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

© Copyright IBM Corp. 1990
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
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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Bits</th>
<th>Significant Digits (Decimal)</th>
<th>Approximate Range (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Integer</td>
<td>16</td>
<td>4</td>
<td>(-32,768 \leq x \leq +32,767)</td>
</tr>
<tr>
<td>Short Integer</td>
<td>32</td>
<td>9</td>
<td>(-2 \times 10^9 \leq x \leq +2 \times 10^9)</td>
</tr>
<tr>
<td>Long Integer</td>
<td>64</td>
<td>19</td>
<td>(-9 \times 10^{18} \leq x \leq +9 \times 10^{18})</td>
</tr>
<tr>
<td>Packed Decimal</td>
<td>80</td>
<td>18</td>
<td>(-9.99 \leq x \leq +9.99) (18 digits)</td>
</tr>
<tr>
<td>Short Real *</td>
<td>32</td>
<td>6 - 7</td>
<td>(8.43 \times 10^{-37} \leq x \leq 3.37 \times 10^{38})</td>
</tr>
<tr>
<td>Long Real *</td>
<td>64</td>
<td>15 - 16</td>
<td>(4.19 \times 10^{-307} \leq x \leq 1.67 \times 10^{308})</td>
</tr>
<tr>
<td>Temporary Real **</td>
<td>80</td>
<td>19</td>
<td>(3.4 \times 10^{-4932} \leq x \leq 1.2 \times 10^{4932})</td>
</tr>
</tbody>
</table>

* 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.

![80386 Addressing Diagram](image)

*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.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Size</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>16, 32, 64 Bits</td>
<td>Load, Store, Compare, Add, Subtract, Multiply, Divide</td>
</tr>
<tr>
<td>Packed BCD*</td>
<td>80 Bits</td>
<td>Load, Store</td>
</tr>
<tr>
<td>Real</td>
<td>32, 64 Bits</td>
<td>Load, Store, Compare, Add, Subtract, Multiply, Divide</td>
</tr>
<tr>
<td>Temporary Real</td>
<td>80 Bits</td>
<td>Add, Subtract, Multiply, Divide, Square Root, Scale, Remainder, Integer Part, Change Sign, Absolute Value, Extract Exponent and Significand, Compare, Examine, Test, Exchange Tangent, Arctangent, $2^x - 1$, $Y^{<em>}\log_2 (X + 1)$, $Y^{</em>}\log_2 (X)$, Load Constant (0.0, $\pi$, etc.), Sine, Cosine, Unordered Compare</td>
</tr>
</tbody>
</table>

* 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.
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Integer</td>
<td>$10^4$</td>
<td>16 Bits</td>
</tr>
<tr>
<td>Short Integer</td>
<td>$10^9$</td>
<td>32 Bits</td>
</tr>
<tr>
<td>Long Integer</td>
<td>$10^{19}$</td>
<td>64 Bits</td>
</tr>
<tr>
<td>Packed BCD</td>
<td>$10^{18}$</td>
<td>18 Digits (2 digits per byte)</td>
</tr>
<tr>
<td>Single Precision</td>
<td>$10^{\pm 38}$</td>
<td>24 Bits</td>
</tr>
<tr>
<td>(Short Real)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Precision</td>
<td>$10^{\pm 308}$</td>
<td>53 Bits</td>
</tr>
<tr>
<td>(Long Real)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Precision</td>
<td>$10^{\pm 4932}$</td>
<td>64 Bits</td>
</tr>
</tbody>
</table>

*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):
<table>
<thead>
<tr>
<th>Bit 30 CD</th>
<th>Bit 29 NW</th>
<th>Operating Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Cache fills disabled, write-through and invalidate disabled</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cache fills disabled, write-through and invalidate enabled</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Cache fills enabled, write-through and invalidate enabled (Normal operating mode)</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>U/S</th>
<th>R/W</th>
<th>WP</th>
<th>User Access</th>
<th>Supervisor Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>None</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>None</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Read/Execute</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Read/Write/Execute</td>
<td>Read/Write/Execute</td>
</tr>
</tbody>
</table>

Figure 7. 80386 Compatible Operation

The new protection feature is given by the following table.

<table>
<thead>
<tr>
<th>U/S</th>
<th>R/W</th>
<th>WP</th>
<th>User Access</th>
<th>Supervisor Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>None</td>
<td>Read/Execute</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>Read/Write/Execute</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Read/Execute</td>
<td>Read/Execute</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Read/Write/Execute</td>
<td>Read/Write/Execute</td>
</tr>
</tbody>
</table>

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 0 mod 0 reg r/m
```

Segment Register to Register/Memory

```
1 0 0 0 1 1 0 mod 0 reg r/m
```
PUSH — Push

Memory

| 11111111 | mod 110 r/w |

Register

| 01010 reg |

Segment Register

| 000 reg 110 |

Immediate

| 011010 s 0 | data | data if s = 0 |

PUSHA — Push All

| 01100000 |

POP — Pop

Register/Memory

| 10001111 | mod 000 r/m |

Register

| 01011 reg |

Segment Register

| 000 reg 111 | reg ≠ 01 |
**POPA = Pop All**

```
01100001
```  

**XCHG = Exchange**

Register/Memory with Register

```
1000011 w  mod reg r/m
```  

Register with Accumulator

```
10010 reg
```  

**IN = Input From**

Fixed Port

```
1110010 w  port
```  

Variable Port

```
1110110 w
```  

**OUT = Output To**

Fixed Port

```
1110011 w  port
```  

Variable Port

```
1110111 w
```  

**XLAT = Translate Byte to AL**

```
11010111
```
LEA = Load EA to Register

\[ \begin{array}{c}
10001101 \\
\text{mod reg r/m}
\end{array} \]

LDS = Load Pointer to DS

\[ \begin{array}{c}
11000101 \\
\text{mod reg r/m mod \neq 11}
\end{array} \]

LES = Load Pointer to ES

\[ \begin{array}{c}
11000100 \\
\text{mod reg r/m mod \neq 11}
\end{array} \]

LAHF = Load AH with Flags

\[ \begin{array}{c}
10011111
\end{array} \]

SAHF = Store AH with Flags

\[ \begin{array}{c}
10011110
\end{array} \]

PUSHF = Push Flags

\[ \begin{array}{c}
10011100
\end{array} \]

POPF = Pop Flags

\[ \begin{array}{c}
10011101
\end{array} \]
# Arithmetic

**ADD** = Add

Register/Memory with Register to Either

| 000000 dw | mod reg r/m |

Immediate to Register/Memory

| 100000 sw | mod 000 r/m | data | data if sw = 0 1 |

Immediate to Accumulator

| 0000010 w | data | data if w = 1 |

**ADC** = Add with Carry

Register/Memory with Register to Either

| 000100 dw | mod reg r/m |

Immediate to Register/Memory

| 100000 sw | mod 010 r/m | data | data if sw = 0 1 |

Immediate to Accumulator

| 0001010 w | data | data if w = 1 |

**INC** = Increment

Register/Memory

| 1111111 w | mod 000 r/m |

Register

| 01000 reg |
**SUB = Subtract**

Register/Memory with Register to Either

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

Immediate from Register/Memory

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

Immediate from Register/Memory

| 1 0 0 0 0 0      | sw | 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      | w  | mod 0 0 1 r/m |

Register

| 0 1 0 0 1        | reg |
CMP = Compare

Register/Memory with Register

\[ 0011101w \quad \text{mod reg } r/m \]

Register with Register/Memory

\[ 0011100w \quad \text{mod reg } r/m \]

Immediate with Register/Memory

\[ 100000sw \quad \text{mod } 111r/m \quad \text{data} \quad \text{data if sw = 01} \]

Immediate with Accumulator

\[ 0011110w \quad \text{data} \quad \text{data if w = 1} \]

NEG = Change Sign

\[ 1111011w \quad \text{mod } 011r/m \]

AAA = ASCII Adjust for Add

\[ 00110111 \]

DAA = Decimal Adjust for Add

\[ 00100111 \]

AAS = ASCII Adjust for Subtract

\[ 00111111 \]

DAS = Decimal Adjust for Subtract

\[ 00101111 \]
MUL = Multiply (Unsigned)

\[
\begin{array}{c|c}
1111011 & \text{mod 100 r/m} \\
\end{array}
\]

IMUL = Integer Multiply (Signed)

\[
\begin{array}{c|c}
1111011 & \text{mod 101 r/m} \\
\end{array}
\]

IIMUL = Integer Immediate Multiply (Signed)

\[
\begin{array}{c|c|c|c}
011010 & \text{reg r/m} & \text{data} & \text{data if s = 0} \\
\end{array}
\]

DIV = Divide (Unsigned)

\[
\begin{array}{c|c}
1111011 & \text{mod 110 r/m} \\
\end{array}
\]

IDIV = Integer Divide (Signed)

\[
\begin{array}{c|c}
1111011 & \text{mod 111 r/m} \\
\end{array}
\]

AAM = ASCII Adjust for Multiply

\[
\begin{array}{c|c}
11010100 & 00001010 \\
\end{array}
\]

AAD = ASCII Adjust for Divide

\[
\begin{array}{c|c}
11010101 & 00001010 \\
\end{array}
\]

CBW = Convert Byte to Word

\[
10011000
\]

CWD = Convert Word to Doubleword

\[
10011001
\]
Logic

Shift/Rotate Instructions

Register/Memory by 1

\[
\begin{array}{|c|c|}
\hline
1101000 \text{w} & \text{mod T T T r/m} \\
\hline
\end{array}
\]

Register/Memory by CL

\[
\begin{array}{|c|c|}
\hline
1101001 \text{w} & \text{mod T T T r/m} \\
\hline
\end{array}
\]

Register/Memory by Count

\[
\begin{array}{|c|c|c|}
\hline
1100000 \text{w} & \text{mod T T T r/m} & \text{count} \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>T T T</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ROL</td>
</tr>
<tr>
<td>001</td>
<td>ROR</td>
</tr>
<tr>
<td>010</td>
<td>RCL</td>
</tr>
<tr>
<td>011</td>
<td>RCR</td>
</tr>
<tr>
<td>100</td>
<td>SHL/SAL</td>
</tr>
<tr>
<td>101</td>
<td>SHR</td>
</tr>
<tr>
<td>111</td>
<td>SAR</td>
</tr>
</tbody>
</table>

AND = And

Register/Memory and Register to Either

\[
\begin{array}{|c|c|}
\hline
001000 \text{dw} & \text{mod reg r/m} \\
\hline
\end{array}
\]

Immediate to Register/Memory

\[
\begin{array}{|c|c|c|}
\hline
1000000 \text{w} & \text{mod 100 r/m} & \text{data} & \text{data if } w = 1 \\
\hline
\end{array}
\]

Immediate to Accumulator

\[
\begin{array}{|c|c|c|}
\hline
0010010 \text{w} & \text{data} & \text{data if } w = 1 \\
\hline
\end{array}
\]
TEST = AND Function to Flags; No Result

Register/Memory and Register

\[
\begin{array}{c|c}
100010w & \text{mod reg r/m} \\
\end{array}
\]

Immediate Data and Register/Memory

\[
\begin{array}{c|c|c}
111011w & \text{mod 000 r/m} & \text{data} & \text{data if } w = 1 \\
\end{array}
\]

Immediate Data and Accumulator

\[
\begin{array}{c|c}
1010100w & \text{data} & \text{data if } w = 1 \\
\end{array}
\]

Or = Or

Register/Memory and Register to Either

\[
\begin{array}{c|c}
00010d w & \text{mod reg r/m} \\
\end{array}
\]

Immediate to Register/Memory

\[
\begin{array}{c|c|c}
100000w & \text{mod 001 r/m} & \text{data} & \text{data if } w = 1 \\
\end{array}
\]

Immediate to Accumulator

\[
\begin{array}{c|c}
000110w & \text{data} & \text{data if } w = 1 \\
\end{array}
\]

XOR = Exclusive OR

Register/Memory and Register to Either

\[
\begin{array}{c|c}
01100d w & \text{mod reg r/m} \\
\end{array}
\]

Immediate to Register/Memory

\[
\begin{array}{c|c|c}
100000w & \text{mod 110 r/m} & \text{data} & \text{data if } w = 1 \\
\end{array}
\]

Immediate to Accumulator

\[
\begin{array}{c|c}
011010w & \text{data} & \text{data if } w = 1 \\
\end{array}
\]
NOT = Invert Register/Memory

1111011w mod 010 r/m

String Manipulation

MOV$ = Move Byte/Word

1010010w

CMPS B/W = Compare Byte/Word

1010011w

SCAS = Scan Byte/Word

1010111w

LODS = Load Byte/Word to AL/AX

1010110w

STOS = Store Byte/Word from AL/AX

1010101w

INS = Input Byte/Word from DX Port

0110110w

OUTS = Output Byte/Word to DX Port

0110111w
**REP/REPNE, REPZ/REPNZ = Repeat String**

<table>
<thead>
<tr>
<th>Repeat Move String</th>
<th>11110011</th>
<th>1010010w</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat Compare String (z/Not z)</td>
<td>1111001z</td>
<td>1010011w</td>
</tr>
<tr>
<td>Repeat Scan String (z/Not z)</td>
<td>1111001z</td>
<td>1010111w</td>
</tr>
<tr>
<td>Repeat Load String</td>
<td>11110011</td>
<td>1010110w</td>
</tr>
<tr>
<td>Repeat Store String</td>
<td>11110011</td>
<td>1010101w</td>
</tr>
<tr>
<td>Repeat Input String</td>
<td>11110011</td>
<td>0110110w</td>
</tr>
<tr>
<td>Repeat Output String</td>
<td>11110011</td>
<td>0110111w</td>
</tr>
</tbody>
</table>
Control Transfer

CALL = Call

Direct within Segment

| 11101000 | disp-low | disp-high |

Register/Memory Indirect within Segment

| 11111111 | mod 010 r/m |

Direct Intersegment

| 10011010 | Segment Offset | Segment Selector |

Indirect Intersegment

| 11111111 | mod 011 r/m (mod ≠ 11) |

JMP = Unconditional Jump

Short/Long

| 11101011 | disp-low |

Direct within Segment

| 11101001 | disp-low | disp-high |

Register/Memory Indirect within Segment

| 11111111 | mod 100 r/m |

Direct Intersegment

| 11101010 | Segment Offset | Segment Selector |
Indirect Intersegment

```
11111111 mod 101 r/m (mod ≠ 11)
```

**RET = Return from Call**

Within Segment

```
11000011
```

Within Segment Adding Immediate to SP

```
11000010 data-low data-high
```

Intersegment

```
11001011
```

Intersegment Adding Immediate to SP

```
11001010 data-low data-high
```

**JE/JZ = Jump on Equal/Zero**

```
01110100 disp
```

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

```
01111100 disp
```

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

```
01111110 disp
```

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

```
01110010 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
01111011 disp

JNO = Jump on Not Overflow
01110001 disp

JNS = Jump on Not Sign
01111001 disp

LOOP = Loop CX Times
11100010 disp

LOOPZ/LOOPE = Loop while Zero/Equal
11100001 disp

LOOPNZ/LOOPNE = Loop while Not Zero/Not Equal
11100000 disp

JCXZ = Jump on CX Zero
11100011 disp

ENTER = Enter Procedure
11001000 data-low data-high

LEAVE = Leave Procedure
11001001
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
\[1111000\]

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 00001110

ESC = Processor Extension Escape

11011TTT  mod LLL r/m
Protection Control

LGDT = Load Global Descriptor Table Register

00001111 00000000 mod 010 r/m

SGDT = Store Global Descriptor Table Register

00001111 00000000 mod 000 r/m

LIDT = Load Interrupt Descriptor Table Register

00001111 00000000 mod 011 r/m

SIDT = Store Interrupt Descriptor Table Register

00001111 00000000 mod 001 r/m

LLDT = Load Local Descriptor Table Register from Register/Memory

00001111 00000000 mod 010 r/m

SLDT = Store Local Descriptor Table Register from Register/Memory

00001111 00000000 mod 000 r/m

LTR = Load Task Register from Register/Memory

00001111 00000000 mod 011 r/m

STR = Store Task Register to Register/Memory

00001111 00000000 mod 001 r/m

LMSW = Load Machine Status Word from Register/Memory

00001111 00000001 mod 110 r/m
SMSW = Store Machine Status Word

\[
\begin{array}{c|c|c}
00001111 & 00000001 & \text{mod 100 r/m}
\end{array}
\]

LAR = Load Access Rights from Register/Memory

\[
\begin{array}{c|c|c}
00001111 & 00000010 & \text{mod reg r/m}
\end{array}
\]

LSL = Load Segment Limit from Register/Memory

\[
\begin{array}{c|c|c}
00001111 & 00000011 & \text{mod reg r/m}
\end{array}
\]

ARPL = Adjust Requested Privilege Level from Register/Memory

\[
\begin{array}{c|c|c}
01100011 & \text{mod reg r/m}
\end{array}
\]

VERR = Verify Read Access; Register/Memory

\[
\begin{array}{c|c|c}
00001111 & 00000000 & \text{mod 100 r/m}
\end{array}
\]

VERW = Verify Write Access

\[
\begin{array}{c|c|c}
00001111 & 00000000 & \text{mod 101 r/m}
\end{array}
\]

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.

<table>
<thead>
<tr>
<th>Reg</th>
<th>Segment Register</th>
<th>Reg</th>
<th>Segment Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ES</td>
<td>10</td>
<td>SS</td>
</tr>
<tr>
<td>01</td>
<td>CS</td>
<td>11</td>
<td>DS</td>
</tr>
</tbody>
</table>

*Figure 9. 2-Bit Register Field*

<table>
<thead>
<tr>
<th>16-Bit (w = 1)</th>
<th>8-Bit (w = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 AX</td>
<td>000 AL</td>
</tr>
<tr>
<td>001 CX</td>
<td>001 CL</td>
</tr>
<tr>
<td>010 DX</td>
<td>010 DL</td>
</tr>
<tr>
<td>011 BX</td>
<td>011 BL</td>
</tr>
<tr>
<td>100 SP</td>
<td>100 AH</td>
</tr>
<tr>
<td>101 BP</td>
<td>101 CH</td>
</tr>
<tr>
<td>110 SI</td>
<td>110 DH</td>
</tr>
<tr>
<td>111 DI</td>
<td>111 BH</td>
</tr>
</tbody>
</table>

*Figure 10. 3-Bit Register Field*

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bit Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape</td>
<td>80286 Extension Escape</td>
<td>Bit Pattern = 11011</td>
</tr>
<tr>
<td>MF</td>
<td>Memory Format</td>
<td>00 = 32-Bit Real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = 32-Bit Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = 64-Bit Real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = 16-Bit Integer</td>
</tr>
<tr>
<td>ST(0)</td>
<td>Current Stack Top</td>
<td></td>
</tr>
<tr>
<td>ST(i)</td>
<td>i\textsuperscript{th} Register Below the Stack Top</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Destination</td>
<td>0 = Destination is ST(0)</td>
</tr>
<tr>
<td>P</td>
<td>Pop</td>
<td>1 = Destination is ST(i)</td>
</tr>
<tr>
<td>R</td>
<td>Reverse(^*)</td>
<td>0 = No pop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Pop ST(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Destination (op) source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Source (op) destination</td>
</tr>
<tr>
<td></td>
<td>* When d = 1, reverse the sense of R.</td>
<td></td>
</tr>
</tbody>
</table>

*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)

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>100 r/m</td>
</tr>
</tbody>
</table>

### ST(i) to ST(0)

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>100 0 ST(i)</td>
</tr>
</tbody>
</table>

### FST = Store

#### ST(0) to Integer/Real Memory

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MF 1</td>
<td>010 r/m</td>
</tr>
</tbody>
</table>

### FSTP = Store and Pop

#### ST(0) to Integer/Real Memory

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MF 1</td>
<td>011 r/m</td>
</tr>
</tbody>
</table>

#### ST(0) to Long Integer Memory

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>111 r/m</td>
</tr>
</tbody>
</table>

#### ST(0) to Temporary Real Memory

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Mod Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>111 r/m</td>
</tr>
<tr>
<td>Instruction</td>
<td>escape code</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ST(0) to BCD Memory</td>
<td>1 1 1</td>
</tr>
<tr>
<td>ST(0) to ST(i)</td>
<td>1 0 1</td>
</tr>
<tr>
<td>FXCH = Exchange ST(i) and ST(0)</td>
<td>0 0 1</td>
</tr>
<tr>
<td><strong>Comparison</strong></td>
<td></td>
</tr>
<tr>
<td>FCOM = Compare</td>
<td></td>
</tr>
<tr>
<td>Integer/Real Memory to ST(0)</td>
<td></td>
</tr>
<tr>
<td>escape MF 0</td>
<td>mod 0 1 0 r/m</td>
</tr>
<tr>
<td>ST(i) to ST(0)</td>
<td>0 0 0</td>
</tr>
<tr>
<td>FCOMP = Compare and Pop</td>
<td></td>
</tr>
<tr>
<td>Integer/Real Memory to ST(0)</td>
<td></td>
</tr>
<tr>
<td>escape MF 0</td>
<td>mod 0 1 1 r/m</td>
</tr>
<tr>
<td>ST(i) to ST(0)</td>
<td>0 0 0</td>
</tr>
<tr>
<td>FCOMPP = Compare ST(1) to ST(0) and Pop Twice</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
FTST = Test ST(0)

\begin{verbatim}
escape 001 11100100
\end{verbatim}

FXAM = Examine ST(0)

\begin{verbatim}
escape 001 11100101
\end{verbatim}

Constants

FLDZ = Load + 0.0 into ST(0)

\begin{verbatim}
escape 001 11101110
\end{verbatim}

FLD1 = Load + 1.0 into ST(0)

\begin{verbatim}
escape 001 11101000
\end{verbatim}

FLDPI = Load \( \pi \) into ST(0)

\begin{verbatim}
escape 001 11101011
\end{verbatim}

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

\begin{verbatim}
escape 001 11101001
\end{verbatim}

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

\begin{verbatim}
escape 001 11101010
\end{verbatim}

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

\begin{verbatim}
escape 001 11101100
\end{verbatim}

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

\begin{verbatim}
escape 001 11101101
\end{verbatim}
### Arithmetic

**FADD = Addition**

Integer/Real Memory with ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape MF 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ST(i) and ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape dP 0</td>
<td>11000</td>
<td>ST(i)</td>
</tr>
</tbody>
</table>

**FSUB = Subtraction**

Integer/Real Memory with ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape MF 0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ST(i) and ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape dP 0</td>
<td>1110</td>
<td>R/m</td>
</tr>
</tbody>
</table>

**FMUL = Multiplication**

Integer/Real Memory with ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape MF 0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

ST(i) and ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape dP 0</td>
<td>1100</td>
<td>1</td>
</tr>
</tbody>
</table>

**FDIV = Division**

Integer/Real Memory with ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape MF 0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ST(i) and ST(0)

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape dP 0</td>
<td>1111</td>
<td>R/m</td>
</tr>
</tbody>
</table>

**FSQRT = Square Root of ST(0)**

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape 001</td>
<td>11111010</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Escape</th>
<th>Mod</th>
<th>R/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape 001</td>
<td>11111101</td>
<td></td>
</tr>
</tbody>
</table>
FPREM = Partial Remainder of ST(0) ÷ ST(1)

```
escape 001 11111000
```

FRNDINT = Round ST(0) to Integer

```
escape 001 11111100
```

FXTRACT = Extract Components of ST(0)

```
escape 001 11110100
```

FABS = Absolute Value of ST(0)

```
escape 001 11100001
```

FCHS = Change Sign of ST(0)

```
escape 001 11100000
```

Transcendental

FPTAN = Partial Tangent of ST(0)

```
escape 001 11110010
```

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

```
escape 001 11110011
```

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

```
escape 001 11110000
```

FYL2X = ST(1) \times \log_2 [ST(0)]

```
escape 001 11110001
```

FYL2XP1 = ST(1) \times \log_2 [ST(0) + 1]
**Processor Control**

**FINIT** = Initialize NPX

| escape 001 | 11111001 |

**FSETPM** = Enter Protected Mode

| escape 011 | 11100011 |

**FSTSW AX** = Store Control Word

| escape 111 | 11100000 |

**FLDCW** = Load Control Word

| escape 001 | mod 101 r/m |

**FSTCW** = Store Control Word

| escape 001 | mod 111 r/m |

**FSTSW** = Store Status Word

| escape 101 | mod 111 r/m |

**FCLEX** = Clear Exceptions

| escape 011 | 11100010 |

**FSTENV** = Store Environment

| escape 001 | mod 110 r/m |
FLDENV = Load Environment

\[
\text{escape 001 \ mod 100 r/m}
\]

FSAVE = Save State

\[
\text{escape 101 \ mod 110 r/m}
\]

FRSTOR = Restore State

\[
\text{escape 101 \ mod 100 r/m}
\]

FINCSTP = Increment Stack Pointer

\[
\text{escape 001 \ 11110111}
\]

FDECSTP = Decrement Stack Pointer

\[
\text{escape 001 \ 11110110}
\]

FFREE = Free ST(i)

\[
\text{escape 101 \ 11000 ST(i)}
\]

FNOP = No Operation

\[
\text{escape 001 \ 11010000}
\]

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.

![Figure 12. 80386 Code and Data Segment Descriptor Format](image)

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:

\[
\text{Segment size} = (N + 1) \times 4\text{KB}
\]

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:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>The opcode may be one or two bytes in length. Within each byte, smaller encoding fields may be defined.</td>
</tr>
<tr>
<td>Mode Specifier</td>
<td>Consists of the &quot;mod r/m&quot; byte and the &quot;scale-index-base&quot; (s-i-b) byte.</td>
</tr>
<tr>
<td></td>
<td>The mod r/m byte specifies the address mode to be used. Format: mod T T T r/m</td>
</tr>
<tr>
<td></td>
<td>The &quot;s-i-b&quot; 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</td>
</tr>
<tr>
<td>Address Displacement</td>
<td>Follows the &quot;mod r/m&quot; byte or &quot;s-i-b&quot; byte. It may be 8, 16, or 32 bits.</td>
</tr>
<tr>
<td>Immediate Data</td>
<td>If specified, follows any displacement bytes and becomes the last field of the instruction. It may be 8, 16, or 32 bits.</td>
</tr>
<tr>
<td></td>
<td>The term &quot;8-bit data&quot; indicates a fixed data length of 8 bits.</td>
</tr>
<tr>
<td></td>
<td>The term &quot;8-, 16-, or 32-bit data&quot; indicates a variable data length. The length is determined by the w field and the current operand size.</td>
</tr>
<tr>
<td></td>
<td>If w = 0, the data is always 8 bits.</td>
</tr>
<tr>
<td></td>
<td>If w = 1, the size is determined by the operand size of the instruction.</td>
</tr>
</tbody>
</table>

*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.

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

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.

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

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:

- \( r/m = 100 \)
- \( \text{mod} = 00, 01, \text{or} 10. \)

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.

<table>
<thead>
<tr>
<th>ss</th>
<th>Scale Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Index</th>
<th>Index Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>No Index Register</td>
</tr>
<tr>
<td>101</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>EDI</td>
</tr>
</tbody>
</table>

*Figure 17. Index Registers (s-i-b Byte Present)*
<table>
<thead>
<tr>
<th>base</th>
<th>Base Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>EAX</td>
</tr>
<tr>
<td>001</td>
<td>ECX</td>
</tr>
<tr>
<td>010</td>
<td>EDX</td>
</tr>
<tr>
<td>011</td>
<td>EBX</td>
</tr>
<tr>
<td>100</td>
<td>ESP</td>
</tr>
<tr>
<td>101</td>
<td>EBP</td>
</tr>
<tr>
<td>110</td>
<td>ESI</td>
</tr>
<tr>
<td>111</td>
<td>EDI</td>
</tr>
</tbody>
</table>

If mod = 00, then EBP is not used to form the EA; immediate 32-bit address displacement follows the mode specifier byte.

*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.

```plaintext
; 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.

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

*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.

<table>
<thead>
<tr>
<th>w</th>
<th>16-Bit Data Operation</th>
<th>32-Bit Data Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 Bits</td>
<td>8 Bits</td>
</tr>
<tr>
<td>1</td>
<td>16 Bits</td>
<td>32 Bits</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>sreg2</th>
<th>sreg3</th>
<th>Segment Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>ES</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>CS</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>SS</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>DS</td>
</tr>
<tr>
<td>-</td>
<td>100</td>
<td>FS</td>
</tr>
<tr>
<td>-</td>
<td>101</td>
<td>GS</td>
</tr>
<tr>
<td>-</td>
<td>110</td>
<td>Reserved</td>
</tr>
<tr>
<td>-</td>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

*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.

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

*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.

<table>
<thead>
<tr>
<th>d</th>
<th>Direction of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Register/Memory &lt;-- Register</td>
</tr>
<tr>
<td></td>
<td>The &quot;reg&quot; field indicates the source operand; &quot;mod r/m&quot; or &quot;mod ss index base&quot; indicates the destination operand.</td>
</tr>
<tr>
<td>1</td>
<td>Register &lt;-- Register/Memory</td>
</tr>
<tr>
<td></td>
<td>The &quot;reg&quot; field indicates the destination operand; &quot;mod r/m&quot; or &quot;mod ss index base&quot; indicates the source operand.</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>s</th>
<th>8-Bit Immediate Data</th>
<th>16/32-Bit Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No effect on data</td>
<td>No effect on data</td>
</tr>
<tr>
<td>1</td>
<td>Sign-extend 8-bit data to fill 16-bit or 32-bit destination</td>
<td>No effect on data</td>
</tr>
</tbody>
</table>

*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 tt defining the condition to test.
<table>
<thead>
<tr>
<th>tttn</th>
<th>Condition</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Overflow</td>
<td>O</td>
</tr>
<tr>
<td>0001</td>
<td>No Overflow</td>
<td>NO</td>
</tr>
<tr>
<td>0010</td>
<td>Below/Not Above or Equal</td>
<td>B/NAE</td>
</tr>
<tr>
<td>0011</td>
<td>Not Below/Above or Equal</td>
<td>NB/AE</td>
</tr>
<tr>
<td>0100</td>
<td>Equal/Zero</td>
<td>E/Z</td>
</tr>
<tr>
<td>0101</td>
<td>Not Equal/Not Zero</td>
<td>NE/NZ</td>
</tr>
<tr>
<td>0110</td>
<td>Below or Equal/Not Above</td>
<td>BE/NA</td>
</tr>
<tr>
<td>0111</td>
<td>Not Below or Equal/Above</td>
<td>NBE/A</td>
</tr>
<tr>
<td>1000</td>
<td>Sign</td>
<td>S</td>
</tr>
<tr>
<td>1001</td>
<td>Not Sign</td>
<td>NS</td>
</tr>
<tr>
<td>1010</td>
<td>Parity/Parity Even</td>
<td>P/PE</td>
</tr>
<tr>
<td>1011</td>
<td>Not Parity/Parity Odd</td>
<td>NP/PO</td>
</tr>
<tr>
<td>1100</td>
<td>Less Than/Not Greater or Equal</td>
<td>L/NGE</td>
</tr>
<tr>
<td>1101</td>
<td>Not Less Than/Greater or Equal</td>
<td>NL/GE</td>
</tr>
<tr>
<td>1110</td>
<td>Less Than or Equal/Not Greater Than</td>
<td>LE/NG</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>eee Code</th>
<th>Interpreted as Control Register</th>
<th>Interpreted as Debug Register</th>
<th>Interpreted as Test Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>CR0</td>
<td>DR0</td>
<td>---</td>
</tr>
<tr>
<td>001</td>
<td>---</td>
<td>DR1</td>
<td>---</td>
</tr>
<tr>
<td>010</td>
<td>CR2</td>
<td>DR2</td>
<td>---</td>
</tr>
<tr>
<td>011</td>
<td>CR3</td>
<td>DR3</td>
<td>---</td>
</tr>
<tr>
<td>100</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>101</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>110</td>
<td>---</td>
<td>DR6</td>
<td>TR6</td>
</tr>
<tr>
<td>111</td>
<td>---</td>
<td>DR7</td>
<td>TR7</td>
</tr>
</tbody>
</table>

Figure 26. Control, Debug, and Test Register Field Encoding
# 80386 Microprocessor Instruction Set

## Data Transfer

**MOV = Move**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000100w</td>
<td>Register to Register/Memory</td>
</tr>
<tr>
<td>1000101w</td>
<td>Register/Memory to Register</td>
</tr>
<tr>
<td>1100011w</td>
<td>Immediate to Register/Memory</td>
</tr>
<tr>
<td>1011w reg</td>
<td>Immediate to Register (Short Form)</td>
</tr>
<tr>
<td>1010000w</td>
<td>Memory to Accumulator (Short Form)</td>
</tr>
<tr>
<td>1010001w</td>
<td>Accumulator to Memory (Short Form)</td>
</tr>
<tr>
<td>10001110</td>
<td>Register/Memory to Segment Register</td>
</tr>
<tr>
<td>10001100</td>
<td>Segment Register to Register/Memory</td>
</tr>
</tbody>
</table>

- **1100011w**: Immediate to Register/Memory
  - `mod 000 r/m`: 8-, 16-, or 32-bit data

- **1011w reg**: Immediate to Register (Short Form)
  - 8-, 16-, or 32-bit data

- **1010000w**: Memory to Accumulator (Short Form)
  - Full 16- or 32-bit displacement

- **1010001w**: Accumulator to Memory (Short Form)
  - Full 16- or 32-bit displacement

- **10001110**: Register/Memory to Segment Register
  - `mod sreg3 r/m`

- **10001100**: Segment Register to Register/Memory
  - `mod sreg3 r/m`
MOV SX = 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 seg 2 1 1 0

Segment Register (FS or GS)
0 0 0 0 1 1 1 1 0 sreg 3 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

\[ 1000111 \quad mod \ 000 \ r/m \]

Register (Short Form)

\[ 01011 \quad reg \]

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

\[ 000 \ sreg2111 \]

Segment Register (FS or GS)

\[ 0000111 \quad 10 \ sreg3001 \]

POPA = Pop All

\[ 01100001 \]

XCHG = Exchange

Register/Memory with Register

\[ 1000011w \quad mod \ reg \ r/m \]

Register with Accumulator (Short Form)

\[ 10010 \quad reg \]

IN = Input From:

Fixed Port

\[ 1110010w \quad port \ number \]

Variable Port

\[ 1110110w \]
OUT = Output To:

Fixed Port

| 11100111w | port number |

Variable Port

| 1110111w |

LEA = Load EA to Register

| 10001101 | mod reg r/m |

Segment Control

LDS = Load Pointer to DS

| 11000101 | mod reg r/m |

LES = Load Pointer to ES

| 11000100 | mod reg r/m |

LFS = Load Pointer to FS

| 00001111 | 10110100 | mod reg r/m |

LGS = Load Pointer to GS

| 00001111 | 10110101 | mod reg r/m |

LSS = Load Pointer to SS

| 00001111 | 10110010 | 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

10011110

STC = Set Carry Flag

11111001

STD = Set Direction Flag

11111101

STI = Set Interrupt Enable Flag

11111011

Arithmetic

ADD = Add

Register to Register

000000d w mod reg r/m

Register to Memory

0000000w mod reg r/m

Memory to Register

000001w mod reg r/m

Immediate to Register/Memory

100000s w mod 000 r/m 8-, 16-, or 32-bit data

Immediate to Accumulator (Short Form)

000010w 8-, 16-, or 32-bit data
**ADC – Add with Carry**

**Register to Register**

| 000100d w | mod reg r/m |

**Register to Memory**

| 0001000 w | mod reg r/m |

**Memory to Register**

| 0001001 w | mod reg r/m |

**Immediate to Register/Memory**

| 100000s w | mod 010 r/m | 8-, 16-, or 32-bit data |

**Immediate to Accumulator (Short Form)**

| 0001010 w | 8-, 16-, or 32-bit data |

**INC – Increment**

**Register/Memory**

| 1111111 w | mod 000 r/m |

**Register (Short Form)**

| 01000 reg |
### SUB = Subtract

**Register from Register**

| 001010d w | mod reg r/m |

**Register from Memory**

| 0010100 w | mod reg r/m |

**Memory from Register**

| 0010101 w | mod reg r/m |

**Immediate from Register/Memory**

| 100000s w | mod 101 r/m | 8-, 16-, or 32-bit data |

**Immediate from Accumulator (Short Form)**

| 0010110 w | 8-, 16-, or 32-bit data |

### SBB = Subtract with Borrow

**Register from Register**

| 000110d w | mod reg r/m |

**Register from Memory**

| 0001100 w | mod reg r/m |

**Memory from Register**

| 0001101 w | mod reg r/m |

**Immediate from Register/Memory**

| 100000s w | mod 011 r/m | 8-, 16-, or 32-bit data |

**Immediate from Accumulator (Short Form)**

| 0001110 w | 8-, 16-, or 32-bit data |
DEC = Decrement

Register/Memory

\[
\begin{array}{c|c}
1111111w & \text{mod 001 r/m} \\
\end{array}
\]

Register (Short Form)

\[
\begin{array}{c}
01001 \text{ reg} \\
\end{array}
\]

CMP = Compare

Register with Register

\[
\begin{array}{c|c}
001110d w & \text{mod reg r/m} \\
\end{array}
\]

Memory with Register

\[
\begin{array}{c|c}
0011100w & \text{mod reg r/m} \\
\end{array}
\]

Register with Memory

\[
\begin{array}{c|c}
0011101w & \text{mod reg r/m} \\
\end{array}
\]

Immediate with Register/Memory

\[
\begin{array}{c|c|c}
100000s w & \text{mod 111 r/m} & 8-, 16-, or 32-bit data \\
\end{array}
\]

Immediate with Accumulator (Short Form)

\[
\begin{array}{c|c}
0011110w & 8-, 16-, or 32-bit data \\
\end{array}
\]

NEG = Change Sign

\[
\begin{array}{c|c}
1111011w & \text{mod 011 r/m} \\
\end{array}
\]

AAA = ASCII Adjust for Add

\[
\begin{array}{c|c}
00110111 & \\
\end{array}
\]
AAS = ASCII Adjust for Subtract

\[ 00111111 \]

DAA = Decimal Adjust for Add

\[ 00100111 \]

DAS = Decimal Adjust for Subtract

\[ 00101111 \]

MUL = Multiply (Unsigned)

Accumulator with Register/Memory

\[
\begin{array}{cc}
1111011w & \text{mod } 100 \text{ r/m} \\
\end{array}
\]

IMUL = Integer Multiply (Signed)

Accumulator with Register/Memory

\[
\begin{array}{cc}
1111011w & \text{mod } 101 \text{ r/m} \\
\end{array}
\]

Register with Register/Memory

\[
\begin{array}{ccc}
00001111 & 10101111 & \text{mod reg r/m} \\
\end{array}
\]

Register/Memory with Immediate to Register

\[
\begin{array}{ccc}
011010s1 & \text{mod reg r/m} & 8-, 16-, or 32-bit data \\
\end{array}
\]

DIV = Divide (Unsigned)

Accumulator by Register/Memory

\[
\begin{array}{cc}
1111011w & \text{mod } 110 \text{ r/m} \\
\end{array}
\]

IDIV = Integer Divide (Signed)

Accumulator by Register/Memory

\[
\begin{array}{cc}
1111011w & \text{mod } 111 \text{ r/m} \\
\end{array}
\]
AAD = ASCII Adjust for Divide

```
11010101 00001010
```

AAM = ASCII Adjust for Multiply

```
11010100 00001010
```

CBW = Convert Byte to Word

```
10011000
```

CWD = Convert Word to Doubleword

```
10011001
```

Logic

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

Register/Memory by 1

```
11010000 w  mod T T T r/m
```

Register/Memory by CL

```
11010001 w  mod T T T r/m
```

Register/Memory by Immediate Count

```
11000000 w  mod T T T r/m  8-bit data
```
Shift/Rotate Instructions
Through Carry (RCL and RCR)

Register/Memory by 1

| 1101000 w | mod T T T r/m |

Register/Memory by CL

| 1101001 w | mod T T T r/m |

Register/Memory by Immediate Count

| 1100000 w | mod T T T r/m | 8-bit data |

<table>
<thead>
<tr>
<th>T T T</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>RCL</td>
</tr>
<tr>
<td>001</td>
<td>ROR</td>
</tr>
<tr>
<td>010</td>
<td>RCL</td>
</tr>
<tr>
<td>011</td>
<td>RCR</td>
</tr>
<tr>
<td>100</td>
<td>SHL/SAL</td>
</tr>
<tr>
<td>101</td>
<td>SHR</td>
</tr>
<tr>
<td>111</td>
<td>SAR</td>
</tr>
</tbody>
</table>

SHLD = Shift Left Double

Register/Memory by Immediate

| 00001111 | 10100100 | mod reg r/m | 8-bit data |

Register/Memory by CL

| 00001111 | 10100101 | mod reg r/m |
**SHRD = Shift Right Double**

<table>
<thead>
<tr>
<th>Register/Memory by Immediate</th>
<th>00001111</th>
<th>10101100</th>
<th>mod reg r/m</th>
<th>8-bit data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Register/Memory by CL</th>
<th>00001111</th>
<th>10101101</th>
<th>mod reg r/m</th>
<th></th>
</tr>
</thead>
</table>

**AND = And**

<table>
<thead>
<tr>
<th>Register to Register</th>
<th>001000d w</th>
<th>mod reg r/m</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Register to Memory</th>
<th>0010000w</th>
<th>mod reg r/m</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory to Register</th>
<th>0010001w</th>
<th>mod reg r/m</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Immediate to Register/Memory</th>
<th>100000s w</th>
<th>mod 100 r/m</th>
<th>8-, 16-, or 32-bit data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Immediate to Accumulator (Short Form)</th>
<th>0010010w</th>
<th>8-, 16-, or 32-bit data</th>
<th></th>
</tr>
</thead>
</table>
## TEST — AND Function to Flags; No Result

### Register/Memory and Register

| 1000010 w | mod reg r/m |

### Immediate Data and Register/Memory

| 1111011 w | mod 000 r/m | 8-, 16-, or 32-bit data |

### Immediate Data and Accumulator (Short Form)

| 1010100 w | 8-, 16-, or 32-bit data |

## OR — Or

### Register to Register

| 000010 d w | mod reg r/m |

### Register to Memory

| 0000100 w | mod reg r/m |

### Memory to Register

| 0000101 w | mod reg r/m |

### Immediate to Register/Memory

| 100000 s w | mod 001 r/m | 8-, 16-, or 32-bit data |

### Immediate to Accumulator (Short Form)

| 0000110 w | 8-, 16-, or 32-bit data |
XOR = Exclusive OR

Register to Register

\[
\begin{array}{|c|c|}
\hline
0011000d & \text{mod reg r/m} \\
\hline
\end{array}
\]

Register to Memory

\[
\begin{array}{|c|c|}
\hline
0011000w & \text{mod reg r/m} \\
\hline
\end{array}
\]

Memory to Register

\[
\begin{array}{|c|c|}
\hline
0011001w & \text{mod reg r/m} \\
\hline
\end{array}
\]

Immediate to Register/Memory

\[
\begin{array}{|c|c|c|}
\hline
1000000s & \text{mod 110 r/m} & 8-, 16-, or 32-bit data \\
\hline
\end{array}
\]

Immediate to Accumulator (Short Form)

\[
\begin{array}{|c|c|}
\hline
0011010w & 8-, 16-, or 32-bit data \\
\hline
\end{array}
\]

NOT = Invert Register/Memory

\[
\begin{array}{|c|c|}
\hline
1111011w & \text{mod 010 r/m} \\
\hline
\end{array}
\]

String Manipulation

CMPS = Compare Byte Word

\[
\begin{array}{|c|}
\hline
1010011w \\
\hline
\end{array}
\]

INS = Input Byte/Word from DX Port

\[
\begin{array}{|c|}
\hline
0110110w \\
\hline
\end{array}
\]
LODS = Load Byte/Word to AL/AX/EAX

1010110w

MOVX = 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
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPNE CMPS</td>
<td>Compare String (Find Match)</td>
<td>11110010 010011</td>
</tr>
</tbody>
</table>
Bit Manipulation

**BSF** = Scan Bit Forward

| 00001111 | 10111100 | mod reg r/m |

**BSR** = Scan Bit Reverse

| 00001111 | 10111101 | mod reg r/m |

**BT** = Test Bit

Register/Memory, Immediate

| 00001111 | 10111010 | mod 100 r/m | 8-bit data |

Register/Memory, Register

| 00001111 | 10100011 | mod reg r/m |

**BTC** = Test Bit and Complement

Register/Memory, Immediate

| 00001111 | 10111010 | mod 111 r/m | 8-bit data |

Register/Memory, Register

| 00001111 | 10111011 | mod reg r/m |

**BTR** = Test Bit and Reset

Register/Memory, Immediate

| 00001111 | 10111010 | mod 110 r/m | 8-bit data |

Register/Memory, Register

| 00001111 | 10110011 | mod reg r/m |
**BTS – Test Bit and Set**

Register/Memory, Immediate

| 00001111 | 1011010 | mod 101 r/m | 8-bit data |

Register/Memory, Register

| 00001111 | 10101011 | mod reg r/m |

**Control Transfer**

**CALL – Call**

Direct within Segment

| 11101000 | full 16- or 32-bit displacement |

Register/Memory Indirect within Segment

| 11111111 | mod 010 r/m |

Direct Intersegment

| 10011010 | offset, selector |

Indirect Intersegment

| 11111111 | mod 011 r/m |

**JMP – Unconditional Jump**

Short

| 11101011 | 8-bit disp. |

Direct within Segment

| 11101001 | full 16- or 32-bit displacement |
Register/Memory Indirect within Segment

11111111 mod 100 r/m

Direct Intersegment

11101010 offset, selector

Indirect Intersegment

11111111 mod 101 r/m

RET = Return from Call

Within Segment

11000011

Within Segment Adding Immediate to SP

11000010 16-bit displacement

Intersegment

11001011

Intersegment Adding Immediate to SP

11001010 16-bit displacement

Conditional Jumps

JO = Jump on Overflow

8-Bit Displacement

01110000 8-bit disp.

Full Displacement

00001111 10000000 full 16- or 32-bit displacement
JNO = Jump on Not Overflow

8-Bit Displacement

| 01110001 | 8-bit disp. |

Full Displacement

| 00001111 | 10000001 | full 16- or 32-bit displacement |

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

8-Bit Displacement

| 01110010 | 8-bit disp. |

Full Displacement

| 00001111 | 10000010 | full 16- or 32-bit displacement |

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

8-Bit Displacement

| 01110011 | 8-bit disp. |

Full Displacement

| 00001111 | 10000011 | full 16- or 32-bit displacement |

JE/JZ = Jump on Equal/Zero

8-Bit Displacement

| 01110100 | 8-bit disp. |

Full Displacement

| 00001111 | 10000100 | full 16- or 32-bit displacement |
### JNE/JNZ - Jump on Not Equal/Not Zero

**8-Bit Displacement**

| 01110101 | 8-bit disp. |

**Full Displacement**

| 00001111 | 10000101 | full 16- or 32-bit displacement |

### JBE/JNA - Jump on Below or Equal/Not Above

**8-Bit Displacement**

| 01110110 | 8-bit disp. |

**Full Displacement**

| 00001111 | 10000110 | full 16- or 32-bit displacement |

### JNBE/JA - Jump on Not Below or Equal/Above

**8-Bit Displacement**

| 01110111 | 8-bit disp. |

**Full Displacement**

| 00001111 | 10000111 | full 16- or 32-bit displacement |

### JS - Jump on Sign

**8-Bit Displacement**

| 01111000 | 8-bit disp. |

**Full Displacement**

| 00001111 | 10001000 | full 16- or 32-bit displacement |
### JNS - Jump on Not Sign

<table>
<thead>
<tr>
<th>8-Bit Displacement</th>
<th>8-bit disp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111001</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Displacement</th>
<th>full 16- or 32-bit displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001111 10001001</td>
<td></td>
</tr>
</tbody>
</table>

### JP/JPE - Jump on Parity/Parity Even

<table>
<thead>
<tr>
<th>8-Bit Displacement</th>
<th>8-bit disp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Displacement</th>
<th>full 16- or 32-bit displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001111 10001010</td>
<td></td>
</tr>
</tbody>
</table>

### JNP/JPO - Jump on Not Parity/Parity Odd

<table>
<thead>
<tr>
<th>8-Bit Displacement</th>
<th>8-bit disp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Displacement</th>
<th>full 16- or 32-bit displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001111 10001011</td>
<td></td>
</tr>
</tbody>
</table>

### JL/JNGE - Jump on Less/Not Greater or Equal

<table>
<thead>
<tr>
<th>8-Bit Displacement</th>
<th>8-bit disp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Displacement</th>
<th>full 16- or 32-bit displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001111 10001100</td>
<td></td>
</tr>
</tbody>
</table>
JNL/JGE = Jump on Not Less/Greater or Equal

8-Bit Displacement
01111101 8-bit disp.

Full Displacement
00001111 10001101 full 16- or 32-bit displacement

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

8-Bit Displacement
01111110 8-bit disp.

Full Displacement
00001111 10001110 full 16- or 32-bit displacement

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

8-Bit Displacement
01111111 8-bit disp.

Full Displacement
00001111 10001111 full 16- or 32-bit displacement

JCXZ = Jump on CX Zero

11100011 8-bit disp.

JE CXZ = Jump on ECX Zero

11100011 8-bit disp.

Note: The operand size prefix differentiates JCXZ from JE CXZ.
**LOOP** = Loop CX Times

| 11100010 | 8-bit disp. |

**LOOPZ/LOOPE** = Loop with Zero/Equal

| 11100001 | 8-bit disp. |

**LOOPNZ/LOOPNE** = Loop while Not Zero

| 11100000 | 8-bit disp. |

**Conditional Byte Set**

**SETO** = Set Byte on Overflow

To Register/Memory

| 00001111 | 10010000 | mod 000 r/m |

**SETNO** = Set Byte on Not Overflow

To Register/Memory

| 00001111 | 10010001 | mod 000 r/m |

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

To Register/Memory

| 00001111 | 10010010 | mod 000 r/m |

**SETNB** = Set Byte on Not Below/Above or Equal

To Register/Memory

| 00001111 | 10010011 | mod 000 r/m |

**SETE/SETZ** = Set Byte on Equal/Zero

To Register/Memory

| 00001111 | 10010100 | mod 000 r/m |
SETNE/SETNZ = Set Byte on Not Equal/Not Zero
To Register/Memory
00001111 10010101 mod 000 r/m

SETBE/SETNA = Set Byte on Below or Equal/Not Above
To Register/Memory
00001111 10010110 mod 000 r/m

SETNBE/SETA = Set Byte on Not Below or Equal/Above
To Register/Memory
00001111 10010111 mod 000 r/m

SETS = Set Byte on Sign
To Register/Memory
00001111 10011000 mod 000 r/m

SETNS = Set Byte on Not Sign
To Register/Memory
00001111 10011001 mod 000 r/m

SETP/SETPE = Set Byte on Parity/Parity Even
To Register/Memory
00001111 10011010 mod 000 r/m

SETNP/SETPO = Set Byte on Not Parity/Parity Odd
To Register/Memory
00001111 10011011 mod 000 r/m

SETL/SETNGE = Set Byte on Less/Not Greater or Equal
To Register/Memory
00001111 10011100 mod 000 r/m
SETNL/SETGE = Set Byte on Not Less/Greater or Equal

To Register/Memory

\[
\begin{array}{ccc}
00001111 & 01111101 & \text{mod 000 r/m}
\end{array}
\]

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

To Register/Memory

\[
\begin{array}{ccc}
00001111 & 10011110 & \text{mod 000 r/m}
\end{array}
\]

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

To Register/Memory

\[
\begin{array}{ccc}
00001111 & 10011111 & \text{mod 000 r/m}
\end{array}
\]

ENTER = Enter Procedure

\[
\begin{array}{ccc}
11001000 & 16\text{-bit displacement} & 8\text{-bit level}
\end{array}
\]

LEAVE = Leave Procedure

\[
11001001
\]

Interrupt Instructions

INT = Interrupt

Type Specified

\[
\begin{array}{ccc}
11001101 & \text{type}
\end{array}
\]

Type 3

\[
11001100
\]

INTO = Interrupt 4 If Overflow Flag Set

\[
11001110
\]
BOUND = Interrupt 5 If Detect Value Out of Range

01100010 mod reg r/m

IRET = Interrupt Return

11001111

Processor Control

HLT = Halt

11110100

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

CR0/CR2/CR3 from Register

00001111 00100010 11 eee reg

Register from CR0-3

00001111 00100000 11 eee reg

DR0-3, DR6-7 from Register

00001111 00100011 11 eee reg

Register from DR0-3, DR6-7

00001111 00100001 11 eee reg

TR6-7 from Register

00001111 00100110 11 eee reg

Register from TR6-7

00001111 00100100 11 eee reg

NOP = No Operation

10010000
WAIT = Wait until BUSY Pin is Negated

10011011

Processor Extension

ESC = Processor Extension Escape

11011TTT  mod LLL 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: 00100110

FS: 01100100

GS: 01100101

SS: 00110110

**Protection Control**

**ARPL = Adjust Requested Privilege Level from Register/Memory**

01100011 mod reg r/m

**LAR = Load Access Rights from Register/Memory**

00001111 00000010 mod reg r/m

**LGDT = Load Global Descriptor Table Register**

00001111 00000001 mod 010 r/m

**LIDT = Load Interrupt Descriptor Table Register**

00001111 00000001 mod 011 r/m

**LLDT = Load Local Descriptor Table Register to Register/Memory**

00001111 00000000 mod 010 r/m
LMSW = Load Machine Status Word from Register/Memory

\[
\begin{array}{ll}
00001111 & 00000001 \\
\text{mod 110 r/m}
\end{array}
\]

LSL = Load Segment Limit from Register/Memory

\[
\begin{array}{ll}
00001111 & 00000011 \\
\text{mod reg r/m}
\end{array}
\]

LTR = Load Task Register from Register/Memory

\[
\begin{array}{ll}
00001111 & 00000000 \\
\text{mod 001 r/m}
\end{array}
\]

SGDT = Store Global Descriptor Table Register

\[
\begin{array}{ll}
00001111 & 00000001 \\
\text{mod 000 r/m}
\end{array}
\]

SIDT = Store Interrupt Descriptor Table Register

\[
\begin{array}{ll}
00001111 & 00000001 \\
\text{mod 001 r/m}
\end{array}
\]

SLDT = Store Local Descriptor Table Register to Register/Memory

\[
\begin{array}{ll}
00001111 & 00000000 \\
\text{mod 000 r/m}
\end{array}
\]

SMSW = Store Machine Status Word

\[
\begin{array}{ll}
00001111 & 00000001 \\
\text{mod 100 r/m}
\end{array}
\]

STR = Store Task Register to Register/Memory

\[
\begin{array}{ll}
00001111 & 00000000 \\
\text{mod 001 r/m}
\end{array}
\]

VERR = Verify Read Access; Register/Memory

\[
\begin{array}{ll}
00001111 & 00000000 \\
\text{mod 100 r/m}
\end{array}
\]
VERW = Verify Write Access

| 00001111 | 00000000 | mod 101 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bit Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape</td>
<td>80386 Extension Escape</td>
<td>Bit Pattern = 11011</td>
</tr>
<tr>
<td>MF</td>
<td>Memory Format</td>
<td>00 = 32-bit Real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = 32-bit integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = 64-bit Real</td>
</tr>
<tr>
<td>ST(0)</td>
<td>Current Stack Top</td>
<td>11 = 16-bit integer</td>
</tr>
<tr>
<td>ST(i)</td>
<td>i&lt;sup&gt;th&lt;/sup&gt; register below the stack top</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Destination</td>
<td>0 = Destination is ST(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Destination is ST(i)</td>
</tr>
<tr>
<td>P</td>
<td>Pop</td>
<td>0 = No pop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Pop ST(0)</td>
</tr>
<tr>
<td>R</td>
<td>Reverse*</td>
<td>0 = Destination (op) source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Source (op) destination</td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Offsets in Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8  7  0</td>
<td></td>
</tr>
<tr>
<td>Control Word</td>
<td>0</td>
</tr>
<tr>
<td>Status Word</td>
<td>2</td>
</tr>
<tr>
<td>Tag Word</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Pointer (IP) Bits 15-0</td>
<td>6</td>
</tr>
<tr>
<td>IP Bits 19-16</td>
<td>8</td>
</tr>
<tr>
<td>Opcode Bits 10-0</td>
<td></td>
</tr>
<tr>
<td>Operand Pointer (OP) Bits 15-0</td>
<td>A</td>
</tr>
<tr>
<td>OP Bits 19-16</td>
<td>C</td>
</tr>
</tbody>
</table>

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

90  Microprocessors and Instruction Sets — October 1990
### Figure 29. Instruction and Pointer Image (16-Bit Protected Mode)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Offset in Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Control Word</td>
</tr>
<tr>
<td>8</td>
<td>Status Word</td>
</tr>
<tr>
<td>7</td>
<td>Tag Word</td>
</tr>
<tr>
<td>6</td>
<td>Instruction Pointer Offset</td>
</tr>
<tr>
<td>8</td>
<td>CS Selector</td>
</tr>
<tr>
<td>A</td>
<td>Operand Offset</td>
</tr>
<tr>
<td>C</td>
<td>Operand Selector</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Offset in Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Control Word</td>
</tr>
<tr>
<td>24</td>
<td>Status Word</td>
</tr>
<tr>
<td>23</td>
<td>Tag Word</td>
</tr>
<tr>
<td>16</td>
<td>IP Bits 15-0</td>
</tr>
<tr>
<td>15</td>
<td>IP Bits 31-16</td>
</tr>
<tr>
<td>8</td>
<td>Opcode Bits 10-0</td>
</tr>
<tr>
<td>7</td>
<td>Operand Pointer Bits 15-0</td>
</tr>
<tr>
<td>0</td>
<td>Operand Pointer Bits 31-16</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Offset in Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Control Word</td>
</tr>
<tr>
<td>24</td>
<td>Status Word</td>
</tr>
<tr>
<td>23</td>
<td>Tag Word</td>
</tr>
<tr>
<td>16</td>
<td>Instruction Pointer Offset</td>
</tr>
<tr>
<td>15</td>
<td>CS Selector</td>
</tr>
<tr>
<td>8</td>
<td>Data Operand Offset</td>
</tr>
<tr>
<td>7</td>
<td>Operand Selector</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
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 (Unsorted 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**

<table>
<thead>
<tr>
<th>Instruction/Real Memory to ST(0)</th>
<th>escape MF 1</th>
<th>mod 0 0 0 r/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Integer Memory to ST(0)</td>
<td>escape 1 1 1</td>
<td>mod 1 0 1 r/m</td>
</tr>
<tr>
<td>Temporary Real Memory to ST(0)</td>
<td>escape 0 1 1</td>
<td>mod 1 0 1 r/m</td>
</tr>
<tr>
<td>BCD Memory to ST(0)</td>
<td>escape 1 1 1</td>
<td>mod 1 0 0 r/m</td>
</tr>
<tr>
<td>ST(i) to ST(0)</td>
<td>escape 0 0 1</td>
<td>1 1 0 0 0 ST(i)</td>
</tr>
</tbody>
</table>

**FST = Store**

<table>
<thead>
<tr>
<th>ST(0) to Integer/Real Memory</th>
<th>escape MF 1</th>
<th>mod 0 1 0 r/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) to ST(i)</td>
<td>escape 1 0 1</td>
<td>1 1 0 1 0 ST(i)</td>
</tr>
</tbody>
</table>

**FSTP = Store and Pop**

<table>
<thead>
<tr>
<th>ST(0) to Integer/Real Memory</th>
<th>escape MF 1</th>
<th>mod 0 1 1 r/m</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) to Long Integer Memory</td>
<td>escape 1 1 1</td>
<td>mod 1 1 1 r/m</td>
</tr>
<tr>
<td>Instruction</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>ST(0) to Temporary Real Memory</td>
<td>escape 0 1 1 mod 1 1 1 r/m</td>
<td></td>
</tr>
<tr>
<td>ST(0) to BCD Memory</td>
<td>escape 1 1 1 mod 1 1 0 r/m</td>
<td></td>
</tr>
<tr>
<td>ST(0) to ST(i)</td>
<td>escape 1 0 1 1 0 1 1 ST(i)</td>
<td></td>
</tr>
<tr>
<td>FXCH = Exchange ST(i) and ST(0)</td>
<td>escape 0 0 1 1 0 0 1 ST(i)</td>
<td></td>
</tr>
<tr>
<td><strong>Comparison</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOM = Compare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer/Real Memory to ST(0)</td>
<td>escape MF 0 mod 0 1 0 r/m</td>
<td></td>
</tr>
<tr>
<td>ST(i) to ST(0)</td>
<td>escape 0 0 0 1 1 0 1 0 ST(i)</td>
<td></td>
</tr>
<tr>
<td>FCOMP = Compare and Pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer/Real Memory to ST(0)</td>
<td>escape MF 0 mod 0 1 1 r/m</td>
<td></td>
</tr>
<tr>
<td>ST(i) to ST(0)</td>
<td>escape 0 0 0 1 1 0 1 1 ST(i)</td>
<td></td>
</tr>
<tr>
<td>FCOMPP = Compare ST(1) to ST(0) and Pop Twice</td>
<td>escape 1 1 0 1 1 0 1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>FUCOM = Unordered Compare Real</td>
<td>escape 1 0 1 1 1 0 0 ST(i)</td>
<td></td>
</tr>
</tbody>
</table>
FUCOMP = Unordered Compare Real and Pop

\[
\text{escape 010 } 11101100
\]

FUCOMPP = Unordered Compare Real and Pop Twice

\[
\text{escape 010 } 11101001
\]

FTST = Test ST(0)

\[
\text{escape 001 } 11100100
\]

FXAM = Examine ST(0)

\[
\text{escape 001 } 11100101
\]

Constants

FLDZ = Load +0.0 Into ST(0)

\[
\text{escape 001 } 11101110
\]

FLD1 = Load +1.0 Into ST(0)

\[
\text{escape 001 } 11101000
\]

FLDPI = Load \pi Into ST(0)

\[
\text{escape 001 } 11101011
\]

FLDL2T = Load \log_2 10 Into ST(0)

\[
\text{escape 001 } 11101001
\]

FLDL2E = Load \log_2 e Into ST(0)

\[
\text{escape 001 } 11101010
\]
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 0 0 0
```

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

```
escape 0 0 1
1 1 1 1 1 0 1
```

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

```
escape 0 0 1
1 1 1 1 0 0 0
```

FPREM1 = IEEE Partial Remainder

```
escape 0 0 1
1 1 1 0 1 0 1
```

FRNDINT = Round ST(0) to Integer

```
escape 0 0 1
1 1 1 1 1 0 0
```

FXTRACT = Extract Components of ST(0)

```
escape 0 0 1
1 1 1 0 1 0 0
```

FABS = Absolute Value of ST(0)

```
escape 0 0 1
1 1 1 0 0 0 1
```

FCHS = Change Sign of ST(0)

```
escape 0 0 1
1 1 1 0 0 0 0
```
Transcendental

FPTAN = Partial Tangent of ST(0)

```
escape 0 0 1 1 1 1 0 0 1 0
```

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

```
escape 0 0 1 1 1 1 0 0 1 1
```

FSIN = Sine

```
escape 0 0 1 1 1 1 1 1 0
```

FCOS = Cosine

```
escape 0 0 1 1 1 1 1 1 1
```

FSINCOS = Sine and Cosine

```
escape 0 0 1 1 1 1 1 0 1 1
```

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

```
escape 0 0 1 1 1 1 0 0 0 0
```

FYL2X = ST(1) x Log₂[ST(0)]

```
escape 0 0 1 1 1 1 0 0 0 1
```

FYL2XP1 = ST(1) x Log₂[ST(0) + 1]

```
escape 0 0 1 1 1 1 1 0 0 1
```

Processor Control

FINIT = Initialize NPX

```
escape 0 1 1 1 1 0 0 0 1 1
```
FSTSW AX = Store Control Word

\[ \text{escape} \quad 111 \quad \text{11100000} \]

FLDCW = Load Control Word

\[ \text{escape} \quad 001 \quad \text{mod 101 r/m} \]

FSTCW = Store Control Word

\[ \text{escape} \quad 001 \quad \text{mod 111 r/m} \]

FSTSW = Store Status Word

\[ \text{escape} \quad 101 \quad \text{mod 111 r/m} \]

FCLEX = Clear Exceptions

\[ \text{escape} \quad 011 \quad \text{11100010} \]

FSTENV = Store Environment

\[ \text{escape} \quad 001 \quad \text{mod 110 r/m} \]

FLDENV = Load Environment

\[ \text{escape} \quad 001 \quad \text{mod 100 r/m} \]

FSAVE = Save State

\[ \text{escape} \quad 101 \quad \text{mod 110 r/m} \]

FRSTOR = Restore State

\[ \text{escape} \quad 101 \quad \text{mod 100 r/m} \]
FINCSTP = Increment Stack Pointer

\[
\begin{array}{|c|c|}
\hline
\text{escape 001} & 11110111 \\
\hline
\end{array}
\]

FDECSTP = Decrement Stack Pointer

\[
\begin{array}{|c|c|}
\hline
\text{escape 001} & 11110110 \\
\hline
\end{array}
\]

FFREE = Free ST(i)

\[
\begin{array}{|c|c|}
\hline
\text{escape 101} & 11000 \text{ST(i)} \\
\hline
\end{array}
\]

FNOP = No Operation

\[
\begin{array}{|c|c|}
\hline
\text{escape 001} & 11010000 \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|}
\hline
00001111 & 11001 \text{ reg} \\
\hline
\end{array}
\]

CMPXCHG = Compare and Exchange

Register 1, Register 2

\[
\begin{array}{|c|c|c|}
\hline
00001111 & 10110000 & 11 \text{ reg}^2 \text{ reg}^1 \\
\hline
\end{array}
\]

Memory, Register 2

\[
\begin{array}{|c|c|c|}
\hline
00001111 & 10110000 & \text{mod reg}^2 \text{ mem} \\
\hline
\end{array}
\]
XADD = Exchange and Add

Register 1, Register 2

| 00001111 | 1100000w | 11 reg² reg¹ |

Memory, Register 2

| 00001111 | 1100000w | mod reg² mem |

INVD = Invalidate Data Cache

| 00001111 | 00001000 |

WBINVD = Write-Back and Invalidate Data Cache

| 00001111 | 00001001 |

INVLPG = Invalidate TLB Entry

| 00001111 | 00000001 | mod 11 mem |