Operand-Size and Address-Size Attributes

When executing an instruction, the 80386 can address memory using either 16 or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. 16-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. 32-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

1- Default Segment Attribute

For programs executed in protected mode, the D-bit in executable-segment descriptors determines the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A value of zero in the D-bit sets the default address size and operand size to 16 bits; a value of one, to 32 bits.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

2-Operand-Size and Address-Size Instruction Prefixes

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. (A later section, "Instruction Format," shows the position of the prefixes in an instruction's encoding.) These prefixes override the default segment attributes for the instruction that follows. Table 17-1 shows the effect of each possible combination of defaults and overrides.

3-Address-Size Attribute for Stack

Instructions that use the stack implicitly (for example: POP EAX also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

The stack address-size attribute is controlled by the B-bit of the data-segment descriptor in the SS register. A value of zero in the B-bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

Table 17-1. Effective Size Attributes
The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, cw, cd, cp: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are--

<table>
<thead>
<tr>
<th>rb</th>
<th>rw</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL = 0</td>
<td>AX = 0</td>
<td>EAX = 0</td>
</tr>
<tr>
<td>CL = 1</td>
<td>CX = 1</td>
<td>ECX = 1</td>
</tr>
<tr>
<td>DL = 2</td>
<td>DX = 2</td>
<td>EDX = 2</td>
</tr>
<tr>
<td>BL = 3</td>
<td>BX = 3</td>
<td>EBX = 3</td>
</tr>
<tr>
<td>AH = 4</td>
<td>SP = 4</td>
<td>ESP = 4</td>
</tr>
<tr>
<td>CH = 5</td>
<td>BP = 5</td>
<td>EBP = 5</td>
</tr>
<tr>
<td>DH = 6</td>
<td>SI = 6</td>
<td>ESI = 6</td>
</tr>
<tr>
<td>BH = 7</td>
<td>DI = 7</td>
<td>EDI = 7</td>
</tr>
</tbody>
</table>

2 Instruction

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8: a relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
rel16, rel32: a relative address within the same code segment as the instruction assembled. rel16 applies to instructions with an operand-size attribute of 16 bits; rel32 applies to instructions with an operand-size attribute of 32 bits.

ptr16:16, ptr16:32: a FAR pointer, typically in a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. ptr16:16 is used when the instruction's operand-size attribute is 16 bits; ptr16:32 is used with the 32-bit attribute.

r8: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32: one of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8: an immediate byte value. imm8 is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16: an immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32768 and +32767 inclusive.

imm32: an immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and -2147483648.

r/m8: a one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16: a word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32: a doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8: a memory byte addressed by DS:SI or ES:DI (used only by string instructions).

m16: a memory word addressed by DS:SI or ES:DI (used only by string instructions).

m32: a memory doubleword addressed by DS:SI or ES:DI (used only by string instructions).
m16:16, M16:32: a memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

m16 & 32, m16 & 16, m32 & 32: a memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. m16 & 16 and m32 & 32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. m16 & 32 is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

moffs8, moffs16, moffs32: (memory offset) a simple memory variable of type BYTE, WORD, or DWORD used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

Sreg: a segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

3 Clocks

The "Clocks" column gives the number of clock cycles the instruction takes to execute. The clock count calculations makes the following assumptions:

- The instruction has been prefetched and decoded and is ready for execution.
- Bus cycles do not require wait states.
- There are no local bus HOLD requests delaying processor access to the bus.
- No exceptions are detected during instruction execution.
- Memory operands are aligned.

Clock counts for instructions that have an r/m (register or memory) operand are separated by a slash. The count to the left is used for a register operand; the count to the right is used for a memory operand.

The following symbols are used in the clock count specifications:

- n, which represents a number of repetitions.
- m, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.
- pm=, a clock count that applies when the instruction executes in Protected Mode. pm= is not given when the clock counts are the same for Protected and Real Address Modes.
When an exception occurs during the execution of an instruction and the exception handler is in another task, the instruction execution time is increased by the number of clocks to effect a task switch. This parameter depends on several factors:

- The type of TSS used to represent the current task (386 TSS or 286 TSS).
- The type of TSS used to represent the new task.
- Whether the current task is in V86 mode.
- Whether the new task is in V86 mode.

Table 17-5 summarizes the task switch times for exceptions.

**Table 17-5. Task Switch Times for Exceptions**

<table>
<thead>
<tr>
<th>New Task</th>
<th>Old Task</th>
<th>386 TSS</th>
<th>286 TSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM = 0</td>
<td>386 TSS</td>
<td>309</td>
<td>282</td>
</tr>
<tr>
<td>VM = 1</td>
<td>386 TSS</td>
<td>314</td>
<td>231</td>
</tr>
<tr>
<td></td>
<td>286 TSS</td>
<td>307</td>
<td>282</td>
</tr>
</tbody>
</table>

4 **Description**

The "Description" column following the "Clocks" column briefly explains the various forms of the instruction. The "Operation" and "Description" sections contain more details of the instruction's operation.

5 **Operation**

The "Operation" section contains an algorithmic description of the instruction which uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

Comments are enclosed within the symbol pairs "(*" and "*)".

Compound statements are enclosed between the keywords of the "if" statement (IF, THEN, ELSE, FI) or of the "do" statement (DO, OD), or of the "case" statement (CASE ... OF, ESAC).

A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI]
indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.

Brackets also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, \([\text{SRC}]\) indicates that the contents of the source operand is a segment-relative offset.

\(\text{A} \leftarrow \text{B};\) indicates that the value of \(\text{B}\) is assigned to \(\text{A}\).

The symbols =, \(<\>, ., \text{ and } .\) are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as \(\text{A} = \text{B}\) is TRUE if the value of \(\text{A}\) is equal to \(\text{B}\); otherwise it is FALSE.

The following identifiers are used in the algorithmic descriptions:
- \(\text{OperandSize}\) represents the operand-size attribute of the instruction, which is either 16 or 32 bits. \(\text{AddressSize}\) represents the address-size attribute, which is either 16 or 32 bits. For example,

\[
\begin{align*}
\text{IF } \text{instruction} & = \text{CMPSW} \\
\text{THEN } & \text{OperandSize} \leftarrow 16; \\
\text{ELSE} & \\
\text{IF } \text{instruction} & = \text{CMPSD} \\
\text{THEN } & \text{OperandSize} \leftarrow 32; \\
& \text{FI}; \\
& \text{FI};
\end{align*}
\]

indicates that the operand-size attribute depends on the form of the CMPS instruction used. Refer to the explanation of address-size and operand-size attributes at the beginning of this chapter for general guidelines on how these attributes are determined.

- \(\text{StackAddrSize}\) represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits, as explained earlier in the chapter.
- \(\text{SRC}\) represents the source operand. When there are two operands, \(\text{SRC}\) is the one on the right.
- \(\text{DEST}\) represents the destination operand. When there are two operands, \(\text{DEST}\) is the one on the left.
- \(\text{LeftSRC}, \text{RightSRC}\) distinguishes between two operands when both are source operands.
- \(\text{eSP}\) represents either the SP register or the ESP register depending on the setting of the B-bit for the current stack segment.

The following functions are used in the algorithmic descriptions:

- \(\text{Truncate to 16 bits(value)}\) reduces the size of the value to fit in 16 bits by discarding the uppermost bits as needed.
- \(\text{Addr(operand)}\) returns the effective address of the operand (the result of the effective address calculation prior to adding the segment base).
- \(\text{ZeroExtend(value)}\) returns a value zero-extended to the operand-size attribute of the instruction. For example, if \(\text{OperandSize} = 32\), ZeroExtend of a byte value of -10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the
value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- SignExtend(value) returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value -10 converts the byte from F6H to a doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- Push(value) pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. The action of Push is as follows: