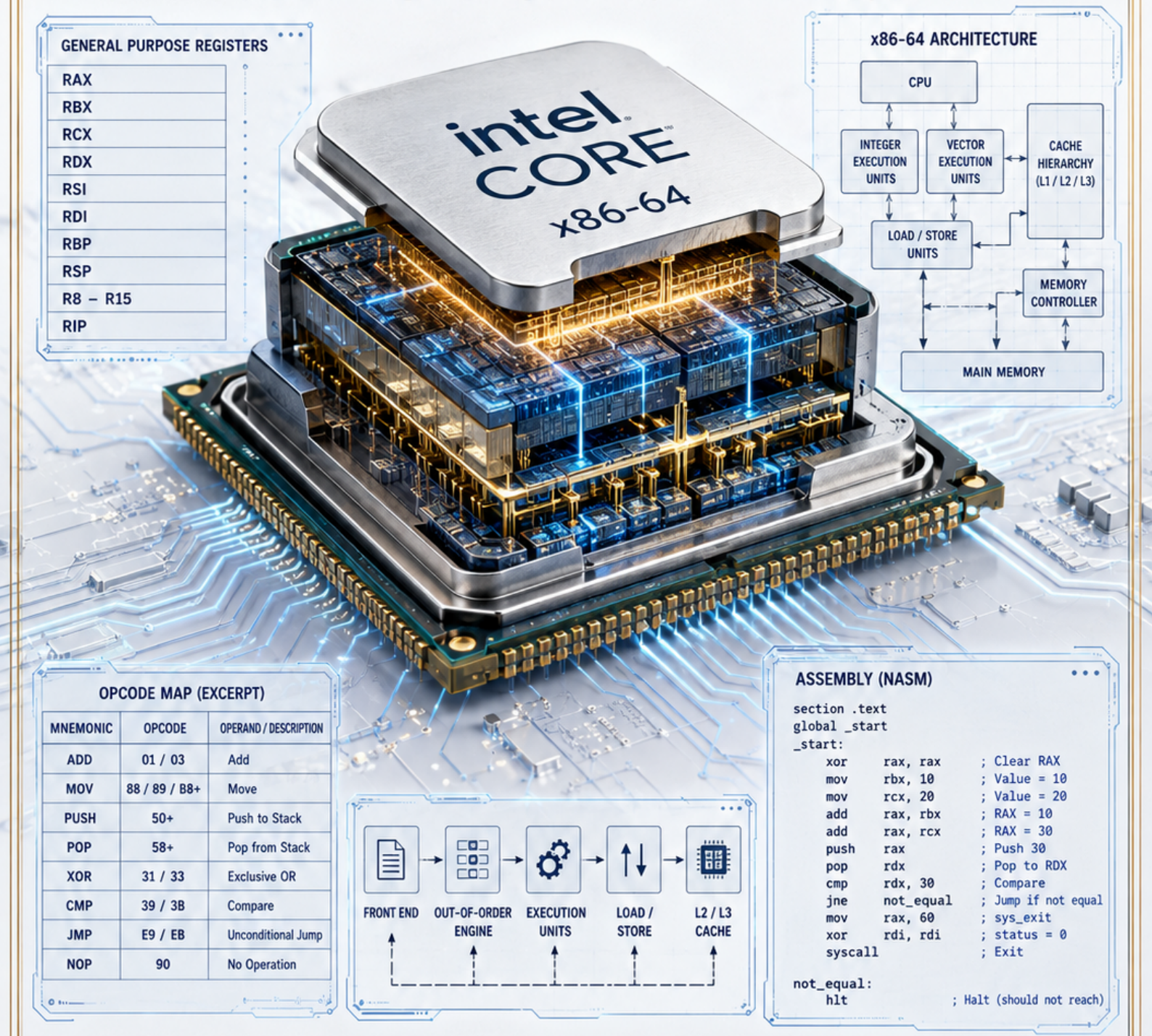
The Comprehensive Reference Guide to Intel x86-64 Instructions

From 8086 to Modern Processors
Instructions, Registers, Flags, and Simplified NASM Examples



OPCODE MAP (EXCERPT)

MNEMONIC	OPCODE	OPERAND / DESCRIPTION
ADD	01 / 03	Add
MOV	88 / 89 / B8+	Move
PUSH	50+	Push to Stack
POP	58+	Pop from Stack
XOR	31 / 33	Exclusive OR
CMP	39 / 3B	Compare
JMP	E9 / EB	Unconditional Jump
NOP	90	No Operation

ASSEMBLY (NASM)

```
section .text
global _start
_start:
    xor     rax, rax        ; Clear RAX
    mov     rbx, 10         ; Value = 10
    mov     rcx, 20         ; Value = 20
    add     rax, rbx        ; RAX = 10
    add     rax, rcx        ; RAX = 30
    push    rax             ; Push 30
    pop     rdx             ; Pop to RDX
    cmp     rdx, 30         ; Compare
    jne     not_equal       ; Jump if not equal
    mov     rax, 60         ; sys_exit
    xor     rdi, rdi        ; status = 0
    syscall                 ; Exit

not_equal:
    hlt                     ; Halt (should not reach)
```


Prepared by Ayman Alheraki

Intel x86-64 Instruction Reference with NASM

The Comprehensive Reference Index for Intel x86-64 Instructions

From 8086 to modern processors - instructions, registers, flags, and simple NASM examples

Prepared by **Ayman Alheraki**

First educational reference edition - July 2026

Introduction

x86-64 instructions are not merely a list of commands. They are layers of compatibility, architectural evolution, SIMD extensions, system instructions, cryptographic acceleration, cache hints, virtualization, and newer AI-oriented features. A learner therefore needs a map before diving into thousands of pages of official Intel manuals.

This booklet is designed as a structured educational index. It begins with the 8086 root, then moves through 32-bit IA-32, Intel 64, SSE, AVX, AVX2, AVX-512, AMX, APX, and AVX10-related references. For each group you will see the processor generation or extension, the idea of the instruction, a short caution, and a simple NASM example.

Important methodological warning

This booklet does not copy or replace the Intel Software Developer's Manual. Exact encodings, exception behavior, valid modes, CPUID feature bits, latency/throughput data, and #UD/#GP cases must be checked in the official Intel documents. The goal here is a readable learning map and a quick index.

How to use this booklet

Start with the registers and flags, then learn the core 8086 instruction families, then 80386 and x86-64, and only after that move to SSE2, AVX2, AVX-512, AMX, and APX.

Table of Contents

How to Read an Instruction Card

An Intel instruction reference page usually contains the syntax, opcode encoding, operand rules, effect on flags, exceptions, valid modes, and feature requirements. This booklet compresses that information into five educational fields:

Field	Practical meaning
Mnemonic	The instruction or instruction family name, such as ADD, VADDPS, or ENDBR64.
Processor / extension	The earliest generation or architectural feature where it belongs pedagogically. Real availability must be verified with CPUID.
Idea	What the instruction does in one short sentence.
Caution / use	A practical warning, common use, or rare use.
NASM example	A minimal example. It does not cover every form or every restriction.

Example: reading ADD

ADD adds the source to the destination and updates arithmetic flags such as CF, ZF, SF, OF, PF, and AF.

```
add rax, rbx
add qword [counter], 1
```

Do not use an instruction just because its name exists

For any modern instruction: check CPUID, OSXSAVE/XGETBV when vector state is involved, OS/ABI support, and assembler support.

Timeline of Intel x86 Instruction Evolution

This timeline is educational and approximate. It organizes the mind; it is not a complete history of every stepping, SKU, or microarchitecture.

Year / period	Processor or extension	Approximate additions	Meaning for the learner
1978	8086/8088	16-bit foundation: MOV, ADD, SUB, CMP, JMP, CALL, RET, strings, FLAGS, segments.	The educational base of x86.
1982	80186	ENTER/LEAVE, PUSHA/POPA, INS/OUTS, BOUND, PUSH immediate.	Some are historical or invalid in 64-bit mode.
1982	80286	Protected mode and descriptor tables: LGDT/LIDT/LTR/VERR/VERW.	Operating-system and protection concepts.
1985	80386	32-bit GPRs, paging, MOVSX/MOVZX, BSF/BSR, BT/BTR/BTS/BTC, SETcc.	The foundation of IA-32.
1989	80486	CMPXCHG, XADD, BSWAP, cache/TLB instructions.	Atomicity and cache control.
1993	Pentium	CPUID, RDTSC, CMPXCHG8B, pipeline-era improvements.	Feature discovery and timing.
1995-1997	P6 / Pentium Pro/II	CMOVcc, FCOMI, SYSENTER/SYSEXIT, FXSAVE.	Performance and system-call evolution.
1997	MMX	64-bit integer SIMD: MM0-MM7.	Historical media/SIMD layer.
1999	SSE	XMM registers, FP32 SIMD, MXCSR, PREFETCH, SFENCE.	Beginning of modern SIMD.
2001	SSE2	FP64 and integer SIMD in XMM; a practical x86-64 baseline.	Very important for modern code.
2004-2008	SSE3/SSSE3/SSE4.x	Horizontal ops, PSHUFB, BLEND, CRC32, string compare.	Media, text, and data processing.
2006-2008	Intel 64 mainstream	RAX-R15, RIP-relative addressing, SYSCALL, CMPXCHG16B.	64-bit programming model.
2010	AES-NI/PCLMUL	AES rounds and carry-less multiply.	Cryptography acceleration.
2011	AVX/F16C	YMM registers, VEX encoding, three-operand forms, FP16 conversion.	256-bit SIMD.
2013	AVX2/FMA/BMI1/2	256-bit integer SIMD, gather, FMA, bit manipulation.	Widely useful on client/server machines.
2015+	AVX-512 families	ZMM registers, opmask registers, EVEX, many feature subsets.	Not uniform across all processors.
2017+	SGX/MPX/CET/CLWB and related history	Security and memory features; some continued, some were deprecated.	Always check CPUID and current guidance.

Year / period	Processor or extension	Approximate additions	Meaning for the learner
2021+	AMX	Tile registers and matrix dot-product operations.	AI/HPC on supporting families.
2024-2026	Core Ultra client	Intel 64 and 64-bit instruction set support; many client SKUs expose SSE4.1/SSE4.2/AVX2, but features vary.	Do not assume AVX-512 on client systems.
2025+	APX / AVX10 references	APX adds EGPRs R16-R31 and new forms; AVX10 aims at vector ISA consolidation.	Modern/future-oriented; verify support.

Register Index in Intel x86-64

Registers are the key to instruction meaning. An instruction only becomes clear when you understand its source, destination, implicit registers, and the flags it leaves behind.

General-Purpose Registers (GPRs) in x86-64

Register	Type / width	Common and less common uses
RAX	Accumulator	Arithmetic results, Linux syscall number, function return value according to the ABI.
RBX	Base register	General-purpose register; usually callee-saved in System V and Windows x64.
RCX	Counter / Windows 1st argument	Loop/string count in some instructions; CL is the shift count; also used by Windows x64 ABI.
RDX	Data / SysV 3rd argument	Used with RAX for multiply/divide results; Linux syscall third argument.
RSI	Source index	Source pointer for string instructions; function argument in System V ABI.
RDI	Destination index	Destination pointer for string instructions; first argument in System V ABI.
RBP	Frame pointer	Stack frame base when frame pointers are enabled; can be used as a GPR when the ABI/compiler allows it.
RSP	Stack pointer	Top of the stack. Keep alignment and calling-convention rules strictly.
R8-R15	Extended GPRs	Additional GPRs added with x86-64; several are used for function arguments depending on ABI.
RIP	Instruction pointer	Current instruction pointer; enables RIP-relative addressing.
R16-R31	APX EGPRs	Additional GPRs introduced by Intel APX when supported.

Subregister Names and Partial Registers

Register	Type / width	Common and less common uses
RAX/EAX/AX/AH/AL	64/32/16/8	Writing to EAX zeroes the upper half of RAX in 64-bit mode.
RBX/EBX/BX/BH/BL	64/32/16/8	High-byte registers AH/BH/CH/DH have restrictions with REX prefixes.
RCX/ECX/CX/CH/CL	64/32/16/8	CL is used as the variable shift/rotate count.
RDX/EDX/DX/DH/DL	64/32/16/8	RDX:RAX participates in multiply and divide operations.
RSI/ESI/SI/SIL	64/32/16/8	SIL is available with 64-bit/REX encodings.
RDI/EDI/DI/DIL	64/32/16/8	DIL is available with 64-bit/REX encodings.
RBP/EBP/BP/BPL	64/32/16/8	BPL is available with 64-bit/REX encodings.
RSP/ESP/SP/SPL	64/32/16/8	SPL is available with 64-bit/REX encodings.
R8-R15 + d/w/b	64/32/16/8	Examples: r8, r8d, r8w, r8b.

Segment Registers

Register	Type / width	Common and less common uses
CS	Code segment	Still relevant to privilege level in 64-bit mode; base/limit are mostly ignored.
DS/ES/SS	Data/extra/stack segments	Limited effect in 64-bit mode; historically central to segmentation.
FS/GS	Thread/local storage	Commonly used for TLS and kernel per-CPU data.

SIMD and Floating-Point Registers

Register	Type / width	Common and less common uses
ST0-ST7	x87 stack	80-bit x87 stack registers; legacy but still architecturally relevant.
MM0-MM7	MMX	Alias x87 state; EMMS is required before returning to x87.
XMM0-XMM31	SSE/AVX/AVX-512	128-bit vector registers; XMM0-XMM15 are standard in x86-64, more with AVX-512/APX contexts.
YMM0-YMM31	AVX/AVX2	256-bit vector registers. Upper state requires OSXSAVE/XGETBV support.
ZMM0-ZMM31	AVX-512	512-bit vector registers; not universal across Intel client processors.
K0-K7	AVX-512 masks	Opmask registers. k0 often means no masking.
TMM0-TMM7	AMX tiles	Tile registers for matrix operations; require OS support.

System and Control Registers

Register	Type / width	Common and less common uses
CR0	Control register	Enables protected mode, paging, and core control features.
CR2	Page-fault linear address	Receives the linear address that caused a page fault.
CR3	Page-table base	Root of paging structures; central to address-space switching.
CR4	Feature control	Enables PAE, OSFXSR, OSXSAVE, SMEP, SMAP, and other features.
CR8	Task-priority register	Interrupt-priority management in x86-64 environments.
DR0-DR7	Debug registers	Hardware breakpoints and debug control.
GDTR/IDTR/LDTR/TR	Descriptor/task registers	Protection, interrupts, descriptor tables, and task structures.
MSRs	Model-specific registers	Read and written with RDMSR/WRMSR; documented in Intel SDM Volume 4.
XCR0	Extended control register	Selects which XSAVE-managed states such as XMM/YMM/ZMM/AMX are enabled.

ABI note

Register usage changes with the ABI. On Linux x86-64 System V, the first function arguments are usually RDI, RSI, RDX, RCX, R8, R9. Linux syscall uses RAX for the syscall number, then RDI, RSI, RDX, R10, R8, R9. Windows x64 uses RCX, RDX, R8, R9 for the first arguments.

RFLAGS / FLAGS Index

Flags are the small architectural memory left by arithmetic, logical, and comparison instructions. Jcc, CMOVcc, SETcc, and many other instructions read them.

Flag	Bit	Name	Practical use
CF	0	Carry	Carry/borrow; vital for unsigned and multi-precision arithmetic.
PF	2	Parity	Parity of the low byte; rarely used in modern application code.
AF	4	Auxiliary carry	BCD-related half-carry; rarely used today.
ZF	6	Zero	Set when the result is zero; basis for JE/JZ.
SF	7	Sign	Sign bit of the result.
TF	8	Trap	Single-step debugging.
IF	9	Interrupt enable	Interrupt enable flag; privileged in normal use.
DF	10	Direction	Direction for string instructions; normally keep it clear with CLD.
OF	11	Overflow	Signed overflow.
IOPL	12	I/O privilege level	Privilege level for I/O operations.
NT	14	Nested task	Historical task-switching flag.
RF	16	Resume	Used around debug exceptions.
VM	17	Virtual-8086	Virtual-8086 mode.
AC	18	Alignment check / SMAP assist	Alignment checking and kernel STAC/CLAC interactions.
VIF/VIP	19	Virtual interrupt flags	Virtualized interrupt flags.
ID	21	CPUID modifiable flag	Historically used to test CPUID availability.

NASM Examples for Flags

```
cmp rax, rbx
je  equal      ; ZF=1
ja  above      ; unsigned: CF=0 and ZF=0
jg  greater    ; signed: ZF=0 and SF=OF

add rax, rbx
jc  carry_out  ; CF=1
jo  overflow   ; OF=1
```

Golden rule

Use JA/JB for unsigned comparisons. Use JG/JL for signed comparisons.

A Minimal NASM Environment on Linux x86-64

The examples in this booklet use NASM syntax where possible. The following complete program prints a message and exits:

```
section .data
    msg db "Hello from NASM", 10
    len equ $ - msg

section .text
    global _start

_start:
    mov rax, 1      ; sys_write
    mov rdi, 1      ; stdout
    lea rsi, [rel msg]
    mov rdx, len
    syscall

    mov rax, 60     ; sys_exit
    xor rdi, rdi
    syscall
```

Typical build commands:

```
nasm -f elf64 hello.asm -o hello.o
ld hello.o -o hello
./hello
```

Warning

System instructions such as HLT, LGDT, MOV CR3, and VMXON cannot be executed from an ordinary user-space program. They may raise protection exceptions. A NASM example is not a permission guarantee.

Instruction Index by Processor and Extension

This is the heart of the booklet. Rows combine individual instructions and instruction families when the family is too large to list every form, especially AVX-512, AMX, and APX. The purpose is to give a learner the map quickly, then send them to Intel SDM for exact forms.

8086 / 8088 - The Educational Root of x86

This group is the practical root of x86 programming. Many instructions still exist in 64-bit mode through updated encodings, while old BCD and segmented-addressing instructions are mostly historical or invalid in long mode.

AAA

8086 **ASCII adjust after addition**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
aaa
```

AAD

8086 **ASCII adjust before division**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
aad
```

AAM

8086 **ASCII adjust after multiply**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
aam
```

AAS

8086 **ASCII adjust after subtraction**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
aas
```

ADC

8086 **Add with carry**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
add rax, rbx
adc rdx, rcx
```

ADD

8086 **Integer addition**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
add rax, 10
```

AND

8086 **Bitwise AND**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
and eax, 0xff
```

CALL

8086 **Call procedure**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
call print_message
```

CBW/CWDE/CDQE

8086/386/x64 **Sign extend accumulator**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cdqe
```

CLC

8086 **Clear carry flag**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
clc
```

CLD

8086 **Clear direction flag**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cld
```

CLI

8086 **Clear interrupt flag**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cli
```

CMC

8086 **Complement carry flag**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cmc
```

CMP

8086 **Compare**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cmp rax, rbx
ja greater
```

CMPSB/CMPSW/CMPSD/CMPSQ

8086/386/x64 **Compare strings**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cld
repe cmpsb
```

CWD/CDQ/CQO

8086/386/x64 **Sign extend for division**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cqo
idiv rbx
```

DAA

8086 **Decimal adjust after addition**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
daa
```

DAS

8086 **Decimal adjust after subtraction**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
das
```

DEC

8086 **Decrement**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
dec rcx
```

DIV

8086 **Unsigned divide**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xor edx, edx
mov eax, 100
div ebx
```

ESC

8086 **Escape to coprocessor**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
; conceptual example - verify support first
```

HLT

8086 **Halt processor**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
hlt
```

IDIV

8086 **Signed divide**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cqo
idiv rbx
```

IMUL

8086/80186 **Signed multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
imul rax, rbx
```

IN

8086 **Input from I/O port**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
in al, dx
```

INC

8086 **Increment**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
inc rax
```

INT

8086 **Software interrupt**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
int 0x80
```

INTO

8086 **Interrupt on overflow**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
into
```

IRET/IRETD/IRETQ

8086/386/x64 **Return from interrupt**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
iretq
```

Jcc

8086 **Conditional jumps**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
cmp rax, 0
je zero
ja above
```

JMP

8086 **Unconditional jump**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
jmp done
```

LAHF

8086 **Load AH from flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lahf
```

LDS/LES

8086 **Load far pointer**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
lds ax, [ptr]
```

LEA

8086 **Load effective address**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lea rax, [rbx + rcx*8 + 16]
```

LOCK

8086 **Atomic prefix**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
lock inc qword [counter]
```

LODSB/LODSW/LODSD/LODSQ

8086/386/x64 **Load string element**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cld
lodsb
```

LOOP/LOOPE/LOOPNE

8086 **Loop with RCX/ECX/CX**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mov ecx, 10
.loop:
; work
loop .loop
```

MOV

8086 **Move data**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mov rax, [value]
```

MOVSB/MOVSW/MOVSD/MOVSQ

8086/386/x64 **Move string**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cld
mov rcx, 64
rep movsb
```

MUL

8086 **Unsigned multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mul rbx
```

NEG

8086 **Two complement negate**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
neg rax
```

NOP

8086 **No operation**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
nop
```

NOT

8086 **Bitwise NOT**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
not rax
```

OR

8086 **Bitwise OR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
or eax, 1
```

OUT

8086 **Output to I/O port**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
out dx, al
```

POP

8086 **Pop from stack**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pop rax
```

POPF/POPFD/POPFQ

8086/386/x64 **Pop flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
popfq
```

PUSH

8086 **Push to stack**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
push rax
```

PUSHF/PUSHFD/PUSHFQ

8086/386/x64 **Push flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pushfq
```

RCL/RCR

8086 **Rotate through carry**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rcl rax, 1
```

REP/REPE/REPNE

8086 **Repeat prefixes**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
rep stosb
```

RET

8086 **Return from procedure**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
ret
```

ROL/ROR

8086 **Rotate bits**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rol eax, 8
```

SAHF

8086 **Store AH into flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sahf
```

SAL/SHL

8086 **Shift left**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
shl rax, 1
```

SAR/SHR

8086 **Shift right**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sar rax, 1
shr rbx, 1
```

SBB

8086 **Subtract with borrow**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sub rax, rbx
sbb rdx, rcx
```

SCASB/SCASW/SCASD/SCASQ

8086/386/x64 **Scan string**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cld
repne scasb
```

STC/STD/STI

8086 **Set flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
stc
```

STOSB/STOSW/STOSD/STOSQ

8086/386/x64 **Store string**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
xor eax, eax
mov rcx, 128
rep stosb
```

SUB

8086 **Subtract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sub rax, 1
```

TEST

8086 **Bit test via AND**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
test rax, rax
jz zero
```

WAIT/FWAIT

8086/8087 **Wait for x87**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fwait
```

XCHG

8086 **Exchange**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xchg rax, rbx
```

XLAT/XLATB

8086 **Table lookup**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xlatb
```

XOR

8086 **Bitwise XOR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xor eax, eax
```

80186 / 80188 - Stack and String Improvements

The 80186 generation added convenient stack, immediate, loop, and string-I/O forms. Some are still valid, but several are slow or invalid in 64-bit mode.

BOUND

80186 **Array bounds check**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
bound ax, [bounds]
```

ENTER

80186 **Create stack frame**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
enter 32, 0
```

LEAVE

80186 **Destroy stack frame**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
leave
```

INSB/INSW/INSD

80186/386 **Input string from port**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rep insb
```

OUTSB/OUTSW/OUTSD

80186/386 **Output string to port**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rep outsb
```

PUSHA/POPA

80186 **Push/pop all legacy GPRs**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
pusha
```

PUSH imm

80186 **Push immediate value**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
push 1234
```

IMUL imm

80186 **Signed multiply immediate**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
imul eax, ebx, 10
```

SHL/SAR/SHR r/m, imm8

80186 **Immediate count shifts**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
shl eax, 3
```

80286 - Protected Mode Foundations

The 80286 introduced protected-mode machinery: descriptor tables, selectors, privilege checks, and task-related instructions. Most of these belong to operating-system code, not ordinary user programs.

ARPL

80286 **Adjust RPL**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
arpl ax, bx
```

CLTS

80286 **Clear task-switched flag**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
clts
```

LAR

80286 **Load access rights**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lar eax, bx
```

LGDT

80286 **Load GDTR**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
lgdt [gdt_descriptor]
```

LIDT

80286 **Load IDTR**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
lidt [idt_descriptor]
```

LLDT

80286 **Load LDTR**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
lldt ax
```

LMSW

80286 **Load machine status word**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
lmsw ax
```

LSL

80286 **Load segment limit**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lsl eax, bx
```

LTR

80286 **Load task register**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
ltr ax
```

SGDT

80286 **Store GDTR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sgdt [buf]
```

SIDT

80286 **Store IDTR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sidt [buf]
```

SLDT

80286 **Store LDTR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sldt ax
```

SMSW

80286 **Store machine status word**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
smsw ax
```

STR

80286 **Store task register**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
str ax
```

VERR/VERW

80286 **Verify segment for read/write**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
verr ax
verw ax
```

80386 - 32-bit, Paging, and Extended Registers

The 80386 was the major 32-bit transition: EAX through EDI, paging, bit-test instructions, sign/zero extension moves, SETcc, and control/debug registers.

BSF

80386 **Bit scan forward**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bsf eax, ebx
```

BSR

80386 **Bit scan reverse**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bsr eax, ebx
```

BT

80386 **Bit test**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bt rax, 5
```

BTC

80386 **Bit test and complement**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
btc qword [flags], 3
```

BTR

80386 **Bit test and reset**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
btr qword [flags], 3
```

BTS

80386 **Bit test and set**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bts qword [flags], 3
```

CDQ

80386 **Sign extend EAX into EDX:EAX**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cdq
idiv ebx
```

CWDE

80386 **Sign extend AX to EAX**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cwde
```

IRETD

80386 **Return from interrupt 32-bit**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
iretd
```

JECXZ

80386 **Jump if ECX zero**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
jecxz done
```

LFS/LGS/LSS

80386 **Load far pointer into FS/GS/SS**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lfs eax, [ptr]
```

MOV CRx/DRx

80386 **Move control/debug registers**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
mov rax, cr3
```

MOVSX

80386 **Move with sign extension**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movsx eax, byte [p]
```

MOVZX

80386 **Move with zero extension**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movzx eax, byte [p]
```

POPAD/PUSHAD

80386 **Push/pop all 32-bit GPRs**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
pushad
```

SETcc

80386 **Set byte on condition**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
cmp eax, ebx
setg al
```

SHLD

80386 **Double precision shift left**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
shld rax, rdx, 4
```

SHRD

80386 **Double precision shift right**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
shrd rax, rdx, 4
```

80486 and Early Pentium - Atomicity and Cache Control

This period introduced core primitives for atomic programming, byte swapping, cache/TLB management, feature discovery, and early timing facilities.

BSWAP

80486 **Byte swap**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bswap eax
```

CMPXCHG

80486 **Compare and exchange**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
lock cmpxchg [ptr], rbx
```

INVD

80486 **Invalidate cache**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
invd
```

INVLPG

80486 **Invalidate TLB entry**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
invlpg [rax]
```

WBINVD

80486 **Write back and invalidate cache**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
wbinvd
```

XADD

80486 **Exchange and add**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
lock xadd [counter], rax
```

CPUID

486/Pentium **CPU identification**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mov eax, 1
cpuid
```

CMPXCHG8B

Pentium **64-bit compare exchange**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
lock cmpxchg8b [mem64]
```

RDTSC

Pentium **Read timestamp counter**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rdtsc
```

RDMSR/WRMSR

Pentium **Read/write model-specific register**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
rdmsr
```

P6 / Pentium Pro - Conditional Moves and x87 Flag Comparisons

P6 brought conditional moves and better flag-producing x87 comparisons. CMOV is useful when a branch would be expensive or unpredictable.

CMOVcc

P6 **Conditional move**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
cmp rax, rbx
cmovl rax, rbx
```

FCMOVcc

P6 x87 **Conditional x87 move**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fcmovb st0, st1
```

FCOMI/FUCOMI

P6 x87 **Compare x87 and set flags**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fcomi st0, st1
```

UD2

P6 **Undefined instruction**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
ud2
```

SYSENTER/SYSEXIT

Pentium II+ **Fast system call legacy**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
sysenter
```

x87 FPU - The Historical Floating-Point Stack

x87 uses a register stack, ST0 through ST7, with extended precision. Modern x86-64 code often prefers SSE2/AVX, but x87 remains important for compatibility and legacy systems.

FLD

8087 **Load floating value**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fld qword [x]
```

FST/FSTP

8087 **Store floating value**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fstp qword [out]
```

FADD/FADDP

8087 **Floating add**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fld qword [a]
fadd qword [b]
```

FSUB/FSUBP

8087 **Floating subtract**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fsub qword [b]
```

FMUL/FMULP

8087 **Floating multiply**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fmul qword [b]
```

FDIV/FDIVP

8087 **Floating divide**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fdiv qword [b]
```

FSQRT

8087 **Square root**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fsqrt
```

FSIN/FCOS/FSINCOS

80387 **Trig functions**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fsincos
```

FPTAN/FPATAN

8087 **Tangent/arctangent**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fptan
```

FLDENV/FNSTENV

8087 **Load/store x87 environment**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fnstenv [buf]
```

FXSAVE/FXRSTOR

Pentium II/SSE **Save/restore FP/SIMD state**

Useful for legacy compatibility and learning. For most modern numeric code, SSE2/AVX or compiler-generated code is easier to maintain.

```
fxsave [area]
```

XSAVE/XRSTOR

Sandy Bridge era **Extended state save/restore**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xsave [area]
```

MMX - 64-bit Integer SIMD

MMX introduced packed integer SIMD in MM0-MM7, aliased with the x87 register file. It is mostly historical today; prefer SSE2/AVX2 for new code.

EMMS

MMX **Empty MMX state**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
emms
```

MOVD/MOVQ

MMX **Move 32/64-bit data**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
movq mm0, [src]
```

PACKSSWB/PACKSSDW

MMX **Signed saturating pack**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
packsswb mm0, mm1
```

PACKUSWB

MMX **Unsigned saturating pack**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
packuswb mm0, mm1
```

PADDB/PADDW/PADDD

MMX **Packed add**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
paddw mm0, mm1
```

PADDUSB/PADDUSW

MMX **Unsigned saturating add**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
paddusb mm0, mm1
```

PADDSB/PADDSW

MMX **Signed saturating add**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
paddsw mm0, mm1
```

PAND/PANDN/POR/PXOR

MMX **Packed bitwise logic**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pxor mm0, mm0
```

PCMPEQB/W/D

MMX **Packed equal compare**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pcmpeqb mm0, mm1
```

PCMPGTB/W/D

MMX **Packed greater compare**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pcmpgtw mm0, mm1
```

PMADDWD

MMX **Multiply add words**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pmaddwd mm0, mm1
```

PMULHW/PMULLW

MMX **Packed multiply words**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pmullw mm0, mm1
```

PSLLW/D/Q

MMX **Packed shift left**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
pslld mm0, 4
```

PSRAW/D

MMX **Packed arithmetic shift right**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
psraw mm0, 1
```

PSRLW/D/Q

MMX **Packed logical shift right**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
psrld mm0, 1
```

PSUBB/W/D

MMX **Packed subtract**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
psubb mm0, mm1
```

PUNPCKHBW/HWD/HDQ

MMX **Unpack high**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
punpckhbw mm0, mm1
```

PUNPCKLBW/LWD/LDQ

MMX **Unpack low**

Historical SIMD. Because MMX aliases x87 state, use EMMS and prefer SSE2/AVX2 in new code.

```
punpcklbw mm0, mm1
```

SSE - XMM Registers and 128-bit Floating Point

SSE introduced XMM registers and the modern SIMD programming style. It also introduced MXCSR and important cache/prefetch hints.

ADDPS/ADDSS

SSE **Packed/scalar FP32 add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
addps xmm0, xmm1
addss xmm2, xmm3
```

SUBPS/SUBSS

SSE **Packed/scalar FP32 subtract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
subps xmm0, xmm1
```

MULPS/MULSS

SSE **Packed/scalar FP32 multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mulps xmm0, xmm1
```

DIVPS/DIVSS

SSE **Packed/scalar FP32 divide**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
divss xmm0, xmm1
```

SQRTPS/SQRTSS

SSE **Square root FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sqrtps xmm0, xmm1
```

RCPPS/RCPSS

SSE **Approx reciprocal**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rcpps xmm0, xmm1
```

RSQRTPS/RSQRTSS

SSE **Approx reciprocal sqrt**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rsqrtps xmm0, xmm1
```

MAXPS/MAXSS

SSE **Maximum FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
maxps xmm0, xmm1
```

MINPS/MINSS

SSE **Minimum FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
minps xmm0, xmm1
```

CMPPS/CMPSS

SSE **Compare FP32**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cmpps xmm0, xmm1, 0
```

COMISS/UCOMISS

SSE **Scalar FP32 compare to flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
ucomiss xmm0, xmm1
```

MOVAPS/MOVUPS

SSE **Aligned/unaligned move**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movups xmm0, [rsi]
```

MOVSS

SSE **Move scalar FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movss xmm0, [x]
```

MOVHPS/MOVLPS/MOVHLPS/MOVLHPS

SSE **Move high/low pairs**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movhlps xmm0, xmm1
```

MOVMSKPS

SSE **Extract sign mask**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movmskps eax, xmm0
```

ANDPS/ANDNPS/ORPS/XORPS

SSE **Bitwise logic on FP vectors**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xorps xmm0, xmm0
```

SHUFPS

SSE **Shuffle FP32 lanes**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
shufps xmm0, xmm1, 0b11100100
```

UNPCKHPS/UNPCKLPS

SSE **Unpack high/low FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
unpcklps xmm0, xmm1
```

LDMXCSR/STMXCSR

SSE **Load/store MXCSR**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
stmxcsr [mxcsr_save]
```

PREFETCHT0/T1/T2/NTA

SSE **Prefetch cache hints**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
prefetcht0 [rsi]
```

SFENCE

SSE **Store fence**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sfence
```

SSE2 - The Practical Baseline of x86-64

SSE2 extended SIMD to double precision and integer operations in XMM registers. It is a practical baseline for 64-bit Intel-compatible software.

ADDPD/ADDSD

SSE2 **Packed/scalar FP64 add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
addpd xmm0, xmm1
addsd xmm2, xmm3
```

SUBPD/SUBSD

SSE2 **FP64 subtract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
subsd xmm0, xmm1
```

MULPD/MULSD

SSE2 **FP64 multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mulpd xmm0, xmm1
```

DIVPD/DIVSD

SSE2 **FP64 divide**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
divsd xmm0, xmm1
```

SQRTPD/SQRTSD

SSE2 **FP64 sqrt**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sqrtsd xmm0, xmm1
```

MAXPD/MAXSD

SSE2 **FP64 max**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
maxsd xmm0, xmm1
```

MINPD/MINSD

SSE2 **FP64 min**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
minpd xmm0, xmm1
```

CMPPD/CMPSD

SSE2 **FP64 compare**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
cmppd xmm0, xmm1, 0
```

COMISD/UCOMISD

SSE2 **Scalar FP64 compare to flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
ucomisd xmm0, xmm1
```

MOVAPD/MOVUPD

SSE2 **Aligned/unaligned FP64 move**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movupd xmm0, [rsi]
```

MOVDQA/MOVDQU

SSE2 **Aligned/unaligned integer move**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movdqu xmm0, [rsi]
```

MOVSD

SSE2 **Move scalar double**

String and memory-scan/copy instruction family. Direction Flag (DF) and RCX/RSI/RDI behavior are essential.

```
movsd xmm0, [x]
```

MOVMSKPD

SSE2 **Extract sign mask FP64**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movmskpd eax, xmm0
```

MOVNTDQ/MOVNTPD/MOVNTI

SSE2 **Non-temporal stores**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movntdq [rdi], xmm0
```

ANDPD/ANDNPD/ORPD/XORPD

SSE2 **Bitwise logic on FP64 vectors**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xorpd xmm0, xmm0
```

PADDQ

SSE2 **Packed qword add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
paddq xmm0, xmm1
```

PSHUFD/PSHUFHW/PSHUFLW

SSE2 **Shuffle integer lanes**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pshufd xmm0, xmm1, 0b00011011
```

PSLLDQ/PSRLDQ

SSE2 **Shift whole XMM bytes**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pslldq xmm0, 4
```

PMULUDQ

SSE2 **Unsigned dword multiply to qword**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmuludq xmm0, xmm1
```

PUNPCKHQDQ/PUNPCKLQDQ

SSE2 **Unpack quadwords**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
punpcklqdq xmm0, xmm1
```

CLFLUSH

SSE2-era **Flush cache line**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
clflush [rdi]
```

LFENCE/MFENCE

SSE2 **Load/full fence**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mfence
```

PAUSE

SSE2-era **Spin-wait hint**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
.spin:
pause
jnz .spin
```

SSE3 / SSSE3 - Horizontal Operations and Shuffle Power

SSE3 and SSSE3 added horizontal arithmetic, byte shuffles, sign operations, alignment helpers, and media-friendly transformations.

ADDSUBPD/ADDSUBPS

SSE3 **Alternating add/sub**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
addsubps xmm0, xmm1
```

HADDPD/HADDPS

SSE3 **Horizontal add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
haddps xmm0, xmm1
```

HSUBPD/HSUBPS

SSE3 **Horizontal subtract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
hsubpd xmm0, xmm1
```

LDDQU

SSE3 **Unaligned load hint**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lddqu xmm0, [rsi]
```

MOVDDUP

SSE3 **Duplicate low double**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movddup xmm0, xmm1
```

MOVSHDUP/MOVSLDUP

SSE3 **Duplicate high/low FP32**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movshdup xmm0, xmm1
```

MONITOR/MWAIT

SSE3 **Monitor address / wait**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
monitor
mwait
```

PABSB/PABSW/PABSD

SSSE3 **Packed absolute value**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pabsd xmm0, xmm1
```

PALIGNR

SSSE3 **Packed align right**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
palignr xmm0, xmm1, 8
```

PHADDW/PHADDD/PHADDSW

SSSE3 **Packed horizontal add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
phaddw xmm0, xmm1
```

PHSUBW/PHSUBD/PHSUBSW

SSSE3 **Packed horizontal subtract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
phsubd xmm0, xmm1
```

PMADDUBSW

SSSE3 **Multiply unsigned/signed bytes add**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmaddubsw xmm0, xmm1
```

PMULHRSW

SSSE3 **Rounded high multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmulhrsw xmm0, xmm1
```

PSHUFB

SSSE3 **Byte shuffle**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pshufb xmm0, xmm1
```

PSIGNB/PSIGNW/PSIGND

SSSE3 **Packed sign**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
psignd xmm0, xmm1
```

SSE4.1 / SSE4.2 - Text, CRC, Blend, and Media Operations

SSE4.1 and SSE4.2 added blending, insert/extract, dot product, string comparison, CRC32, and other practical data-processing instructions.

BLENDPD/BLENDPS

SSE4.1 **Blend lanes by immediate**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
blendps xmm0, xmm1, 0b1010
```

BLENDVPD/BLENDVPS

SSE4.1 **Blend lanes by mask**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
blendvps xmm0, xmm1
```

DPPD/DPPS

SSE4.1 **Dot product**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
dpps xmm0, xmm1, 0xff
```

EXTRACTPS

SSE4.1 **Extract FP32 lane**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
extractps eax, xmm0, 2
```

INSERTPS

SSE4.1 **Insert FP32 lane**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
insertps xmm0, xmm1, 0x10
```

MOVNTDQA

SSE4.1 **Streaming load**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movntdqa xmm0, [rsi]
```

MPSADBW

SSE4.1 **Sum absolute differences**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mpsadbw xmm0, xmm1, 0
```

PACKUSDW

SSE4.1 **Pack signed dwords to unsigned words**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
packusdw xmm0, xmm1
```

PBLENDVB/PBLENDW

SSE4.1 **Blend integer lanes**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pblendw xmm0, xmm1, 0b01010101
```

PCMPEQQ

SSE4.1 **Compare qword equal**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pcmpeqq xmm0, xmm1
```

PEXTRB/PEXTRD/PEXTRQ/PEXTRW

SSE4.1 **Extract integer element**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pextrd eax, xmm0, 1
```

PHMINPOSUW

SSE4.1 **Horizontal min unsigned word**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
phminposuw xmm0, xmm1
```

PINSRB/PINSRD/PINSRQ

SSE4.1 **Insert integer element**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pinsrd xmm0, eax, 2
```

PMAXSB/PMAXSD/PMAXUW/PMAXUD

SSE4.1 **Packed maximum**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmaxsd xmm0, xmm1
```

PMINSB/PMINSD/PMINUW/PMINUD

SSE4.1 **Packed minimum**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pminuw xmm0, xmm1
```

PMOVSX*

SSE4.1 **Packed sign extension**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmovsxbd xmm0, [rsi]
```

PMOVZX*

SSE4.1 **Packed zero extension**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmovzxbd xmm0, [rsi]
```

PMULDQ/PMULLD

SSE4.1 **Packed dword multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pmulld xmm0, xmm1
```

PTEST

SSE4.1 **Packed bit test**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
ptest xmm0, xmm1
```

ROUNDPS/ROUNDPD/ROUNDSS/ROUNDSD

SSE4.1 **Rounding**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
roundps xmm0, xmm1, 0
```

CRC32

SSE4.2 **CRC32C update**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
crc32 eax, byte [rsi]
```

PCMPESTRI/PCMPESTRM

SSE4.2 **Explicit length string compare**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pcmpestri xmm0, xmm1, 0
```

PCMPISTRI/PCMPISTRM

SSE4.2 **Implicit length string compare**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pcmpistri xmm0, xmm1, 0
```

PCMPGTQ

SSE4.2 **Packed qword greater compare**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pcmpgtq xmm0, xmm1
```

AES-NI / PCLMUL / SHA - Cryptographic Instructions

These instructions accelerate common cryptographic primitives. They must still be used through correct algorithms, modes, padding, authentication, and side-channel-aware libraries.

AESENC

AES-NI **AES round encrypt**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aesenc xmm0, xmm1
```

AESENCLAST

AES-NI **AES final encrypt round**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aesenclast xmm0, xmm1
```

AESDEC

AES-NI **AES round decrypt**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aesdec xmm0, xmm1
```

AESDECLAST

AES-NI **AES final decrypt round**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aesdeclast xmm0, xmm1
```

AESIMC

AES-NI **Inverse mix columns**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aesimc xmm0, xmm1
```

AESKEYGENASSIST

AES-NI **Assist key expansion**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
aeskeygenassist xmm1, xmm0, 0x1
```

PCLMULQDQ

PCLMULQDQ **Carry-less multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pclmulqdq xmm0, xmm1, 0x00
```

SHA1MSG1/SHA1MSG2

SHA ext **SHA-1 message schedule**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sha1msg1 xmm0, xmm1
```

SHA1NEXTE/SHA1RNDS4

SHA ext **SHA-1 rounds**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sha1rnds4 xmm0, xmm1, 0
```

SHA256MSG1/SHA256MSG2

SHA ext **SHA-256 schedule**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sha256msg1 xmm0, xmm1
```

SHA256RNDS2

SHA ext **SHA-256 rounds**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
sha256rnds2 xmm0, xmm1
```

RDRAND

Ivy Bridge era **Hardware random**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
.retry:
rdrand rax
jnc .retry
```

RDSEED

Broadwell era **Seed random**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
.retry:
rdseed rax
jnc .retry
```

AVX / FMA / F16C - VEX Encoding and 256-bit SIMD

AVX introduced VEX encodings, three-operand forms, and 256-bit YMM registers. FMA and F16C widened the numerical and conversion toolbox.

VADDPS/VADDPD

AVX **Packed FP add**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vaddps ymm0, ymm1, ymm2
```

VADDSS/VADDSD

AVX **Scalar FP add**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vaddsd xmm0, xmm1, xmm2
```

VSUB*/VMUL*/VDIV*

AVX **FP arithmetic families**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vmulps ymm0, ymm1, ymm2
```

VMOVAPS/VMOVUPS/VMOVAPD/VMOVUPD

AVX **Vector moves**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vmovups ymm0, [rsi]
```

VBROADCASTSS/SD/F128

AVX **Broadcast scalar/block**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vbroadcastss ymm0, [x]
```

VINSERTF128/VEXTRACTF128

AVX **Insert/extract 128-bit lane**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vinsertf128 ymm0, ymm1, xmm2, 1
```

VPERMILPS/VPERMILPD

AVX **Permute lanes**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpermilps ymm0, ymm1, 0b10110001
```

VTESTPS/VTESTPD

AVX **Vector bit test**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vtestps ymm0, ymm1
```

VZEROUPPER

AVX **Zero upper YMM bits**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vzeroupper
```

VZEROALL

AVX **Zero all vector regs**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vzeroall
```

VCVTPH2PS

F16C **FP16 to FP32**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vcvtph2ps ymm0, xmm1
```

VCVTPS2PH

F16C **FP32 to FP16**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vcvtps2ph xmm0, ymm1, 0
```

VFMADD132PS/213PS/231PS

FMA3 **Fused multiply-add**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vfmadd231ps ymm0, ymm1, ymm2
```

VFMSUB*/VFNMADD*/VFNMSUB*

FMA3 **FMA variants**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vfmsub231pd ymm0, ymm1, ymm2
```

AVX2 / BMI / ADX - Integer SIMD and Bit Manipulation

AVX2 brought 256-bit integer SIMD and gather operations. BMI1/BMI2 and ADX added powerful scalar bit-manipulation and multi-precision arithmetic primitives.

VPBROADCASTB/W/D/Q

AVX2 **Broadcast integer**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpbroadcastd ymm0, eax
```

VINSERTI128/VEXTRACTI128

AVX2 **Insert/extract integer lane**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vinserti128 ymm0, ymm1, xmm2, 1
```

VPERM2I128

AVX2 **Permute 128-bit lanes**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vperm2i128 ymm0, ymm1, ymm2, 0x31
```

VPERMD/VPERMPS

AVX2 **Permute dword/float lanes**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpermd ymm0, ymm1, ymm2
```

VPGATHERDD/DQ/QD/QQ

AVX2 **Integer gather**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpgatherdd ymm0, [rdi+ymm1*4], ymm2
```

VGATHERDPS/VGATHERQPS

AVX2 **FP gather**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vgatherdps ymm0, [rdi+ymm1*4], ymm2
```

VPSLLVD/VPSLLVQ

AVX2 **Variable shift left**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpsllvd ymm0, ymm1, ymm2
```

VPSRLVD/VPSRLVQ

AVX2 **Variable logical shift right**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpsrlvd ymm0, ymm1, ymm2
```

VPSRAVD

AVX2 **Variable arithmetic shift right**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpsravd ymm0, ymm1, ymm2
```

VPBLENDD

AVX2 **Blend dwords**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpblendd ymm0, ymm1, ymm2, 0b10101010
```

VPMASKMOVD/Q

AVX2 **Masked load/store**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpmaskmovd ymm0, ymm1, [rsi]
```

ANDN

BMI1 **Bitwise AND NOT**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
andn rax, rbx, rcx
```

BEXTR

BMI1 **Bit field extract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bextr rax, rbx, rcx
```

BLSI/BLSMSK/BLSR

BMI1 **Lowest set bit helpers**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
blsi rax, rbx
```

TZCNT

BMI1 **Count trailing zeros**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
tzcnt rax, rbx
```

LZCNT

ABM/LZCNT **Count leading zeros**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lzcnt rax, rbx
```

MULX

BMI2 **Unsigned multiply without flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mulx rdx, rax, rbx
```

PDEP/PEXT

BMI2 **Parallel bit deposit/extract**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
pext rax, rbx, rcx
```

RORX

BMI2 **Rotate right without flags**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rorx rax, rbx, 13
```

BZHI

BMI2 **Zero high bits**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bzhi rax, rbx, rcx
```

ADCX/ADOX

ADX **Dual carry chains**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
adcx rax, rbx
adox rcx, rdx
```

CLAC/STAC

SMAP **Clear/set AC flag**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
stac
; copy user
clac
```

Intel 64 / x86-64 - 64-bit Registers and SYSCALL

Intel 64 adds 64-bit general-purpose registers, RIP-relative addressing, canonical addresses, SYSCALL/SYSRET, and long-mode restrictions on old instructions.

SYSCALL

Intel 64 **Fast system call**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
mov rax, 60
xor rdi, rdi
syscall
```

SYSRET

Intel 64 **Return from system call**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
sysret
```

SWAPGS

Intel 64 **Swap GS base**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
swapgs
```

MOVSXD

Intel 64 **Sign extend dword to qword**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
movsxd rax, dword [rsi]
```

CDQE/CQO

Intel 64 **Sign extension forms**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cdqe
cqo
```

CMPXCHG16B

Intel 64 era **128-bit compare exchange**

Important for atomic or multi-precision code. Be precise about flags, memory ordering, and the ABI.

```
lock cmpxchg16b [mem128]
```

RIP-relative addressing

Intel 64 **Position-independent data access**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
lea rdi, [rel message]
```

REX prefix

Intel 64 **Register/operand extension**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mov r8, rax
```

AVX-512 / EVEX - ZMM Registers, Masks, and Many Families

AVX-512 is not a single uniform feature. It is a collection of families with ZMM registers, opmask registers, EVEX encodings, and different availability across Intel products.

KADD/KAND/KOR/KXOR/KNOT

AVX-512F/BW **Mask logic**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
kxorw k1, k1, k1
```

KMOVB/W/D/Q

AVX-512F/BW/DQ **Move mask**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
kmovw k1, eax
```

VADDPS/VADDPD {k}{z}

AVX-512F **Masked vector add**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vaddps zmm0 {k1}{z}, zmm1, zmm2
```

VCOMPRESSPS/PD

AVX-512F **Compress lanes**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vcompressps [rdi]{k1}, zmm0
```

VEXPANDPS/PD

AVX-512F **Expand lanes**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vexpandps zmm0{k1}{z}, [rsi]
```

VGATHER*/VSCATTER*

AVX-512F/DQ **Gather/scatter**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vgatherdps zmm0{k1}, [rdi+zmm1*4]
```

VPTERNLOGD/Q

AVX-512F **Ternary boolean logic**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpternlogd zmm0, zmm1, zmm2, 0x96
```

VPERMT2*/VPERMI2*

AVX-512F/VBMI **Two-source permute**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpermt2d zmm0, zmm1, zmm2
```

VPMOV*

AVX-512F/BW/DQ **Vector convert/narrow/widen**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpmovdb [rdi]{k1}, zmm0
```

VPMULLQ

AVX-512DQ **Packed qword multiply**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpmullq zmm0, zmm1, zmm2
```

VPMADD52LUQ/HUQ

AVX-512IFMA **52-bit integer FMA**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpmadd52luq zmm0, zmm1, zmm2
```

VPDPBUSD/VPDPWSSD

AVX-512VNNI **Dot-product integer**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vpdpbusd zmm0, zmm1, zmm2
```

VCVTNEPS2BF16/VCVTNE2PS2BF16

AVX-512 BF16 **Convert FP32 to BF16**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vcvtneps2bf16 ymm0, zmm1
```

VP2INTERSECTD/Q

AVX-512 VP2INTERSECT **Set intersection**

Requires feature detection. Check CPUID, OSXSAVE/XGETBV state support, assembler support, and the target processor family.

```
vp2intersectd k1, zmm0, zmm1
```

VAESENC/VAESDEC

VAES **Vector AES**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
vaesenc zmm0, zmm1, zmm2
```

VPCLMULQDQ

VPCLMULQDQ **Vector carry-less multiply**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
vpclmulqdq zmm0, zmm1, zmm2, 0
```

System, Virtualization, Protection, and Cache Instructions

These instructions belong mainly to kernels, hypervisors, firmware, runtimes, and performance-sensitive system code. Many are privileged and will fault in user mode.

CPUID

Pentium+ **Feature discovery**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
mov eax, 7
xor ecx, ecx
cpuid
```

XGETBV/XSETBV

XSAVE **Read/write XCR registers**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
xor ecx, ecx
xgetbv
```

RDPMC

Pentium Pro+ **Read performance counter**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rdpmc
```

RDPID

RDPID **Read processor ID**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rdpid rax
```

RDTSCP

RDTSCP **Read TSC ordered with AUX**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rdtscp
```

VMXON/VMXOFF

VMX **Enter/leave VMX operation**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
vmxon [vmxon_region]
```

VMLAUNCH/VMRESUME

VMX **Launch/resume VM**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
vmlaunch
```

VMREAD/VMWRITE

VMX **Access VMCS fields**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
vmread rax, rbx
```

VMCALL

VMX **Call hypervisor**

Control-flow related. Understand which flags are read and whether the comparison is signed or unsigned.

```
vmcall
```

INVEPT/INVVPID

VMX **Invalidate EPT/VPID translations**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
invept rax, [desc]
```

GETSEC

SMX **Safer mode extensions**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
getsec
```

ENCLS/ENCLU/ENCLV

SGX **SGX enclave instructions**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
enclu
```

ENDBR32/ENDBR64

CET IBT **Indirect branch target**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
endbr64
```

WRSS/WRUSS

CET shadow stack **Write shadow stack**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
wrssq [rdi], rax
```

RSTORSSP/SAVEPREVSSP

CET shadow stack **Restore/save shadow stack pointer**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
rstorssp [rdi]
```

CLDEMOTE

Cache hint **Demote cache line**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
cldemote [rdi]
```

CLFLUSHOPT/CLWB

Cache mgmt **Optimized flush/writeback**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
clwb [rdi]
```

SERIALIZE

Serialization **Serialize instruction stream**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
serialize
```

UMONITOR/UMWAIT/TPAUSE

WAITPKG **User wait instructions**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
umonitor rax
umwait ecx, edx
```

AMX - Tiles and Matrix Operations for AI/HPC

AMX introduces tile registers and matrix-oriented operations aimed at AI and HPC. It requires operating-system support for extended state management.

LDTILECFG

AMX-TILE **Load tile configuration**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
ldtilecfg [tilecfg]
```

STTILECFG

AMX-TILE **Store tile configuration**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
sttilecfg [tilecfg]
```

TILELOADD

AMX-TILE **Load tile**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tileloadd tmm0, [rsi+rax]
```

TILELOADDT1

AMX-TILE **Load tile hint T1**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tileloaddt1 tmm0, [rsi+rax]
```

TILESTORED

AMX-TILE **Store tile**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
tilestored [rdi+rax], tmm0
```

TILERELEASE

AMX-TILE **Release tile state**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tilerelease
```

TILEZERO

AMX-TILE **Zero tile**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tilezero tmm0
```

TDPBSSD

AMX-INT8 **Dot product signed/signed bytes**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpbssd tmm0, tmm1, tmm2
```

TDPBSUD

AMX-INT8 **Dot product signed/unsigned**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpbsud tmm0, tmm1, tmm2
```

TDPBUSD

AMX-INT8 **Dot product unsigned/signed**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpbusd tmm0, tmm1, tmm2
```

TDPBUUD

AMX-INT8 **Dot product unsigned/unsigned**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpbuud tmm0, tmm1, tmm2
```

TDPBF16PS

AMX-BF16 **BF16 dot product to FP32**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpbf16ps tmm0, tmm1, tmm2
```

TDPFP16PS

AMX-FP16 **FP16 dot product to FP32**

Requires AMX feature bits and OS support for tile state. Usually used through tuned libraries or compiler intrinsics, not handwritten beginner code.

```
tdpfp16ps tmm0, tmm1, tmm2
```

Intel APX and AVX10 - Modern/Future Features to Verify Carefully

APX and AVX10 represent newer directions in the Intel ISA. Treat them as feature-dependent: check the latest documents, toolchain support, and CPUID before relying on them.

R16-R31

APX **Extended GPRs**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
; conceptual example - verify support first
add r16, r17
```

NDD forms

APX **New data destination**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
; conceptual example - verify support first
```

NF forms

APX **No flags update**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
; conceptual example - verify support first
```

PUSH2/POP2

APX **Push/pop register pair**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
push2 r16, r17
```

JMPABS

APX **Absolute indirect jump form**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
jmpabs rax
```

CCMP/CTEST

APX **Conditional compare/test**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
ccmp rax, rbx
```

AVX10.1

AVX10 **Unified vector ISA direction**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
; conceptual example - verify support first
```

APX-extended BMI

APX + BMI **EVEX forms of BMI instructions**

Modern or future-oriented feature. Verify the exact APX/AVX10 specification revision, CPU support, OS/ABI readiness, and assembler encoding support.

```
blsr r16, r17
```

APX-extended VMX

APX + VMX **EVEX forms for selected system instr**

Mostly privileged or system-level. Study it to understand kernels, firmware, or hypervisors; do not expect it to run in user space.

```
; conceptual example - verify support first
```

Rare, Deprecated, or Undocumented Instructions

This section is deliberately cautionary. Undocumented or deprecated instructions are not a stable programming contract. Use documented Intel instructions and CPUID-based dispatch.

SALC

Undocumented historical **Set AL from CF**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
; conceptual example - verify support first
```

ICEBP / INT1

Debug/undocumented history **Debug trap**

System-call and interrupt related. The calling convention and validity depend on the operating system and execution mode.

```
db 0xF1 ; ICEBP/INT1
```

LOADALL

80286/80386 undocumented **Load internal state**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
; conceptual example - verify support first
```

AAM/AAD imm8 variants

8086 undocumented encodings **BCD adjust variants**

Legacy or invalid/restricted in 64-bit mode. Keep it for history, compatibility reading, or emulator work rather than new x86-64 code.

```
db 0xD4, 0x0A ; AAM 10
```

UD0/UD1

Undefined opcodes **Undefined instruction forms**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
ud2    ; see Intel SDM for exact constraints
```

MPX family

Intel MPX **Deprecated bounds checking**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
bndcl bnd0, [rsi]
```

TSX HLE prefixes

TSX/HLE **Hardware lock elision hints**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
xacquire lock add [mem], eax
```

3DNow!

AMD, not Intel focus **Legacy SIMD**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
; conceptual example - verify support first
```

Long mode invalid legacy

Intel 64 **Instructions invalid in 64-bit**

Core instruction family. Learn its operands, affected flags, addressing forms, and how NASM encodes the form you use.

```
; conceptual example - verify support first
```

Special Notes on Rare, Hidden, and Unsupported Instructions

Safe practical position

If an instruction is not documented by Intel, does not have a CPUID feature bit, or is not part of an official Intel specification, do not build production code on it. It may be a historical artifact, an internal opcode, or behavior that changes across processors or microcode updates.

When studying unusual instructions, separate them into four categories:

Category	Example	How to treat it
Documented but privileged	HLT, LGDT, MOV CR3, VMXON	Study for operating systems and hypervisors; do not execute in user space.
Documented but invalid/ restricted in 64-bit mode	PUSHA, POPA, BOUND, AAA/AAD/ AAM/AAS in certain contexts	Mention historically and avoid in modern x86-64 code.
Documented but deprecated or limited	MPX BND*, HLE hints	Check processor support, OS support, and current Intel guidance.
Historically undocumented	SALC, ICEBP, LOADALL	Do not depend on it. Use UD2 for a documented intentional trap.

Practical Appendices

Appendix A: Minimal CPUID Template in NASM

```
section .text
global _start
_start:
    mov eax, 1
    cpuid
    ; ECX/EDX now contain feature bits for leaf 1.

    mov eax, 7
    xor ecx, ecx
    cpuid
    ; EBX/ECX/EDX contain features such as AVX2/BMI/AVX-512,
    ; depending on the leaf/subleaf and the Intel SDM tables.

    mov eax, 60
    xor edi, edi
    syscall
```

Appendix B: Check OSXSAVE Before AVX

```
; General idea:
; 1) CPUID leaf 1: verify XSAVE, OSXSAVE, and AVX.
; 2) XGETBV XCR0: verify that the OS saves XMM/YMM state.
xor ecx, ecx
xgetbv
; Check bit 1 for XMM and bit 2 for YMM.
```

Appendix C: A Quick Table of Beginner-Essential Instructions

Area	Start with these instructions
Data movement	MOV, LEA, PUSH, POP, XCHG
Arithmetic	ADD, ADC, SUB, SBB, INC, DEC, NEG, IMUL, MUL, IDIV, DIV
Logic and bits	AND, OR, XOR, NOT, TEST, BT, BTS, BTR, BTC
Control flow	CMP, JMP, Jcc, CALL, RET, CMOVcc, SETcc
Strings	MOVSB/MOVSQ, STOSB/STOSQ, CMPSB, SCASB, REP
Linux system calls	SYSCALL with the Linux syscall-number table
Starter SIMD	MOVDQU/MOVDQA, ADDPS/ADDPD, PADDD, PXOR, PSHUFB
Modern SIMD	VADDPS, VMOVUPS, VPBROADCASTD, VPERMD, VZEROUPPER

Appendix D: Reading Mnemonic Prefix Patterns

Pattern	Typical meaning
V*	Often AVX/AVX2/AVX-512 using VEX or EVEX encodings.

Pattern	Typical meaning
P*	Often packed integer/SIMD, such as PADDD or PSHUFB.
K*	Often AVX-512 mask-register related.
TILE* / TDP*	Often AMX tile or matrix operations.
RD* / WR*	Usually read/write a register, counter, state, or model-specific register.
INV*	Often invalidates cache, TLB, or virtualization translations.

Closing Summary

To master x86-64 Assembly, do not memorize instructions as a dictionary. Connect every instruction to the registers it uses, the flags it changes, the addressing modes it accepts, and the architectural layer in which it appeared. x86 is not a small ISA; it is a stack of compatible layers.

Suggested learning plan

1) Master GPRs and RFLAGS. 2) Write small NASM programs using SYSCALL. 3) Learn the stack and ABI. 4) Learn addressing modes and LEA. 5) Move to SSE2, then AVX2. 6) Study AVX-512, AMX, and APX only after you understand CPUID, OSXSAVE, and OS constraints.

Intel SDM remains the decisive reference. This booklet is a compact and organized roadmap to help you know where to look, what to ask, and how to read instructions without getting lost.

Official and Trusted References

This educational index is based on the following official Intel references, with NASM used to simplify examples:

1. **Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
2. **Intel 64 and IA-32 Architectures SDM Volume 2A-2D - Instruction Set Reference A-Z**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
3. **Intel SDM Volume 1 - Basic Architecture**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
4. **Intel SDM Volume 3A-3D - System Programming Guide**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
5. **Intel SDM Volume 4 - Model-Specific Registers**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
6. **Intel Architecture Instruction Set Extensions and Future Features Programming Reference**
   https://cdrdv2-public.intel.com/865891/319433-059-architecture-instruction-set-extensions-programming-reference_2025-0930.pdf
7. **Intel Advanced Performance Extensions (Intel APX) Architecture Specification, Revision 7.0**
   https://cdrdv2-public.intel.com/861610/355828-007-intel-apx-spec.pdf
8. **Intel Software Development Emulator (Intel SDE)**
   https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html
9. **Intel Intrinsics Guide**
   https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
10. **Intel Core Ultra 9 Processor 185H Specifications**
    https://www.intel.com/content/www/us/en/products/sku/236849/intel-core-ultra-9-processor-185h-24m-cache-up-to-5-10-ghz/specifications.html
11. **Intel Core Ultra 9 Processor 285K Specifications**
    https://www.intel.com/content/www/us/en/products/sku/241060/intel-core-ultra-9-processor-285k-36m-cache-up-to-5-70-ghz/specifications.html
12. **NASM Documentation**
    https://www.nasm.us/doc/

```
Important: Intel documentation changes over time. Always download the latest Intel SDM and
extension references before writing production code that depends on a specific instruction or
CPUID bit.
```