x86asm.net

X86 Opcode and Instruction Reference

MazeGen, 2017-02-18 Revision: 1.12

This reference is intended to be precise opcode and instruction set reference (including x86-64). Its principal aim is exact definition of instruction parameters and attributes.

Quick Navigation

coder32, coder32-abc, geek32, geek32-abc

coder64, coder64-abc, geek64, geek64-abc

coder, coder-abc, geek, geek-abc (these contain both x86-32 and x64 instructions).

In contrast to other references, primary source of this reference is an XML document, which guarantees clear and structured information base and therefore ability to extract many various informations such as a list of instructions from requested groups, etc.

The reference is primarily based on Intel manuals as Intel is the originator of x86 architecture. Additionally, it describes undocumented instructions as well. On appropriate places, it gives a notice if an opcode act differently on AMD architecture. Support for Cyrix, NexGen etc. specific instructions is not scheduled at all.

HTML Editions

These editions are available at the moment: The *coder* suite is intended to more common use and contains the following editions: coder32, coder64, and coder (sorted by opcode), and coder32-abc, coder64-abc, and coder-abc (sorted by mnemonic). The *geek* suite is intended for deeper research of x86 architectures' instruction set. This includes geek32, geek64, and geek editions (by opcode) and geek32-abc, geek64-abc, and geek-abc editions (by mnemonic). More on the purpose and use of this suite see close below.

Don't get confused by geek(-abc) and coder(-abc) editions. Both of them contains instruction set of both x86-32 and x86-64 architectures. If you don't have a particular reason to use them (such as to view the differencies between the architectures), the other editions would probably suit you better.

Editions coder32 a geek32 relate exclusively to x86-32 architecture. Similarly, editions coder64 and geek64 relate exclusively to x86-64 architecture.

The following chart illustrates the differencies between editions for current release:

Edition		coder	coder32	coder64	geek	geek32	geek64
Supported Architectures		both	pure x86-32	pure x86-64	both	pure x86-32	pure x86-64
Operand Codes		traditional	traditional	traditional	special	special	special
Abandoned Instructions		no	no	no	yes	yes	yes
Opcode Bitfields Information		no	no	no	yes	yes	yes
Instruction Extension Indicated		yes	yes	yes	yes	yes	yes
Instruction Group Indicated		no	no	no	yes	yes	yes
	general	yes	yes	yes	yes	yes	yes
	system	yes	yes	yes	yes	yes	yes
	x87 FPU	yes	yes	yes	yes	yes	yes
Present Instructions	**MMX**	yes	yes	yes	yes	yes	yes
	Intel SSE (all)	yes	yes	yes	yes	yes	yes
	VMX	yes	yes	yes	yes	yes	yes
	SMX	yes	yes	yes	yes	yes	yes

Itanium	no	no	no	yes	yes	yes

The Purpose of Geek Editions in Short

The geek editions contains as much complete information from the source XML document as possible. That's why they may seem quite unclear. You appreciate them only if you need to get to know the instruction set deeply or if you investigate the source XML and you need to visualize it better.

These editions use specific operand codes (which are described in Instruction Operand Codes chapter below). These codes may look strange and obscure at the first sight. The reason to use them is that they hold more information than the more common ones. One example can be operand combination `rAX, imm16/32`, such as in instruction `ADD rAX, imm16/32` in coder64 edition. One can determine that the destination operand is either `ax`, `eax`, or `rax`, and the source one is either `imm16` or `imm32`. A problem arises when one needs to determine what magic is behind `rax, imm32` combination. If one is just getting started with x64 architecture, it is not clear how exactly is 32-bit immediate added to 64-bit `rax`. This question is answered by corresponding geek edition, `ADD rAX, Ivds` in geek64 edition. The immediate value is encoded there using `Ivds` code. `I` code means *Immediate*, `v` means *word* or *doubleword* (`imm16` or `imm32`). The most important part is `ds` code, which means *doubleword, sign-extended to 64 bits for 64-bit operand size*. Now is it clear.

As for Itanium-specific instructions, they are added just for the sake of interest - they give a notice that the appropriate opcodes are already used.

Hypertext Reference to Particular Opcode

If you want to refer to particular opcode (in any edition), e. g., `0FA0 PUSH FS`, it can be easily achieved this way:

`ref.x86asm.net/geek.html#x0FA0` (try it)

It works for opcode extension similarly, e. g., `83 /7 CMP`:

`ref.x86asm.net/coder32.html#x83_7` (try it)

Using HTML Editions

Since HTML editions can look complicated at first sight, here goes an outline how to work with them. Following examples come from coder32's edition because it is easier to use than geek's editions.

Example: ADC Instruction

Let's start with more known instruction, such as `ADC`. We find something similar to the following:

```
|pf|0F|po|so|flds|o|proc|st|m|rl|l|mnemonic|op1     |op2   |op3|op4|iext|grp1|grp2 |tested f|modif f |def f   |undef f|f values|description, notes|
|  |  |11|  |    |r|    |  | |  |L|ADC     |r/m16/32|r16/32|   |   |    |gen |arith|.......c|o..szapc|o..szapc|       |        |Add with Carry    |
```

First column pf (Prefix) is empty. That means the instruction's opcode doesn't contain any fixed prefix.

Next column 0F is just allocated for `0F` prefix for multiple-byte opcodes so it is empty.

Next column po (Primary Opcode) holds primary opcode value itself.

Because the instruction's opcode doesn't contain any added byte, the column so (Secondary Opcode) is empty too.

The opcode doesn't contain any specific bits so the column flds (Opcode Fields) is empty.

The column o (Register/Opcode Field) here holds "r", which indicates that the instruction uses "full" ModR/M byte (no opcode extension).

Because this instruction is supported since 8086 processor, proc column (Introduced with Processor) is empty.

This instruction is officially documented so st column is empty too.

Instruction `ADC` can work on any ring level so the column rl, Ring Level, is empty.

The column x holds "L", which means that `LOCK` prefix is allowed with this instruction.

Next three columns, mnemonic, op1 and op2 show instruction's syntax. The destination operand of this instruction is set up using bold, what always means the operand is modified by the instruction.

The column iext (Instruction Extension Group) is empty because the instruction doesn't belong to any instruction set extension.

Columns grp1 and grp2 classify the instruction among general arithmetic instructions.

`ADC` instruction is influenced by CF flag, what represents tested f column.

This instruction influences (overwrites) all status flags. These can be found in next column modif f column.

All of these flags are defined (don't contain random values) so the same flags are in next def f column, and undef f column must be empty.

No flag is set to a fixed value (all modified flags depend on input operands) so f values column is empty.

Last column description, notes contains only a general description of the instruction.

Example: Opcode Extensions

Some opcodes (only a few) depend on Opcode Extension Field in ModR/M byte. Using this field, the opcode is actually extended by three bits. In most cases, different extension of the same opcode means more or less different instruction. An example can be opcode `F6`. We choose last three

extensions of the opcode:

pf	0F	po	so	flds	o	proc	st	m	rl	l	mnemonic	op1	op2	op3	op4	iext	grp1	grp2	tested f	modif f	def f	undef f	f values	description, notes
		F6		5							IMUL	***AX***	*AL*	r/m8			gen	arith		o..szapc	o.....c	...szap.		Signed Multiply
		F6		6							DIV	***AL***	***AH***	*AX*	r/m8		gen	arith		o..szapc		o..szapc		Unsigned Divide
		F6		7							IDIV	***AL***	***AH***	*AX*	r/m8		gen	arith		o..szapc		o..szapc		Signed Divide

The opcode extension can be a value from 0 through 7. These values are indicated in o (Register/Opcode Field) column. In this example, values 5, 6, and 7 are chosen.

Additionally, this example shows that operands, which are not explicitly used (AL, AH, and AX operands), are set up using italic. It also shows that DIV and IDIV instructions always destroy all status flags: both modif f and undef f column contain these flags.

Example: One Opcode, More Syntaxes

Some opcodes are represented by more instructions with the same meaning, using different syntaxes. (This doesn't apply to the case when an opcode depends on Opcode Extension field in ModR/M byte. In this case, these instructions act more or less differently). Best known example are conditional jumps, for example JZ/JE, where we find something similar:

pf	0F	po	so	flds	o	proc	st	m	rl	l	mnemonic	op1	op2	op3	op4	iext	grp1	grp2	tested f	modif f	def f	undef f	f values	description, notes
		74									JZ	rel8					gen	branch	....z...					Jump short if zero/equal (ZF=0)
											JE	rel8												

Each syntax has dedicated row in mnemonic column and in columns with instruction operands.

More complex case is, for example, MOVS/MOVSW/MOVSD instruction:

pf	0F	po	so	flds	o	proc	st	m	rl	l	mnemonic	op1	op2	op3	op4	iext	grp1	grp2	tested f	modif f	def f	undef f	f values	description, notes
		A5									MOVS	**m16**	m16				gen	datamov	.d.....					Move Data from String to String
											MOVSW	***m16***	*m16*					string						
		A5			03+						MOVS	**m16/32**	m16/32				gen	datamov	.d.....					Move Data from String to String
											MOVSD	***m32***	*m32*					string						

Here, the opcode's record is complicated by the fact that since 80386 processor, the syntax is extended (thanks to 32-bit operands) with MOVSD mnemonic and MOVS syntax is changed. That's why all four syntaxes have to be split by twos.

More examples with multiple syntaxes: PUSHA/PUSHAD, SHL/SAL, or SLDT.

Example: Undocumented Instruction SETALC

All main editions contain a few undocumented instructions (from the Intel manual point of view). No that in this reference, undocumented doesn't equal invalid. All undocumented instructions mentioned by this reference work well in their shape. It is, for example, SETALC instruction:

pf	0F	po	so	flds	o	proc	st	m	rl	l	mnemonic	op1	op2	op3	op4	iext	grp1	grp2	tested f	modif f	def f	undef f	f values	description, notes
		D6			02+	D[5]					*undefined*													Undefined and Reserved; Does not Generate #UD
		D6			02+	U[6]					SALC	***AL***					gen	datamov	c					Set AL If Carry
											SETALC	***AL***												

In this case, the documented meaning goes first, as indicated in st column by "D" value. Since this opcode's documented meaning is not a common one, there is additional reference to the description where the opcode is documented. The column mnemonic implies by the value "undefined" (which is set up using italic, which always means here that it is not an original mnemonic) that the documented meaning of this opcode is "undefined and reserved". This is also stated in the last column.

Below goes the undocumented meaning of the opcode - st column holds "U" value. Each undocumented meaning should contain a reference to the description where is the opcode unofficially documented, like in this case.

More examples of undocumented instructions: INT1/ICEBP or TEST.

Columns Description

Quick navigation:

- pf Prefix
- 0F 0F Prefix
- po Primary Opcode
- so Secondary Opcode
- flds Opcode Fields
- o Register/Opcode Field
- proc Introduced with Processor
- st Documentation Status
- m Mode of Operation
- rl Ring Level
- x Lock Prefix/FPU Push/FPU Pop
- mnemonic Instruction Mnemonic
- op1, op2, ... Instruction Operands
- iext Instruction Extension Group
- grp1, grp2, grp3 Main Group, Sub-group, Sub-sub-group
- tested f, modif f, def f, undef f Tested, Modified, Defined, and Undefined Flags
- f values Flags Values

- description, notes

Name	Meaning	Description, Examples
pf	Prefix	Fixed extraordinary prefix, which may change the semantic of the Primary Opcode. Usually used in case of waiting x87 FPU instructions, and many SSE instructions. F390 PAUSE, 9BD9/7 FSTCW, F30F10 MOVSS
0F	0F Prefix	Dedicated for 0F Prefix. two-byte opcodes
po	Primary Opcode	Basic opcode. Second opcode byte in case of two- and three-byte opcodes. For coder's editions: +r means a register code, from 0 through 7, added to the value. 50 PUSH
so	Secondary Opcode	Fixed appended value to the primary opcode. It is used in some special cases, x87 FPU instructions and for new **three-byte** instructions. D40A AAM, D50A AAD, D5F8 FLD1, three-byte escape 0F38
flds	Opcode Fields	This column is present only in geek's editions. It contain present Primary Opcode binary fields. These are: • +r means a register code, from 0 through 7, added to the basic value of the Primary Opcode. 40 INC The following fields are case-sensitive: if a letter of the code is set up in lower case, it means the appropriate bit is cleared, otherwise is set. • w means bit w (bit index 0, *operand size*) is present; may be combined with bits d or s. 04 ADD • s means bit s (bit index 1, *Sign-extend*) is present; may be combined with bit w. 6B IMUL • d means bit d (bit index 1, *Direction*) is present; may be combined with bit w. 00 ADD • tttn means bit field tttn (4 bits, bit index 0, *condition*). Used only with conditional instructions. 70 JO • sr means segment register specifier - a code of one of original four segment registers (2 bits, bit index 3). See also S2 addressing method. 06 PUSH • sre means segment register specifier - a code of any segment registers (3 bits, bit index 0 or 3). See also S30 and S33 addressing methods. 0FA0 PUSH • mf means bit field MF (2 bits, bit index 1, *memory format*); used only with x87 FPU instructions coded with second floating-point instruction format. DA/0 FIADD
o	Register/ Opcode Field	1. The value of the opcode extension (values from 0 through 7). group 80 2. r indicates that the ModR/M byte contains a register operand and an r/m operand. 00 ADD
proc	Introduced with Processor	Indicates the instruction's introductory processor (code in curves apply to XML reference): • 00: 8086 • 01: 80186 • 02: 80286 • 03: 80386 • 04: 80486 • P1 (05): Pentium (1) • PX (06): Pentium with MMX • PP (07): Pentium Pro • P2 (08): Pentium II • P3 (09): Pentium III • P4 (10): Pentium 4 • C1 (11): Core (1) • C2 (12): Core 2 • C7 (13): Core i7 • IT (99): Itanium (only geek editions) The opcodes that are not forward-compatible (the ones which have been abandoned) are present only in geek's editions. 1. If the processor marking is a range (e.g., 03-04), it means that the instruction is unsupported in latter processors. 0F24 MOV 2. + (e. g., 00+) means the instruction is supported in any of latter processors and also in 64-bit mode, if the next row doesn't explicitly say otherwise. 06 PUSH ES 3. ++ (e. g., P4++) the same meaning, but only in the latter steppings of the processor (e. g., SSE3 instruction extensions). 0FA2 CPUID If this column is empty: In case of 32-bit editions, it means 00+ (8086 and all latter processors). In case of 64-bit editions, it means P4++ (P4, latter stepping, and all latter processors), because Intel 64 Architecture is available since latter stepping of the Pentium 4 processor.
st	Document. Status	Indicates how is the instruction documented in the Intel manuals: 1. D means fully documented. It can contain a reference to description which chapter in Intel manual it is documented in, if it may be unclear. D6 2. M means documented only marginally. 66 (SSE2) 3. U undocumented at all. It should contain a reference to description of the source. Note that in this reference, undocumented doesn't equal invalid. All mentioned undocumented instructions should work well in their scope. D6 SALC

Name	Meaning	Description, Examples
m	Mode of Operation	If this column is empty, it means D (documented with no further notes). Indicates the mode, which is the instruction valid on. Virtual-8086 Mode is not taken into account. 1. R applies for real, protected and 64-bit mode. SMM is not taken into account. 2. P applies for protected and 64-bit mode. SMM is not taken into account. group 0F00 3. E applies for 64-bit mode. SMM is not taken into account. 63 MOVSXD 4. S applies for SMM. 0FAA RSM If this column is empty, it means R. For 64-bit editions, E code indicates in most cases that the semantics of the opcode is specific to 64-bit mode.
rl	Ring Level	The ring level, which is the instruction valid (3 or 0) from; f indicates that the level depends on a flag(s) and it should contain a reference to the description of that flag, if the flag is not too complex. If this column is empty, it means ring 3. INT, INS, RDTSC
x	Lock Prefix FPU Push/ FPU Pop	L indicates that the instruction is basically valid with F0 LOCK prefix. 00 ADD The following codes apply only to x87 FPU instructions (none of them can use LOCK prefix). • s incidates that the opcode performs additional pu**s**h of a value to the register stack. D9 /0 FLD • p incidates that the opcode performs additional **p**op of the register stack. D9 /3 FSTP • P incidates the same like p, but pops twice. DA /5 FUCOMPP
mnemonic	Instr. Mnemonic	The instruction mnemonic itself. If there is no mnemonic, it holds additional information about the mnemonic or instruction: • If the mnemonic is set up using italic, there is no oficial mnemonic and the present one is just suggested one. D4 AMX, D5 ADX, 0FB9 UD • *no mnemonic* means that there is no mnemonic for the opcode. 66 • *invalid* means that the opcode is invalid. This option is not used everywhere the opcode is invalid, but only in some cases. 06 (64-bit mode) • *undefined* means that the behaviour of the instruction is according to official documentation undefined. D6 • *nop* means that the opcode is treated as integer NOP instruction. It should contain a reference to description of the source. no mnenonic nop • *null* means that the prefix has no meaning (no operation). 26 (64-bit mode) If there is a mnemonic, it can hold additional attributes of the instruction: • *nop* means that the instruction is treated as integer NOP instruction (except NOP instructions themselves). It should contain a reference to description of the source. DBE0 FNENI Only geek's editions: • *alias* means that the opcode is an alias to another opcode. The attribute should be a reference to that instruction. group 82, C0 /6 SAL • *part alias* means not true alias. It should contain a reference to the description of the differences between referenced instructions. F1 INT1
op1, op2, ...	Instr. Operands	Instruction operands. Geek's editions use special operand codes, explained in Instruction Operand Codes chapter below. If an operand is set up using italic, it is an implicit operand, which is not explicitly used. If an operand is set up using boldface, it is modified by the instruction.
iext	Instr. Extension Group	The instruction extension group, which was the opcode released on: 1. MMX MMX Technology 2. SSE1 Streaming SIMD Extensions (1) 3. SSE2 Streaming SIMD Extensions 2 4. SSE3 Streaming SIMD Extensions 3 5. SSSE3 Supplemental Streaming SIMD Extensions 3 6. SSE41 Streaming SIMD Extensions 4.1 7. SSE42 Streaming SIMD Extensions 4.2 8. VMX Virtualization Technology Extensions 9. SMX Safer Mode Extensions

grp1, grp2, grp3	Main Group, Sub-group, Sub -sub-group	These columns are present only in geek's editions. They classifies the instruction among groups. These groups don't match the instruction groups given by the Intel manual (I found them too loose). One instruction may fit into more groups.
		
		1. *prefix*
		  1. *segreg* segment register
		  2. *branch*
		  1. *cond* conditional
		  3. *x87fpu*
		  1. *control* (only WAIT)
		2. *obsol* obsolete
		  1. *control*

Name	Meaning	Description, Examples

3. *gen* general
    1. *datamov* data movement
    2. *stack*
    3. *conver* type conversion
    4. *arith* arithmetic
        1. *binary*
        2. *decimal*
    5. *logical*
    6. *shftrot* shift&rotate
    7. *bit* bit manipulation
    8. *branch*
        1. *cond* conditional
    9. *break* interrupt
    10. *string* (means that the instruction can make use of the REP family prefixes)
    11. *inout* I/O
    12. *flgctrl* flag control
    13. *segreg* segment register manipulation
    14. *control*
4. system
    1. *branch*
        1. *trans* transitional (implies sensitivity to operand-size attribute)
5. *x87fpu* x87 FPU
    1. *datamov* data movement
    2. *arith* basic arithmetic
    3. *compar* comparison
    4. *trans* transcendental
    5. *ldconst* load constant
    6. *control*
    7. *conv* conversion
6. *sm* x87 FPU and SIMD state management

MMX instruction extensions technology groups. Note that these groups are just experimental and may change in future.

1. *datamov* data movement
2. *arith* packed arithmetic
3. *compar* comparison
4. *conver* conversion
5. *logical*
6. *shift*
7. *unpack* unpacking

SSE1 instruction extensions groups. Note that these groups are just experimental and may change in future.

1. *simdfp* SIMD single-precision floating-point
    1. *datamov* data movement
    2. *arith* packed arithmetic
    3. *compar* comparison
    4. *logical*
    5. *shunpck* shuffle&unpacking
2. *conver* conversion instructions
3. *simdint* 64-bit SIMD integer
4. *mxcsrsm* MXCSR state management
5. *cachect* cacheability control
6. *fetch* prefetch
7. *order* instruction ordering

SSE2 instruction extensions groups. Note that these groups are just experimental and may change in future.

1. *pcksclr* packed and scalar double-precision floating-point
    1. *datamov* data movement
    2. *conver* conversion
    3. *arith* packed arithmetic
    4. *compar* comparison
    5. *logical*
    6. *shunpck* shuffle&unpacking
2. *pcksp* packed single-precision floating-point
3. *simdint* 128-bit SIMD integer
    1. *datamov* data movement
    2. *arith* packed arithmetic
    3. *shunpck* shuffle&unpacking
    4. *shift*
    5. *compar* comparison
    6. *conver* conversion
    7. *logical*
4. *cachect* cacheability control

Name	Meaning	Description, Examples

		5. *order* instruction ordering
		
		SSE3 instruction extensions groups. Note that these groups are just experimental and may change in future.
		
		1. *simdfp* SIMD single-precision floating-point (SIMD packed)
		  1. *datamov* data movement
		  2. *arith* packed arithmetic
		2. *cachect* cacheability control
		3. *sync* agent synchronization
		
		SSSE3 instruction extensions group. Note that these groups are just experimental and may change in future.
		
		1. *simdint* SIMD integer
		
		SSE4.1 instruction extensions group. Note that these groups are just experimental and may change in future.
		
		1. *simdint* SIMD integer
		  1. *datamov* data movement
		  2. *arith* packed arithmetic
		  3. *compar* comparison
		  4. *conver* conversion
		2. *simdfp* SIMD SIMD floating-point
		  1. *datamov* data movement
		  2. *arith* packed arithmetic
		  3. *conver* conversion
		3. *cachect* cacheability control
		
		SSE4.2 instruction extensions group. Note that these groups are just experimental and may change in future.
		
		1. *simdint* SIMD integer
		  1. *compar* comparison
		2. *strtxt* string and text processing
		
		VMX and SMX instruction extensions has no groups at the moment. The grouping may be added in future.

tested f, modif f, def f, undef f	Tested, Modified, Defined, and Undefined Flags	• For rFlags register, indicates these flags using *odiszapc* pattern. Present flag fits in with the appropriate group. group C0 • For x87 FPU flags, indicates these flags using *1234* x87 FPU flag pattern. Present flag fits in with the appropriate group. DB/7 FSTP Note that if a flag is present in both Defined and Undefined column, the flag fits in under further conditions, which are not described by this reference.
f values	Flags Values	• For rFlags register, indicates the values of flags, which are always set or cleared, using case-sensitive *odiszapc* flag pattern. Lower-case flag means cleared flag, upper-case means set flag. STC • For x87 FPU flags, indicates these flags using *1234* x87 FPU flag pattern. Present flag holds its value. DBE3 FNINIT
description, notes		Short desciption of the opcode. For now, the descriptions are very general. They will be improved in future perhaps.

Instruction Operand Codes

These codes come from official codes used in Intel manual Instruction Set Reference, N-Z for Pentium 4 processor, revision 17. The reason of using this particular, out-of-date revision is that the codes from this revision are most apposite ones. In next revisions the codes changed unfortunately. These codes were modified and completed mainly because of the possibility to code operands simultaneously for 64-bit mode. Ideally, it would be the best to make brand new codes, but I'm afraid those wouldn't be widely acceptable.

The State column says if the code is original, added or changed.

The "Geek" part in these tables in the first column indicates codes used in HTML geek's editions and in the source XML document as well. The "Coder" part indicates alternative codes used in HTML coder's editions. These are used also within instruction reference in Intel manual.

Codes for Addressing Method

The following abbreviations are used for addressing methods:

Geek Coder	State	Description
A	Original	Direct address. The instruction has no ModR/M byte; the address of the operand is encoded in the instruction; no base

Geek Coder	State	Description
ptr		register, index register, or scaling factor can be applied (for example, far JMP (EA)).
BA m	Added	Memory addressed by DS:EAX, or by rAX in 64-bit mode (only 0F01C8 MONITOR).
BB m	Added	Memory addressed by DS:eBX+AL, or by rBX+AL in 64-bit mode (only XLAT). (This code changed from single B in revision 1.00)
BD m	Added	Memory addressed by DS:eDI or by RDI (only 0FF7 MASKMOVQ and 660FF7 MASKMOVDQU) (This code changed from YD (introduced in 1.00) in revision 1.02)
C CRn	Original	The reg field of the ModR/M byte selects a control register (only MOV (0F20, 0F22)).
D DRn	Original	The reg field of the ModR/M byte selects a debug register (only MOV (0F21, 0F23)).
E r/m	Original	A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, or a displacement.
ES STi/m	Added	(Implies original E). A ModR/M byte follows the opcode and specifies the operand. The operand is either a x87 FPU stack register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, or a displacement.
EST STi	Added	(Implies original E). A ModR/M byte follows the opcode and specifies the x87 FPU stack register.
F -	Original	rFLAGS register.
G r	Original	The reg field of the ModR/M byte selects a general register (for example, AX (000)).
H r	Added	The r/m field of the ModR/M byte always selects a general register, regardless of the mod field (for example, MOV (0F20)).
I imm	Original	Immediate data. The operand value is encoded in subsequent bytes of the instruction.
J rel	Original	The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (E9), LOOP)).
M m	Original	The ModR/M byte may refer only to memory: mod != 11bin (BOUND, LEA, CALLF, JMPF, LES, LDS, LSS, LFS, LGS, CMPXCHG8B, CMPXCHG16B, F20FF0 LDDQU).
N mm	Original	The R/M field of the ModR/M byte selects a packed quadword MMX technology register.
O moffs	Original	The instruction has no ModR/M byte; the offset of the operand is coded as a word, double word or quad word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (only MOV (A0, A1, A2, A3)).
P mm	Original	The reg field of the ModR/M byte selects a packed quadword MMX technology register.
Q mm/m64	Original	A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
R r	Original	The mod field of the ModR/M byte may refer only to a general register (only MOV (0F20-0F24, 0F26)).
S Sreg	Original	The reg field of the ModR/M byte selects a segment register (only MOV (8C, 8E)).
SC -	Added	Stack operand, used by instructions which either push an operand to the stack or pop an operand from the stack. Pop-like instructions are, for example, POP, RET, IRET, LEAVE. Push-like are, for example, PUSH, CALL, INT. No Operand type is provided along with this method because it depends on source/destination operand(s).

Geek Coder	State	Description
T TRn	Original	The reg field of the ModR/M byte selects a test register (only MOV (0F24, 0F26)).
U xmm	Original	The R/M field of the ModR/M byte selects a 128-bit XMM register.
V xmm	Original	The reg field of the ModR/M byte selects a 128-bit XMM register.
W xmm/m	Original	A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement
X m	Original	Memory addressed by the DS:eSI or by RSI (only MOVS, CMPS, OUTS, and LODS). In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit modes, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.
Y m	Original	Memory addressed by the ES:eDI or by RDI (only MOVS, CMPS, INS, STOS, and SCAS). In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit modes, only 32-bit (EDI) and 16-bit (DI) address sizes are supported. The implicit ES segment register cannot be overriden by a segment prefix.
Z r	Added	The instruction has no ModR/M byte; the three least-significant bits of the opcode byte selects a general-purpose register

The following abbreviations are used for addressing methods only in case of direct segment registers and are accessible only in HTML geek's editions as segment register's title. As for source XML document, they are used within *address* atribute of *syntax/dst* or *syntax/src* elements. All of them are added:

- S2 The two bits at bit index three of the opcode byte selects one of original four segment registers (for example, PUSH ES).
- S30 The three least-significant bits of the opcode byte selects segment register SS, FS, or GS (for example, LSS).
- S33 The three bits at bit index three of the opcode byte selects segment register FS or GS (for example, PUSH FS).

Codes for Operand Type

The following abbreviations are used for operand types:

Geek Coder	State	Description
a 16/32&16/32	Original	Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (only BOUND).
b 8	Original	Byte, regardless of operand-size attribute.
bcd 80dec	Added	Packed-BCD. Only x87 FPU instructions (for example, FBLD).
bs 8	Added; simplified bsq	Byte, sign-extended to the size of the destination operand.
bsq -	Original; replaced by bs	(Byte, sign-extended to 64 bits.)
bss 8	Original	Byte, sign-extended to the size of the stack pointer (for example, PUSH (6A)).
c ?	Original	Byte or word, depending on operand-size attribute. (unused even by Intel?)
d 32	Original	Doubleword, regardless of operand-size attribute.
di 32int	Added	Doubleword-integer. Only x87 FPU instructions (for example, FIADD).
dq	Original	Double-quadword, regardless of operand-size attribute (for example, CMPXCHG16B).

Geek Coder	State	Description
128		
dqp 32/64	Added; combines d and qp	Doubleword, or quadword, promoted by REX.W in 64-bit mode (for example, MOVSXD).
dr 64real	Added	Double-real. Only x87 FPU instructions (for example, FADD).
ds 32	Original	Doubleword, sign-extended to 64 bits (for example, CALL (E8).
e 14/28	Added	x87 FPU environment (for example, FSTENV).
er 80real	Added	Extended-real. Only x87 FPU instructions (for example, FLD).
p 16:16/32	Original	32-bit or 48-bit pointer, depending on operand-size attribute (for example, CALLF (9A).
pi (64)	Original	Quadword MMX technology data.
pd	Original	128-bit packed double-precision floating-point data.
ps (128)	Original	128-bit packed single-precision floating-point data.
psq 64	Added	64-bit packed single-precision floating-point data.
pt -	Original; replaced by ptp	(80-bit far pointer.)
ptp 16:16/32/64	Added	32-bit or 48-bit pointer, depending on operand-size attribute, or 80-bit far pointer, promoted by REX.W in 64-bit mode (for example, CALLF (FF /3)).
q 64	Original	Quadword, regardless of operand-size attribute (for example, CALL (FF /2)).
qi 64int	Added	Qword-integer. Only x87 FPU instructions (for example, FILD).
qp 64	Original	Quadword, promoted by REX.W (for example, IRETQ).
s -	Changed to Changed from	6-byte pseudo-descriptor, or 10-byte pseudo-descriptor in 64-bit mode (for example, SGDT). *6-byte pseudo-descriptor*.
sd -	Original	Scalar element of a 128-bit packed double-precision floating data.
si ?	Original	Doubleword integer register (e. g., eax). (unused even by Intel?)
sr 32real	Added	Single-real. Only x87 FPU instructions (for example, FADD).
ss -	Original	Scalar element of a 128-bit packed single-precision floating data.
st 94/108	Added	x87 FPU state (for example, FSAVE).
stx	Added	x87 FPU and SIMD state (FXSAVE and FXRSTOR).

Geek Coder	State	Description
512 t -	Original; replaced by ptp	10-byte far pointer.
v 16/32	Original	Word or doubleword, depending on operand-size attribute (for example, INC (40), PUSH (50)).
vds 16/32	Added; combines v and ds	Word or doubleword, depending on operand-size attribute, or doubleword, sign-extended to 64 bits for 64-bit operand size.
vq 64/16	Original	Quadword (default) or word if operand-size prefix is used (for example, PUSH (50)).
vqp 16/32/64	Added; combines v and qp	Word or doubleword, depending on operand-size attribute, or quadword, promoted by REX.W in 64-bit mode.
vs 16/32	Original	Word or doubleword sign extended to the size of the stack pointer (for example, PUSH (68)).
w 16	Original	Word, regardless of operand-size attribute (for example, ENTER).
wi 16int	Added	Word-integer. Only x87 FPU instructions (for example, FIADD).

The following abbreviations are used for operand types and are accessible only in HTML geek's editions as operand's code title. They are issued to indicate a dependency on address-size attribute instead of operand-size attribute. As for source XML document, they are used within *address* atribute of *syntax/dst* or *syntax/src* elements. All of them are added:

- va Word or doubleword, according to address-size attribute (only REP and LOOP families).
- dqa Doubleword or quadword, according to address-size attribute (only REP and LOOP families).
- wa Word, according to address-size attribute (only JCXZ instruction).
- wo Word, according to current operand size (e. g., MOVSW instruction).
- ws Word, according to current stack size (only PUSHF and POPF instructions in 64-bit mode).
- da Doubleword, according to address-size attribute (only JECXZ instruction).
- do Doubleword, according to current operand size (e. g., MOVSD instruction).
- qa Quadword, according to address-size attribute (only JRCXZ instruction).
- qs Quadword, according to current stack size (only PUSHFQ and POPFQ instructions).

Current State

In this version, the reference is almost complete. It contains general, system, x87 FPU, MMX, SSE, SSE1, SSE2, SSE3, SSSE3, SSE4, VMX, and SMX instructions (both one-byte and two-byte ones). We are working on AMD-specific instructions and Intel AVX instructions now.

The MMX and SSE* instruction classification among groups is considered experimental and may change in future.

Note that from the point of project's progress, modifications of any of HTML editions is almost useless. A HTML edition is just a result of transformation of source XML file, so all modifications need to be done there.

Implementations

Bukowski's disassembler is first public implementation of the XML reference.

Mediana, maintained by Mikae, is table-based x86/x86-64 disassembler engine. However, the transformation from source XML file is not a part of the project.

License

Since version 1.12, the reference is licensed under GPL-3.0. For more see its GitHub repository.

The old license (used up to version 1.12) is not available anymore.

Resources

This reference has been completed using the following resources:

Intel manuals

Sandpile.org

AMD manuals

Intel iAPX 86/88, 186/188 User's manual

Credits

Thanks to all these geeks involved in some way in this project:

Christian Ludloff: maintainer of Sandpile.org site, one of important sources for this project

Martin Mocko a.k.a. **vid**: many design ideas for HTML editions

Anthony Lopes: great XML and XSL contributions

Aquila: many great contributions

EliCZ: bug reports, design ideas

Cephexin: many great contributions to XML

Miloslav Ponkrác: helped with PHP and JavaScript on this site

William Whistler: valuable reviews and bug reports

Mikae: reviews, bug reports

References

Handily-organized x86 instruction and opcode references

x86オペコードリファレンスその後の後

Referencia de Instrucciones y Códigos de Operación (OPCodes) x86

Download

The source files can be downloaded from GitHub repository.

HTML Editions Files

coder.html	coder-abc.html
coder32.html	coder32-abc.html
coder64.html	coder64-abc.html
geek.html	geek-abc.html
geek32.html	geek32-abc.html
geek64.html	geek64-abc.html

Comments

My contact information is here.

Revisions

2017-02-18	1.12	Various bugfixes. See GitHub releases for details.	MazeGen
2010-01-20	1.11	Mostly a bugfix release	MazeGen
2009-08-19	1.10	• All SSE4 instructions (Aquila contribution) • All VMX instructions • (the only) SMX instruction • All new general instructions: `POPCNT`, `MOVBE` • All new system instructions: `XGETBV`, `XSETBV`, `RDTSCP`, `XSAVE`, `XRSTOR` • Processor code `C7` to indicate Core i7 • Implicate register operand group `xcr` (extended control register) added because of `XGETBV` and `XSETBV` instructions	MazeGen
2009-06-30	1.02	The first version considered stable	MazeGen
2008-12-17	1.01β	Various bugfixes and updates	MazeGen

2008-10-19	1.00β	• All SSE, SSE2, SSE3, and SSSE3 instructions added (Aquila and Cephexin contributions) • Alphabetically sorted editions (postfixed with *-abc*)	MazeGen
2008-05-15	0.40β	All MMX instructions added (Anthony Lopes contribution)	MazeGen
2008-03-11	0.30β	All x87 FPU instructions added, including new ones	MazeGen
2007-11-29	0.21β	Various changes	MazeGen
2007-11-06	0.20β	Added coder, coder32, coder64, geek32, and geek64 editions. All main project's files modified. Project's documentation completed.	MazeGen
2007-06-04	0.10β	First public version	MazeGen

(dates format correspond to ISO 8601)