encoding and should be sign-extended and added to the effective address. The displacement field is always the last field in the encoding, so *disp8*=05h. The *reg*=000b value selects the AX register as the destination of the MOV instruction. The *r/m* field has the value 100b, which means that the SIB byte is included in the encoding and will specify the effective address.

The next byte is the SIB byte (0Bh = 00001011b) and can be broken down as follows. The *scale* field is 00, which means that the index register will not be multiplied before use. The *index* field value of 001b selects the ECX register as the index register. Finally, the *base* field value of 011b indicates that the EBX register will be used as the base and selects the DS register for memory access. The resultant effective address is DS:[EBX+ECX+5].

The instruction represented by this series of bytes is therefore MOV AX,WORD PTR DS:[EBX][ECX*1][5]. You can verify this by assembling the instruction and looking at the resulting listing.

Example 2: *Assume that an 80486 is running in 8086-emulation mode and determine the encoding for the instruction*

```
MOV [ESI*4][ESP],EAX.
```

Since the processor is in real mode, the defaults will be 16-bit effective addresses and 16-bit operands. This instruction uses a 32-bit effective address and is moving 32 bits of data. Therefore both the address size (67h) and operand size (66h) override prefixes will be required.

The form of the instruction is MOV *mem,reg*. The instruction listing gives the encoding for this form as 100010*dw*. The direction of data flow is to memory from a register, so *d*=0. The operand size is 32 bits, so *w*=1. The opcode byte is therefore 89h.

The next byte to encode will be the addressing mode byte. The *mod* field must be set to 00b to indicate that no displacement field is present. The *r/m* field must be set to 100b to indicate that the SIB byte is being used to generate the effective address. (This is required because, among other reasons, the effective address uses scaling.) The *reg* field is used to specify the source operand, EAX, and is set to 000b. The addressing mode byte is 00000100b=04h.

The base register for the effective address will be ESP. (Even though ESI is first, it cannot be the base because it is scaled.) This sets the *base* field to 100b. The index register is scaled by 4, which indicates that *scale*=10b. Finally, the *index* field is set to 110b to select ESI. The SIB byte is then 10110100b=B4h.

The final encoding for the instruction MOV [ESI*4][ESP],EAX is 67h 66h 8Ah 04h B5h. You can verify this by assembling the instruction and looking at the resulting listing or by entering the bytes into a debugger that supports the 80386.

Instruction Execution Time

Assembly code is the realm of the small and the fast, and the best programmers will spend inordinate amounts of time crafting their routines to remove one

more byte or reduce execution time by a few more clocks. Optimization efforts should be concentrated in the sections of the code that are executed repeatedly such as loops, sorts, and searches.

The execution time for each instruction is given in the instruction reference in Appendix A. The timings represent the number of clock cycles required to execute each particular form of an instruction. The duration of each clock period is determined from the system clock speed by this formula:

1 clock = 1000/(system speed in MHz) nanoseconds (nS)

Determining the execution time for a series of instructions is not as straightforward as simply adding up the timings for the individual instructions. A number of variables regarding the instructions themselves can influence execution time significantly. In addition, events that occur outside of the system (including interrupts, exceptions, memory refresh, and so on) will also affect the real-world elapsed time. This section explains the adjustments that must be made to the timings shown in Appendix A to arrive at a better estimate of execution time.

Many instructions have multiple timings. For example, the timing for conditional jumps will vary depending on whether control is transferred (the jump is taken) or the jump falls through (not taken). Other instructions will list different times for execution in real mode and in protected mode. The timing for these two conditions is listed separately in the instruction reference in Appendix A.

If an effective address has to be calculated, this may also add to execution time. The 8086 and 8088 (and to a lesser extent the more advanced processors) will all show increased execution time when accessing memory operands. Execution time for repeating instructions (such as the shift and rotate instructions) can vary as a function of the repeat factor. And finally, the alignment of data operands will affect the time required to fetch them. The effect of all these factors on timing is explained in detail later in this section.

Instruction Timing Assumptions and Penalties

The timings for individual instructions, for all the processors, assume the following conditions. Additional assumptions for particular processors are provided later in this section.

- The instruction is prefetched and waiting in the instruction queue.
- Control transfer statement timings include any additional clocks needed to reinitialize the instruction queue.
- All memory operands are aligned. (See the processor-specific assumptions below.)
- Bus cycles do not require wait states.
- No other processors contend for bus access.
- Internal components of the processor do not contend for bus access.
- No exceptions are detected during execution of the instruction.

During normal program operation, the instruction to be executed has already been prefetched. This assumption is generally valid during normal program execution. A series of instructions that take fewer than two clocks per byte to execute, however, can deplete the queue and the processor will idle until more instructions can be prefetched.

Wait states increase the time between a processor request for data and the receipt of that data. Wait states do not necessarily add to execution time, unless the instruction queue becomes depleted, since bus access and instruction execution can take place simultaneously.

As shown, certain assumptions about the external environment are necessary as well. Most PC configurations use a single processor, and the assumption that there are no local bus HOLD requests delaying processor access to the bus is valid. The listed timings cannot account for external interrupts that occur during execution. It must also be assumed that no exceptions are detected during execution of the instruction. For example, the timing for the DIV instruction will not be valid if the division error exception is generated.

The internal components of the processor may compete for access to the bus. For example, if part of the processor (the execution unit) is executing an instruction that accesses memory, another part of the processor (the bus interface unit) will have to wait to fetch an instruction and vice versa.

Due to all the factors that cannot be predicted with certainty, an average program will take approximately 5-10 percent longer to execute than would be indicated by the timings. This error can be significant for programmers working in real-time environments where the timing and response speed of the processor are critical. In that case, special hardware must be used to determine the exact execution time.

8086 and 8088 On the 8086 and 8088, the timings presented are subject to the following additional assumption:

- Memory operands are aligned. On the 8086, a four-clock penalty is assigned for each reference to a 16-bit operand located at an odd memory address. Since all 8088 accesses are 8-bit, no alignment is required, but four additional clocks must be added for each access to a word memory operand.

The 80286 On the 80286, the timings presented are subject to the following additional assumptions.

- Memory operands are aligned. A two-clock penalty is assigned for accessing a 16-bit memory operand at an odd physical address.
- Effective address calculations do not use the base + index + displacement form. If the base + index + displacement form is used add a one-clock penalty.
- No task switch is required. The clock penalty required for a task switch is given later in this section.

The 80386 On the 80386, the following assumptions apply when using the timings listed in the instruction set reference:

- Memory operands are aligned.

  80386DX: A two-clock penalty is assigned for accessing a 32-bit memory operand at a physical address that is not evenly divisible by 4.

  80386SX: A two-clock penalty is assigned for accessing a 16-bit memory operand at an odd physical address.

  80386SX: A two-clock penalty is assigned for accessing a 32-bit memory operand at an even physical address.

  80386SX: A four-clock penalty is assigned for accessing a 32-bit memory operand at an odd physical address.
- Effective address calculations do not use two general register components. One register, scaling and a displacement can be used with the indicated timing. If the effective address calculation uses two general register components, add a one-clock penalty.
- No task switch is required. The clock penalty required for a task switch is described in detail later in this section.
- Some timings are dependent on the subsequent instruction. The clock penalty required is described in detail later in this section.

The 80486 On the 80486, assumptions must be made about the state of each of the internal components of the processor in order to specify timings. These additional assumptions and the penalties incurred when they are invalid are listed below.

- Both data and instruction accesses "hit" in the cache. The 80486 timings assume that memory fetches of both data and instructions can be found in the cache. Intel claims a combined instruction and data cache hit rate of over 90 percent. If a cache miss occurs, the 80486 will be required to use an external bus cycle to transfer the required code or data into the cache. The additional clocks required for a cache miss are noted in the instruction reference in Appendix A. The cache miss penalty bytes are based on the 80486 using the fastest bus it can support. The processor 32-bit burst speed is defined as r-b-w, where r, b, and w are defined as:

r	The number of clocks in the first cycle of a burst read or the number of clocks per data cycle in a nonburst read.
b	The number of clocks for the second and subsequent cycles in a burst read.
w	The number of clocks for a write.

  The fastest bus supported is 2-1-2 with zero wait states. The cache miss clocks assume this bus. For slower buses, r-2 clocks should be added for the first doubleword accessed.

- Instructions that read multiple consecutive data items and miss the cache are assumed to start their first access on a 16-byte (paragraph) boundary. If not, an extra cache fill may be required, which will add up to r+3*b clocks to the cache miss penalty.
- The cache is filled before subsequent access to the line is allowed. If a read operation misses in the cache while a cache fill is already in progress, due to a previous read or prefetch, the read must wait for the cache to fill. A read or write must also wait for a cache line to fill completely before it can be accessed.
- Memory operands are aligned. For each access to an unaligned operand, add three clocks. The number of memory accesses for each instruction is listed in the instruction reference in Appendix A.
- The target of a jump instruction is assumed to be in the cache. If not, add r clocks to allow time for accessing the target instruction of a jump. If the target instruction is not completely contained in the first doubleword read, add a maximum of 3*b clocks. If the target instruction is not completely contained in the first 16-byte burst (due to misalignment), add a maximum of r+3*b clocks.
- Page translations "hit" in the TLB. A TLB miss will add 13, 21, or 28 clocks to the timing depending on whether the accessed and/or dirty bits in neither, one, or both of the page entries needs to be set in memory. This penalty assumes that neither page table entry is in the data cache and that a page fault does not occur on the address translation.
- No invalidate cycles contend with the instruction for use of the cache. A one-clock penalty is imposed for each invalidate cycle that contends with the CPU for the internal cache/external bus.
- Effective address calculations use a base and no index register. If the effective address calculation uses both a base and index register, a one-clock penalty *may* be added to the timing.
- A base register used in an effective address calculation is not the destination of the preceding instruction. A one-clock penalty is imposed for back-to-back operations on the same register. Note that specialized on-chip hardware is used to ensure that the PUSH and POP instructions do not incur this penalty.
- A displacement field and immediate field are not used in the same instruction. If used together, an additional one clock *may* be required.
- There are no write-buffer delays. If there is no write-buffer delay but all the write buffers are full, then a w clock penalty is added. Intel documentation specifies that this case rarely occurs.

The Effective Address Calculation

The effective address calculation must be performed whenever a memory operand is used. On the 8086 and 8088, this calculation adds a specified number of clocks to the instruction's execution time depending on the components used in the effective address. Table 6.15 shows the possible addressing combinations and the number of clocks required to calculate the resulting effective address.

TABLE 6.15 **Additional Clocks Required for Effective Address Calculations**

Effective	Address Components	Additional Clocks Required			
		86/88	286	386	486
Displacement	[*disp*]	6	Ø	Ø	Ø
Base or Index	[BX] [BP] [SI] [DI]	5	Ø	Ø	Ø
Base + *disp* Index + *disp*	[BX + *disp*] [BP + *disp*] [SI + *disp*] [DI + *disp*]	9	Ø	Ø	Ø
Base + Index	[BX + SI] [BP + DI]	7	Ø	Ø	Ø
	[BX + DI] [BP + SI]	8	Ø	Ø	Ø
Base + Index + *disp*	[BX + SI + *disp*] [BP + DI + *disp*]	11	1	Ø	Ø
	[BX + DI + *disp*] [BP + SI + *disp*]	12	1	Ø	Ø
Segment Override	*sreg*:[]	2			
Base + scale • index (32-bit mode)	[*reg32* + *scale* • *reg32*]	–	–	1	1*

* One clock may be added to the 8Ø486 execution time depending on the state of the processor.

On the 80286 and later processors, the effective address calculation is performed by dedicated hardware and generally does not add to the execution time. The exception to this rule is that [base+index+displacement] addressing adds one clock to the 80286 and 80386 timings listed in Appendix A. The additional clock *may* be required on the 80486, depending on the internal state of the processor.

Other Timing Factors

There are several additional factors that influence the timing of an instruction. These factors generally represent different modes of operation and must be taken into account when determining timings.

Control Transfer Instructions The timings shown for the control transfer instructions, including JMP, CALL, and INT, include any additional clocks required to reinitialize the instruction queue starting at the target of the transfer. In addition, the time required to fetch the target instruction is also included.

LTR

Load Task Register (PM)

DESCRIPTION `LTR op`

LTR loads the task register from the operand specified by *op* and marks the loaded Task State Segment (TSS) busy. No task switch is performed.

The LTR instruction is typically used in operating-system software.

ALGORITHM `TR=op`

FLAGS

O	D	I	T	S	Z	A	P	C

No flags are affected

LTR *reg16/mem16*

00001111	00000000	*mod*	011	*r/m*	*disp*

disp=0- or 2-byte displacement (16-bit address mode)
0- or 4-byte displacement (32-bit address mode)

TIMING

Operands	x	88	86	286	386	486
reg16	0	–	–	PM:17	PM:23	PM:20
mem16	1	–	–	PM:19	PM:27	PM:20

MOV

Move Data

DESCRIPTION `MOV` op_1,op_2

MOV transfers a byte, word, or doubleword from op_2 to op_1.

The MOV instruction is the workhorse of the 80x86 family for transferring data between registers and memory. And while all programmers should be familiar with the different instruction forms, the assembler will usually generate the most efficient form for the move.

Note that if the MOV instruction is to be valid, both operands must be the same size. The instruction MOV BX,AL, for example, is both logically and syntactically incorrect.

The CS, IP, and FLAGS registers cannot be accessed with this instruction. Special forms are provided for the special registers available on the advanced processors.

When a 16-bit segment register is specified as the source operand, the MOV instruction behaves differently depending on whether 16- or 32-bit operand size is in effect. If the operand size is 32-bits and the destination (op_1) is a register, the segment register is copied into the low-order 16 bits of the destination register and the high-order 16 bits of the destination register are undefined. If the operand size is 16-bits, the segment register is copied into the low-order 16 bits of the destination register and the high-order 16 bits of the destination register are unchanged. If the destination is a memory operand, the segment register is written to memory as a 16-bit quantity, regardless of the current operand size; bits 16-31 of the destination should be cleared if necessary.

ALGORITHM $op_1=op_2$

NOTES

1. When the MOV *sreg,reg* form is used on the 8088 and 8086, interrupts will be cleared until the following instruction has executed. This property was designed to allow the stack segment (SS) and stack pointer (SP) registers to be set without an interrupt occurring between the two instructions. If such an interrupt did occur, the SS:SP register pair would most likely point to an invalid stack and cause the system to crash. Early versions of the 8088 and 8086 chips did not implement this function correctly. The 80286 and later processors disable interrupts only when *sreg*=SS.
2. On some versions of the 80386, breakpoints specified by the debug registers DR0-DR3 may produce spurious breakpoints after a MOV from CR3, TR6, or TR7 is executed. The contents of DR0-DR3 are not affected. This condition will persist until the processor executes the next jump instruction. The breakpoint instruction (opcode CCh) and the single-step trap are not affected.

To avoid this situation, before MOVing from CR3, TR6, or TR7, breakpoints should be disabled. After the MOV, a jump should be executed, then breakpoints should be re-enabled.

3. The 80386 executes the MOV to/from special registers (CR*n*, DR*n*, TR*n*) regardless of the setting of the MOD field. The MOD field should be set to 11b, but an early 80386 documentation error indicated that the MOD field value was a don't care. Early versions of the 80486 detect a MOD≠11b as an illegal opcode. This was changed in later versions to ignore the value of MOD. Assemblers that generate MOD≠11b for these instructions will fail on some 80486s.
4. On the 80386, the MOV to/from DR4 and DR5 instructions were aliased to MOV to/from DR6 and DR7, respectively. Early versions of the 80486 generate an invalid opcode for this encoding. Later versions of the 80486 execute the aliased instructions.
5. If a prefetch is pending when a MOV TR5,*reg32* instruction is executed with the control bits (bits 0 and 1) set for a cache read, write, or flush, the A-C0 step of the 80486 processor may hang. To avoid this, use the following code sequence:

```
          JMP    Label    ;Flush the prefetch queue and
                          ; begin the first prefetch at Label
ALIGN 16                  ;Start on a 16-byte boundary
Label:    NOP             ;Lets 2nd prefetch begin
          IN     AL,port  ;Ensures both prefetches complete
          MOV    TR5,EAX  ;EAX contains the new TR5 value
          NOP             ;These NOPs ensure no new prefetch
          NOP             ; started until MOV TR5 completes
```

6. On the A-C0 step of the 80486, if the MOV CR0,*reg32* instruction is used to disable the cache, a line in the cache may be corrupted. Since this instruction is not typically used by applications, this problem should not occur. The code shown below will avoid the problem. Note that the NMI and faults/traps should not occur during this code sequence.

```
          PUSHFD
          CLI
          MOV      BL,CS:Label
          MOV      CRØ,EAX
Label:    POPFD
```

FLAGS

O	D	I	T	S	Z	A	P	C

No flags are affected

89 7e fc

MOV *reg/mem,reg/mem*

7e 01111110
1 7 6

	2	3	3	
100010dw	*mod*	*reg*	*r/m*	*disp*

d=0, *op*$_1$=*mod*+*r/m* *op*$_2$=*reg*
1, *op*$_1$=*reg* *op*$_2$=*mod*+*r/m*

w=0 operands are 8-bit
1 operands are 16-bit (16-bit operand mode)
32-bit (32-bit operand mode)

disp=0- or 2-byte displacement (16-bit address mode)
0- or 4-byte displacement (32-bit address mode)

TIMING

Operands	x	88	86	286	386	486
reg,reg	0	2	2	2	2	1
reg,mem	1	B:8+EA W:12+EA	8+EA	3	4	1
mem,reg	1	B:9+EA W:13+EA	9+EA	5	2	1

MOV *reg,immed*

1011w	*reg*	*immed*

w=0 operands are 8-bit
1 operands are 16-bit (16-bit operand mode)
32-bit (32-bit operand mode)

immed=1- or 2-byte immediate data (16-bit address mode)
1- or 4-byte immediate data (32-bit address mode)

TIMING

Operands	x	88	86	286	386	486
reg,immed	0	4	4	2	2	1

MOV *reg/mem,immed*

1100011w	*mod*	000	*r/m*	*immed*

w=0 operands are 8-bit
1 operands are 16-bit (16-bit operand mode)
32-bit (32-bit operand mode)

immed=1- or 2-byte immediate data (16-bit address mode)
1- or 4-byte immediate data (32-bit address mode)

Note: This encoding is not normally used to move immediate data into a register. The MOV *reg,immed* instruction is provided for this purpose and the opcode occupies only a single byte.