg32	
UMWAIT	reg32, reg_edx, reg_eax	ND
WBNOINVD		

### F.1.55 Galois field operations (GFNI)

GF2P8AFFINEINVB	xmmreg, xmmrm128, imm8	GFNI, SSE
VGF2P8AFFINEINVB	xmmreg, xmmreg*, xmmrm128, imm8	GFNI, AVX
VGF2P8AFFINEINVB	ymmreg, ymmreg*, ymmrm256, imm8	GFNI, AVX
VGF2P8AFFINEINVB	xmmreg mask z, xmmreg*, xmmrm128 b64, imm8	AVX512VL, GFNI
VGF2P8AFFINEINVB	ymmreg mask z, ymmreg*, ymmrm256 b64, imm8	AVX512VL, GFNI
VGF2P8AFFINEINVB	zmmreg mask z, zmmreg*, zmmrm512 b64, imm8	AVX512, GFNI
GF2P8AFFINEQB	xmmreg, xmmrm128, imm8	GFNI, SSE
VGF2P8AFFINEQB	xmmreg, xmmreg*, xmmrm128, imm8	GFNI, AVX
VGF2P8AFFINEQB	ymmreg, ymmreg*, ymmrm256, imm8	GFNI, AVX
VGF2P8AFFINEQB	xmmreg mask z, xmmreg*, xmmrm128 b64, imm8	AVX512VL, GFNI
VGF2P8AFFINEQB	ymmreg mask z, ymmreg*, ymmrm256 b64, imm8	AVX512VL, GFNI
VGF2P8AFFINEQB	zmmreg mask z, zmmreg*, zmmrm512 b64, imm8	AVX512, GFNI
GF2P8MULB	xmmreg, xmmrm128	GFNI, SSE
VGF2P8MULB	xmmreg, xmmreg*, xmmrm128	GFNI, AVX
VGF2P8MULB	ymmreg, ymmreg*, ymmrm256	GFNI, AVX
VGF2P8MULB	xmmreg mask z, xmmreg*, xmmrm128	AVX512VL, GFNI
VGF2P8MULB	ymmreg mask z, ymmreg*, ymmrm256	AVX512VL, GFNI
VGF2P8MULB	zmmreg mask z, zmmreg*, zmmrm512	AVX512, GFNI

### F.1.56 AVX512 Vector Bit Manipulation Instructions 2

VPCOMPRESSB	mem128 mask, xmmreg	AVX512VBMI2/VL
VPCOMPRESSB	mem256 mask, ymmreg	AVX512VBMI2/VL
VPCOMPRESSB	mem512 mask, zmmreg	AVX512VBMI2
VPCOMPRESSB	xmmreg mask z, xmmreg	AVX512VBMI2/VL
VPCOMPRESSB	ymmreg mask z, ymmreg	AVX512VBMI2/VL
VPCOMPRESSB	zmmreg mask z, zmmreg	AVX512VBMI2
VPCOMPRESSW	mem128 mask, xmmreg	AVX512VBMI2/VL
VPCOMPRESSW	mem256 mask, ymmreg	AVX512VBMI2/VL
VPCOMPRESSW	mem512 mask, zmmreg	AVX512VBMI2
VPCOMPRESSW	xmmreg mask z, xmmreg	AVX512VBMI2/VL
VPCOMPRESSW	ymmreg mask z, ymmreg	AVX512VBMI2/VL
VPCOMPRESSW	zmmreg mask z, zmmreg	AVX512VBMI2
VPEXPANDB	xmmreg mask z, xmmrm128	AVX512VBMI2/VL
VPEXPANDB	ymmreg mask z, ymmrm256	AVX512VBMI2/VL
VPEXPANDB	zmmreg mask z, zmmrm512	AVX512VBMI2
VPEXPANDW	xmmreg mask z, xmmrm128	AVX512VBMI2/VL
VPEXPANDW	ymmreg mask z, ymmrm256	AVX512VBMI2/VL
VPEXPANDW	zmmreg mask z, zmmrm512	AVX512VBMI2
VPSHLDW	xmmreg mask z, xmmreg*, xmmrm128, imm8	AVX512VBMI2/VL
VPSHLDW	ymmreg mask z, ymmreg*, ymmrm256, imm8	AVX512VBMI2/VL
VPSHLDW	zmmreg mask z, zmmreg*, zmmrm512, imm8	AVX512VBMI2
VPSHLDD	xmmreg mask z, xmmreg*, xmmrm128 b32, imm8	AVX512VBMI2/VL
VPSHLDD	ymmreg mask z, ymmreg*, ymmrm256 b32, imm8	AVX512VBMI2/VL
VPSHLDD	zmmreg mask z, zmmreg*, zmmrm512 b32, imm8	AVX512VBMI2

VPSHLDQ	xmmreg mask z, xmmreg*, xmmrm128 b64, imm8	AVX512VBMI2/VL
VPSHLDQ	ymmreg mask z, ymmreg*, ymmrm256 b64, imm8	AVX512VBMI2/VL
VPSHLDQ	zmmreg mask z, zmmreg*, zmmrm512 b64, imm8	AVX512VBMI2
VPSHLDVW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VBMI2/VL
VPSHLDVW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VBMI2/VL
VPSHLDVW	zmmreg mask z, zmmreg*, zmmrm512	AVX512VBMI2
VPSHLDVD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VBMI2/VL
VPSHLDVD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VBMI2/VL
VPSHLDVD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VBMI2
VPSHLDVQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VBMI2/VL
VPSHLDVQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VBMI2/VL
VPSHLDVQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512VBMI2
VPSHRDW	xmmreg mask z, xmmreg*, xmmrm128, imm8	AVX512VBMI2/VL
VPSHRDW	ymmreg mask z, ymmreg*, ymmrm256, imm8	AVX512VBMI2/VL
VPSHRDW	zmmreg mask z, zmmreg*, zmmrm512, imm8	AVX512VBMI2
VPSHRDD	xmmreg mask z, xmmreg*, xmmrm128 b32, imm8	AVX512VBMI2/VL
VPSHRDD	ymmreg mask z, ymmreg*, ymmrm256 b32, imm8	AVX512VBMI2/VL
VPSHRDD	zmmreg mask z, zmmreg*, zmmrm512 b32, imm8	AVX512VBMI2
VPSHRDQ	xmmreg mask z, xmmreg*, xmmrm128 b64, imm8	AVX512VBMI2/VL
VPSHRDQ	ymmreg mask z, ymmreg*, ymmrm256 b64, imm8	AVX512VBMI2/VL
VPSHRDQ	zmmreg mask z, zmmreg*, zmmrm512 b64, imm8	AVX512VBMI2
VPSHRDVW	xmmreg mask z, xmmreg*, xmmrm128	AVX512VBMI2/VL
VPSHRDVW	ymmreg mask z, ymmreg*, ymmrm256	AVX512VBMI2/VL
VPSHRDVW	zmmreg mask z, zmmreg*, zmmrm512	AVX512VBMI2
VPSHRDVD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VBMI2/VL
VPSHRDVD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VBMI2/VL
VPSHRDVD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VBMI2
VPSHRDVQ	xmmreg mask z, xmmreg*, xmmrm128 b64	AVX512VBMI2/VL
VPSHRDVQ	ymmreg mask z, ymmreg*, ymmrm256 b64	AVX512VBMI2/VL
VPSHRDVQ	zmmreg mask z, zmmreg*, zmmrm512 b64	AVX512VBMI2

### F.1.57 AVX512 VNNI

VPDPBUSD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VNNI/VL
VPDPBUSD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VNNI/VL
VPDPBUSD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VNNI
VPDPBUSDS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VNNI/VL
VPDPBUSDS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VNNI/VL
VPDPBUSDS	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VNNI
VPDPWSSD	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VNNI/VL
VPDPWSSD	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VNNI/VL
VPDPWSSD	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VNNI
VPDPWSSDS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512VNNI/VL
VPDPWSSDS	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512VNNI/VL
VPDPWSSDS	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512VNNI

### F.1.58 AVX512 Bit Algorithms

VPOPCNTB	xmmreg mask z, xmmrm128	AVX512BITALG/VL
VPOPCNTB	ymmreg mask z, ymmrm256	AVX512BITALG/VL
VPOPCNTB	zmmreg mask z, zmmrm512	AVX512BITALG
VPOPCNTW	xmmreg mask z, xmmrm128	AVX512BITALG/VL
VPOPCNTW	ymmreg mask z, ymmrm256	AVX512BITALG/VL
VPOPCNTW	zmmreg mask z, zmmrm512	AVX512BITALG
VPOPCNTD	xmmreg mask z, xmmrm128	AVX512VPOPCNTDQ/VL
VPOPCNTD	ymmreg mask z, ymmrm256	AVX512VPOPCNTDQ/VL
VPOPCNTD	zmmreg mask z, zmmrm512	AVX512VPOPCNTDQ
VPOPCNTQ	xmmreg mask z, xmmrm128	AVX512VPOPCNTDQ/VL
VPOPCNTQ	ymmreg mask z, ymmrm256	AVX512VPOPCNTDQ/VL
VPOPCNTQ	zmmreg mask z, zmmrm512	AVX512VPOPCNTDQ
VPSHUFBITQMB	kreg mask, xmmreg, xmmrm128	AVX512BITALG/VL
VPSHUFBITQMB	kreg mask, ymmreg, ymmrm256	AVX512BITALG/VL
VPSHUFBITQMB	kreg mask, zmmreg, zmmrm512	AVX512BITALG

### F.1.59 AVX512 4-iteration Multiply-Add

V4FMADDPs	zmmreg mask z, zmmreg rs4, mem	AVX512FMAPS, SO
V4FNADDPs	zmmreg mask z, zmmreg rs4, mem	AVX512FMAPS, SO



V4FMADDSS	zmmreg mask z, zmmreg rs4, mem	AVX5124FMAPS, SO
V4FNMADDSS	zmmreg mask z, zmmreg rs4, mem	AVX5124FMAPS, SO

### F.1.60 AVX512 4-iteration Dot Product

V4DPWSSDS	zmmreg mask z, zmmreg rs4, mem	AVX5124VNNIW, SO
V4DPWSSD	zmmreg mask z, zmmreg rs4, mem	AVX5124VNNIW, SO

### F.1.61 Intel Software Guard Extensions (SGX)

ENCLS	SGX
ENCLU	SGX
ENCLV	SGX

### F.1.62 Intel Control-Flow Enforcement Technology (CET)

CLRSSBSY	mem64	CET
ENDBR32		CET
ENDBR64		CET
INCSSPD	reg32	CET
INCSSPQ	reg64	CET, LONG
RDSSPD	reg32	CET
RDSSPQ	reg64	CET, LONG
RSTORSSP	mem64	CET
SAVEPREVSSP		CET
SETSSBSY		CET
WRUSSD	mem32, reg32	CET
WRUSSQ	mem64, reg64	CET, LONG
WRSSD	mem32, reg32	CET
WRSSQ	mem64, reg64	CET, LONG

### F.1.63 Instructions from ISE doc 319433-040, June 2020

ENQCMD	reg16, mem512	ENQCMD, SZ, NO LONG
ENQCMD	reg32, mem512	ENQCMD, SZ, NO LONG, ND
ENQCMD	reg32, mem512	ENQCMD, SZ
ENQCMD	reg64, mem512	ENQCMD, SZ, LONG
ENQCMDs	reg16, mem512	ENQCMD, SZ, NO LONG, PRIV
ENQCMDs	reg32, mem512	ENQCMD, SZ, NO LONG, PRIV, ND
ENQCMDs	reg32, mem512	ENQCMD, SZ, PRIV
ENQCMDs	reg64, mem512	ENQCMD, SZ, PRIV, LONG
PCONFIG		PCONFIG, PRIV
SERIALIZE		SERIALIZE
WBNOINVD		WBNOINVD, PRIV
XRESLDRK		TSXLDTRK
XSUSLDRK		TSXLDTRK

### F.1.64 AVX512 Bfloat16 instructions

VCVTNE2PS2BF16	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512BF16
VCVTNE2PS2BF16	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512BF16
VCVTNE2PS2BF16	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512BF16
VCVTNEPS2BF16	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512BF16
VCVTNEPS2BF16	ymmreg mask z, ymmreg*, ymmrm256 b32	AVX512BF16
VCVTNEPS2BF16	zmmreg mask z, zmmreg*, zmmrm512 b32	AVX512BF16
VDPBF16PS	xmmreg mask z, xmmreg*, xmmrm128 b32	AVX512BF16
VDPBF16PS	ymmreg mask z, ymmreg*, ymmrm128 b32	AVX512BF16
VDPBF16PS	zmmreg mask z, zmmreg*, zmmrm128 b32	AVX512BF16

### F.1.65 AVX512 mask intersect instructions

VP2INTERSECTD	kreg rs2, xmmreg, xmmrm128 b32	AVX512BF16
VP2INTERSECTD	kreg rs2, ymmreg, ymmrm128 b32	AVX512BF16
VP2INTERSECTD	kreg rs2, zmmreg, zmmrm128 b32	AVX512BF16

### F.1.66 Intel Advanced Matrix Extensions (AMX)

LD TILECFG	mem512	AMXTILE, SZ, LONG
ST TILECFG	mem512	AMXTILE, SZ, LONG

TDPBF16PS	tmmreg, tmmreg, tmmreg	AMXBF16, LONG
TDPBSSD	tmmreg, tmmreg, tmmreg	AMXINT8, LONG
TDPBSUD	tmmreg, tmmreg, tmmreg	AMXINT8, LONG
TDPBUSD	tmmreg, tmmreg, tmmreg	AMXINT8, LONG
TDPBUUD	tmmreg, tmmreg, tmmreg	AMXINT8, LONG
TILELOADD	tmmreg, mem	AMXTILE, MIB, SIB, SX, LONG
TILELOADDT1	tmmreg, mem	AMXTILE, MIB, SIB, SX, LONG
TILERELEASE		AMXTILE, LONG
TILESTORED	mem, tmmreg	AMXTILE, MIB, SIB, SX, LONG
TILEZERO	tmmreg	AMXTILE, LONG

### F.1.67 Intel AVX512-FP16 instructions

VADDPH	xmmreg mask z, xmmreg*, xmmrm16 b16	AVX512FP16/VL
VADDPH	ymmreg mask z, ymmreg*, ymmrm16 b16	AVX512FP16/VL
VADDPH	zmmreg mask z, zmmreg*, zmmrm16 b16 er	AVX512FP16
VADDSH	xmmreg mask z, xmmreg*, xmmrm16 er	AVX512FP16
VCMPPH	kreg mask, xmmreg*, xmmrm16 b16, imm8	AVX512FP16/VL
VCMPPH	kreg mask, ymmreg*, ymmrm16 b16, imm8	AVX512FP16/VL
VCMPPH	kreg mask, zmmreg*, zmmrm16 b16 sae, imm8	AVX512FP16
VCMPSH	kreg mask, xmmreg*, xmmrm16 sae, imm8	AVX512FP16
VCOMISH	xmmreg, xmmrm16 sae	AVX512FP16
VCVTDQ2PH	xmmreg mask z, xmmrm128 b32	AVX512FP16/VL
VCVTDQ2PH	ymmreg mask z, ymmrm256 b32	AVX512FP16/VL
VCVTDQ2PH	zmmreg mask z, zmmrm512 b32 er	AVX512FP16
VCVTPD2PH	xmmreg mask z, xmmrm128 b64	AVX512FP16/VL
VCVTPD2PH	ymmreg mask z, ymmrm256 b64	AVX512FP16/VL
VCVTPD2PH	zmmreg mask z, zmmrm512 b64 er	AVX512FP16
VCVTPH2DQ	xmmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2DQ	ymmreg mask z, xmmrm128 b16	AVX512FP16/VL
VCVTPH2DQ	zmmreg mask z, ymmrm256 b16 er	AVX512FP16
VCVTPH2PD	xmmreg mask z, xmmrm32 b16	AVX512FP16/VL
VCVTPH2PD	ymmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2PD	zmmreg mask z, xmmrm128 b16 sae	AVX512FP16
VCVTPH2PS	xmmreg, xmmrm64	AVX512FC16
VCVTPH2PS	ymmreg, xmmrm128	AVX512FC16
VCVTPH2PS	xmmreg mask z, xmmrm64	AVX512VL
VCVTPH2PS	ymmreg mask z, xmmrm128	AVX512VL
VCVTPH2PS	zmmreg mask z, ymmrm256 sae	AVX512
VCVTPH2PSX	xmmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2PSX	ymmreg mask z, xmmrm128 b16	AVX512FP16/VL
VCVTPH2PSX	zmmreg mask z, ymmrm256 b16 sae	AVX512FP16
VCVTPH2QQ	xmmreg mask z, xmmrm32 b16	AVX512FP16/VL
VCVTPH2QQ	ymmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2QQ	zmmreg mask z, xmmrm128 b16 er	AVX512FP16
VCVTPH2UDQ	xmmreg mask z, xmmrm32 b16	AVX512FP16/VL
VCVTPH2UDQ	ymmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2UDQ	zmmreg mask z, xmmrm128 b16 er	AVX512FP16
VCVTPH2UQQ	xmmreg mask z, xmmrm32 b16	AVX512FP16/VL
VCVTPH2UQQ	ymmreg mask z, xmmrm64 b16	AVX512FP16/VL
VCVTPH2UQQ	zmmreg mask z, xmmrm128 b16 er	AVX512FP16
VCVTPH2UW	xmmreg mask z, xmmrm128 b16	AVX512FP16/VL
VCVTPH2UW	ymmreg mask z, ymmrm256 b16	AVX512FP16/VL
VCVTPH2UW	zmmreg mask z, zmmrm512 b16 er	AVX512FP16
VCVTPH2W	xmmreg mask z, xmmrm128 b16	AVX512FP16/VL
VCVTPH2W	ymmreg mask z, ymmrm256 b16	AVX512FP16/VL
VCVTPH2W	zmmreg mask z, zmmrm512 b16 er	AVX512FP16
VCVTPS2PH	xmmrm64, xmmreg, imm8	AVX512FC16/VL
VCVTPS2PH	xmmrm128, ymmreg, imm8	AVX512FC16/VL
VCVTPS2PH	xmmreg mask z, xmmreg, imm8	AVX512VL
VCVTPS2PH	mem64 mask, xmmreg, imm8	AVX512VL
VCVTPS2PH	xmmreg mask z, ymmreg, imm8	AVX512VL
VCVTPS2PH	mem128 mask, ymmreg, imm8	AVX512VL
VCVTPS2PH	ymmreg mask z, zmmreg sae, imm8	AVX512
VCVTPS2PH	mem256 mask, zmmreg sae, imm8	AVX512
VCVTPS2PH	xmmreg mask z, xmmrm128 b32	AVX512FP16/VL
VCVTPS2PH	ymmreg mask z, ymmrm256 b32	AVX512FP16/VL

VCVTPS2PH	zmmreg mask z, zmmrm512 b32 er AVX512FP16
VCVTQQ2PH	xmmreg mask z, xmmrm128 b64 AVX512FP16/VL
VCVTQQ2PH	ymmreg mask z, ymmrm256 b64 AVX512FP16/VL
VCVTQQ2PH	zmmreg mask z, zmmrm512 b64 er AVX512FP16/VL
VCVTSD2SH	xmmreg mask z, xmmreg*, xmmrm64 er AVX512FP16
VCVTSH2SD	xmmreg, xmmreg*, xmmrm16 sae AVX512FP16
VCVTSH2SI	reg32, xmmrm16 er AVX512FP16
VCVTSH2SI	reg64, xmmrm16 er AVX512FP16
VCVTSH2SS	xmmreg mask z, xmmreg*, xmmrm16 sae AVX512FP16
VCVTSH2USI	reg32, xmmrm16 er AVX512FP16
VCVTSH2USI	reg64, xmmrm16 er AVX512FP16
VCVTSI2SH	xmmreg, xmmreg*, rm32 er AVX512FP16
VCVTSI2SH	xmmreg, xmmreg*, rm64 er AVX512FP16
VCVTSS2SH	xmmreg, xmmreg*, xmmrm32 er AVX512FP16
VCVTTPH2DQ	xmmreg mask z, xmmrm64 b16 AVX512FP16/VL
VCVTTPH2DQ	ymmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTTPH2DQ	zmmreg mask z, ymmrm256 b16 sae AVX512FP16
VCVTTPH2QQ	xmmreg mask z, xmmrm32 b16 AVX512FP16/VL
VCVTTPH2QQ	ymmreg mask z, xmmrm64 b16 AVX512FP16/VL
VCVTTPH2QQ	zmmreg mask z, xmmrm128 b16 sae AVX512FP16
VCVTTPH2UDQ	xmmreg mask z, xmmrm64 b16 AVX512FP16/VL
VCVTTPH2UDQ	ymmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTTPH2UDQ	zmmreg mask z, ymmrm256 b16 sae AVX512FP16
VCVTTPH2UQQ	xmmreg mask z, xmmrm32 b16 AVX512FP16/VL
VCVTTPH2UQQ	ymmreg mask z, xmmrm64 b16 AVX512FP16/VL
VCVTTPH2UQQ	zmmreg mask z, xmmrm128 b16 sae AVX512FP16
VCVTTPH2UW	xmmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTTPH2UW	ymmreg mask z, ymmrm256 b16 AVX512FP16/VL
VCVTTPH2UW	zmmreg mask z, zmmrm512 b16 sae AVX512FP16
VCVTTPH2W	xmmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTTPH2W	ymmreg mask z, ymmrm256 b16 AVX512FP16/VL
VCVTTPH2W	zmmreg mask z, zmmrm512 b16 sae AVX512FP16
VCVTTS2SI	reg32, xmmrm16 sae AVX512FP16
VCVTTS2SI	reg64, xmmrm16 sae AVX512FP16
VCVTTS2USI	reg32, xmmrm16 sae AVX512FP16
VCVTTS2USI	reg64, xmmrm16 sae AVX512FP16
VCVTUDQ2PH	xmmreg mask z, xmmrm128 b32 AVX512FP16/VL
VCVTUDQ2PH	ymmreg mask z, ymmrm256 b32 AVX512FP16/VL
VCVTUDQ2PH	zmmreg mask z, zmmrm512 b32 AVX512FP16
VCVTUQ2PH	xmmreg mask z, xmmrm128 b32 AVX512FP16/VL
VCVTUQ2PH	ymmreg mask z, ymmrm256 b32 AVX512FP16/VL
VCVTUQ2PH	zmmreg mask z, zmmrm512 b32 AVX512FP16
VCVTUSI2SH	xmmreg, xmmreg er, rm32 er AVX512FP16
VCVTUSI2SS	xmmreg, xmmreg er, rm64 er AVX512FP16
VCVTUW2PH	xmmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTUW2PH	ymmreg mask z, ymmrm256 b16 AVX512FP16/VL
VCVTUW2PH	zmmreg mask z, zmmrm512 b16 er AVX512FP16
VCVTW2PH	xmmreg mask z, xmmrm128 b16 AVX512FP16/VL
VCVTW2PH	ymmreg mask z, ymmrm256 b16 AVX512FP16/VL
VCVTW2PH	zmmreg mask z, zmmrm512 b16 er AVX512FP16
VDIVPH	xmmreg mask z, xmmreg*, xmmrm128 b16 AVX512FP16/VL
VDIVPH	ymmreg mask z, ymmreg*, ymmrm256 b16 AVX512FP16/VL
VDIVPH	zmmreg mask z, zmmreg*, zmmrm512 b16 er AVX512FP16
VDIVSH	xmmreg mask z, xmmreg*, xmmrm16 er AVX512FP16
VFCMADDCPH	xmmreg mask z, xmmreg*, xmmrm128 b32 AVX512FP16/VL
VFCMADDCPH	ymmreg mask z, ymmreg*, ymmrm256 b32 AVX512FP16/VL
VFCMADDCPH	zmmreg mask z, zmmreg*, zmmrm512 b32 er AVX512FP16/VL
VFMADDCPH	xmmreg mask z, xmmreg*, xmmrm128 b32 AVX512FP16/VL
VFMADDCPH	ymmreg mask z, ymmreg*, ymmrm256 b32 AVX512FP16/VL
VFMADDCPH	zmmreg mask z, zmmreg*, zmmrm512 b32 er AVX512FP16/VL
VFCMADDCSH	xmmreg mask z, xmmreg*, xmmrm32 er AVX512FP16
VFMADDCSH	xmmreg mask z, xmmreg*, xmmrm32 er AVX512FP16
VFCMULCPCH	xmmreg mask z, xmmreg*, xmmrm128 b32 AVX512FP16/VL
VFCMULCPCH	ymmreg mask z, ymmreg*, ymmrm256 b32 AVX512FP16/VL
VFCMULCPCH	zmmreg mask z, zmmreg*, zmmrm512 b32 er AVX512FP16/VL
VFMULCPCH	xmmreg mask z, xmmreg*, xmmrm128 b32 AVX512FP16/VL
VFMULCPCH	ymmreg mask z, ymmreg*, ymmrm256 b32 AVX512FP16/VL

[illegible]



AAND	mem64, reg64	RAOINT, LONG
AXOR	mem32, reg32	RAOINT, SD
AXOR	mem64, reg64	RAOINT, LONG

### F.1.69 User interrupts

CLUI		UINTR, LONG
SENDUIPI	reg64	UINTR, LONG
STUI		UINTR, LONG
TESTUI		UINTR, LONG
UIRET		UINTR, LONG

### F.1.70 Compare, exchange and add conditional

CMPccXADD	mem32, reg32, reg32	CMPCCXADD, LONG, SD
CMPccXADD	mem64, reg64, reg64	CMPCCXADD, LONG

### F.1.71 Flexible Return and Exception Delivery

ERETS		FRED, PRIV, LONG
ERETU		FRED, PRIV, LONG
LKGS	mem	LKGS, PRIV, LONG, SW
LKGS	reg16	LKGS, PRIV, LONG
LKGS	reg32	LKGS, PRIV, LONG, ND
LKGS	reg64	LKGS, PRIV, LONG, ND, OPT
LKGS	reg64	LKGS, PRIV, LONG

### F.1.72 WRMSRNS and MSRLIST instructions

WRMSRNS		WRMSRNS, PRIV, LONG
RDMSRLIST		MSRLIST, PRIV, LONG
WRMSRLIST		MSRLIST, PRIV, LONG

### F.1.73 History reset

HRESET	imm, reg_eax	HRESET, PRIV
HRESET	imm	HRESET, PRIV, ND

### F.1.74 Systematic names for the hinting nop instructions

HINT_NOP0	rm16	P6, UNDOC
HINT_NOP0	rm32	P6, UNDOC
HINT_NOP0	rm64	X86_64, LONG, UNDOC
HINT_NOP1	rm16	P6, UNDOC
HINT_NOP1	rm32	P6, UNDOC
HINT_NOP1	rm64	X86_64, LONG, UNDOC
HINT_NOP2	rm16	P6, UNDOC
HINT_NOP2	rm32	P6, UNDOC
HINT_NOP2	rm64	X86_64, LONG, UNDOC
HINT_NOP3	rm16	P6, UNDOC
HINT_NOP3	rm32	P6, UNDOC
HINT_NOP3	rm64	X86_64, LONG, UNDOC
HINT_NOP4	rm16	P6, UNDOC
HINT_NOP4	rm32	P6, UNDOC
HINT_NOP4	rm64	X86_64, LONG, UNDOC
HINT_NOP5	rm16	P6, UNDOC
HINT_NOP5	rm32	P6, UNDOC
HINT_NOP5	rm64	X86_64, LONG, UNDOC
HINT_NOP6	rm16	P6, UNDOC
HINT_NOP6	rm32	P6, UNDOC
HINT_NOP6	rm64	X86_64, LONG, UNDOC
HINT_NOP7	rm16	P6, UNDOC
HINT_NOP7	rm32	P6, UNDOC
HINT_NOP7	rm64	X86_64, LONG, UNDOC
HINT_NOP8	rm16	P6, UNDOC
HINT_NOP8	rm32	P6, UNDOC
HINT_NOP8	rm64	X86_64, LONG, UNDOC
HINT_NOP9	rm16	P6, UNDOC

HINT_NOP9	rm32	P6, UNDOC
HINT_NOP9	rm64	X86_64, LONG, UNDOC
HINT_NOP10	rm16	P6, UNDOC
HINT_NOP10	rm32	P6, UNDOC
HINT_NOP10	rm64	X86_64, LONG, UNDOC
HINT_NOP11	rm16	P6, UNDOC
HINT_NOP11	rm32	P6, UNDOC
HINT_NOP11	rm64	X86_64, LONG, UNDOC
HINT_NOP12	rm16	P6, UNDOC
HINT_NOP12	rm32	P6, UNDOC
HINT_NOP12	rm64	X86_64, LONG, UNDOC
HINT_NOP13	rm16	P6, UNDOC
HINT_NOP13	rm32	P6, UNDOC
HINT_NOP13	rm64	X86_64, LONG, UNDOC
HINT_NOP14	rm16	P6, UNDOC
HINT_NOP14	rm32	P6, UNDOC
HINT_NOP14	rm64	X86_64, LONG, UNDOC
HINT_NOP15	rm16	P6, UNDOC
HINT_NOP15	rm32	P6, UNDOC
HINT_NOP15	rm64	X86_64, LONG, UNDOC
HINT_NOP16	rm16	P6, UNDOC
HINT_NOP16	rm32	P6, UNDOC
HINT_NOP16	rm64	X86_64, LONG, UNDOC
HINT_NOP17	rm16	P6, UNDOC
HINT_NOP17	rm32	P6, UNDOC
HINT_NOP17	rm64	X86_64, LONG, UNDOC
HINT_NOP18	rm16	P6, UNDOC
HINT_NOP18	rm32	P6, UNDOC
HINT_NOP18	rm64	X86_64, LONG, UNDOC
HINT_NOP19	rm16	P6, UNDOC
HINT_NOP19	rm32	P6, UNDOC
HINT_NOP19	rm64	X86_64, LONG, UNDOC
HINT_NOP20	rm16	P6, UNDOC
HINT_NOP20	rm32	P6, UNDOC
HINT_NOP20	rm64	X86_64, LONG, UNDOC
HINT_NOP21	rm16	P6, UNDOC
HINT_NOP21	rm32	P6, UNDOC
HINT_NOP21	rm64	X86_64, LONG, UNDOC
HINT_NOP22	rm16	P6, UNDOC
HINT_NOP22	rm32	P6, UNDOC
HINT_NOP22	rm64	X86_64, LONG, UNDOC
HINT_NOP23	rm16	P6, UNDOC
HINT_NOP23	rm32	P6, UNDOC
HINT_NOP23	rm64	X86_64, LONG, UNDOC
HINT_NOP24	rm16	P6, UNDOC
HINT_NOP24	rm32	P6, UNDOC
HINT_NOP24	rm64	X86_64, LONG, UNDOC
HINT_NOP25	rm16	P6, UNDOC
HINT_NOP25	rm32	P6, UNDOC
HINT_NOP25	rm64	X86_64, LONG, UNDOC
HINT_NOP26	rm16	P6, UNDOC
HINT_NOP26	rm32	P6, UNDOC
HINT_NOP26	rm64	X86_64, LONG, UNDOC
HINT_NOP27	rm16	P6, UNDOC
HINT_NOP27	rm32	P6, UNDOC
HINT_NOP27	rm64	X86_64, LONG, UNDOC
HINT_NOP28	rm16	P6, UNDOC
HINT_NOP28	rm32	P6, UNDOC
HINT_NOP28	rm64	X86_64, LONG, UNDOC
HINT_NOP29	rm16	P6, UNDOC
HINT_NOP29	rm32	P6, UNDOC
HINT_NOP29	rm64	X86_64, LONG, UNDOC
HINT_NOP30	rm16	P6, UNDOC
HINT_NOP30	rm32	P6, UNDOC
HINT_NOP30	rm64	X86_64, LONG, UNDOC
HINT_NOP31	rm16	P6, UNDOC
HINT_NOP31	rm32	P6, UNDOC

HINT_NOP31	rm64	X86_64, LONG, UNDOC
HINT_NOP32	rm16	P6, UNDOC
HINT_NOP32	rm32	P6, UNDOC
HINT_NOP32	rm64	X86_64, LONG, UNDOC
HINT_NOP33	rm16	P6, UNDOC
HINT_NOP33	rm32	P6, UNDOC
HINT_NOP33	rm64	X86_64, LONG, UNDOC
HINT_NOP34	rm16	P6, UNDOC
HINT_NOP34	rm32	P6, UNDOC
HINT_NOP34	rm64	X86_64, LONG, UNDOC
HINT_NOP35	rm16	P6, UNDOC
HINT_NOP35	rm32	P6, UNDOC
HINT_NOP35	rm64	X86_64, LONG, UNDOC
HINT_NOP36	rm16	P6, UNDOC
HINT_NOP36	rm32	P6, UNDOC
HINT_NOP36	rm64	X86_64, LONG, UNDOC
HINT_NOP37	rm16	P6, UNDOC
HINT_NOP37	rm32	P6, UNDOC
HINT_NOP37	rm64	X86_64, LONG, UNDOC
HINT_NOP38	rm16	P6, UNDOC
HINT_NOP38	rm32	P6, UNDOC
HINT_NOP38	rm64	X86_64, LONG, UNDOC
HINT_NOP39	rm16	P6, UNDOC
HINT_NOP39	rm32	P6, UNDOC
HINT_NOP39	rm64	X86_64, LONG, UNDOC
HINT_NOP40	rm16	P6, UNDOC
HINT_NOP40	rm32	P6, UNDOC
HINT_NOP40	rm64	X86_64, LONG, UNDOC
HINT_NOP41	rm16	P6, UNDOC
HINT_NOP41	rm32	P6, UNDOC
HINT_NOP41	rm64	X86_64, LONG, UNDOC
HINT_NOP42	rm16	P6, UNDOC
HINT_NOP42	rm32	P6, UNDOC
HINT_NOP42	rm64	X86_64, LONG, UNDOC
HINT_NOP43	rm16	P6, UNDOC
HINT_NOP43	rm32	P6, UNDOC
HINT_NOP43	rm64	X86_64, LONG, UNDOC
HINT_NOP44	rm16	P6, UNDOC
HINT_NOP44	rm32	P6, UNDOC
HINT_NOP44	rm64	X86_64, LONG, UNDOC
HINT_NOP45	rm16	P6, UNDOC
HINT_NOP45	rm32	P6, UNDOC
HINT_NOP45	rm64	X86_64, LONG, UNDOC
HINT_NOP46	rm16	P6, UNDOC
HINT_NOP46	rm32	P6, UNDOC
HINT_NOP46	rm64	X86_64, LONG, UNDOC
HINT_NOP47	rm16	P6, UNDOC
HINT_NOP47	rm32	P6, UNDOC
HINT_NOP47	rm64	X86_64, LONG, UNDOC
HINT_NOP48	rm16	P6, UNDOC
HINT_NOP48	rm32	P6, UNDOC
HINT_NOP48	rm64	X86_64, LONG, UNDOC
HINT_NOP49	rm16	P6, UNDOC
HINT_NOP49	rm32	P6, UNDOC
HINT_NOP49	rm64	X86_64, LONG, UNDOC
HINT_NOP50	rm16	P6, UNDOC
HINT_NOP50	rm32	P6, UNDOC
HINT_NOP50	rm64	X86_64, LONG, UNDOC
HINT_NOP51	rm16	P6, UNDOC
HINT_NOP51	rm32	P6, UNDOC
HINT_NOP51	rm64	X86_64, LONG, UNDOC
HINT_NOP52	rm16	P6, UNDOC
HINT_NOP52	rm32	P6, UNDOC
HINT_NOP52	rm64	X86_64, LONG, UNDOC
HINT_NOP53	rm16	P6, UNDOC
HINT_NOP53	rm32	P6, UNDOC
HINT_NOP53	rm64	X86_64, LONG, UNDOC



HINT_NOP54	rm16	P6, UNDOC
HINT_NOP54	rm32	P6, UNDOC
HINT_NOP54	rm64	X86_64, LONG, UNDOC
HINT_NOP55	rm16	P6, UNDOC
HINT_NOP55	rm32	P6, UNDOC
HINT_NOP55	rm64	X86_64, LONG, UNDOC
HINT_NOP56	rm16	P6, UNDOC
HINT_NOP56	rm32	P6, UNDOC
HINT_NOP56	rm64	X86_64, LONG, UNDOC
HINT_NOP57	rm16	P6, UNDOC
HINT_NOP57	rm32	P6, UNDOC
HINT_NOP57	rm64	X86_64, LONG, UNDOC
HINT_NOP58	rm16	P6, UNDOC
HINT_NOP58	rm32	P6, UNDOC
HINT_NOP58	rm64	X86_64, LONG, UNDOC
HINT_NOP59	rm16	P6, UNDOC
HINT_NOP59	rm32	P6, UNDOC
HINT_NOP59	rm64	X86_64, LONG, UNDOC
HINT_NOP60	rm16	P6, UNDOC
HINT_NOP60	rm32	P6, UNDOC
HINT_NOP60	rm64	X86_64, LONG, UNDOC
HINT_NOP61	rm16	P6, UNDOC
HINT_NOP61	rm32	P6, UNDOC
HINT_NOP61	rm64	X86_64, LONG, UNDOC
HINT_NOP62	rm16	P6, UNDOC
HINT_NOP62	rm32	P6, UNDOC
HINT_NOP62	rm64	X86_64, LONG, UNDOC
HINT_NOP63	rm16	P6, UNDOC
HINT_NOP63	rm32	P6, UNDOC
HINT_NOP63	rm64	X86_64, LONG, UNDOC



# Index

---

! operator	41	%elifnmacro	64
!= operator	40	%elifnum	66
\$		%elifnstr	66
current address	39	%elifntoken	66
prefix	31, 36	%elifnum	66
\$\$ token	39, 110	%elifstr	66
%,	52	%eliftoken	66
%[	49	%else	64
%[]	46	%endrep	67
% operator	40	%error	73
% prefix to DB lists	32	%eval()	51, 54
%!	46, 75	%exitrep	67
\$\$ and \$\$\$ prefixes	69	%fatal	73
%% operator	40, 58	%hex()	54
%*?	50	%iassign	51
%*??	50	%idefalias	52
%+	46, 49	%idefine	46
%+1 and %-1 syntax	62	%idefstr	51
%?	50	%ideftok	51
%%?	50	%if	45, 54, 63, 65
%00	61	%ifctx	64, 70
%0 parameter count	60	%ifdef	64
%abs()	53	%ifempty	66
%arg	72	%ifenv	66
%assign	51, 54	%ifid	65
%clear	75, 77	%ifidn	65
%cond()	54	%ifidni	65
%count()	54	%ifmacro	64
%defalias	52	%ifn	63, 65
%define	25, 46	%ifnctx	64
%defstr	51, 56	%ifndef	64
%deftok	51, 56	%ifnempty	66
%depend	68	%ifnenv	66
%elif	64, 65	%ifnid	66
%elifctx	64	%ifnidn	65
%elifdef	64	%ifnidni	65
%elifempty	66	%ifnmacro	64
%elifenv	66	%ifnum	66
%elifid	66	%ifnstr	66
%elifidn	65	%ifntoken	66
%elifidni	65	%ifnum	65
%elifmacro	64	%ifstr	65
%elifn	63, 65	%iftoken	66
%elifnctx	64	%imacro	45, 57
%elifndef	64	%include	25, 67
%elifnempty	66	%is()	54
%elifnenv	66	%ixdefine	48
%elifnid	66	%line	28, 45, 46, 75
%elifnidn	65	%local	73
%elifnidni	65	%macro	45, 57

%map()	54	-I option	25
%num()	55	-i option	25, 154
%pathsearch	25, 68	-k	154
%pop	68, 69	-L option	22
%pragma	74	-l option	22
%push	68, 69	-M option	23
%rep	34, 45, 66	-MD option	23
%repl	70	-MF option	23
%rotate	61	-MG option	23
%sel()	54, 56	-MP option	23
%stacksize	72	-MQ option	23
%str()	51, 56	-MT option	23
%strcat	52, 56	-MW option	23
%strcat()	56	-O option	26
%strlen	53, 56	-o option	21, 153
%strlen()	56	-P option	25
%substr	53, 56	-p option	25, 68
%substr()	56	-r	153
%tok()	52, 56	-s option	24, 154
%undef	25, 50	-soname	133
%unimacro	63	-t	26
%unmacro	63	-U option	25
%use	68, 83	-u option	25, 153
%warning	73	-v option	27
%xdefine	48	-w option	27
& operator	40	-w option	27
&& operator	39	-Werror option	27
* operator	40	-Wno-error option	27
+ modifier	59	-X option	24
+ operator		-Z option	24
binary	40	..@ symbol prefix	43, 58
unary	40	..got	110
- operator		..gotoff	110
binary	40	..gotpc	110
unary	40	..gottpoff	111
--before	28	..plt	110
--(g l)postfix	27	..start	100, 115
--(g l)prefix	27	..sym	110
--keep-all	28	..tlsie	110
--limit-X	28	.bss	109, 112
--no-line	28, 75	.COM	95, 117
--pragma	27	.comment	109
--reproducible	28	.data	109, 112
--v	27	.drectve	101
-a option	26, 154	.EXE	97, 115
-b	153	.lbss	109
-D option	25	.ldata	109
-d option	25	.lrodata	109
-E option	26	.nolist	63
-e option	26, 154	.obj	115
-F option	24	.rodata	109
-f option	22, 95	.SYS	95, 118
-g option	24	.tbss	109
-h	153	.tdata	109

.text	109, 112	__?SNaN?__	38
/ operator	40	__?TIME?__	78
// operator	40	__?TIME_NUM?__	78
< operator	40	__?USE_*?__	79
<< operator	40	__?UTC_DATE?__	78
<<< operator	40	__?UTC_DATE_NUM?__	78
<= operator	40	__?UTC_TIME?__	78
<=> operator	40	__?UTC_TIME_NUM?__	78
<=> operator	40	__?utf16?__	37
<> operator	40	__?utf16be?__	37
= operator	40	__?utf16le?__	37
== operator	40	__?utf32?__	37
> operator	40	__?utf32be?__	37
>= operator	40	__?utf32le?__	37
>> operator	40	__NASMDEFSEG	97
>>> operator	40	__DATA	119
?		__GLOBAL_OFFSET_TABLE__	110
data syntax	32, 33	__TEXT	119
operator	39	'nowait'	30
\c{section	95	operator	40
^ operator	40	operator	39
^^ operator	39	~ operator	41
__?ALIGNMODE?__	83	1's complement	41
__?bfloat16?__	37	2's complement	40
__?BITS?__	78	16-bit mode, versus 32-bit mode	87
__?DATE?__	78	64-bit displacement	138
__?DATE_NUM?__	78	64-bit immediate	137
__?DEBUG_FORMAT?__	78	a.out	
__?FILE?__	77	BSD version	112
__?FLOAT?__	93	Linux version	112
__?FLOAT_DAZ?__	93	A16	31, 136
__?FLOAT_ROUND?__	93	A32	31, 136
__?float8?__	37	A64	31, 136
__?float16?__	37	a86	29, 30
__?float32?__	37	ABS	35, 88
__?float64?__	38	ABSOLUTE	89, 98
__?float80e?__	38	addition	40
__?float80m?__	38	address-size prefixes	31
__?float128h?__	38	addressing, mixed-size	135
__?float128l?__	38	algebra	34
__?Infinity?__	38	ALIGN	81, 83, 95, 98
__?LINE?__	77	smart	83
__?NaN?__	38	align, elf attribute	109
__?NASM_MAJOR?__	77	ALIGNB	81
__?NASM_MINOR?__	77	alignment	81
__?NASM_PATCHLEVEL?__	77	in bin sections	96
__?NASM_SNAPSHOT?__	77	in ELF sections	109
__?NASM_SUBMINOR?__	77	in obj sections	98
__?NASM_VER?__	77	in win32 sections	101
__?NASM_VERSION_ID?__	77	of ELF common variables	111
__?OUTPUT_FORMAT?__	78	ALIGNMODE	83
__?PASS?__	79	ALINK	115
__?QNaN?__	38	alink.sourceforge.net	115
__?SECT?__	88, 89	alloc	109

alternate register names	83	coff	107
altreg	83	colon	31
ambiguity	30	comdat section, in win32	102
aout	112	comdat symbol, in win32	102
aoutb	112, 130	comdat, win32 attribute	102
arg	122, 129	comma	60
as86	112	command-line	21, 95
assembler directives	87	commas in macro parameters	59
assembly-time options	25	comment	31, 45
ASSUME	30	ending in \	45
AT	80	removal	45
auto-sync	154	syntax	45
baddb	33	COMMON	91, 98
bf16	84	ELF extensions to	111
bfloat16	38	obj extensions to	100
bin output format	22, 95	Common Object File Format	107
bin, multisection	96	common variables	91
binary	36	alignment in ELF	111
binary files	33	element size	100
bit shift	40	comp.os.msdos.programmer	118
BITS	87, 95	comparison operators	39
bitwise AND	40	concatenating macro parameters	62
bitwise OR	40	concatenating strings	52
bitwise XOR	40	concatenation	46
block IFs	70	condition codes as macro parameters	62
BND	88	conditional assembly	45, 63
boolean		conditional comma operator	52
AND	39	conditional jumps	141
OR	39	conditional-return macro	62
XOR	39	constants	35
boot loader	95	context fall-through lookup	69
boot sector	141	context stack	68, 70
Borland		context-local labels	46, 69
Pascal	123	context-local single-line macros	69
Win32 compilers	97	continuation line	45
braces		continuation lines	45
after % sign	62	counting macro parameters	60
around macro parameters	57	CPU	92
BSD	130	CPUID	37
bug tracker	199	creating contexts	69
bugs	199	critical expression	33, 42, 51, 54, 89, 144
BYTE	141	cv8	103
C calling convention	119, 127	daily development snapshots	199
C symbol names	118	data	111
c16.mac	122, 124	data structure	121, 129
c32.mac	129	DB	32, 37
CALL FAR	41	dbg	112
case sensitivity	29, 47, 48, 51, 57, 65, 99	DD	32, 37
changing sections	88	debug information	24
character constant	32, 37	debug information format	24
character strings	36	decimal	36
circular references	47	declaring structures	79
CLASS	98	DEFAULT	88
CodeView debugging format	103	default	111

default macro parameters	60	extension	21, 95
default name	95	EXTERN	90
default-WRT mechanism	100	elf extensions to	111
defining sections	88	obj extensions to	100
design goals	29	extracting substrings	53
detokenization	46	far call	30
DevPac	42	far common variables	100
directives	46	far pointer	41
disabling listing expansion	63	FARCODE	122, 124
division	40	fini_array	109
signed	40	FLAT	98
unsigned	40	flat memory model	127
DJGPP	107, 127	flat-form binary	95
djlink	115	FLOAT	93
DLL symbols		floating-point	
exporting	99	constants	37, 93
importing	99	packed BCD constants	39
DO	32, 37	floating-point	30, 31, 32, 37
DOS	24	follows=	96
DQ	32, 37	format-specific directives	87
DT	32, 37	fp	84
DUP	32, 33, 34	frame pointer	120, 123, 127
DW	32, 37	FreeBSD	112, 130
DWORD	32	FreeLink	115
DY	32, 37	ftp.simtel.net	115
DZ	32	function	111
effective addresses	31, 34	functions	
element size, in common variables	100	C calling convention	119, 127
ELF	108	PASCAL calling convention	123
16-bit code	112	(G L)POSTFIX	91
debug formats	112	(G L)PREFIX	91
shared libraries	111	git	197
elf32	108	GLOBAL	90
elf64	108	aoutb extensions to	111
elfx32	108	ELF extensions to	111
endproc	122, 129	global offset table	130
ENDSTRUC	79, 89	rdf extensions to	
environment	28	GOT	110, 130
EQU	32, 34	GOT relocations	131
error messages	24	GOTOFF relocations	131
error reporting format	24	GOTPC relocations	130
escape sequences	36	graphics	33
EVEN	81	greedy macro parameters	58
exact matches	63	GROUP	98
EXE_begin	116	groups	41
EXE_end	116	here token	39
EXE_stack	116	hexadecimal	36
EXE2BIN	117	hidden	111
exebin.mac	116	hybrid syntaxes	29
exec	109	IEND	80
Executable and Linkable Format	108	ifunc	84
EXPORT	99	ilog2()	84
exporting symbols	90	ilog2c()	84
expressions	26, 39	ilog2cw()	84

<code>ilog2e()</code>	84	macro-local labels	46, 58
<code>ilog2f()</code>	84	macros	34
<code>ilog2fw()</code>	84	makefile dependencies	23
<code>ilog2w()</code>	84	map files	96
<code>IMPORT</code>	99	MASM	29, 32, 34, 84, 97
import library	99	masm compatibility	84
importing symbols	90	DB syntax	32, 33
unconditionally	90	memory models	30, 119
<code>INCBIN</code>	32, 33, 37	memory operand	32
include search path	25	memory references	29, 34
including other files	67	merge	109
inefficient code	141	Microsoft OMF	97
infinite loop	39	minifloat	38
infinity	38	Minix	112
informational section	101	misc subdirectory	116, 122, 129
<code>init_array</code>	109	mixed-language program	118
inline expansions	46	mixed-size addressing	135
instances of structures	80	mixed-size instruction	135
instruction list	201	modulo	40
integer functions	84	signed	40
integer logarithms	84	unsigned	40
intel hex	96	motorola s-records	96
Intel number formats	38	MS-DOS	95
internal	111	MS-DOS device drivers	118
<code>ISTRUC</code>	80	multi-line macro	45, 46
iterating over macro parameters	61	multi-line macro expansion	46
<code>ith</code>	96	multi-line macros	46, 57, 147
<code>Jcc NEAR</code>	141	multipass optimization	26
<code>JMP DWORD</code>	135	multiple section names	95
jumps, mixed-size	135	multiplication	40
label preceding macro	61	multiplier	109
label prefix	43	multi-push macro	61
label-orphan	31	multisection	96
last	60	NaN	38
<code>ld86</code>	112	<code>nasm -h</code>	22
license	19	NASM version	
linker, free	115	history	157
Linux		ID macro	77
a.out	112	macros	77
as86	112	string macro	77
ELF	108	<code>nasm-devel</code>	199
list of warning classes	143	<code>nasm.out</code>	21
listing file	22	<code>NASMENV</code>	28
little-endian	37	<code>ndisasm</code>	153
local labels	42	near call	30
Mach, object file format	107	near common variables	100
Mach-O, object file format	107	negation	
<code>macho32</code>	107	arithmetic	40
<code>macho64</code>	107	bitwise	41
MacOS X	107	bitwise	41
macro indirection	46, 49	boolean	41
macro library	25	boolean	41
macro parameters range	59	NetBSD	112, 130
macro processor	45	new releases	199



no_dead_strip	108	preinit_array	109
noalloc	109	preprocess-only mode	26
nobits	96, 109	preprocessor	26, 34, 40, 45
NOBND	88	preprocessor directives	45, 46
noexec	109	preprocessor functions	46, 52, 53
note	109	comment removal	45
nowrite	109	conditionals	45
NSIS	197	continuation line	45
Nullsoft Scriptable Installer	197	expansions	45
numeric constants	32, 36	expressions	26
016	31, 136	loops	45, 66
032	31, 136	variables	51
064	31	primitive directives	87
obj	97	PRIVATE	98
object	111	private_extern	108
octal	36	proc	122, 129
OF_DEFAULT	22	procedure linkage table	110, 131, 132
OFFSET	29	processor mode	87
OMF	97	progbits	96, 109
omitted parameters	60	program entry point	100, 115
OpenBSD	112, 130	program origin	95
operand-size prefixes	31	protected	111
operands	31	pseudo-instructions	32
operating system	95	PUBLIC	90, 98
writing	135	pure binary	95
operators	39	quick start	29
unary	40	QWORD	32
ORG	95, 117, 141	redirecting errors	24
OS/2	97, 98	REL	35, 88
osabi	109	release candidates	199
other preprocessor directives	75	relocations, PIC-specific	110
out of range, jumps	141	removing contexts	69
output file format	22	renaming contexts	70
output formats	95	repeating	34, 66
overlapping segments	41	reporting bugs	199
OVERLAY	98	REQUIRED	90
overloading		RESB	32, 33
multi-line macros	57	RESD	32, 33
single-line macros	47	RES0	32, 33
paradox	42	RESQ	32, 33
PASCAL	124	REST	32, 33
Pascal calling convention	123	RESW	32, 33
period	42	RESY	32, 33
PharLap	98	RESZ	32, 33
PIC	110, 112, 130	{rex}	159
PLT	110, 131, 132	rotating macro parameters	61
PLT relocations	110, 131, 132	searching for include files	67
plt relocations	132	SECTALIGN	82
pointer, elf attribute	109	SECTION	88
position-independent code	110, 112, 130	section alignment	
pre-defining macros	25, 48	in bin	96
pre-including files	25	in ELF	109
precedence	39	in obj	98
preferred	41	in win32	101

ELF extensions to	109	symbols	
macho extensions to	107	exporting from DLLs	99
Windows extensions to	101	importing from DLLs	99
SEG	41, 97	specifying sizes	111
SEGMENT	88	specifying types	111
segment address	41	synchronization	154
segment alignment		tasm	26, 29, 97
in bin	96	TBYTE	30
in obj	98	test subdirectory	115
segment names, Borland Pascal	124	testing	
obj extensions to	97	arbitrary numeric expressions	65
segment override	30, 31	context stack	64
segments	41	exact text identity	65
groups of	98	multi-line macro existence	64
separator character	29	single-line macro existence	64
shared libraries	112, 130	token types	65
shift command	61	thread local storage	
signed		in ELF	110
bit shift	40	in macho	108
division	40	TIMES	32, 34, 42, 141, 142
modulo	40	TLINK	117
single-line macros	46, 147	tls	108, 109, 110
size, of symbols	111	trailing colon	31
smartalign	83	TWORD	30, 32
snapshots, daily development	199	type, of symbols	111
Solaris x86	108	unary operators	40
sound	33	undefining macros	25
source-listing file	22	underscore, in C symbols	118
square brackets	29, 34	Unicode	37
srec	96	uninitialized	32, 33
STACK	98	uninitialized storage	30, 32
stack relative preprocessor directives	72	Unix	
standard macro packages	83	SCO	108
standard macros	77	System V	108
standardized section names	88, 101, 109, 112	UnixWare	108
start=	96	unrolled loops	34
STATIC	91	unsigned	
stderr	24	bit shift	40
stdout	24	division	40
STRICT	41	modulo	40
string	36	UPPERCASE	29, 99
constants	32, 36, 37	USE16	88, 98
length	53	USE32	88, 98
manipulation in macros	52	user-defined errors	73
strings, elf attribute	109	user-level directives	77, 87
strong	111	UTF-8	37
STRUC	79, 89, 121, 129	UTF-16	37
structure data types	79	UTF-32	37
stub preprocessor	26	VAL	115
subsections_via_symbols	108	valid characters	31
subtraction	40	variable types	30
suppressing preprocessing	26	variables, preprocessor	51
switching between sections	88	version	27
		version number of nasm	77

vfollows=	96	pp-macro-params	147
Visual C++	101	pp-macro-params-legacy	147
vstart=	96	pp-macro-params-multi	147
[WARNING]	27, 93	pp-macro-params-single	147
warning class	143	pp-macro-redef-multi	147
all	143	pp-open	148
bad-pragma	143	pp-open-braces	148
bnd	143	pp-open-brackets	148
db-empty	143	pp-open-string	148
ea	143	pp-rep-negative	148
ea-absolute	143	pp-sel-range	148
ea-dispsize	143	pp-trailing	148
environment	143	pragma	148
float	143	pragma-bad	148
float-denorm	143	pragma-empty	148
float-overflow	143	pragma-na	148
float-toolong	143	pragma-unknown	149
float-underflow	144	prefix	149
forward	144	prefix-bnd	149
hle	144	prefix-hle	149
label	144	prefix-lock	149
label-orphan	144	prefix-lock-error	149
label-redef	144	prefix-lock-xchg	149
label-redef-late	144	prefix-opsiz	149
lock	144	prefix-seg	149
macro-def-case-single	144	ptr	149
macro-def-greedy-single	144	regsize	150
macro-def-param-single	144	reloc	150
macro-defaults	144	reloc-abs	150
macro-params-legacy	145	reloc-abs-byte	150
macro-params-multi	145	reloc-abs-dword	150
macro-params-single	145	reloc-abs-qword	150
negative-rep	145	reloc-abs-word	150
not-my-pragma	145	reloc-rel	150
number-overflow	145	reloc-rel-byte	150
obsolete	145	reloc-rel-dword	151
obsolete-nop	145	reloc-rel-qword	151
obsolete-removed	145	reloc-rel-word	151
obsolete-valid	145	unknown-pragma	151
orphan-labels	145	unknown-warning	151
other	145	user	151
phase	145	warn-stack-empty	151
pp	146	zeroing	151
pp-else	146	zext-reloc	151
pp-else-elif	146	warning classes, list	143
pp-else-else	146	warnings	27
pp-empty-braces	146	weak	111
pp-environment	146	website	199
pp-macro	146	Win32	97, 101, 127
pp-macro-def	146	Win64	103, 137
pp-macro-def-case-single	146	Windows	115
pp-macro-def-greedy-single	147	debugging formats	103
pp-macro-def-param-single	147	write	109
pp-macro-defaults	147	writing operating systems	135

WRT	41, 97, 108, 110, 112
WRT ..got	131
WRT ..gotoff	131
WRT ..gotpc	130
WRT ..plt	132
WRT ..sym	132
www.delorie.com	115
www.pcorner.com	115
x2ftp.oulu.fi	115
x32	108