Microprocessors and Instruction Sets

80286 Microprocessor 1
Real-Address Mode 1
Protected Virtual Address Mode 1
80287 Math Coprocessor 2
Programming Interface 2
Hardware Interface 3
80386 Microprocessor 4
Real Address Mode 5
Protected Virtual Address Mode 5
Virtual 8086 Mode 6
80386 Paging Mechanism 7
80387 Math Coprocessor 9
80387 To 80486 Math Coprocessor Compatibility 10
Programming Interface 10
Hardware Interface 11
80486 Microprocessor 13
Cache Control 13
Cache Paging Control 14
Page Protection Feature 15
New Alignment Check 16
New Instructions 16
80286 Microprocessor Instruction Set 17
Data Transfer 17
Arithmetic 21
Logic 25
String Manipulation 27
Control Transfer 29
Processor Control 33
Protection Control 35
80287 Math Coprocessor Instruction Set 38
Data Transfer 38
Comparison 40
Constants 41
Arithmetic 42
Transcendental 43
Processor Control 44
Introduction to the 80386 Instruction Set 45
Code and Data Segment Descriptors 46
Prefixes 47
Instruction Format 48
Encoding 50
Address Mode 50

Operand Length (w) Field 53
Segment Register (sreg) Field 54
General Register (reg) Field 54
Operation Direction (d) Field 55
Sign-Extend (s) Field 55
Conditional Test (tttn) Field 55
Control, Debug, or Test Register (eee) Field 56
80386 Microprocessor Instruction Set 57
Data Transfer 57
Segment Control 60
Flag Control 61
Arithmetic 62
Logic 67
String Manipulation 71
Repeated String Manipulation 72
Bit Manipulation 74
Control Transfer 75
Conditional Jumps 76
Conditional Byte Set 81
Interrupt Instructions 83
Processor Control 84
Processor Extension 85
Prefix Bytes 85
Protection Control 86
Introduction to the 80387 Instruction Set 89
80387 Usage of the Scale-Index-Base Byte 89
Instruction and Data Pointers 89
New Instructions 92
80387 Math Coprocessor Instruction Set 93
Data Transfer 93
Comparison 94
Constants 95
Arithmetic 96
Transcendental 98
Processor Control 98
80486 Microprocessor Instruction Set 100

Figures

1. 80287 Data Types . . . 3
2. 80386 Addressing . . . 6
3. Paging Mechanism . . . 8
4. Data Type Classifications and Instructions . . . 9
5. 80387 Data Types . . . 11
6. Control Register 0 . . . 13
7. 80386 Compatible Operation . . . 15
8. 80486 Protection Operation . . . 15
9. 2-Bit Register Field . . . 37
10. 3-Bit Register Field . . . 37
11. 80287 Encoding Field Summary . . . 38
12. 80386 Code and Data Segment Descriptor Format . . . 46
13. Instruction Format . . . 48
14. 80386 Instruction Set Encoding Field Summary . . . 49
15. Effective Address (16-Bit and 32-Bit Address Modes) . . . 50
16. Scale Factor (s-i-b Byte Present) . . . 51
17. Index Registers (s-i-b Byte Present) . . . 51
18. Base Registers (s-i-b Byte Present) . . . 52
19. Effective Address (32-Bit Address Mode — s-i-b Byte Present) . . . 53
20. Operand Length Field Encoding . . . 53
21. Segment Register Field Encoding . . . 54
22. General Register Field Encoding . . . 54
23. Operand Direction Field Encoding . . . 55
24. Sign-Extend Field Encoding . . . 55
25. Conditional Test Field Encoding . . . 56
26. Control, Debug, and Test Register Field Encoding . . . 56
27. 80387 Encoding Field Summary . . . 89
28. Instruction and Pointer Image (16-Bit Real Address Mode) 90
29. Instruction and Pointer Image (16-Bit Protected Mode) . . . 91
30. Instruction and Pointer Image (32-Bit Real Address Mode) 91
31. Instruction and Pointer Image (32-Bit Protected Mode) . . . 91

Notes:

80286 Microprocessor

The 80286 microprocessor subsystem has the following:

- 24-bit address
- 16-bit data interface
- Extensive instruction set, including string I/O
- Hardware fixed-point multiply and divide
- Two operational modes:
  - 8086-compatible Real Address
  - Protected Virtual Address.
- 16MB (MB equals 1,048,576 or 2^{20} bytes) of physical address space
- 1GB (GB equals 1,073,741,824 or 2^{30} bytes) of virtual address space.

Real-Address Mode

In the real-address mode, the address space of the system microprocessor is a contiguous array of up to 1MB. The system microprocessor generates 20-bit physical addresses to address memory.

The segment portion of the pointer is interpreted as the upper 16 bits of a 20-bit segment address; the lower 4 bits are always 0. Therefore, segment addresses begin on multiples of 16 bytes.

All segments in the real-address mode are 64KB (KB equals 1024 bytes) and can be read, written, or executed. An exception or interrupt can occur if data operands or instructions attempt to wrap around the end of a segment (for example, a word with its low-order byte at offset hex FFFF and its high-order byte at hex 0000). If, in the real-address mode, the information contained in the segment does not use the full 64KB, the unused end of the segment can be overlaid by another segment to reduce physical memory requirements.

Protected Virtual Address Mode

The protected virtual address mode (hereafter called protected mode) offers extended physical and virtual memory address space, memory protection mechanisms, and new operations to support operating systems and virtual memory.

The protected mode provides a virtual address space of 1GB for each task mapped into a 16MB physical address space. The virtual address space may be larger than the physical address space, because any use of an address that does not map to a physical memory location will cause a restartable exception.

Like the real-address mode, the protected mode uses 32-bit pointers, consisting of 16-bit selector and offset components. The selector specifies an index into a memory-resident table rather than the upper 16 bits of a real address. The 24-bit base address of the desired segment is obtained from a table in memory. The 16-bit offset is added to the segment base address to form the physical address. The system microprocessor automatically refers to the tables whenever a segment register is loaded with a selector. All instructions that load a segment register refer to the table without additional program support. Each entry in a table is 8-bytes wide.

80287 Math Coprocessor

The optional 80287 Math Coprocessor enables the system to perform high-speed arithmetic, logarithmic, and trigonometric operations. The coprocessor works in parallel with the microprocessor. The parallel operation decreases operating time by allowing the coprocessor to do mathematical calculations while the microprocessor continues to do other functions.

The coprocessor works with seven numeric data types, which are divided into the following three classes:

- Binary integers (three types)
- Decimal integers (one type)
- Real numbers (three types).

Programming Interface

The coprocessor offers extended data types, registers, and instructions to the microprocessor. The coprocessor has eight 80-bit registers, which provide the equivalent capacity of forty 16-bit registers. This register space allows constants and temporary results to be held in registers during calculations, thus reducing memory access, improving speed, and increasing bus availability. The register space can be used as a stack or as a fixed register set. When used as a stack, only the top two stack elements are operated on.

The following figure shows representations of large and small numbers in each data type.

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
Word Integer	16	4	$-32{,}768 \leq x \leq +32{,}767$
Short Integer	32	9	$-2 \times 10^9 \leq x \leq +2 \times 10^9$
Long Integer	64	19	$-9 \times 10^{18} \leq x \leq +9 \times 10^{18}$
Packed Decimal	80	18	$-9..99 \leq x \leq +9..99$ (18 digits)
Short Real *	32	6 - 7	$8.43 \times 10^{-37} \leq x \leq 3.37 \times 10^{38}$
Long Real *	64	15 - 16	$4.19 \times 10^{-307} \leq x \leq 1.67 \times 10^{308}$
Temporary Real **	80	19	$3.4 \times 10^{-4932} \leq x \leq 1.2 \times 10^{4932}$

* The short-real and long-real data types correspond to the single-precision and double-precision data types.

** The temporary-real data type corresponds to the extended-precision data Type.

Figure 1. 80287 Data Types

Hardware Interface

The coprocessor uses the same clock generator as the microprocessor and operates in the asynchronous mode. The coprocessor is wired so that it functions as an I/O device through I/O port addresses hex 00F8, 00FA, and 00FC. The microprocessor sends opcodes and operands through these I/O ports. It also receives and stores results through the same I/O ports. The coprocessor 'busy' signal informs the microprocessor that it is executing; the microprocessor Wait instruction forces the microprocessor to wait until the coprocessor is finished executing.

The coprocessor detects six different exception conditions that can occur during instruction execution:

- Invalid operation
- Denormal operand
- Zero-divide
- Overflow
- Underflow
- Precision.

If the appropriate exception-mask bit within the coprocessor is not set, the coprocessor activates the 'error' signal. The 'error' signal generates a hardware interrupt (IRQ 13) causing the 'busy' signal to be held in the busy state. The 'busy' signal may be cleared by an 8-bit I/O Write command to address hex 00F0, with D7 through D0 equal to 0. This action also clears IRQ 13.

The power-on self-test code in the system ROM enables IRQ 13 and sets up its vector to point to a routine in ROM. The ROM routine clears the 'busy' signal latch and then transfers control to the address pointed to by the nonmaskable interrupt (NMI) vector. This maintains code compatibility across the IBM Personal Computer and Personal System/2 product lines. The NMI handler reads the coprocessor status to determine if the coprocessor generated the NMI. If it was not generated by the coprocessor, control is passed to the original NMI handler.

The coprocessor has two operating modes: real-address mode and protected mode. They are similar to the two modes of the microprocessor. The coprocessor is in the real-address mode if reset by a power-on reset, system reset, or I/O write operation to port hex 00F1. This mode is compatible with the 8087 Math Coprocessor used in IBM Personal Computers. The coprocessor is placed in the protected mode by executing the SETPM ESC instruction. It is placed back in the real-address mode by an I/O write operation to port hex 00F1, with D7 through D0 equal to 0.

Detailed information for the internal functions of the 80287 Math Coprocessor is in the books listed in the Bibliography. Also see "Compatibility" for more information.

80386 Microprocessor

The 80386 microprocessor subsystem has the following:

- 32-bit address
- 32-bit data interface
- Extensive instruction set, including string I/O
- Hardware fixed-point multiply and divide
- Three operational modes:
  - Real Address
  - Protected Virtual Address
  - Virtual 8086.

- 4GB of physical address space
- 8 general-purpose 32-bit registers
- 64TB (TB equals 1,099,511,627,776 or 2^{40} bytes) of total virtual-address space.

Real Address Mode

In the real-address mode, the address space of the system microprocessor is a contiguous array of up to 1MB. The system microprocessor generates 20-bit physical addresses to address memory.

The segment portion of the pointer is interpreted as the upper 16 bits of a 20-bit segment address; the lower 4 bits are always 0. Therefore, segment addresses begin on multiples of 16 bytes.

All segments in the real-address mode are 64KB and can be read, written, or executed. An exception or interrupt can occur if data operands or instructions attempt to wrap around the end of a segment (for example, a word with its low-order byte at offset hex FFFF and its high-order byte at hex 0000). If, in the real-address mode, the information contained in the segment does not use the full 64KB, the unused end of the segment can be overlaid by another segment to reduce physical memory requirements.

Protected Virtual Address Mode

The protected virtual-address mode offers extended physical and virtual memory address space, memory protection mechanisms, and new operations to support operating systems and virtual memory.

The protected mode provides up to 64TB of virtual address space for each task mapped into a 4GB physical address space.

From a programmer's point of view, the main difference between the real-address mode and protected mode is the increased address space and the method of calculating the base address. The protected mode uses 32- or 48-bit pointers, consisting of 16-bit selector and 16- or 32-bit offset components. The selector specifies an index into one of two memory-resident tables, the global descriptor table (GDT) or the local descriptor table (LDT). These tables contain the 32-bit base address of a given segment. The 32-bit effective offset is added to the segment base address to form the physical address. The system microprocessor automatically refers to the tables whenever a segment register is loaded with a selector. All instructions that load a segment register refer to the memory-resident tables without additional program support. The memory-resident tables contain 8-byte values called descriptors.

The paging option provides an additional way of managing memory in the very large segments of the 80386. Paging operates in the protected mode only, beneath segmentation. The paging mechanism translates the protected linear address (which comes from the segmentation unit) into a physical address. When paging is not enabled, the physical address is the same as the linear address. The following figure shows the 80386 addressing mechanism.

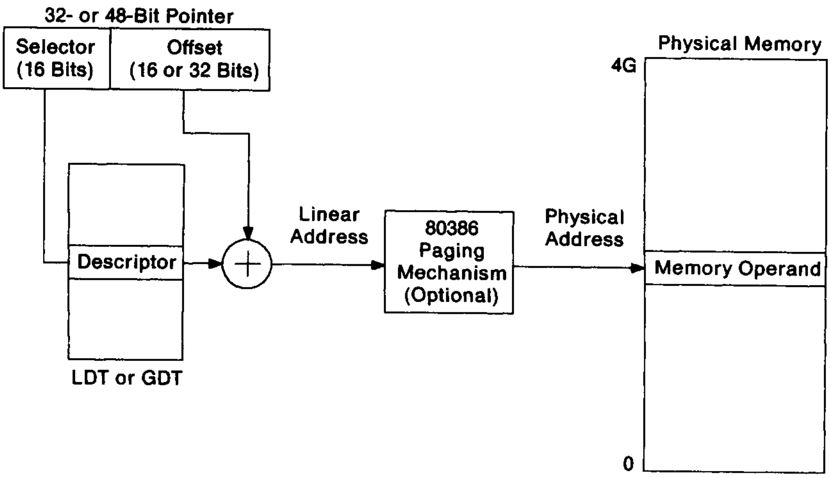

Figure 2. 80386 Addressing

Virtual 8086 Mode

The virtual-8086 mode ensures compatibility of programs written for 8086- and 8088-based systems by establishing a protected 8086 environment within the 80386 multitasking framework.

Since the address space of an 8086 is limited to 1MB, the logical addresses generated by the virtual-8086 mode lie within the first 1MB of the 80386 linear address space. To support multiple virtual-8086 tasks, paging can be used to give each virtual-8086 task a 1MB address space anywhere in the 80386 physical address space.

On a task-by-task basis, the value of the virtual-8086 flag (VM86 flag in the Flags register) determines whether the 80386 behaves as an 80386 or as an 8086. Some instructions, such as Clear Interrupt Flag, can disrupt all operations in a multitasking environment. The 80386 raises an exception when a virtual-8086 mode task attempts to execute an I/O instruction, interrupt-related instruction, or other sensitive instruction. Anytime an exception or interrupt occurs, the 80386 leaves the virtual 8086 mode, making the full resources of the 80386 available to an interrupt handler or exception handler. These handlers can determine if the source of the exception was a virtual-8086 mode task by inspecting the VM86 flag in the Flags image on the stack. If the source is a virtual-8086 mode task, the handler calls on a routine in the operating system to simulate an 8086 instruction and return to the virtual-8086 mode.[1]

80386 Paging Mechanism

The 80386 uses two levels of tables to translate the linear address from the segmentation unit into a physical address. There are three components to the paging mechanism:

- Page directory
- Page tables
- Page frame (the page itself).

The figure on the following page shows how the two-level paging mechanism works.

---

[1] The routine in the operating system, called a *virtual machine monitor*, simulates a limited number of 8086 instructions.

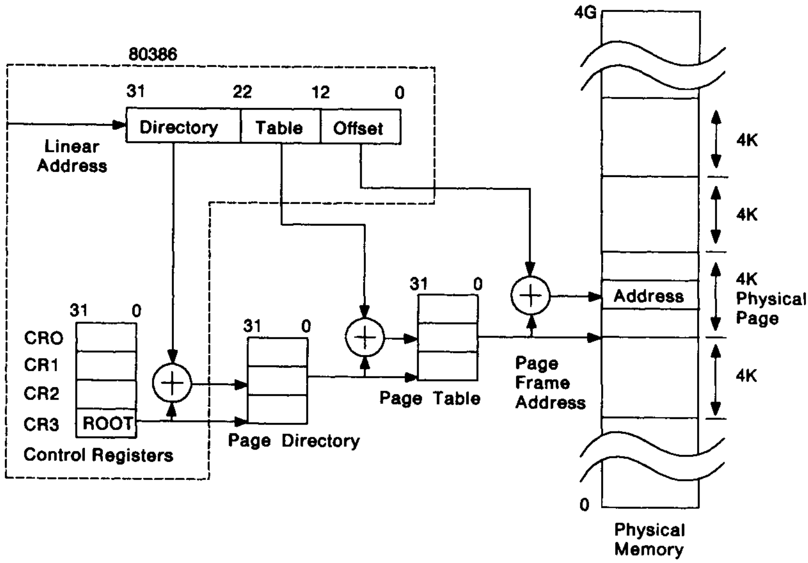

Figure 3. Paging Mechanism

CR2 is the Page-Fault Linear-Address register. It holds the 32-bit linear address that caused the last detected page fault.

CR3 is the Page Directory Physical Base Address register. It contains the physical starting address of the page directory.

The page directory is 4KB and allows up to 1024 page-directory entries. Each page-directory entry contains the address of the next level of tables, the page tables, and information about the page tables. The upper 10 bits of the linear address (A22 through A31) are used as an index to select the correct page-directory entry.

Each page table is 4KB and holds up to 1024 page-table entries. Page-table entries contain the starting address of the page frame and statistical information about the page. Address bits A12 through A21 are used as an index to select one of the 1024 page-table entries. The upper 20 bits of the page-frame address (from the page-table entry) are linked with the lower 12 bits of the linear address to form the physical address. The page-frame address bits become the most-significant bits; the linear-address bits become the least-significant bits.

80387 Math Coprocessor

The optional 80387 Math Coprocessor enables the system to perform high-speed arithmetic, logarithmic, and trigonometric operations. The 80387 effectively extends the 80386 register and instruction set for existing data types and also adds several new data types. The following figure shows the four data type classifications and the instructions associated with each.

Classification	Size	Instructions
Integer	16, 32, 64 Bits	Load, Store, Compare, Add, Subtract, Multiply, Divide
Packed BCD*	80 Bits	Load, Store
Real	32, 64 Bits	Load, Store, Compare, Add, Subtract, Multiply, Divide
Temporary Real	80 Bits	Add, Subtract, Multiply, Divide, Square Root, Scale, Remainder, Integer Part, Change Sign, Absolute Value, Extract Exponent and Significand, Compare, Examine, Test, Exchange Tangent, Arctangent, 2^x-1, $Y^*\text{Log}_2(X+1)$, $Y^*\text{Log}_2(X)$, Load Constant (0.0, π, etc.), Sine, Cosine, Unordered Compare

* BCD = Binary-coded decimal

Figure 4. Data Type Classifications and Instructions

The 80386/80387 configuration fully conforms to the ANSI[2] and IEEE[3] floating-point standard and are upward, object-code compatible from 80286/80287- and 8086/8087-based systems.

2 American National Standards Institute

3 Institute of Electrical and Electronics Engineers

80387 To 80486 Math Coprocessor Compatibility

The 80387 floating-point coprocessor is integrated into the 80486 microprocessor. All numeric 80387 instructions are fully compatible with the 80486 floating-point unit. The 80486 microprocessor supports the 80486 floating-point error reporting modes to ensure DOS compatibility with 80386/80387 systems.

The coprocessor presence test will always show the presence of a coprocessor in the 80486.

Programs for the 80386/80387 systems that explicitly reset the coprocessor by writing to hex 00F1 will no longer function because the coprocessor is an integral part of the microprocessor. Coprocessor reset or initialization must be accomplished through FINIT/FSAVE.

For DOS compatibility, the numeric exception bit Control Register 0 must be set to 0.

Programming Interface

The 80387 is not sensitive to the processing mode of the 80386. The 80387 functions the same whether the 80386 is executing in real-address mode, protected mode, or virtual-8086 mode. All memory access is handled by the 80386; the 80387 merely operates on instructions and values passed to it by the 80386.

All communication between the 80386 and 80387 is transparent to application programs. The 80386 automatically controls the 80387 whenever a numeric instruction is executed. All physical and virtual memory is available for storage of instructions and operands of programs that use the 80387. All memory address modes, including use of displacement, base register, index register, and scaling are available for addressing numeric operands.

The coprocessor has eight 80-bit registers. The total capacity of these eight registers is equivalent to twenty 32-bit registers. This register space allows constants and temporary results to be held in registers during calculations, thus reducing memory access, improving speed, and increasing bus availability. The register space can be used as a stack or as a fixed register set. When it is used as a stack, only the top two stack elements are operated on.

The following figure shows the seven data types supported by the 80387 Math Coprocessor.

Data Type	Range	Precision
Word Integer	10^4	16 Bits
Short Integer	10^9	32 Bits
Long Integer	10^{19}	64 Bits
Packed BCD	10^{18}	18 Digits (2 digits per byte)
Single Precision (Short Real)	$10^{\pm 38}$	24 Bits
Double Precision (Long Real)	$10^{\pm 308}$	53 Bits
Extended Precision (Temporary Real)	$10^{\pm 4932}$	64 Bits

Figure 5. 80387 Data Types

Hardware Interface

The 80387 Math Coprocessor uses the same clock generator as the 80386 system microprocessor. The coprocessor is wired so that it functions as an I/O device through I/O port addresses hex 00F8, 00FA, and 00FC. The system microprocessor sends opcodes and operands through these I/O ports. The coprocessor 'busy' signal informs the system microprocessor that it is executing an instruction; the system microprocessor Wait instruction forces the system microprocessor to wait until the coprocessor is finished executing the instruction.

The coprocessor detects six different exception conditions that can occur during instruction execution:

- Invalid operation
- Denormal operand
- Zero-divide
- Overflow
- Underflow
- Precision.

If the appropriate exception mask bit within the coprocessor is not set, the coprocessor activates the 'error' signal. The 'error' signal generates a hardware interrupt (IRQ 13) causing the 'busy' signal to be held in the busy state. The 'busy' signal can be cleared by an 8-bit I/O Write command to address hex 00F0, with D7 through D0 equal to 0. This action also clears IRQ 13.

The power-on self-test code in the system ROM enables IRQ 13 and sets up its vector to point to a routine in ROM. The ROM routine clears the 'busy' signal latch and then transfers control to the address pointed to by the (NMI) vector. This maintains code compatibility across the IBM Personal Computer and Personal System/2 product lines. The NMI handler reads the status of the coprocessor to determine if the coprocessor generated the NMI. If it was not generated by the coprocessor, control is passed to the original NMI handler.

Detailed information about the internal functions of the 80387 Math Coprocessor is in the books listed in the Bibliography. Also see "Compatibility" for more information.

80486 Microprocessor

The 80486 microprocessor subsystem has the following:

- 32-bit address
- 32-bit data interface
- Extensive instruction set, including string I/O
- Hardware fixed-point multiply and divide
- Three operational modes:
  - Real Address
  - Protected Virtual Address
  - Virtual 8086
- 4GB of physical address space
- 8 general-purpose 32-bit registers
- 64TB of total virtual-address space
- Internal 8KB, set-associative cache with controller
- Internal 80387 coprocessor.

The 80486 microprocessor is compatible with the 80386 in the following areas:

- Real Address Mode
- Protected Virtual Address Mode
- Virtual 8086 Mode
- 80386 Paging Mechanism
- All published 80386 instructions
- All published 80387 instructions.

The complete 80387 Math Coprocessor instruction set and register set have been included in the 80486 as a floating-point unit. No I/O cycles are executed during floating-point instructions. The 80486 microprocessor is 80386/80387 compatible except for resets to the floating-point unit. Software must use FINIT/FSAVE to reset the floating-point unit (math coprocessor). The instruction and data pointers are set to zero after FINIT/FSAVE.

Cache Control

The 80486 microprocessor contains an 8KB integrated cache for code and data. The cache is managed in two ways, and the operation of the cache has no effect on the operation of any program.

The cache is managed by bit 30 — Cache Disable (CD) and bit 29 — Not Write Through (NW) in Control Register 0 (CR0):

Bit 30 CD	Bit 29 NW	Operating Mode
1	1	Cache fills disabled, write-through and invalidate disabled
1	0	Cache fills disabled, write-through and invalidate enabled
0	1	Reserved
0	0	Cache fills enabled, write-through and invalidate enabled (Normal operating mode)

Figure 6. Control Register 0

Cache Paging Control

The page-write-through (PWT) bit and the page-cache-disabled (PCD) bit are two new bits defined in entries in both levels of the page table structure, the page-directory table and the page-table entry, and in Control Register 3.

The PWT bit (bit 4) controls cache write policy. When this bit is set to 1, a write-through policy for the current 4KB page is defined. When this bit is set to 0, it allows the possibility of write-back policy. This bit is ignored internally because the 80486 microprocessor has a write-through-only cache. The PWT bit can be used to control the write policy of a second-level (external) cache.

The PCD bit (bit 3),in conjunction with the KEN# (cache enabled) input signal and the cache-enable and write-transparent bits in Control Register 0 (CR0), controls the ability of cache. When this bit is set to 1, caching is disabled for the 4KB page regardless of the KEN#, cache-enable bit, and write-through bit. These two bits are also driven external to the processor during memory access to manage a second-level cache, if one exists.

The page-write-through and page-cache-disable bits for a bus cycle are obtained either from Control Register 3, the page-directory entry, or the page-table entry, depending on the type of cycle performed.

Page Protection Feature

The 80486 microprocessor has a new protection feature. The write-protect (WP) bit in CR0 has been added to the 80486 microprocessor to protect read-only pages from supervisor write accesses. The 80386 microprocessor allows a read-only page to be written from protection level 0, 1, or 2. When the WP bit is set to 0, the 80486 microprocessor is in the 80386-compatible mode. When the WP bit is set to 0, the supervisor write access to a read-only page (Read/Write is set to 0) causes a page fault (exception 14).

The write-protect bit has a new feature. This feature involves the use of three new bits in CR0:

- User/Supervisor – U/S
- Read/Write – R/W
- Write/Protect – WP.

The compatible protection feature is described by the following table.

U/S	R/W	WP	User Access	Supervisor Access
0	0	0	None	Read/Write/Execute
0	1	0	None	Read/Write/Execute
1	0	0	Read/Execute	Read/Write/Execute
1	1	0	Read/Write/Execute	Read/Write/Execute

Figure 7. 80386 Compatible Operation

The new protection feature is given by the following table.

U/S	R/W	WP	User Access	Supervisor Access
0	0	1	None	Read/Execute
0	1	1	None	Read/Write/Execute
1	0	1	Read/Execute	Read/Execute
1	1	1	Read/Write/Execute	Read/Write/Execute

Figure 8. 80486 Protection Operation

New Alignment Check

The Flag register in the 80486 microprocessor contains a new bit not available in the 80386. The new bit, alignment check, is bit 18 of the Flag register and enables fault reporting on accesses to misaligned data (through interrupt 17 with an error code 0).

When alignment check is set to 1, it enables fault reporting if memory reference is to a misaligned address. A misaligned address is a word access to an odd address, a doubleword access to an address not on a doubleword boundary, or an 8-byte reference to an address that is not on a 64-bit boundary.

Alignment faults are generated only by a program running at privilege level 3. The alignment-check bit is ignored at privilege levels 0, 1, and 2.

The alignment-check bit is conditioned by a new alignment mask bit, defined as bit 18 in Control Register 0. The alignment-mask bit controls whether the alignment-check bit in the Flag register can allow an alignment fault. When the alignment-mask bit is set to 0, the alignment-check bit is disabled and compatible with the 80386 microprocessor. When the alignment-mask bit is set to 1, the alignment-check bit is enabled.

New Instructions

In addition, the 80486 has six unique instructions that control cache operation:

- Byte Swap (BSWAP)
- Compare and Exchange (CMPXCHG)
- Exchange-and-Add (XADD)
- Invalidate Data Cache (INVD)
- Invalidate TLBN Entry (INVLPG).
- Write-Back and Invalidate Data Cache (WBINVD).

80286 Microprocessor Instruction Set

Data Transfer

MOV = Move

Register to Register/Memory

1 0 0 0 1 0 0 w	mod reg r/m

Register/Memory to Register

1 0 0 0 1 0 1 w	mod reg r/m

Immediate to Register/Memory

1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1

Immediate to Register

1 0 1 1 wreg	data	data if w = 1

Memory to Accumulator

1 0 1 0 0 0 0 w	addr-low	addr-high

Accumulator to Memory

1 0 1 0 0 0 1 w	addr-low	addr-high

Register/Memory to Segment Register

1 0 0 0 1 1 1 0	mod 0 reg r/m

Segment Register to Register/Memory

1 0 0 0 1 1 0 0	mod 0 reg r/m

PUSH = Push

Memory

1 1 1 1 1 1 1 1	mod 1 1 0 r/w

Register

0 1 0 1 0 reg

Segment Register

0 0 0 reg 1 1 0

Immediate

0 1 1 0 1 0 s 0	data	data if s = 0

PUSHA = Push All

0 1 1 0 0 0 0 0

POP = Pop

Register/Memory

1 0 0 0 1 1 1 1	mod 0 0 0 r/m

Register

0 1 0 1 1 reg

Segment Register

0 0 0 reg 1 1 1	reg ≠ 0 1

POPA = Pop All

0 1 1 0 0 0 0 1

XCHG = Exchange

Register/Memory with Register

1 0 0 0 0 1 1 w	mod reg r/m

Register with Accumulator

1 0 0 1 0 reg

IN = Input From

Fixed Port

1 1 1 0 0 1 0 w	port

Variable Port

1 1 1 0 1 1 0 w

OUT = Output To

Fixed Port

1 1 1 0 0 1 1 w	port

Variable Port

1 1 1 0 1 1 1 w

XLAT = Translate Byte to AL

1 1 0 1 0 1 1 1

LEA = Load EA to Register

10001101	mod reg r/m

LDS = Load Pointer to DS

11000101	mod reg r/m mod ≠ 11

LES = Load Pointer to ES

11000100	mod reg r/m mod ≠ 11

LAHF = Load AH with Flags

10011111

SAHF = Store AH with Flags

10011110

PUSHF = Push Flags

10011100

POPF = Pop Flags

10011101

Arithmetic

ADD = Add

Register/Memory with Register to Either

0 0 0 0 0 0 dw	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 sw	mod 0 0 0 r/m	data	data if sw = 0 1

Immediate to Accumulator

0 0 0 0 0 1 0 w	data	data if w = 1

ADC = Add with Carry

Register/Memory with Register to Either

0 0 0 1 0 0 dw	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 sw	mod 0 1 0 r/m	data	data if sw = 0 1

Immediate to Accumulator

0 0 0 1 0 1 0 w	data	data if w = 1

INC = Increment

Register/Memory

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

Register

0 1 0 0 0 reg

SUB = Subtract

Register/Memory with Register to Either

0 0 1 0 1 0 d w	mod reg r/m

Immediate from Register/Memory

1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if sw = 0 1

Immediate from Accumulator

0 0 1 0 1 1 0 w	data	data if w = 1

SBB = Subtract with Borrow

Register/Memory with Register to Either

0 0 0 1 1 0 d w	mod reg r/m

Immediate from Register/Memory

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if sw = 0 1

Immediate from Accumulator

0 0 0 1 1 1 0 w	data	data if w = 1

DEC = Decrement

Register/Memory

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

Register

0 1 0 0 1 reg

CMP = Compare

Register/Memory with Register

0 0 1 1 1 0 1 w	mod reg r/m

Register with Register/Memory

0 0 1 1 1 0 0 w	mod reg r/m

Immediate with Register/Memory

1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if sw = 0 1

Immediate with Accumulator

0 0 1 1 1 1 0 w	data	data if w = 1

NEG = Change Sign

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

AAA = ASCII Adjust for Add

0 0 1 1 0 1 1 1

DAA = Decimal Adjust for Add

0 0 1 0 0 1 1 1

AAS = ASCII Adjust for Subtract

0 0 1 1 1 1 1 1

DAS = Decimal Adjust for Subtract

0 0 1 0 1 1 1 1

MUL = Multiply (Unsigned)

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

IMUL = Integer Multiply (Signed)

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

IIMUL = Integer Immediate Multiply (Signed)

0 1 1 0 1 0 s 1	mod reg r/m	data	data if s = 0

DIV = Divide (Unsigned)

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

IDIV = Integer Divide (Signed)

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

AAM = ASCII Adjust for Multiply

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

AAD = ASCII Adjust for Divide

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

CBW = Convert Byte to Word

1 0 0 1 1 0 0 0

CWD = Convert Word to Doubleword

1 0 0 1 1 0 0 1

Logic

Shift/Rotate Instructions

Register/Memory by 1

1101000w	mod TTT r/m

Register/Memory by CL

1101001w	mod TTT r/m

Register/Memory by Count

1100000w	mod TTT r/m	count

TTT	Instruction
000	ROL
001	ROR
010	RCL
011	RCR
100	SHL/SAL
101	SHR
111	SAR

AND = And

Register/Memory and Register to Either

001000dw	mod reg r/m

Immediate to Register/Memory

1000000w	mod 100 r/m	data	data if w = 1

Immediate to Accumulator

0010010w	data	data if w = 1

TEST = AND Function to Flags; No Result

Register/Memory and Register

1 0 0 0 0 1 0 w	mod reg r/m

Immediate Data and Register/Memory

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1

Immediate Data and Accumulator

1 0 1 0 1 0 0 w	data	data if w = 1

Or = Or

Register/Memory and Register to Either

0 0 0 0 1 0 d w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1

Immediate to Accumulator

0 0 0 0 1 1 0 w	data	data if w = 1

XOR = Exclusive OR

Register/Memory and Register to Either

0 0 1 1 0 0 d w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1

Immediate to Accumulator

0 0 1 1 0 1 0 w	data	data if w = 1

NOT = Invert Register/Memory

1 1 1 1 0 1 1 w	mod 0 1 0 r/m

String Manipulation

MOVS = Move Byte Word

1 0 1 0 0 1 0 w

CMPS B/W = Compare Byte/Word

1 0 1 0 0 1 1 w

SCAS = Scan Byte/Word

1 0 1 0 1 1 1 w

LODS = Load Byte/Word to AL/AX

1 0 1 0 1 1 0 w

STOS = Store Byte/Word from AL/AX

1 0 1 0 1 0 1 w

INS = Input Byte/Word from DX Port

0 1 1 0 1 1 0 w

OUTS = Output Byte/Word to DX Port

0 1 1 0 1 1 1 w

REP/REPNE, REPZ/REPNZ = Repeat String

Repeat Move String

1 1 1 1 0 0 1 1	1 0 1 0 0 1 0 w

Repeat Compare String (z/Not z)

1 1 1 1 0 0 1 z	1 0 1 0 0 1 1 w

Repeat Scan String (z/Not z)

1 1 1 1 0 0 1 z	1 0 1 0 1 1 1 w

Repeat Load String

1 1 1 1 0 0 1 1	1 0 1 0 1 1 0 w

Repeat Store String

1 1 1 1 0 0 1 1	1 0 1 0 1 0 1 w

Repeat Input String

1 1 1 1 0 0 1 1	0 1 1 0 1 1 0 w

Repeat Output String

1 1 1 1 0 0 1 1	0 1 1 0 1 1 1 w

Control Transfer

CALL = Call

Direct within Segment

1 1 1 0 1 0 0 0	disp-low	disp-high

Register/Memory Indirect within Segment

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

Direct Intersegment

1 0 0 1 1 0 1 0	Segment Offset	Segment Selector

Indirect Intersegment

1 1 1 1 1 1 1 1	mod 0 1 1 r/m (mod ≠ 11)

JMP = Unconditional Jump

Short/Long

1 1 1 0 1 0 1 1	disp-low

Direct within Segment

1 1 1 0 1 0 0 1	disp-low	disp-high

Register/Memory Indirect within Segment

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

Direct Intersegment

1 1 1 0 1 0 1 0	Segment Offset	Segment Selector

Indirect Intersegment

1 1 1 1 1 1 1 1	mod 1 0 1 r/m (mod ≠ 11)

RET = Return from Call

Within Segment

1 1 0 0 0 0 1 1

Within Segment Adding Immediate to SP

1 1 0 0 0 0 1 0	data-low	data-high

Intersegment

1 1 0 0 1 0 1 1

Intersegment Adding Immediate to SP

1 1 0 0 1 0 1 0	data-low	data-high

JE/JZ = Jump on Equal/Zero

0 1 1 1 0 1 0 0	disp

JL/JNGE = Jump on Less/Not Greater, or Equal

0 1 1 1 1 1 0 0	disp

JLE/JNG = Jump on Less, or Equal/Not Greater

0 1 1 1 1 1 1 0	disp

JB/JNAE = Jump on Below/Not Above, or Equal

0 1 1 1 0 0 1 0	disp

JBE/JNA = Jump on Below, or Equal/Not Above

01110110	disp

JP/JPE = Jump on Parity/Parity Even

01111010	disp

JO = Jump on Overflow

01110000	disp

JS = Jump on Sign

01111000	disp

JNE/JNZ = Jump on Not Equal/Not Zero

01110101	disp

JNL/JGE = Jump on Not Less/Greater, or Equal

01111101	disp

JNLE/JG = Jump on Not Less, or Equal/Greater

01111111	disp

JNB/JAE = Jump on Not Below/Above, or Equal

01110011	disp

JNBE/JA = Jump on Not Below, or Equal/Above

01110111	disp

JNP/JPO = Jump on Not Parity/Parity Odd

0 1 1 1 1 0 1 1	disp

JNO = Jump on Not Overflow

0 1 1 1 0 0 0 1	disp

JNS = Jump on Not Sign

0 1 1 1 1 0 0 1	disp

LOOP = Loop CX Times

1 1 1 0 0 0 1 0	disp

LOOPZ/LOOPE = Loop while Zero/Equal

1 1 1 0 0 0 0 1	disp

LOOPNZ/LOOPNE = Loop while Not Zero/Not Equal

1 1 1 0 0 0 0 0	disp

JCXZ = Jump on CX Zero

1 1 1 0 0 0 1 1	disp

ENTER = Enter Procedure

1 1 0 0 1 0 0 0	data-low	data-high

LEAVE = Leave Procedure

1 1 0 0 1 0 0 1

INT = Interrupt

Type Specified

11001101

Type 3

11001100

INTO = Interrupt on Overflow

11001110

IRET = Interrupt Return

11001111

BOUND = Detect Value Out of Range

01100010	mod reg r/m

Processor Control

CLC = Clear Carry

11111000

CMC = Complement Carry

11110101

STC = Set Carry

11111001

CLD = Clear Direction

11111100

STD = Set Direction

11111101

CLI = Clear Interrupt

11111010

STI = Set Interrupt Enable Flag

11111011

HLT = Halt

11110100

WAIT = Wait

10011011

LOCK = Bus Lock Prefix

11110000

CTS = Clear Task Switched Flag

00001111	00000110

ESC = Processor Extension Escape

11011TTT	mod LLL r/m

Protection Control

LGDT = Load Global Descriptor Table Register

00001111	00000001	mod 0 1 0 r/m

SGDT = Store Global Descriptor Table Register

00001111	00000001	mod 0 0 0 r/m

LIDT = Load Interrupt Descriptor Table Register

00001111	00000001	mod 0 1 1 r/m

SIDT = Store Interrupt Descriptor Table Register

00001111	00000001	mod 0 0 1 r/m

LLDT = Load Local Descriptor Table Register from Register/Memory

00001111	00000000	mod 0 1 0 r/m

SLDT = Store Local Descriptor Table Register from Register/Memory

00001111	00000000	mod 0 0 0 r/m

LTR = Load Task Register from Register/Memory

00001111	00000000	mod 0 1 1 r/m

STR = Store Task Register to Register/Memory

00001111	00000000	mod 0 0 1 r/m

LMSW = Load Machine Status Word from Register/Memory

00001111	00000001	mod 1 1 0 r/m

SMSW = Store Machine Status Word

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod 1 0 0 r/m

LAR = Load Access Rights from Register/Memory

0 0 0 0 1 1 1 1	0 0 0 0 0 0 1 0	mod reg r/m

LSL = Load Segment Limit from Register/Memory

0 0 0 0 1 1 1 1	0 0 0 0 0 0 1 1	mod reg r/m

ARPL = Adjust Requested Privilege Level from Register/Memory

0 1 1 0 0 0 1 1	mod reg r/m

VERR = Verify Read Access; Register/Memory

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod 1 0 0 r/m

VERW = Verify Write Access

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod 1 0 1 r/m

The effective address (EA) of the memory operand is computed according to the mod and r/m fields:

If mod = 11, then r/m is treated as a reg field.
If mod = 00, then disp = 0, disp-low and disp-high are absent.
If mod = 01, then disp = disp-low sign-extended to 16 bits, disp-high is absent.
If mod = 10, then disp = disp-high:disp-low.

If r/m = 000, then EA = (BX) + (SI) + DISP
If r/m = 001, then EA = (BX) + (DI) + DISP
If r/m = 010, then EA = (BP) + (SI) + DISP
If r/m = 011, then EA = (BP) + (DI) + DISP
If r/m = 100, then EA = (SI) + DISP
If r/m = 101, then EA = (DI) + DISP
If r/m = 110, then EA = (BP) + DISP
If r/m = 111, then EA = (BX) + DISP

The disp field follows the second byte of the instruction (before data if required).

Note: An exception to the above statements occurs when mod = 00 and r/m = 110, in which case EA = disp-high; disp-low.

Segment Override Prefix

0 0 1 reg 1 1 0

The 2-bit and 3-bit reg fields are defined in the following figures.

Reg	**Segment Register**	**Reg**	**Segment Register**
00	ES	10	SS
01	CS	11	DS

Figure 9. 2-Bit Register Field

Figure 10. 3-Bit Register Field

16-Bit (w = 1)	**8-Bit (w = 0)**
000 AX	000 AL
001 CX	001 CL
010 DX	010 DL
011 BX	011 BL
100 SP	100 AH
101 BP	101 CH
110 SI	110 DH
111 DI	111 BH

The physical addresses of all operands addressed by the BP register are computed using the SS Segment register. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden.

80287 Math Coprocessor Instruction Set

The following is an instruction-set summary for the 80287 Math Coprocessor.

The following figure shows abbreviations used in the summary.

Field	Description	Bit Information
escape	80286 Extension Escape	Bit Pattern = 11011
MF	Memory Format	00 = 32-Bit Real 01 = 32-Bit Integer 10 = 64-Bit Real 11 = 16-Bit Integer
ST(0)	Current Stack Top	
ST(i)	i^{th} Register Below the Stack Top	
d	Destination	0 = Destination is ST(0) 1 = Destination is ST(i)
P	Pop	0 = No pop 1 = Pop ST(0)
R	Reverse*	0 = Destination (op) source 1 = Source (op) destination

* When d = 1, reverse the sense of R.

Figure 11. 80287 Encoding Field Summary

Data Transfer

FLD = Load

Integer/Real Memory to ST(0)

escape MF 1

Long Integer Memory to ST(0)

escape 1 1 1	mod 1 0 1 r/m

Temporary Real Memory to ST(0)

escape 0 1 1	mod 1 0 1 r/m

BCD Memory to ST(0)

escape 1 1 1	mod 1 0 0 r/m

ST(i) to ST(0)

escape 0 0 1	1 1 0 0 0 ST(i)

FST = Store

ST(0) to Integer/Real Memory

escape MF 1	mod 0 1 0 r/m

ST(0) to ST(i)

escape 1 0 1	1 1 0 1 0 ST(i)

FSTP = Store and Pop

ST(0) to Integer/Real Memory

escape MF 1	mod 0 1 1 r/m

ST(0) to Long Integer Memory

escape 1 1 1	mod 1 1 1 r/m

ST(0) to Temporary Real Memory

escape 0 1 1	mod 1 1 1 r/m

ST(0) to BCD Memory

escape 1 1 1	mod 1 1 0 r/m

ST(0) to ST(i)

escape 1 0 1	1 1 0 1 1 ST(i)

FXCH = Exchange ST(I) and ST(0)

escape 0 0 1	1 1 0 0 1 ST(i)

Comparison

FCOM = Compare

Integer/Real Memory to ST(0)

escape MF 0	mod 0 1 0 r/m

ST(i) to ST(0)

escape 0 0 0	1 1 0 1 0 ST(i)

FCOMP = Compare and Pop

Integer/Real Memory to ST(0)

escape MF 0	mod 0 1 1 r/m

ST(i) to ST(0)

escape 0 0 0	1 1 0 1 1 ST(i)

FCOMPP = Compare ST(1) to ST(0) and Pop Twice

escape 1 1 0	1 1 0 1 1 0 0 1

FTST = Test ST(0)

escape 0 0 1	1 1 1 0 0 1 0 0

FXAM = Examine ST(0)

escape 0 0 1	1 1 1 0 0 1 0 1

Constants

FLDZ = Load + 0.0 Into ST(0)

escape 0 0 1	1 1 1 0 1 1 1 0

FLD1 = Load + 1.0 Into ST(0)

escape 0 0 1	1 1 1 0 1 0 0 0

FLDPI = Load π Into ST(0)

escape 0 0 1	1 1 1 0 1 0 1 1

FLDL2T = Load $\log_2 10$ Into ST(0)

escape 0 0 1	1 1 1 0 1 0 0 1

FLDL2E = Load $\log_2 e$ Into ST(0)

escape 0 0 1	1 1 1 0 1 0 1 0

FLDLG2 = Load $\log_{10} 2$ Into ST(0)

escape 0 0 1	1 1 1 0 1 1 0 0

FLDLN2 = Load $\log_e 2$ Into ST(0)

escape 0 0 1	1 1 1 0 1 1 0 1

Arithmetic

FADD = Addition

Integer/Real Memory with ST(0)

escape MF 0	mod 0 0 0 r/m

ST(i) and ST(0)

escape dP0	1 1 0 0 0 ST(i)

FSUB = Subtraction

Integer/Real Memory with ST(0)

escape MF 0	mod 1 0 R r/m

ST(i) and ST(0)

escape dP 0	1110R r/m

FMUL = Multiplication

Integer/Real Memory with ST(0)

escape MF 0	mod 0 0 1 r/m

ST(i) and ST(0)

escape dP 0	1 1 0 0 1 r/m

FDIV = Division

Integer/Real Memory with ST(0)

escape MF 0	mod 1 1 R r/m

ST(i) and ST(0)

escape dP 0	1 1 1 1 R r/m

FSQRT = Square Root of ST(0)

escape 0 0 1	1 1 1 1 1 0 1 0

FSCALE = Scale ST(0) by ST(1)

escape 0 0 1	1 1 1 1 1 1 0 1

FPREM = Partial Remainder of ST(0) ÷ ST(1)

escape 0 0 1	1 1 1 1 1 0 0 0

FRNDINT = Round ST(0) to Integer

escape 0 0 1	1 1 1 1 1 1 0 0

FXTRACT = Extract Components of ST(0)

escape 0 0 1	1 1 1 1 0 1 0 0

FABS = Absolute Value of ST(0)

escape 0 0 1	1 1 1 0 0 0 0 1

FCHS = Change Sign of ST(0)

escape 0 0 1	1 1 1 0 0 0 0 0

Transcendental

FPTAN = Partial Tangent of ST(0)

escape 0 0 1	1 1 1 1 0 0 1 0

FPATAN = Partial Arctangent of ST(1) ÷ ST(0)

escape 0 0 1	1 1 1 1 0 0 1 1

F2XM1 = $2^{ST(0)}$ -1

escape 0 0 1	1 1 1 1 0 0 0 0

FYL2X = ST(1) x Log_2 [ST(0)]

escape 0 0 1	1 1 1 1 0 0 0 1

FYL2XP1 = ST(1) x Log_2 [ST(0) + 1]

escape 0 0 1	1 1 1 1 1 0 0 1

Processor Control

FINIT = Initialize NPX

escape 0 1 1	1 1 1 0 0 0 1 1

FSETPM = Enter Protected Mode

escape 0 1 1	1 1 1 0 0 1 0 0

FSTSW AX = Store Control Word

escape 1 1 1	1 1 1 0 0 0 0 0

FLDCW = Load Control Word

escape 0 0 1	mod 1 0 1 r/m

FSTCW = Store Control Word

escape 0 0 1	mod 1 1 1 r/m

FSTSW = Store Status Word

escape 1 0 1	mod 1 1 1 r/m

FCLEX = Clear Exceptions

escape 0 1 1	1 1 1 0 0 0 1 0

FSTENV = Store Environment

escape 0 0 1	mod 1 1 0 r/m

FLDENV = Load Environment

escape 0 0 1	mod 1 0 0 r/m

FSAVE = Save State

escape 1 0 1	mod 1 1 0 r/m

FRSTOR = Restore State

escape 1 0 1	mod 1 0 0 r/m

FINCSTP = Increment Stack Pointer

escape 0 0 1	1 1 1 1 0 1 1 1

FDECSTP = Decrement Stack Pointer

escape 0 0 1	1 1 1 1 0 1 1 0

FFREE = Free ST(I)

escape 1 0 1	1 1 0 0 0 ST(i)

FNOP = No Operation

escape 0 0 1	1 1 0 1 0 0 0 0

Introduction to the 80386 Instruction Set

The 80386 instruction set is an extended version of the 8086 and 80286 instruction sets. The instruction sets have been extended in two ways:

- The instructions have extensions that allow operations on 32-bit operands, registers, and memory.
- A 32-bit addressing mode allows flexible selection of registers for base and index as well as index scaling capabilities (x2, x4, x8) for computing a 32-bit effective address. The 32-bit effective address yields a 4GB address range.

Note: The effective address size must be less than 64KB in the real-address or virtual-address modes to avoid an exception.

Code and Data Segment Descriptors

Although the 80386 supports all 80286 Code and Data segment descriptors, there are some differences in the format. The 80286 segment descriptors contain a 24-bit base address and a 16-bit limit field, while the 80386 segment descriptors have a 32-bit base address, a 20-bit limit field, a default bit, and a granularity bit.

<table>
<tr><th>31 24</th><th colspan="5">23 16</th><th>15 08</th><th>07 00</th><th>Offset</th></tr>
<tr><td colspan="6">Segment Base (SB) Bits 15-0</td><td colspan="2">Segment Limit (SL) Bits 15-0</td><td>0</td></tr>
<tr><td>SB Bits 31-24</td><td>G</td><td>D</td><td>0</td><td>0</td><td>SL 19-16</td><td>Access Rights Byte</td><td>SB Bits 23-16</td><td>4</td></tr>
</table>

Figure 12. 80386 Code and Data Segment Descriptor Format

Note: Bits 31 through 16 shown at offset 4 are set to 0 for all 80286 segment descriptors.

The default (D) bit of the code segment register is used to determine whether the instruction is carried out as a 16-bit or 32-bit instruction. Code segment descriptors are not used in either the real-address mode or the virtual-8086 mode. When the system microprocessor is operating in either of these modes, a D-bit value of 0 is assumed and operations default to a 16-bit length compatible with 8086 and 80286 programs.

The granularity (G) bit is used to determine the granularity of the segment length (1 = page granular, 0 = byte granular). If the value of the 20 segment-limit bits is defined as *N*, a G-bit value of 1 defines the segment size as follows:

Segment size = $(N + 1) \times 4KB$

4KB represents the size of a page.

Prefixes

Two prefixes have been added to the instruction set. The Operand Size prefix overrides the default selection of the operand size; the Effective Address Size prefix overrides the effective address size. The presence of either prefix toggles the default setting to its opposite condition. For example:

- If the operand size defaults to 32-bit data operations, the presence of the Operand Size prefix sets it for 16-bit data operations.
- If the effective address size is 16-bits, the presence of the Effective Address Size prefix toggles the instruction to use 32-bit effective address computations.

The prefixes are available in all 80386 modes, including the real-address mode and the virtual-8086 mode. Since the default of these modes is always 16 bits, the prefixes are used to specify 32-bit operations. If needed, either or both of the prefixes may precede any opcode bytes and affect only the instruction they precede.

Instruction Format

The instructions are presented in this format:

Opcode	Mode Specifier	Address Displacement	Immediate Data

Term	Description
Opcode	The opcode may be one or two bytes in length. Within each byte, smaller encoding fields may be defined.
Mode Specifier	Consists of the "mod r/m" byte and the "scale-index-base" (s-i-b) byte. The mod r/m byte specifies the address mode to be used. Format: mod T T T r/m The "s-i-b" byte is optional and can be used only in 32-bit address modes. It follows the mod r/m byte to fully specify the manner in which the effective address is computed. Format: ss index base
Address Displacement	Follows the "mod r/m" byte or "s-i-b" byte. It may be 8, 16, or 32 bits.
Immediate Data	If specified, follows any displacement bytes and becomes the last field of the instruction. It may be 8, 16, or 32 bits. The term "8-bit data" indicates a fixed data length of 8 bits. The term "8-, 16-, or 32-bit data" indicates a variable data length. The length is determined by the w field and the current operand size. If w = 0, the data is always 8 bits. If w = 1, the size is determined by the operand size of the instruction.

Figure 13. Instruction Format

The instructions use a variety of fields to indicate register selection, the addressing mode, and so on. The following figure is a summary of the fields.

Field Name	Description	Bit Information
w	Specifies if data is byte or full size. (Full size is either 16 or 32 bits.)	1
d	Specifies the direction of data operation.	1
s	Specifies if an immediate data field must be sign-extended.	1
reg	General address specifier.	3
mod r/m	Address mode specifier (effective address can be a general register).	2 for mod; 3 for r/m
ss	Scale factor for scaled index address mode.	2
index	General register to be used as an index register.	3
base	General register to be used as base register.	3
sreg2	Segment register specifier for CS, SS, DS, and ES.	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, and GS.	3
tttn	For conditional instructions; specifies a condition asserted or a condition negated.	4

Figure 14. 80386 Instruction Set Encoding Field Summary

Encoding

This section defines the encoding of the fields used in the instruction sets.

Address Mode

The first addressing byte is the "mod r/m" byte. The effective address (EA) of the memory operand is computed according to the mod and r/m fields. The mod r/m byte can be interpreted as either a 16-bit or 32-bit addressing mode specifier. Interpretation of the byte depends on the address components used to calculate the EA. The following figure defines the encoding of 16-bit and 32-bit addressing modes with the mod r/m byte.

mod r/m	16-Bit Mode	32-Bit Mode (No s-i-b byte)
00 000	DS:[BX + SI]	DS:[EAX]
00 001	DS:[BX + DI]	DS:[ECX]
00 010	SS:[BP + SI]	DS:[EDX]
00 011	SS:[BP + DI]	DS:[EBX]
00 100	DS:[SI]	s-i-b present (see Figure 19 on page 53)
00 101	DS:[DI]	DS:d32
00 110	d16	DS:[ESI]
00 111	DS:[BX]	DS:[EDI]
01 000	DS:[BX + SI + d8]	DS:[EAX + d8]
01 001	DS:[BX + DI + d8]	DS:[ECX + d8]
01 010	SS:[BP + SI + d8]	DS:[EDX + d8]
01 011	SS:[BP + DI + d8]	DS:[EBX + d8]
01 100	DS:[SI + d8]	s-i-b present (see Figure 19 on page 53)
01 101	DS:[DI + d8]	SS:[EBP + d8]
01 110	SS:[BP + d8]	DS:[ESI + d8]
01 111	DS:[BX + d8]	DS:[EDI + d8]
10 000	DS:[BX + SI + d16]	DS:[EAX + d32]
10 001	DS:[BX + DI + d16]	DS:[ECX + d32]
10 010	SS:[BP + SI + d16]	SS:[EDX + d32]
10 011	SS:[BP + DI + d16]	DS:[EBX + d32]
10 100	DS:[SI + d16]	s-i-b present (see Figure 19 on page 53)
10 101	DS:[DI + d16]	SS:[EBP + d32]
10 110	SS:[BP + d16]	DS:[ESI + d32]
10 111	DS:[BX + d16]	DS:[EDI + d32]

Figure 15. Effective Address (16-Bit and 32-Bit Address Modes)

The displacement follows the second byte of the instruction (before data, if required).

The scale-index-base (s-i-b) byte can be specified as a second byte of addressing information. The s-i-b byte is specified when using a 32-bit addressing mode and the mod r/m byte has the following values:

- r/m = 100
- mod = 00, 01, or 10.

When the s-i-b byte is present, the 32-bit effective address is a function of the mod, ss, index, and base fields. The following figures show the scale factor, Index register selected, and base register selected when the s-i-b byte is present.

ss	Scale Factor
00	1
01	2
10	4
11	8

Figure 16. Scale Factor (s-i-b Byte Present)

Index	Index Register	
000	EAX	
001	ECX	
010	EDX	
011	EBX	
100	No Index Register	The ss field must equal 00 when the index field is 100; if not, the effective address is undefined.
101	EBP	
110	ESI	
111	EDI	

Figure 17. Index Registers (s-i-b Byte Present)

base	Base Register	
000	EAX	
001	ECX	
010	EDX	
011	EBX	
100	ESP	
101	EBP	If mod = 00, then EBP is not used to form the EA; immediate 32-bit address displacement follows the mode specifier byte.
110	ESI	
111	EDI	

Figure 18. Base Registers (s-i-b Byte Present)

The scaled-index information is determined by multiplying the contents of the Index register by the scale factor. The following example shows the use of the 32-bit addressing mode with scaling where:

- EAX is the base of ARRAY_A
- ECX is the index of the desired element
- 2 is the scale factor.

```
; ARRAY_A is an array of words
MOV EAX, offset ARRAY_A
MOV ECX, element_number
MOV BX, [EAX][ECX*2]
```

The following figure defines the encoding of the 32-bit addressing mode when the s-i-b byte is present.

Note: The mod field is from the mod r/m byte. The base field and scaled-index information are from the s-i-b byte.

Mod Base	32-Bit Address Mode
00 000	DS:[EAX + (scaled index)]
00 001	DS:[ECX + (scaled index)]
00 010	DS:[EDX + (scaled index)]
00 011	DS:[EBX + (scaled index)]
00 100	SS:[ESP + (scaled index)]
00 101	DS:[d32 + (scaled index)]
00 110	DS:[ESI + (scaled index)]
00 111	DS:[EDI + (scaled index)]
01 000	DS:[EAX + (scaled index) + d8]
01 001	DS:[ECX + (scaled index) + d8]
01 010	DS:[EDX + (scaled index) + d8]
01 011	DS:[EBX + (scaled index) + d8]
01 100	SS:[ESP + (scaled index) + d8]
01 101	SS:[EBP + (scaled index) + d8]
01 110	DS:[ESI + (scaled index) + d8]
01 111	DS:[EDI + (scaled index) + d8]
10 000	DS:[EAX + (scaled index) + d32]
10 001	DS:[ECX + (scaled index) + d32]
10 010	DS:[EDX + (scaled index) + d32]
10 011	DS:[EBX + (scaled index) + d32]
10 100	SS:[ESP + (scaled index) + d32]
10 101	SS:[EBP + (scaled index) + d32]
10 110	DS:[ESI + (scaled index) + d32]
10 111	DS:[EDI + (scaled index) + d32]

Figure 19. Effective Address (32-Bit Address Mode — s-i-b Byte Present)

Operand Length (w) Field

For an instruction performing a data operation, the instruction is executed as either a 32-bit or 16-bit operation. Within the constraints of the operation size, the w field encodes the operand size as either one byte or full operation.

w	16-Bit Data Operation	32-Bit Data Operation
0	8 Bits	8 Bits
1	16 Bits	32 Bits

Figure 20. Operand Length Field Encoding

Segment Register (sreg) Field

The 2-bit segment register field (sreg2) allows one of the four 80286 segment registers to be specified. The 3-bit segment register (sreg3) allows the 80386 FS and GS segment registers to be specified.

sreg2	sreg3	Segment Register
00	000	ES
01	001	CS
10	010	SS
11	011	DS
--	100	FS
--	101	GS
--	110	Reserved
--	111	Reserved

Figure 21. Segment Register Field Encoding

General Register (reg) Field

The general register is specified by the reg field, which may appear in the primary opcode bytes as the reg field of the mod reg r/m byte, or as the r/m field of the mod reg r/m byte when mod = 11.

reg	16-Bit w/o w	16-Bit w=0	16-Bit w=1	32-Bit w/o w	32-Bit w=0	32-Bit w=1
000	AX	AL	AX	EAX	AL	EAX
001	CX	CL	CX	ECX	CL	ECX
010	DX	DL	DX	EDX	DL	EDX
011	BX	BL	BX	EBX	BL	EBX
100	SP	AH	SP	ESP	AH	ESP
101	BP	CH	BP	EBP	CH	EBP
110	SI	DH	SI	ESI	DH	ESI
111	DI	BH	DI	EDI	BH	EDI

Figure 22. General Register Field Encoding

The physical addresses of all operands addressed by the BP register are computed using the SS Segment register. For string primitive operations (those addressed by the DI register), addresses of the destination operands are computed using the ES segment, which may not be overridden.

Operation Direction (d) Field

The operation direction (d) field is used in many two-operand instructions to indicate which operand is the source and which is the destination.

d	Direction of Operation
0	Register/Memory <-- Register The "reg" field indicates the source operand; "mod r/m" or "mod ss index base" indicates the destination operand.
1	Register<-- Register/Memory The "reg" field indicates the destination operand; "mod r/m" or "mod ss index base" indicates the source operand.

Figure 23. Operand Direction Field Encoding

Sign-Extend (s) Field

The sign-extend (s) field appears primarily in instructions having immediate data fields. The s field affects only 8-bit immediate data being placed in a 16-bit or 32-bit destination.

s	8-Bit Immediate Data	16/32-Bit Immediate Data
0	No effect on data	No effect on data
1	Sign-extend 8-bit data to fill 16-bit or 32-bit destination	No effect on data

Figure 24. Sign-Extend Field Encoding

Conditional Test (tttn) Field

For conditional instructions (conditional jumps and set-on condition), the conditional test (tttn) field is encoded, with n indicating whether to use the condition (n = 0) or its negation (n = 1), and ttt defining the condition to test.

tttn	Condition	Mnemonic
0000	Overflow	O
0001	No Overflow	NO
0010	Below/Not Above or Equal	B/NAE
0011	Not Below/Above or Equal	NB/AE
0100	Equal/Zero	E/Z
0101	Not Equal/Not Zero	NE/NZ
0110	Below or Equal/Not Above	BE/NA
0111	Not Below or Equal/Above	NBE/A
1000	Sign	S
1001	Not Sign	NS
1010	Parity/Parity Even	P/PE
1011	Not Parity/Parity Odd	NP/PO
1100	Less Than/Not Greater or Equal	L/NGE
1101	Not Less Than/Greater or Equal	NL/GE
1110	Less Than or Equal/Not Greater Than	LE/NG
1111	Not Less or Equal/Greater Than	NLE/G

Figure 25. Conditional Test Field Encoding

Control, Debug, or Test Register (eee) Field

The following shows the encoding for loading and storing the Control, Debug, and Test registers (eee).

eee Code	Interpreted as Control Register	Interpreted as Debug Register	Interpreted as Test Register
000	CR0	DR0	---
001	---	DR1	---
010	CR2	DR2	---
011	CR3	DR3	---
100	---	---	---
101	---	---	---
110	---	DR6	TR6
111	---	DR7	TR7

Figure 26. Control, Debug, and Test Register Field Encoding

80386 Microprocessor Instruction Set

Data Transfer

MOV = Move

Register to Register/Memory

1 0 0 0 1 0 0 w	mod reg r/m

Register/Memory to Register

1 0 0 0 1 0 1 w	mod reg r/m

Immediate to Register/Memory

1 1 0 0 0 1 1 w	mod 0 0 0 r/m	8-, 16-, or 32-bit data

Immediate to Register (Short Form)

1 0 1 1 w reg	8-, 16-, or 32-bit data

Memory to Accumulator (Short Form)

1 0 1 0 0 0 0 w	full 16- or 32-bit displacement

Accumulator to Memory (Short Form)

1 0 1 0 0 0 1 w	full 16- or 32-bit displacement

Register/Memory to Segment Register

1 0 0 0 1 1 1 0	mod sreg3 r/m

Segment Register to Register/Memory

1 0 0 0 1 1 0 0	mod sreg3 r/m

MOVSX = Move with Sign Extension

Register from Register/Memory

0 0 0 0 1 1 1 1	1 0 1 1 1 1 1 w	mod reg r/m

MOVZX = Move with Zero Extension

Register from Register/Memory

0 0 0 0 1 1 1 1	1 0 1 1 0 1 1 w	mod reg r/m

PUSH = Push

Register/Memory

1 1 1 1 1 1 1 1	mod 1 1 0 r/m

Register (Short Form)

0 1 0 1 0 reg

Segment Register (ES, CS, SS, or DS) Short Form

0 0 0 sreg2 1 1 0

Segment Register (FS or GS)

0 0 0 0 1 1 1 1	1 0 sreg3 0 0 0

Immediate

0 1 1 0 1 0 s 0	8-, 16-, or 32-bit data

PUSHA = Push All

0 1 1 0 0 0 0 0

POP = Pop

Register/Memory

1 0 0 0 1 1 1 1	mod 0 0 0 r/m

Register (Short Form)

0 1 0 1 1 reg

Segment Register (ES, SS, or DS) Short Form

0 0 0 sreg2 1 1 1

Segment Register (FS or GS)

0 0 0 0 1 1 1 1	1 0 sreg3 0 0 1

POPA = Pop All

0 1 1 0 0 0 0 1

XCHG = Exchange

Register/Memory with Register

1 0 0 0 0 1 1 w	mod reg r/m

Register with Accumulator (Short Form)

1 0 0 1 0 reg

IN = Input From:

Fixed Port

1 1 1 0 0 1 0 w	port number

Variable Port

1 1 1 0 1 1 0 w

OUT = Output To:

Fixed Port

1 1 1 0 0 1 1 w	port number

Variable Port

1 1 1 0 1 1 1 w

LEA = Load EA to Register

1 0 0 0 1 1 0 1	mod reg r/m

Segment Control

LDS = Load Pointer to DS

1 1 0 0 0 1 0 1	mod reg r/m

LES = Load Pointer to ES

1 1 0 0 0 1 0 0	mod reg r/m

LFS = Load Pointer to FS

0 0 0 0 1 1 1 1	1 0 1 1 0 1 0 0	mod reg r/m

LGS = Load Pointer to GS

0 0 0 0 1 1 1 1	1 0 1 1 0 1 0 1	mod reg r/m

LSS = Load Pointer to SS

0 0 0 0 1 1 1 1	1 0 1 1 0 0 1 0	mod reg r/m

Flag Control

CLC = Clear Carry Flag

11111000

CLD = Clear Direction Flag

11111100

CLI = Clear Interrupt Enable Flag

11111010

CLTS = Clear Task Switched Flag

00001111	00000110

CMC = Complement Carry Flag

11110101

LAHF = Load AH Into Flag

10011111

POPF = Pop Flags

10011101

PUSHF = Push Flags

10011100

SAHF = Store AH Into Flags

1 0 0 1 1 1 1 0

STC = Set Carry Flag

1 1 1 1 1 0 0 1

STD = Set Direction Flag

1 1 1 1 1 1 0 1

STI = Set Interrupt Enable Flag

1 1 1 1 1 0 1 1

Arithmetic

ADD = Add

Register to Register

0 0 0 0 0 0 d w	mod reg r/m

Register to Memory

0 0 0 0 0 0 0 w	mod reg r/m

Memory to Register

0 0 0 0 0 0 1 w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 s w	mod 0 0 0 r/m	8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

0 0 0 0 0 1 0 w	8-, 16-, or 32-bit data

ADC = Add with Carry

Register to Register

0 0 0 1 0 0 d w	mod reg r/m

Register to Memory

0 0 0 1 0 0 0 w	mod reg r/m

Memory to Register

0 0 0 1 0 0 1 w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 s w	mod 0 1 0 r/m	8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

0 0 0 1 0 1 0 w	8-, 16-, or 32-bit data

INC = Increment

Register/Memory

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

Register (Short Form)

0 1 0 0 0 reg

SUB = Subtract

Register from Register

0 0 1 0 1 0 d w	mod reg r/m

Register from Memory

0 0 1 0 1 0 0 w	mod reg r/m

Memory from Register

0 0 1 0 1 0 1 w	mod reg r/m

Immediate from Register/Memory

1 0 0 0 0 0 s w	mod 1 0 1 r/m	8-, 16-, or 32-bit data

Immediate from Accumulator (Short Form)

0 0 1 0 1 1 0 w	8-, 16-, or 32-bit data

SBB = Subtract with Borrow

Register from Register

0 0 0 1 1 0 d w	mod reg r/m

Register from Memory

0 0 0 1 1 0 0 w	mod reg r/m

Memory from Register

0 0 0 1 1 0 1 w	mod reg r/m

Immediate from Register/Memory

1 0 0 0 0 0 s w	mod 0 1 1 r/m	8-, 16-, or 32-bit data

Immediate from Accumulator (Short Form)

0 0 0 1 1 1 0 w	8-, 16-, or 32-bit data

DEC = Decrement

Register/Memory

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

Register (Short Form)

0 1 0 0 1 reg

CMP = Compare

Register with Register

0 0 1 1 1 0 d w	mod reg r/m

Memory with Register

0 0 1 1 1 0 0 w	mod reg r/m

Register with Memory

0 0 1 1 1 0 1 w	mod reg r/m

Immediate with Register/Memory

1 0 0 0 0 0 s w	mod 1 1 1 r/m	8-, 16-, or 32-bit data

Immediate with Accumulator (Short Form)

0 0 1 1 1 1 0 w	8-, 16-, or 32-bit data

NEG = Change Sign

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

AAA = ASCII Adjust for Add

0 0 1 1 0 1 1 1

AAS = ASCII Adjust for Subtract

0 0 1 1 1 1 1 1

DAA = Decimal Adjust for Add

0 0 1 0 0 1 1 1

DAS = Decimal Adjust for Subtract

0 0 1 0 1 1 1 1

MUL = Multiply (Unsigned)

Accumulator with Register/Memory

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

IMUL = Integer Multiply (Signed)

Accumulator with Register/Memory

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

Register with Register/Memory

0 0 0 0 1 1 1 1	1 0 1 0 1 1 1 1	mod reg r/m

Register/Memory with Immediate to Register

0 1 1 0 1 0 s 1	mod reg r/m	8-, 16-, or 32-bit data

DIV = Divide (Unsigned)

Accumulator by Register/Memory

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

IDIV = Integer Divide (Signed)

Accumulator by Register/Memory

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

AAD = ASCII Adjust for Divide

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

AAM = ASCII Adjust for Multiply

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

CBW = Convert Byte to Word

1 0 0 1 1 0 0 0

CWD = Convert Word to Doubleword

1 0 0 1 1 0 0 1

Logic

Shift/Rotate Instructions Not Through Carry (ROL, ROR, SAL, SAR, SHL, and SHR)

Register/Memory by 1

1 1 0 1 0 0 0 w	mod T T T r/m

Register/Memory by CL

1 1 0 1 0 0 1 w	mod T T T r/m

Register/Memory by Immediate Count

1 1 0 0 0 0 0 w	mod T T T r/m	8-bit data

Shift/Rotate Instructions Through Carry (RCL and RCR)

Register/Memory by 1

1 1 0 1 0 0 0 w	mod T T T r/m

Register/Memory by CL

1 1 0 1 0 0 1 w	mod T T T r/m

Register/Memory by Immediate Count

1 1 0 0 0 0 0 w	mod T T T r/m	8-bit data

T T T	Instruction
0 0 0	ROL
0 0 1	ROR
0 1 0	RCL
0 1 1	RCR
1 0 0	SHL/SAL
1 0 1	SHR
1 1 1	SAR

SHLD = Shift Left Double

Register/Memory by Immediate

0 0 0 0 1 1 1 1	1 0 1 0 0 1 0 0	mod reg r/m	8-bit data

Register/Memory by CL

0 0 0 0 1 1 1 1	1 0 1 0 0 1 0 1	mod reg r/m

SHRD = Shift Right Double

Register/Memory by Immediate

0 0 0 0 1 1 1 1	1 0 1 0 1 1 0 0	mod reg r/m	8-bit data

Register/Memory by CL

0 0 0 0 1 1 1 1	1 0 1 0 1 1 0 1	mod reg r/m

AND = And

Register to Register

0 0 1 0 0 0 d w	mod reg r/m

Register to Memory

0 0 1 0 0 0 0 w	mod reg r/m

Memory to Register

0 0 1 0 0 0 1 w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 s w	mod 1 0 0 r/m	8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

0 0 1 0 0 1 0 w	8-, 16-, or 32-bit data

TEST = AND Function to Flags; No Result

Register/Memory and Register

1 0 0 0 0 1 0 w	mod reg r/m

Immediate Data and Register/Memory

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	8-, 16-, or 32-bit data

Immediate Data and Accumulator (Short Form)

1 0 1 0 1 0 0 w	8-, 16-, or 32-bit data

OR = Or

Register to Register

0 0 0 0 1 0 d w	mod reg r/m

Register to Memory

0 0 0 0 1 0 0 w	mod reg r/m

Memory to Register

0 0 0 0 1 0 1 w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 s w	mod 0 0 1 r/m	8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

0 0 0 0 1 1 0 w	8-, 16-, or 32-bit data

XOR = Exclusive OR

Register to Register

0 0 1 1 0 0 d w	mod reg r/m

Register to Memory

0 0 1 1 0 0 0 w	mod reg r/m

Memory to Register

0 0 1 1 0 0 1 w	mod reg r/m

Immediate to Register/Memory

1 0 0 0 0 0 s w	mod 1 1 0 r/m	8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

0 0 1 1 0 1 0 w	8-, 16-, or 32-bit data

NOT = Invert Register/Memory

1 1 1 1 0 1 1 w	mod 0 1 0 r/m

String Manipulation

CMPS = Compare Byte Word

1 0 1 0 0 1 1 w

INS = Input Byte/Word from DX Port

0 1 1 0 1 1 0 w

LODS = Load Byte/Word to AL/AX/EAX

1010110w

MOVS = Move Byte Word

1010010w

OUTS = Output Byte/Word to DX Port

0110111w

SCAS = Scan Byte Word

1010111w

STOS = Store Byte/Word from AL/AX/EX

1010101w

XLAT = Translate String

11010111

Repeated String Manipulation

Repeated by Count in CX or ECX

REPE CMPS = Compare String (Find Non-Match)

11110011	1010011w

REPNE CMPS = Compare String (Find Match)

11110010	1010011w

REP INS = Input String

11110010	0110110w

REP LODS = Load String

11110010	1010110w

REP MOVS = Move String

11110010	1010010w

REP OUTS = Output String

11110010	0110111w

REPE SCAS = Scan String (Find Non-AL/AX/EAX)

11110011	1010111w

REPNE SCAS = Scan String (Find AL/AX/EAX)

11110010	1010111w

REP STOS = Store String

11110010	1010101w

Bit Manipulation

BSF = Scan Bit Forward

0 0 0 0 1 1 1 1	1 0 1 1 1 1 0 0	mod reg r/m

BSR = Scan Bit Reverse

0 0 0 0 1 1 1 1	1 0 1 1 1 1 0 1	mod reg r/m

BT = Test Bit

Register/Memory, Immediate

0 0 0 0 1 1 1 1	1 0 1 1 1 0 1 0	mod 1 0 0 r/m	8-bit data

Register/Memory, Register

0 0 0 0 1 1 1 1	1 0 1 0 0 0 1 1	mod reg r/m

BTC = Test Bit and Complement

Register/Memory, Immediate

0 0 0 0 1 1 1 1	1 0 1 1 1 0 1 0	mod 1 1 1 r/m	8-bit data

Register/Memory, Register

0 0 0 0 1 1 1 1	1 0 1 1 1 0 1 1	mod reg r/m

BTR = Test Bit and Reset

Register/Memory, Immediate

0 0 0 0 1 1 1 1	1 0 1 1 1 0 1 0	mod 1 1 0 r/m	8-bit data

Register/Memory, Register

0 0 0 0 1 1 1 1	1 0 1 1 0 0 1 1	mod reg r/m

BTS = Test Bit and Set

Register/Memory, Immediate

0 0 0 0 1 1 1 1	1 0 1 1 1 0 1 0	mod 1 0 1 r/m	8-bit data

Register/Memory, Register

0 0 0 0 1 1 1 1	1 0 1 0 1 0 1 1	mod reg r/m

Control Transfer

CALL = Call

Direct within Segment

1 1 1 0 1 0 0 0	full 16- or 32-bit displacement

Register/Memory Indirect within Segment

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

Direct Intersegment

1 0 0 1 1 0 1 0	offset, selector

Indirect Intersegment

1 1 1 1 1 1 1 1	mod 0 1 1 r/m

JMP = Unconditional Jump

Short

1 1 1 0 1 0 1 1	8-bit disp.

Direct within Segment

1 1 1 0 1 0 0 1	full 16- or 32-bit displacement

Register/Memory Indirect within Segment

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

Direct Intersegment

1 1 1 0 1 0 1 0	offset, selector

Indirect Intersegment

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

RET = Return from Call

Within Segment

1 1 0 0 0 0 1 1

Within Segment Adding Immediate to SP

1 1 0 0 0 0 1 0	16-bit displacement

Intersegment

1 1 0 0 1 0 1 1

Intersegment Adding Immediate to SP

1 1 0 0 1 0 1 0	16-bit displacement

Conditional Jumps

JO = Jump on Overflow

8-Bit Displacement

0 1 1 1 0 0 0 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 0	full 16- or 32-bit displacement

JNO = Jump on Not Overflow

8-Bit Displacement

0 1 1 1 0 0 0 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 1	full 16- or 32-bit displacement

JB/JNAE = Jump on Below/Not Above or Equal

8-Bit Displacement

0 1 1 1 0 0 1 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 0	full 16- or 32-bit displacement

JNB/JAE = Jump on Not Below/Above or Equal

8-Bit Displacement

0 1 1 1 0 0 1 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 1	full 16- or 32-bit displacement

JE/JZ = Jump on Equal/Zero

8-Bit Displacement

0 1 1 1 0 1 0 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 0	full 16- or 32-bit displacement

JNE/JNZ = Jump on Not Equal/Not Zero

8-Bit Displacement

0 1 1 1 0 1 0 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 1	full 16- or 32-bit displacement

JBE/JNA = Jump on Below or Equal/Not Above

8-Bit Displacement

0 1 1 1 0 1 1 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 0	full 16- or 32-bit displacement

JNBE/JA = Jump on Not Below or Equal/Above

8-Bit Displacement

0 1 1 1 0 1 1 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 1	full 16- or 32-bit displacement

JS = Jump on Sign

8-Bit Displacement

0 1 1 1 1 0 0 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 0 0 0	full 16- or 32-bit displacement

JNS = Jump on Not Sign

8-Bit Displacement

0 1 1 1 1 0 0 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 0 0 1	full 16- or 32-bit displacement

JP/JPE = Jump on Parity/Parity Even

8-Bit Displacement

0 1 1 1 1 0 1 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 0 1 0	full 16- or 32-bit displacement

JNP/JPO = Jump on Not Parity/Parity Odd

8-Bit Displacement

0 1 1 1 1 0 1 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 0 1 1	full 16- or 32-bit displacement

JL/JNGE = Jump on Less/Not Greater or Equal

8-Bit Displacement

0 1 1 1 1 1 0 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 1 0 0	full 16- or 32-bit displacement

JNL/JGE = Jump on Not Less/Greater or Equal

8-Bit Displacement

0 1 1 1 1 1 0 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 1 0 1	full 16- or 32-bit displacement

JLE/JNG = Jump on Less or Equal/Not Greater

8-Bit Displacement

0 1 1 1 1 1 1 0	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 1 1 0	full 16- or 32-bit displacement

JNLE/JG = Jump on Not Less or Equal/Greater

8-Bit Displacement

0 1 1 1 1 1 1 1	8-bit disp.

Full Displacement

0 0 0 0 1 1 1 1	1 0 0 0 1 1 1 1	full 16- or 32-bit displacement

JCXZ = Jump on CX Zero

1 1 1 0 0 0 1 1	8-bit disp.

JECXZ = Jump on ECX Zero

1 1 1 0 0 0 1 1	8-bit disp.

Note: The operand size prefix differentiates JCXZ from JECXZ.

LOOP = Loop CX Times

1 1 1 0 0 0 1 0	8-bit disp.

LOOPZ/LOOPE = Loop with Zero/Equal

1 1 1 0 0 0 0 1	8-bit disp.

LOOPNZ/LOOPNE = Loop while Not Zero

1 1 1 0 0 0 0 0	8-bit disp.

Conditional Byte Set

SETO = Set Byte on Overflow

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 0	mod 0 0 0 r/m

SETNO = Set Byte on Not Overflow

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 1	mod 0 0 0 r/m

SETB/SETNAE = Set Byte on Below/Not Above or Equal

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 0 1 0	mod 0 0 0 r/m

SETNB = Set Byte on Not Below/Above or Equal

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 0 1 1	mod 0 0 0 r/m

SETE/SETZ = Set Byte on Equal/Zero

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 1 0 0	mod 0 0 0 r/m

SETNE/SETNZ = Set Byte on Not Equal/Not Zero

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 1 0 1	mod 0 0 0 r/m

SETBE/SETNA = Set Byte on Below or Equal/Not Above

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 1 1 0	mod 0 0 0 r/m

SETNBE/SETA = Set Byte on Not Below or Equal/Above

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 0 1 1 1	mod 0 0 0 r/m

SETS = Set Byte on Sign

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 0 0 0	mod 0 0 0 r/m

SETNS = Set Byte on Not Sign

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 0 0 1	mod 0 0 0 r/m

SETP/SETPE = Set Byte on Parity/Parity Even

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 0 1 0	mod 0 0 0 r/m

SETNP/SETPO = Set Byte on Not Parity/Parity Odd

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 0 1 1	mod 0 0 0 r/m

SETL/SETNGE = Set Byte on Less/Not Greater or Equal

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 1 0 0	mod 0 0 0 r/m

SETNL/SETGE = Set Byte on Not Less/Greater or Equal

To Register/Memory

0 0 0 0 1 1 1 1	0 1 1 1 1 1 0 1	mod 0 0 0 r/m

SETLE/SETNG = Set Byte on Less or Equal/Not Greater

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 1 1 0	mod 0 0 0 r/m

SETNLE/SETG = Set Byte on Not Less or Equal/Greater

To Register/Memory

0 0 0 0 1 1 1 1	1 0 0 1 1 1 1 1	mod 0 0 0 r/m

ENTER = Enter Procedure

1 1 0 0 1 0 0 0	16-bit displacement	8-bit level

LEAVE = Leave Procedure

1 1 0 0 1 0 0 1

Interrupt Instructions

INT = Interrupt

Type Specified

1 1 0 0 1 1 0 1	type

Type 3

1 1 0 0 1 1 0 0

INTO = Interrupt 4 If Overflow Flag Set

1 1 0 0 1 1 1 0

BOUND = Interrupt 5 If Detect Value Out of Range

0 1 1 0 0 0 1 0	mod reg r/m

IRET = Interrupt Return

1 1 0 0 1 1 1 1

Processor Control

HLT = Halt

1 1 1 1 0 1 0 0

MOV = Move to and from Control/Debug/Test Registers

CR0/CR2/CR3 from Register

0 0 0 0 1 1 1 1	0 0 1 0 0 0 1 0	1 1 eee reg

Register from CR0-3

0 0 0 0 1 1 1 1	0 0 1 0 0 0 0 0	1 1 eee reg

DR0-3, DR6-7 from Register

0 0 0 0 1 1 1 1	0 0 1 0 0 0 1 1	1 1 eee reg

Register from DR0-3, DR6-7

0 0 0 0 1 1 1 1	0 0 1 0 0 0 0 1	1 1 eee reg

TR6-7 from Register

0 0 0 0 1 1 1 1	0 0 1 0 0 1 1 0	1 1 eee reg

Register from TR6-7

0 0 0 0 1 1 1 1	0 0 1 0 0 1 0 0	1 1 eee reg

NOP = No Operation

1 0 0 1 0 0 0 0

WAIT = Wait until BUSY Pin is Negated

10011011

Processor Extension

ESC = Processor Extension Escape

11011TTT	mod L L L r/m

Note: TTT and LLL bits are opcode information for the coprocessor.

Prefix Bytes

Address Size Prefix

01100111

Operand Size Prefix

01100110

LOCK = Bus Lock Prefix

11110000

Note: The use of LOCK is restricted to an exchange with memory, or bit test and reset type of instruction.

Segment Override Prefix

CS:

00101110

DS:

00111110

ES:

0 0 1 0 0 1 1 0

FS:

0 1 1 0 0 1 0 0

GS:

0 1 1 0 0 1 0 1

SS:

0 0 1 1 0 1 1 0

Protection Control

ARPL = Adjust Requested Privilege Level from Register/Memory

0 1 1 0 0 0 1 1	mod reg r/m

LAR = Load Access Rights from Register/Memory

0 0 0 0 1 1 1 1	0 0 0 0 0 0 1 0	mod reg r/m

LGDT = Load Global Descriptor Table Register

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod 0 1 0 r/m

LIDT = Load Interrupt Descriptor Table Register

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod 0 1 1 r/m

LLDT = Load Local Descriptor Table Register to Register/Memory

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0	mod 0 1 0 r/m

LMSW = Load Machine Status Word from Register/Memory

00001111	00000001	mod 1 1 0 r/m

LSL = Load Segment Limit from Register/Memory

00001111	00000011	mod reg r/m

LTR = Load Task Register from Register/Memory

00001111	00000000	mod 0 0 1 r/m

SGDT = Store Global Descriptor Table Register

00001111	00000001	mod 0 0 0 r/m

SIDT = Store Interrupt Descriptor Table Register

00001111	00000001	mod 0 0 1 r/m

SLDT = Store Local Descriptor Table Register to Register/Memory

00001111	00000000	mod 0 0 0 r/m

SMSW = Store Machine Status Word

00001111	00000001	mod 1 0 0 r/m

STR = Store Task Register to Register/Memory

00001111	00000000	mod 0 0 1 r/m

VERR = Verify Read Access; Register/Memory

00001111	00000000	mod 1 0 0 r/m

VERW = Verify Write Access

00001111	00000000	mod 1 0 1 r/m

Introduction to the 80387 Instruction Set

The 80387 instructions use many of the same fields defined earlier in this section for the 80386 instructions. Additional fields used by the 80387 instructions are defined in the following figure.

Field	Description	Bit Information
escape	80386 Extension Escape	Bit Pattern = 11011
MF	Memory Format	00 = 32-bit Real 01 = 32-bit integer 10 = 64-bit Real 11 = 16-bit integer
ST(0) ST(i)	Current Stack Top ith register below the stack top	
d	Destination	0 = Destination is ST(0) 1 = Destination is ST(i)
P	Pop	0 = No pop 1 = Pop ST(0)
R	Reverse*	0 = Destination (op) source 1 = Source (op) destination
* When d=1, reverse the sense of R.		

Figure 27. 80387 Encoding Field Summary

Within the 80387 Instruction Set:

- Temporary (Extended) Real is 80-bit Real.
- Long Integer is a 64-bit integer.

80387 Usage of the Scale-Index-Base Byte

The "mod r/m" byte of an 80387 instruction can be followed by a scale-index-base (s-i-b) byte having the same address mode definition as in the 80386 instruction. The mod field in the 80387 instruction is never equal to 11.

Instruction and Data Pointers

The parallel operation of the 80386 and 80387 may allow errors detected by the 80387 to be reported after the 80386 has executed the ESC instruction that caused the error. The 80386/80387 provides two pointer registers to identify the failing numeric instruction. The pointer registers supply the address of the failing numeric instruction and the address of its numeric memory operand when applicable.

Although the pointer registers are located in the 80386, they appear to be located in the 80387 because they are accessed by the ESC instructions FLDENV, FSTENV, FSAVE, and FRSTOR. Whenever the 80386 decodes a new ESC instruction, it saves the address of the instruction along with any prefix bytes that may be present, the address of the operand (if present), and the opcode.

The instruction and data pointers appear in one of four available formats:

- 16-bit Real Mode/Virtual 8086 Mode
- 32-bit Real Mode
- 16-bit Protected Mode
- 32-bit Protected Mode

The Real Mode formats are used whenever the 80386 is in the Real Mode or Virtual 8086 Mode. The Protected Mode formats are used when the 80386 is in the Protected Mode. The Operand Size Prefix can also be used with the 80387 instructions. The operand size of the 80387 instruction determines whether the 16-bit or 32-bit format is used.

Note: FSAVE and FRSTOR have an additional eight fields (10 bytes per field) that contain the current contents of ST(0) through ST(7). These fields follow the instruction and data pointer image shown in the following figures.

The following figures show the instruction and data pointer image format used in the various address modes. The ESC instructions FLDENV, FSTENV, FSAVE, and FRSTOR are used to transfer these values between the 80386/80387 registers and memory.

Bits

15		8 | 7 … 0	Offset in Block
Control Word			0
Status Word			2
Tag Word			4
Instruction Pointer (IP) Bits 15-0			6
IP Bits 19-16	0	Opcode Bits 10-0	8
Operand Pointer (OP) Bits 15-0			A
OP Bits 19-16	0	0 0 0 0 0 0 0 0 0 0 0	C

Figure 28. Instruction and Pointer Image (16-Bit Real Address Mode)

Bits

<table>
<tr><th>15 — 8</th><th>7 — 0</th><th>Offset in Block</th></tr>
<tr><td colspan="2">Control Word</td><td>0</td></tr>
<tr><td colspan="2">Status Word</td><td>2</td></tr>
<tr><td colspan="2">Tag Word</td><td>4</td></tr>
<tr><td colspan="2">Instruction Pointer Offset</td><td>6</td></tr>
<tr><td colspan="2">CS Selector</td><td>8</td></tr>
<tr><td colspan="2">Operand Offset</td><td>A</td></tr>
<tr><td colspan="2">Operand Selector</td><td>C</td></tr>
</table>

Figure 29. Instruction and Pointer Image (16-Bit Protected Mode)

Bits

<table>
<tr><th colspan="2">31 — 24 | 23 — 16</th><th colspan="3">15 — 8 | 7 — 0</th><th>Offset in Block</th></tr>
<tr><td colspan="2">Reserved</td><td colspan="3">Control Word</td><td>0</td></tr>
<tr><td colspan="2">Reserved</td><td colspan="3">Status Word</td><td>4</td></tr>
<tr><td colspan="2">Reserved</td><td colspan="3">Tag Word</td><td>8</td></tr>
<tr><td colspan="2">Reserved</td><td colspan="3">IP Bits 15-0</td><td>C</td></tr>
<tr><td>0 0 0 0</td><td colspan="2">IP Bits 31-16</td><td>0</td><td>Opcode Bits 10-0</td><td>10</td></tr>
<tr><td colspan="2">Reserved</td><td colspan="3">Operand Pointer Bits 15-0</td><td>14</td></tr>
<tr><td>0 0 0 0</td><td colspan="2">Operand Pointer Bits 31-16</td><td colspan="2">0 0 0 0 0 0 0 0 0 0 0 0</td><td>18</td></tr>
</table>

Figure 30. Instruction and Pointer Image (32-Bit Real Address Mode)

Bits

<table>
<tr><th>31 — 24 | 23 — 16</th><th>15 — 8 | 7 — 0</th><th>Offset in Block</th></tr>
<tr><td>Reserved</td><td>Control Word</td><td>0</td></tr>
<tr><td>Reserved</td><td>Status Word</td><td>4</td></tr>
<tr><td>Reserved</td><td>Tag Word</td><td>8</td></tr>
<tr><td colspan="2">Instruction Pointer Offset</td><td>C</td></tr>
<tr><td>Reserved</td><td>CS Selector</td><td>10</td></tr>
<tr><td colspan="2">Data Operand Offset</td><td>14</td></tr>
<tr><td>Reserved</td><td>Operand Selector</td><td>18</td></tr>
</table>

Figure 31. Instruction and Pointer Image (32-Bit Protected Mode)

New Instructions

Several new instructions are included in the 80387 instruction set that are not available to the 80287 or 8087 Math Coprocessors. The new instructions are:

FUCOM (Unordered Compare Real)
FUCOMP (Unordered Compare Real and Pop)
FUCOMPP (Unordered Compare Real and Pop Twice)
FPREM1 (IEEE Partial Remainder)
FSINE (Sine)
FCOS (Cosine)
FSINCOS (Sine and Cosine).

80387 Math Coprocessor Instruction Set

The following is an instruction set summary for the 80387 coprocessor. In the following, the bit pattern for escape is 11011.

Data Transfer

FLD = Load

Integer/Real Memory to ST(0)

escape MF 1	mod 0 0 0 r/m

Long Integer Memory to ST(0)

escape 1 1 1	mod 1 0 1 r/m

Temporary Real Memory to ST(0)

escape 0 1 1	mod 1 0 1 r/m

BCD Memory to ST(0)

escape 1 1 1	mod 1 0 0 r/m

ST(i) to ST(0)

escape 0 0 1	1 1 0 0 0 ST(i)

FST = Store

ST(0) to Integer/Real Memory

escape MF 1	mod 0 1 0 r/m

ST(0) to ST(i)

escape 1 0 1	1 1 0 1 0 ST(i)

FSTP = Store and Pop

ST(0) to Integer/Real Memory

escape MF 1	mod 0 1 1 r/m

ST(0) to Long Integer Memory

escape 1 1 1	mod 1 1 1 r/m

ST(0) to Temporary Real Memory

escape 0 1 1	mod 1 1 1 r/m

ST(0) to BCD Memory

escape 1 1 1	mod 1 1 0 r/m

ST(0) to ST(i)

escape 1 0 1	1 1 0 1 1 ST(i)

FXCH = Exchange ST(i) and ST(0)

escape 0 0 1	1 1 0 0 1 ST(i)

Comparison

FCOM = Compare

Integer/Real Memory to ST(0)

escape MF 0	mod 0 1 0 r/m

ST(i) to ST(0)

escape 0 0 0	1 1 0 1 0 ST(i)

FCOMP = Compare and Pop

Integer/Real Memory to ST(0)

escape MF 0	mod 0 1 1 r/m

ST(i) to ST(0)

escape 0 0 0	1 1 0 1 1 ST(i)

FCOMPP = Compare ST(1) to ST(0) and Pop Twice

escape 1 1 0	1 1 0 1 1 0 0 1

FUCOM = Unordered Compare Real

escape 1 0 1	1 1 1 0 0 ST(i)

FUCOMP = Unordered Compare Real and Pop

escape 1 0 1	1 1 1 0 1 ST(i)

FUCOMPP = Unordered Compare Real and Pop Twice

escape 0 1 0	1 1 1 0 1 0 0 1

FTST = Test ST(0)

escape 0 0 1	1 1 1 0 0 1 0 0

FXAM = Examine ST(0)

escape 0 0 1	1 1 1 0 0 1 0 1

Constants

FLDZ = Load +0.0 into ST(0)

escape 0 0 1	1 1 1 0 1 1 1 0

FLD1 = Load +1.0 into ST(0)

escape 0 0 1	1 1 1 0 1 0 0 0

FLDPI = Load π into ST(0)

escape 0 0 1	1 1 1 0 1 0 1 1

FLDL2T = Load $\log_2 10$ into ST(0)

escape 0 0 1	1 1 1 0 1 0 0 1

FLDL2E = Load $\log_2 e$ into ST(0)

escape 0 0 1	1 1 1 0 1 0 1 0

FLDLG2 = Load $\log_{10}$ 2 into ST(0)

escape 0 0 1	1 1 1 0 1 1 0 0

FLDLN2 = Load $\log_e$ 2 into ST(0)

escape 0 0 1	1 1 1 0 1 1 0 1

Arithmetic

FADD = Addition

Integer/Real Memory with ST(0)

escape MF 0	mod 0 0 0 r/m

ST(i) and ST(0)

escape d P 0	1 1 0 0 0 ST(i)

FSUB = Subtraction

Integer/Real Memory with ST(0)

escape MF 0	mod 1 0 R r/m

ST(i) and ST(0)

escape d P 0	1 1 1 0 R r/m

FMUL = Multiplication

Integer/Real Memory with ST(0)

escape MF 0	mod 0 0 1 r/m

ST(i) and ST(0)

escape d P 0	1 1 0 0 1 r/m

FDIV = Division

Integer/Real Memory with ST(0)

escape MF 0	mod 1 1 R r/m

ST(i) and ST(0)

escape d P 0	1 1 1 1 R r/m

FSQRT = Square Root of ST(0)

escape 0 0 1	1 1 1 1 1 0 1 0

FSCALE = Scale ST(0) by ST(1)

escape 0 0 1	1 1 1 1 1 1 0 1

FPREM = Partial Remainder of ST(0) ÷ ST(1)

escape 0 0 1	1 1 1 1 1 0 0 0

FPREM1 = IEEE Partial Remainder

escape 0 0 1	1 1 1 1 0 1 0 1

FRNDINT = Round ST(0) to Integer

escape 0 0 1	1 1 1 1 1 1 0 0

FXTRACT = Extract Components of ST(0)

escape 0 0 1	1 1 1 1 0 1 0 0

FABS = Absolute Value of ST(0)

escape 0 0 1	1 1 1 0 0 0 0 1

FCHS = Change Sign of ST(0)

escape 0 0 1	1 1 1 0 0 0 0 0

Transcendental

FPTAN = Partial Tangent of ST(0)

escape 0 0 1	1 1 1 1 0 0 1 0

FPATAN = Partial Arctangent of ST(1) ÷ ST(0)

escape 0 0 1	1 1 1 1 0 0 1 1

FSIN = Sine

escape 0 0 1	1 1 1 1 1 1 1 0

FCOS = Cosine

escape 0 0 1	1 1 1 1 1 1 1 1

FSINCOS = Sine and Cosine

escape 0 0 1	1 1 1 1 1 0 1 1

F2XM1 = $2^{ST(0)}$ -1

escape 0 0 1	1 1 1 1 0 0 0 0

FYL2X = ST(1) x Log_2 [ST(0)]

escape 0 0 1	1 1 1 1 0 0 0 1

FYL2XP1 = ST(1) x Log_2 [ST(0) + 1]

escape 0 0 1	1 1 1 1 1 0 0 1

Processor Control

FINIT = Initialize NPX

escape 0 1 1	1 1 1 0 0 0 1 1

FSTSW AX = Store Control Word

escape 1 1 1	1 1 1 0 0 0 0 0

FLDCW = Load Control Word

escape 0 0 1	mod 1 0 1 r/m

FSTCW = Store Control Word

escape 0 0 1	mod 1 1 1 r/m

FSTSW = Store Status Word

escape 1 0 1	mod 1 1 1 r/m

FCLEX = Clear Exceptions

escape 0 1 1	1 1 1 0 0 0 1 0

FSTENV = Store Environment

escape 0 0 1	mod 1 1 0 r/m

FLDENV = Load Environment

escape 0 0 1	mod 1 0 0 r/m

FSAVE = Save State

escape 1 0 1	mod 1 1 0 r/m

FRSTOR = Restore State

escape 1 0 1	mod 1 0 0 r/m

FINCSTP = Increment Stack Pointer

escape 0 0 1	1 1 1 1 0 1 1 1

FDECSTP = Decrement Stack Pointer

escape 0 0 1	1 1 1 1 0 1 1 0

FFREE = Free ST(I)

escape 1 0 1	1 1 0 0 0 ST(i)

FNOP = No Operation

escape 0 0 1	1 1 0 1 0 0 0 0

80486 Microprocessor Instruction Set

The 80486 microprocessor uses the same instruction set that the 80386 microprocessor and the 80387 Math Coprocessor. In addition, the 80486 has six unique instructions that control cache operation:

- Byte Swap (BSWAP)
- Compare and Exchange (CMPXCHG)
- Exchange-and-Add (XADD)
- Invalidate Data Cache (INVD)
- Invalidate TLBN Entry (INVLPG)
- Write-Back and Invalidate Data Cache (WBINVD).

BSWAP = Byte Swap

0 0 0 0 1 1 1 1	1 1 0 0 1 reg

CMPXCHG = Compare and Exchange

Register 1, Register 2

0 0 0 0 1 1 1 1	1 0 1 1 0 0 0 w	1 1 reg^2 reg^1

Memory, Register 2

0 0 0 0 1 1 1 1	1 0 1 1 0 0 0 w	mod reg^2 mem

XADD = Exchange and Add

Register 1, Register 2

0 0 0 0 1 1 1 1	1 1 0 0 0 0 0 w	1 1 reg2 reg1

Memory, Register 2

0 0 0 0 1 1 1 1	1 1 0 0 0 0 0 w	mod reg2 mem

INVD = Invalidate Data Cache

0 0 0 0 1 1 1 1	0 0 0 0 1 0 0 0

WBINVD = Write-Back and Invalidate Data Cache

0 0 0 0 1 1 1 1	0 0 0 0 1 0 0 1

INVLPG = Invalidate TLB Entry

0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 1	mod 1 1 mem

Notes: