Table 2-17. Key to Instruction Coding Formats

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction.
source-table	XLAT	Name of memory translation table addressed by register BX.
target	JMP, CALL	A label to which control is to be transferred directly, or a register or memory location whose *content* is the address of the location to which control is to be transferred indirectly.
short-label	cond. transfer, iteration control	A label to which control is to be conditionally transferred; must lie within −128 to +127 bytes of the first byte of the next instruction.
accumulator	IN, OUT	Register AX for word transfers, AL for bytes.
port	IN, OUT	An I/O port number; specified as an immediate value of 0-255, or register DX (which contains port number in range 0-64k).
source-string	string ops.	Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered.
dest-string	string ops.	Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.
count	shifts, rotates	Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0-255).
interrupt-type	INT	Immediate value of 0-255 identifying interrupt pointer number.
optional-pop-value	RET	Number of bytes (0-64k, ordinarily an even number) to discard from stack.
external-opcode	ESC	Immediate value (0-63) that is encoded in the instruction for use by an external processor.

Table 2-18. Key to Flag Effects

IDENTIFIER	EXPLANATION
(blank)	not altered
0	cleared to 0
1	set to 1
X	set or cleared according to result
U	undefined—contains no reliable value
R	restored from previously-saved value

For control transfer instructions, the timings given include any additional clocks required to reinitialize the instruction queue as well as the time required to fetch the target instruction. For instructions executing on an 8086, four clocks should be added for each instruction reference to a word operand located at an odd memory address to reflect any additional operand bus cycles required. Similarly for instructions executing on an 8088, four clocks should be added to each instruction reference to a 16-bit memory operand; this includes all stack operations. The required number of data references is listed in table 2-21 for each instruction to aid in this calculation.

Several additional factors can increase actual execution time over the figures shown in table 2-21. The time provided assumes that the instruction has already been prefetched and that it is waiting in the instruction queue, an assumption that is valid under most, but not all, operating conditions. A series of fast executing (fewer than two clocks per opcode byte) instructions can drain the queue and increase execution time. Execution time also is slightly impacted by the interaction of the EU and BIU when memory operands must be read or written. If the EU needs access to memory, it may have to wait for up to one clock if the BIU has already started an instruction fetch bus cycle. (The EU can detect the need for a memory operand and post a bus request far enough in advance of its need for this operand to avoid waiting a full 4-clock bus cycle). Of course the EU does not have to wait if the queue is full, because the BIU is idle. (This discussion assumes

Table 2-19. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	An 8- or 16-bit general register
reg 16	A 16-bit general register
seg-reg	A segment register
accumulator	Register AX or AL
immediate	A constant in the range 0-FFFFH
immed8	A constant in the range 0-FFH
memory	An 8- or 16-bit memory location(1)
mem8	An 8-bit memory location(1)
mem16	A 16-bit memory location(1)
source-table	Name of 256-byte translate table
source-string	Name of string addressed by register SI
dest-string	Name of string addressed by register DI
DX	Register DX
short-label	A label within −128 to +127 bytes of the end of the instruction
near-label	A label in current code segment
far-label	A label in another code segment
near-proc	A procedure in current code segment
far-proc	A procedure in another code segment
memptr16	A word containing the offset of the location in the current code segment to which control is to be transferred(1)
memptr32	A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred(1)
regptr16	A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred
repeat	A string instruction repeat prefix

(1)Any addressing mode—direct, register indirect, based, indexed, or based indexed—may be used (see section 2.8).

Table 2-20. Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP + DI, BX + SI	7
	BP + SI, BX + DI	8
Displacement + Base + Index	BP + DI + DISP BX + SI + DISP	11
	BP + SI + DISP BX + DI + DISP	12

*Add 2 clocks for segment override

that the BIU can obtain the bus on demand, i.e., that no other processors are competing for the bus.)

With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings given in table 2-21. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions, however, is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the SDK-86 or the iSBC 86/12™ board.

Table 2-21. Instruction Set Reference Data

AAA	**AAA** (no operands) ASCII adjust for addition			**Flags** O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAA

AAD	**AAD** (no operands) ASCII adjust for division			**Flags** O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	60	—	2	AAD

AAM	**AAM** (no operands) ASCII adjust for multiply			**Flags** O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	83	—	1	AAM

AAS	**AAS** (no operands) ASCII adjust for subtraction			**Flags** O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

ADC	**ADC** destination,source Add with carry			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADC AX, SI
register, memory	9+EA	1	2-4	ADC DX, BETA [SI]
memory, register	16+EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17+EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

ADD	**ADD** destination,source Addition			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADD CX, DX
register, memory	9+EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16+EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory, immediate	17+EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

AND	**AND** destination,source Logical and			**Flags** O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	AND AL,BL
register, memory	9+EA	1	2-4	AND CX,FLAG_WORD
memory, register	16+EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	—	3-4	AND CX,0F0H
memory, immediate	17+EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

CALL	**CALL** target Call a procedure			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FAR_PROC
memptr 16	21+EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37+EA	4	2-4	CALL [BX].TASK [SI]

CBW	**CBW** (no operands) Convert byte to word			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CBW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

CLC	CLC (no operands) Clear carry flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLC

CLD	CLD (no operands) Clear direction flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLD

CLI	CLI (no operands) Clear interrupt flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLI

CMC	CMC (no operands) Complement carry flag			Flags O D I T S Z A P C X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CMC

CMP	CMP destination,source Compare destination to source			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	CMP BX, CX
register, memory	9+EA	1	2-4	CMP DH, ALPHA
memory, register	9+EA	1	2-4	CMP [BP+2], SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10+EA	1	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

CMPS	CMPS dest-string,source-string Compare string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9+22/rep	2/rep	1	REPE CMPS ID, KEY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

CWD	CWD (no operands) Convert word to doubleword			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	5	—	1	CWD

DAA	DAA (no operands) Decimal adjust for addition			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAA

DAS	DAS (no operands) Decimal adjust for subtraction			Flags O D I T S Z A P C U X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAS

DEC	DEC destination Decrement by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15+EA	2	2-4	DEC ARRAY [SI]

DIV	DIV source Division, unsigned			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) +EA	1	2-4	DIV ALPHA
mem16	(150-168) +EA	1	2-4	DIV TABLE [SI]

ESC	ESC external-opcode,source Escape			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immediate, memory	8+EA	1	2-4	ESC 6,ARRAY [SI]
immediate, register	2	—	2	ESC 20,AL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

HLT	**HLT** (no operands) Halt			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	HLT

IDIV	**IDIV** source Integer division			**Flags** O D I T S Z A P C U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) +EA	1	2-4	IDIV DIVISOR_BYTE [SI]
mem16	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL	**IMUL** source Integer multiplication			**Flags** O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) +EA	1	2-4	IMUL RATE_BYTE
mem16	(134-160) +EA	1	2-4	IMUL RATE_WORD [BP] [DI]

IN	**IN** accumulator,port Input byte or word			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed8	10	1	2	IN AL, 0FFEAH
accumulator, DX	8	1	1	IN AX, DX

INC	**INC** destination Increment by 1			**Flags** O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI] [BX]

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

INT	**INT** interrupt-type Interrupt			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8 (type = 3) immed8 (type ≠ 3)	52 51	5 5	1 2	INT 3 INT 67

INTR†	**INTR** (external maskable interrupt) Interrupt if INTR and IF=1			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	61	7	N/A	N/A

INTO	**INTO** (no operands) Interrupt if overflow			**Flags** O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	53 or 4	5	1	INTO

IRET	**IRET** (no operands) Interrupt Return			**Flags** O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	24	3	1	IRET

JA/JNBE	**JA/JNBE** short-label Jump if above/Jump if not below nor equal			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JA ABOVE

JAE/JNB	**JAE/JNB** short-label Jump if above or equal/Jump if not below			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JAE ABOVE_EQUAL

JB/JNAE	**JB/JNAE** short-label Jump if below/Jump if not above nor equal			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JB BELOW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†INTR is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JBE/JNA	JBE/JNA short-label Jump if below or equal/Jump if not above			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNA NOT_ABOVE

JC	JC short-label Jump if carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JC CARRY_SET

JCXZ	JCXZ short-label Jump if CX is zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	JCXZ COUNT_DONE

JE/JZ	JE/JZ short-label Jump if equal/Jump if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JZ ZERO

JG/JNLE	JG/JNLE short-label Jump if greater/Jump if not less nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JG GREATER

JGE/JNL	JGE/JNL short-label Jump if greater or equal/Jump if not less			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JGE GREATER_EQUAL

JL/JNGE	JL/JNGE short-label Jump if less/Jump if not greater nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JL LESS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JLE/JNG	JLE/JNG short-label Jump if less or equal/Jump if not greater			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNG NOT_GREATER

JMP	JMP target Jump			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN_SEGMENT
far-label	15	—	5	JMP FAR_LABEL
memptr16	18+EA	1	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24+EA	2	2-4	JMP OTHER.SEG [SI]

JNC	JNC short-label Jump if not carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ short-label Jump if not equal/Jump if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNE NOT_EQUAL

JNO	JNO short-label Jump if not overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNO NO_OVERFLOW

JNP/JPO	JNP/JPO short-label Jump if not parity/Jump if parity odd			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPO ODD_PARITY

JNS	JNS short-label Jump if not sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNS POSITIVE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JO	JO short-label Jump if overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JO SIGNED_OVRFLW

JP/JPE	JP/JPE short-label Jump if parity/Jump if parity even			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPE EVEN_PARITY

JS	JS short-label Jump if sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JS NEGATIVE

LAHF	LAHF (no operands) Load AH from flags			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	LAHF

LDS	LDS destination,source Load pointer using DS			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LDS SI,DATA.SEG [DI]

LEA	LEA destination,source Load effective address			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem16	2+EA	—	2-4	LEA BX, [BP] [DI]

LES	LES destination,source Load pointer using ES			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LES DI, [BX].TEXT_BUFF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

LOCK	LOCK (no operands) Lock bus			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	LOCK XCHG FLAG,AL

LODS	LODS source-string Load string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string (repeat) source-string	12 9+13/rep	1 1/rep	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	LOOP short-label Loop			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17/5	—	2	LOOP AGAIN

LOOPE/LOOPZ	LOOPE/LOOPZ short-label Loop if equal/Loop if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	LOOPE AGAIN

LOOPNE/LOOPNZ	LOOPNE/LOOPNZ short-label Loop if not equal/Loop if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	—	2	LOOPNE AGAIN

NMI†	NMI (external nonmaskable interrupt) Interrupt if NMI = 1			Flags O S I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†NMI is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

MOV	**MOV** destination,source Move			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX,CX
register, memory	8+EA	1	2-4	MOV BP, STACK_TOP
memory, register	9+EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10+EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9+EA	1	2-4	MOV [BX].SEG_SAVE, CS

MOVS	**MOVS** dest-string,source-string Move string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9+17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** (no operands) Move string (byte/word)			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	18	2	1	MOVSB
(repeat) (no operands)	9+17/rep	2/rep	1	REP MOVSW

MUL	**MUL** source Multiplication, unsigned			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	70-77	—	2	MUL BL
reg16	118-133	—	2	MUL CX
mem8	(76-83) +EA	1	2-4	MUL MONTH [SI]
mem16	(124-139) +EA	1	2-4	MUL BAUD_RATE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

NEG	NEG destination Negate			Flags O D I T S Z A P C X X X X X 1*
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register memory	3 16+EA	— 2	2 2-4	NEG AL NEG MULTIPLIER

*0 if destination = 0

NOP	NOP (no operands) No Operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3	—	1	NOP

NOT	NOT destination Logical not			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register memory	3 16+EA	— 2	2 2-4	NOT AX NOT CHARACTER

OR	OR destination,source Logical inclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	OR AL, BL
register, memory	9+EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16+EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	—	2-3	OR AL, 01101100B
register, immediate	4	—	3-4	OR CX,01H
memory, immediate	17+EA	2	3-6	OR [BX].CMD_WORD,0CFH

OUT	OUT port,accumulator Output byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8, accumulator	10	1	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP	POP destination Pop word off stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17+EA	2	2-4	POP PARAMETER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

POPF	**POPF** (no operands) Pop flags off stack			**Flags** O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	8	1	1	POPF

PUSH	**PUSH** source Push word onto stack			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16+EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	**PUSHF** (no operands) Push flags onto stack			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	10	1	1	PUSHF

RCL	**RCL** destination,count Rotate left through carry			**Flags** O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCL CX, 1
register, CL	8+4/bit	—	2	RCL AL, CL
memory, 1	15+EA	2	2-4	RCL ALPHA, 1
memory, CL	20+EA+4/bit	2	2-4	RCL [BP].PARM, CL

RCR	**RCR** designation,count Rotate right through carry			**Flags** O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCR BX, 1
register, CL	8+4/bit	—	2	RCR BL, CL
memory, 1	15+EA	2	2-4	RCR [BX].STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	RCR ARRAY [DI], CL

REP	**REP** (no operands) Repeat string operation			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REP MOVS DEST, SRCE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

REPE/REPZ	REPE/REPZ (no operands) Repeat string operation while equal/while zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPE CMPS DATA, KEY

REPNE/REPNZ	REPNE/REPNZ (no operands) Repeat string operation while not equal/not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPNE SCAS INPUT_LINE

RET	RET optional-pop-value Return from procedure			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(intra-segment, no pop)	8	1	1	RET
(intra-segment, pop)	12	1	3	RET 4
(inter-segment, no pop)	18	2	1	RET
(inter-segment, pop)	17	2	3	RET 2

ROL	ROL destination,count Rotate left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Examples**
register, 1	2	—	2	ROL BX, 1
register, CL	8+4/bit	—	2	ROL DI, CL
memory, 1	15+EA	2	2-4	ROL FLAG_BYTE [DI],1
memory, CL	20+EA+4/bit	2	2-4	ROL ALPHA , CL

ROR	ROR destination,count Rotate right			Flags O D I T S Z A P C X X
Operand	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	ROR AL, 1
register, CL	8+4/bit	—	2	ROR BX, CL
memory, 1	15+EA	2	2-4	ROR PORT_STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	ROR CMD_WORD, CL

SAHF	SAHF (no operands) Store AH into flags			Flags O D I T S Z A P C R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	SAHF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SAL/SHL	SAL/SHL destination,count Shift arithmetic left/Shift logical left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
register,1	2	—	2	SAL AL,1
register, CL	8+4/bit	—	2	SHL DI, CL
memory,1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20+EA+4/bit	2	2-4	SAL STORE_COUNT, CL

SAR	SAR destination,source Shift arithmetic right			Flags O D I T S Z A P C X X X U X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SAR DX, 1
register, CL	8+4/bit	—	2	SAR DI, CL
memory, 1	15+EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20+EA+4/bit	2	2-4	SAR N_BLOCKS, CL

SBB	SBB destination,source Subtract with borrow			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SBB BX, CX
register, memory	9+EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16+EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS	SCAS dest-string Scan string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9+15/rep	1/rep	1	REPNE SCAS BUFFER

SEGMENT†	SEGMENT override prefix Override to specified segment			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	MOV SS:PARAMETER, AX

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†ASM-86 incorporates the segment override prefix into the operand specification and not as a separate instruction. SEGMENT is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SHR	SHR destination,count Shift logical right			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SHR SI, 1
register, CL	8+4/bit	—	2	SHR SI, CL
memory, 1	15+EA	2	2-4	SHR ID_BYTE [SI] [BX], 1
memory, CL	20+EA+4/bit	2	2-4	SHR INPUT_WORD, CL

SINGLE STEP†	SINGLE STEP (Trap flag interrupt) Interrupt if TF = 1			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

STC	STC (no operands) Set carry flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STC

STD	STD (no operands) Set direction flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STD

STI	STI (no operands) Set interrupt enable flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STI

STOS	STOS dest-string Store byte or word string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†SINGLE STEP is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

SUB	**SUB** destination,source Subtraction			**Flags** O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SUB CX, BX
register, memory	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16+EA	2	2-4	SUB [BP+2], CL
accumulator, immediate	4	—	2-3	SUB AL, 10
register, immediate	4	—	3-4	SUB SI, 5280
memory, immediate	17+EA	2	3-6	SUB [BP].BALANCE, 1000

TEST	**TEST** destination,source Test or non-destructive logical and			**Flags** O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	TEST SI, DI
register, memory	9+EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	—	2-3	TEST AL, 00100000B
register, immediate	5	—	3-4	TEST BX, 0CC4H
memory, immediate	11+EA	—	3-6	TEST RETURN_CODE, 01H

WAIT	**WAIT** (no operands) Wait while $\overline{\text{TEST}}$ pin not asserted			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3 + 5n	—	1	WAIT

XCHG	**XCHG** destination,source Exchange			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, reg16	3	—	1	XCHG AX, BX
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	—	2	XCHG AL, BL

XLAT	**XLAT** source-table Translate			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-table	11	1	1	XLAT ASCII_TAB

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Table 2-21. Instruction Set Reference Data (Cont'd.)

XOR	XOR destination,source Logical exclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	XOR CX, BX
register, memory	9+EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16+EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

2.8 Addressing Modes

The 8086 and 8088 provide many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways. These addressing modes greatly extend the flexibility and convenience of the instruction set. This section briefly describes register and immediate operands and then covers the 8086/8088 memory and I/O addressing modes in detail.

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all instruction forms. This is because the register "addresses" are encoded in instructions in just a few bits, and because these operations are performed entirely within the CPU (no bus cycles are run). Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; like a register operand, no bus cycles need to be run to obtain an immediate operand. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the EU has direct access to register and immediate operands, memory operands must be transferred to or from the CPU over the bus. When the EU needs to read or write a memory operand, it must pass an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register producing a 20-bit physical address and then executes the bus cycle(s) needed to access the operand.

The Effective Address

The offset that the EU calculates for a memory operand is called the operand's effective address or EA. It is an unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides. The EU can calculate the effective address in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

Figure 2-34 shows that the execution unit calculates the EA by summing a displacement, the content of a base register and the content of an index register. The fact that any combination of these three components may be present in a given instruction gives rise to the variety of 8086/8088 memory addressing modes.

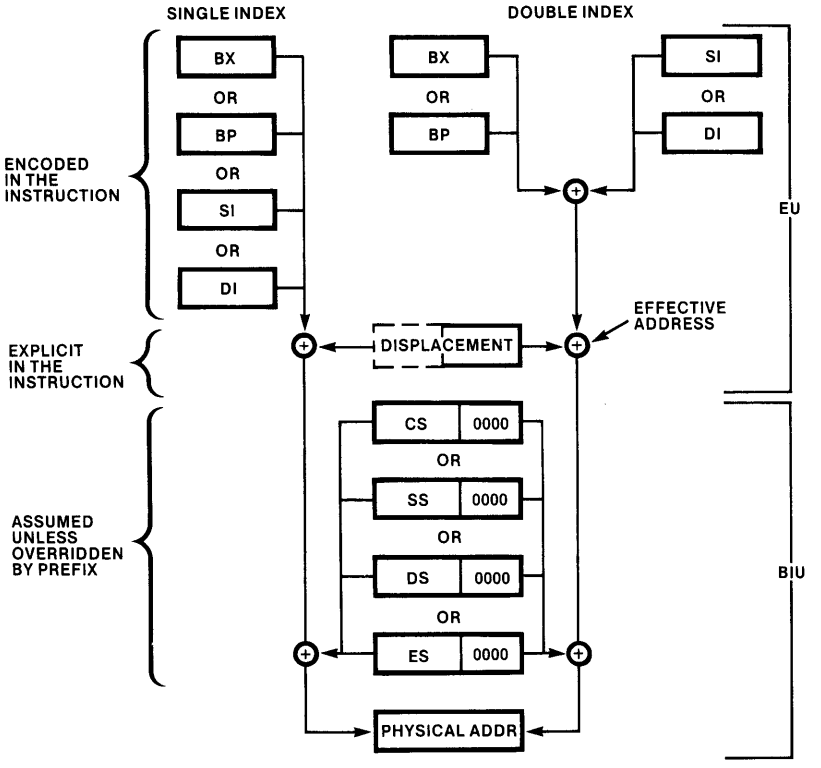

Figure 2-34. Memory Address Computation

The displacement element is an 8- or 16-bit number that is contained in the instruction. The displacement generally is derived from the position of the operand name (a variable or label) in the program. It also is possible for a programmer to modify this value or to specify the displacement explicitly.

A programmer may specify that either BX or BP is to serve as a base register whose content is to be used in the EA computation. Similarly, either SI or DI may be specified as an index register. Whereas the displacement value is a constant, the contents of the base and index registers may change during execution. This makes it possible for one instruction to access different memory locations as determined by the current values in the base and/or index registers.

It takes time for the EU to calculate a memory operand's effective address. In general, the more elements in the calculation, the longer it takes. Table 2-20 shows how much time is required to compute an effective address for any combination of displacement, base register and index register.

Direct Addressing

Direct addressing (see figure 2-35) is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing typically is used to access simple variables (scalars).

Register Indirect Addressing

The effective address of a memory operand may be taken directly from one of the base or index registers as shown in figure 2-36. One instruction can operate on many different memory locations if the value in the base or index register is updated

appropriately. The LEA (load effective address) and arithmetic instructions might be used to change the register value.

Note that *any* 16-bit general register may be used for register indirect addressing with the JMP or CALL instructions.

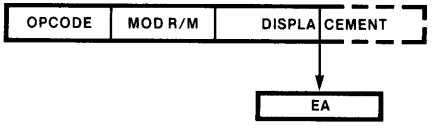

Figure 2-35. Direct Addressing

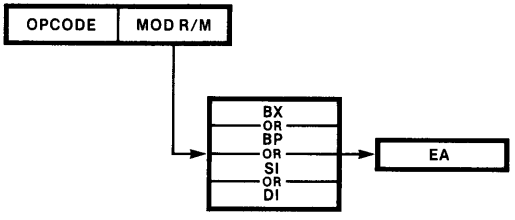

Figure 2-36. Register Indirect Addressing

Based Addressing

In based addressing (figure 2-37), the effective address is the sum of a displacement value and the content of register BX or register BP. Recall that specifying BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data (see section 2.10 for examples).

Based addressing also provides a straightforward way to address structures which may be located at different places in memory (see figure 2-38). A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

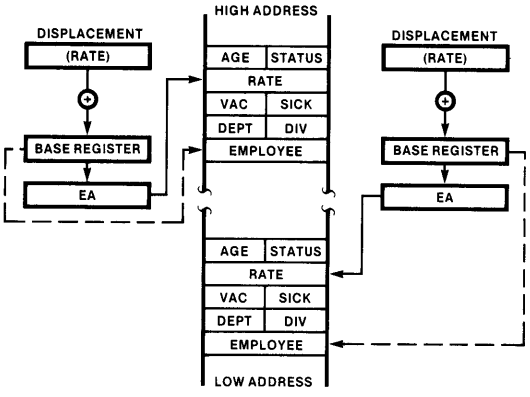

Figure 2-38. Accessing a Structure With Based Addressing

Indexed Addressing

In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register (SI or DI) as shown in figure 2-39. Indexed addressing often is

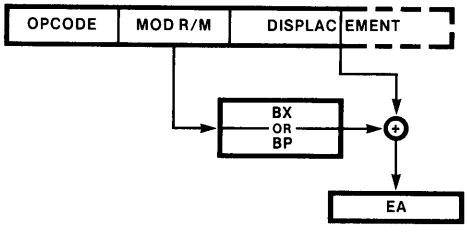

Figure 2-37. Based Addressing

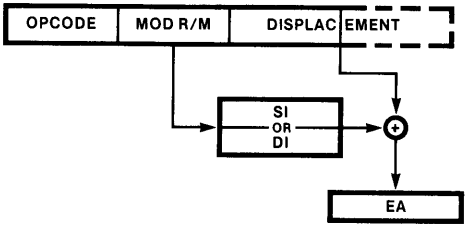

Figure 2-39. Indexed Addressing

used to access elements in an array (see figure 2-40). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). Since all array elements are the same length, simple arithmetic on the index register will select any element.

Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register and a displacement (see figure 2-41). Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack (see figure 2-42). Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

Arrays contained in structures and matrices (two-dimension arrays) also could be accessed with based indexed addressing.

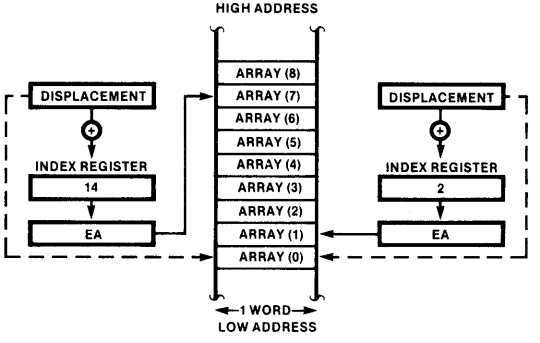

Figure 2-40. Accessing an Array With Indexed Addressing

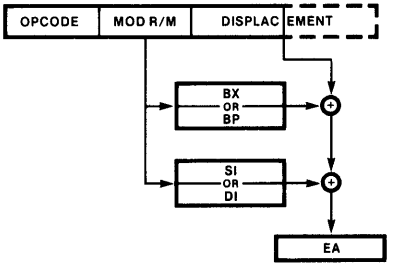

Figure 2-41. Based Indexed Addressing

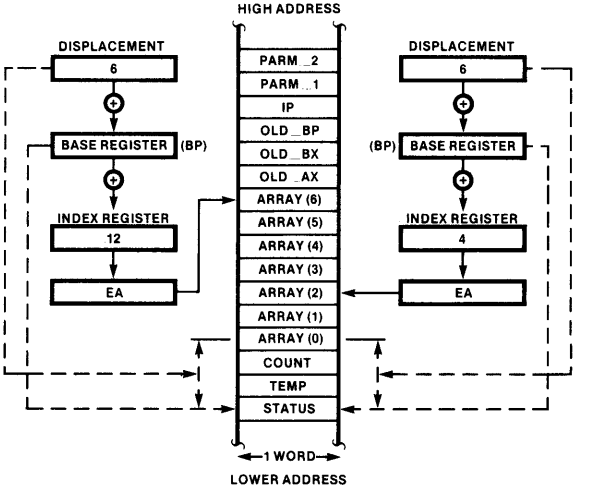

Figure 2-42. Accessing a Stack Array With Based Indexed Addressing

String Addressing

String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly as shown in figure 2-43. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, the CPUs automatically adjust SI and DI to obtain subsequent bytes or words.

I/O Port Addressing

If an I/O port is memory mapped, any of the memory operand addressing modes may be used to access the port. For example, a group of terminals can be accessed as an "array." String instructions also can be used to transfer data to memory-mapped ports with an appropriate hardware interface. Section 2.10 contains examples of addressing memory-mapped I/O ports.

Two different addressing modes can be used to access ports located in the I/O space; these are illustrated in figure 2-44. In direct port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and can range from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

2.9 Programming Facilities

A comprehensive integrated set of tools supports 8086/8088 software development. These tools are programs that run on Intellec® 800 or Series II Microcomputer Development Systems under the ISIS-II operating system, the same hardware and operating system used to develop software for the 8080 and the 8085. Since the 8086 and 8088 are software-compatible with one another, the same tools are used for both processors to provide programmers with a uniform development environment.

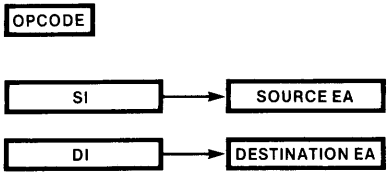

Figure 2-43. String Operand Addressing

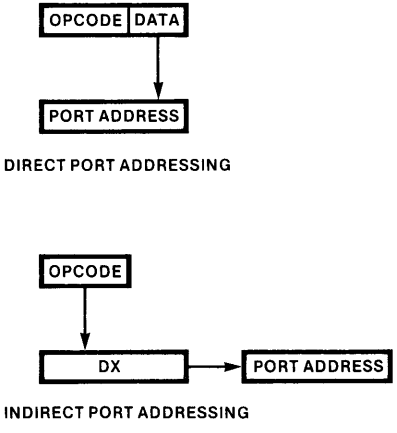

Figure 2-44. I/O Port Addressing

Software Development Overview

A program that will ultimately execute on an 8086- or 8088-based system is developed in steps (see figure 2-45). The overall program is composed of functional units called modules. For purposes of this discussion, a module is a section of code that is separately created, edited, and compiled or assembled. A very small program might consist of a single module; a large program could be comprised of 100 or more modules. The 8086/8088 LINK-86 utility binds modules together into a single program. (The module structure of a program is critical to its successful development and maintenance; see section 2.10 for guidelines.)

8086 and 8088 modules can be written in either PL/M-86 or ASM-86 (see table 2-22). PL/M-86 is a high-level language suitable for most microprocessor applications. It is easy to use, even by programmers who have little experience with microprocessors. Because it reduces software development time, PL/M-86 is ideal for most of the programming in any application, especially applications that must get to market quickly.

ASM-86 is the 8086/8088 assembly language. ASM-86 provides the programmer who is familiar with the CPU architecture, access to all processor features. For critical code segments within programs that make sophisticated use of the hardware, have extremely demanding performance or memory constraints, ASM-86 is the best choice.

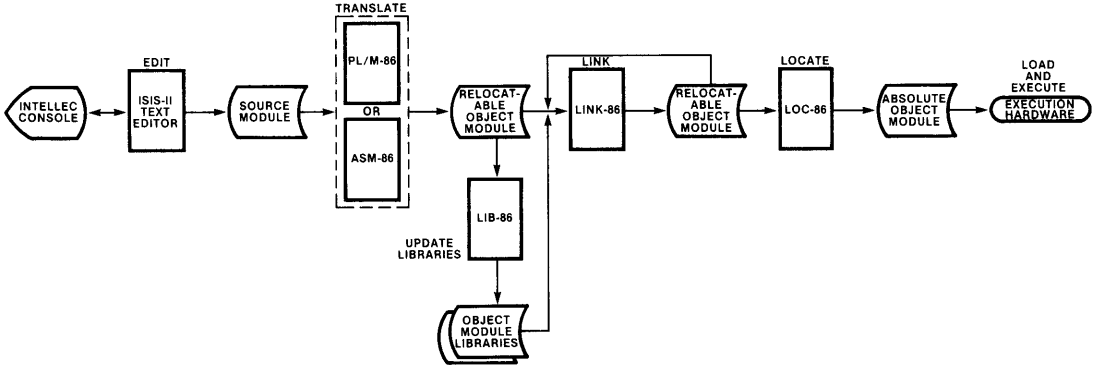

Figure 2-45. Software Development Process

Table 2-22. PL/M-86/ASM-86 Characteristics

PL/M-86	ASM-86
• Fast Development	• Fastest Execution Speed
• Less Programmer Training	• Smallest Memory Requirements
• Detailed Hardware Knowledge Not Required	• Access To All Processor Facilities

The languages are completely compatible, and a judicious combination of the two often makes good sense. Prototype software can be developed rapidly with PL/M-86. When the system is operating correctly, it can be analyzed to see which sections can best profit from being written in ASM-86. Since the logic of these sections already has been debugged, selective rewriting can be done quickly and with low risk.

Each PL/M-86 or ASM-86 module (called a source moduel) is keyed into the Intellec® system using the ISIS-II text editor and is stored as a diskette file. This source file is then input to the appropriate language translator (ASM-86 assembler or PL/M-86 compiler). The language translator creates a diskette file from the source file, which is called a relocatable object module. The translator also lists the program and flags any errors detected during the translation. The relocatable object module contains the 8086/8088 machine instructions that the translator created from the statements in the source module. The term "relocatable" refers to the fact that all references to memory locations in the module are relative, rather than being absolute memory addresses. The module generally is not executable until the relative references are changed to the actual memory locations where the module will reside in the execution system's memory. The process of changing the relative references to absolute memory locations is called locating.

There are very good reasons for not locating modules when they are translated. First, the execution system's physical memory configuration (where RAM and ROM/PROM segments are actually located in the megabyte memory space) may not be known at the time the modules are written. Second, it is desirable to be able to use a common module (e.g., a square root routine) in more than one system. If absolute addresses were assigned at translation time, the common module would either have to occupy the same physical addresses in every system, or separate versions with different addresses would have to be maintained for each system. When locating is deferred, a single version of a common routine can be used by any number of systems. Finally, the locations of modules typically change as a system is developed, maintained and enhanced. Separating the location process from the translation process means that as modifications are made, unchanged modules only need to be relocated, not retranslated.

Relocatable object modules may be placed into special files called libraries, using the LIB-86 library manager program. Libraries provide a convenient means of collecting groups of related modules so that they can be accessed automatically by the LINK-86 program.

When enough relocatable object modules have been created to test the system, or part of it, the modules are linked and located. Linking combines all the separate modules into a single program. Locating changes the relative memory references in the program to the actual memory locations where the program will be loaded in the execution system. The link and locate process also is referred to as R & L, for relocation and linkage.

Two other programs round out the software development tools available for the 8086 and 8088. OH-86 converts an absolute object file into a hexadecimal format used by some PROM programmers and system loaders (for example, the SDK-86 and iSBC 957™ loaders). CONV-86 can do most of the conversion work required to translate 8080/8085 assembly language source modules into ASM-86 source modules.

The 8086/8088 software development facilities are covered in more detail in the remainder of this section. However, these are only introductions to

the use of these tools. Complete documentation is available in the following publications available from Intel's Literature Department:

ISIS-II:

ISIS-II System User's Guide, Order No. 9800306

ASM-86:

MCS-86 Assembly Language Reference Manual, Order No. 9800640

MCS-86 Assembler Operating Instructions for ISIS-II Users, Order No. 9800641

PL/M-86:

PL/M-86 Programming Manual, Order No. 9800466

ISIS-II PL/M-86 Compiler Operator's Manual, Order No. 9800478

LINK-86, LOC-86, LIB-86, OH-86:

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, Order No. 9800639

CONV-86:

MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users, Order No. 9800642

PL/M-86

PL/M-86 is a general-purpose, high-level language for programming the 8086 and 8088 microprocessors. It is an extension of PL/M-80, the most widely-used, high-level programming language for microprocessors. (PL/M-80 source programs can be processed by the PL/M-86 compiler; the resulting object program is generally reduced by 15-30% in size.) PL/M-86 is suitable for all types of microprocessor software from operating systems to application programs.

PL/M-86's purpose is simple: to reduce the time and cost of developing and maintaining software for the 8086 and 8088. It accomplishes this by creating a programming environment that, for the most part, is distinct from the architecture of the CPUs. Registers, segments, addressing modes, stacks, etc., are effectively "invisible" to the PL/M-86 programmer. Instead, the processors appear to respond to simple commands and familiar algebraic expressions. The responsibility for translating these source statements into the machine instructions ultimately required to execute on the 8086/8088 is assumed by the PL/M-86 compiler. By "hiding" the details of the machine architecture, PL/M-86 encourages programmers to concentrate on solving the problem at hand. Furthermore, because PL/M-86 is closer to natural language, it is easier to "think in PL/M-86" than it is to "think in assembly language." This speeds up the expression of a program solution, and, equally important, makes that solution easier for someone other than the original programmer to understand. PL/M-86 also contains all the constructs necessary for structured programming.

Statements and Comments

A programmer builds a PL/M-86 program by writing statements and comments (see figure 2-46). There are several different types of statements in PL/M-86; they always end with a semicolon. Blanks can be used freely before, within, and after statements to improve readability. A statement also may span more than one line.

The characters "/*" start a comment, and the characters "*/" end it; any characters may be used in between. Comments do not affect the execution of a PL/M-86 program, but all good programs are thoughtfully commented. Comments are notes that document and clarify the program's operation; they may be written virtually anywhere in a PL/M-86 program.

Data Definition

Most PL/M-86 programs begin by defining the data items (variables) with which they are going to work. An individual PL/M-86 data element is called a scalar. Every scalar variable has a programmer-supplied name up to 31 characters long, and a type. PL/M-86 supports five types of scalars: byte, word, integer, real, and pointer. Table 2-23 lists the characteristics of these PL/M-86 data types.

```
/*TRAFFIC DATA RECORDER CONTROL PROGRAM*
 *VERSION 2.2, RELEASE 5, 23APR79.*
 *THIS RELEASE FIXES THREE BUGS*
 *DOCUMENTED IN PROBLEM REPORT #16.*/

/*COMPUTE TOTAL PAYMENT DUE*/
TOTAL = PRINCIPAL + INTEREST;

IF TERMINAL$READY
    THEN CALL FILL$BUFFER;
    ELSE CALL WAIT (50);      /*WAIT 50 MS FOR RESPONSE*/
```

Figure 2-46. PL/M-86 Statements and Comments

Table 2-23. PL/M-86 Data Types

TYPE	BYTES	RANGE	USAGE
BYTE	1	0 to 255	Unsigned Integer, Character
WORD	2	0 to 65,535	Unsigned Integer
INTEGER	2	−32,768 to +32,767	Signed Integer
REAL	4	1×10^{-38} to $3.37 \times 10^{+38}$	Floating Point
POINTER	2/4	N/A	Address Manipulation

Variables are defined by writing a DECLARE statement of this form:

```
DECLARE scalar-name type;
```

Options of the DECLARE statement can be used to specify an initial value for the scalar and to define a series of items in a shorthand form.

Besides scalar variables, scalar constants may be used in PL/M-86 programs (see figure 2-47). Constants may be written "as is" or may be given names to improve program clarity.

Scalars can be aggregated into named collections of data such as arrays and structures. An array is a collection of scalars of the same type (all integer, all real, etc.). Arrays are useful for representing data that has a repetitive nature. For example, monthly rainfall samples could be represented as an array of 12 elements, one for each month:

```
DECLARE RAINFALL (12) REAL;
```

Each element in an array is accessible by a number called a subscript which is the element's relative location in the array. In PL/M-86, the first element in an array has a subscript of 0; it is considered the "0th" element. Thus, RAINFALL (11) refers to December's sample. The subscript need not be a constant; variables and expressions also may be used as subscripts.

Strings of character data are typically defined as byte arrays. Characters can be accessed with subscripts or with powerful string-handling functions built into PL/M-86.

```
      10  /*DECIMAL NUMBER*/
     0AH  /*HEXADECIMAL NUMBER*/
     12Q  /*OCTAL NUMBER*/
00001010B /*BINARY NUMBER*/
    10.0  /*FLOATING POINT NUMBER*/
   1.0E1  /*FLOATING POINT NUMBER*/
     'A'  /*CHARACTER*/

/*CONSTANTS MAY BE GIVEN NAMES*/
DECLARE STATUS$PORT LITERALLY '0FFEH';
DECLARE THRESHOLD LITERALLY '98.6';
```

Figure 2-47. PL/M-86 Constants

A structure is a collection of related data elements that do not necessarily have the same type. The elements are related by virtue of "belonging" to the entity represented by the structure. Here is a simple structure declaration:

```
DECLARE BRIDGE STRUCTURE
        (SPAN          WORD,
        YR$BUILT       BYTE,
        AVG$TRAFFIC REAL);
```

The year the bridge was built could be accessed by writing BRIDGE.YR$BUILT; the structure element name is "qualified" by the dot and the structure name. This allows structures with the same element names to be distinguished from each other (e.g., HIGHWAY.YR$BUILT).

Arrays and structures can be combined into more complex data aggregates:

- array elements may be structures rather than scalars,
- a structure element may be an array,
- structures in arrays may themselves contain arrays.

Figure 2-48 provides sample PL/M-86 data declarations.

Assignment Statement

Data that has been defined can be operated on with PL/M-86 executable statements. The fundamental executable statement is the assignment statement, written in this form:

```
variable-name = expression;
```

This means "evaluate the expression and assign (move) the result to the variable."

There are three basic classes of expressions in PL/M-86; arithmetic, relational and logical (see table 2-24 and figure 2-49). All expressions are combinations of operands and operators, although an expression can consist of a single operand. Operands are variables and constants; operators vary according to the type of expression. Evaluation of an expression always yields a single result; different classes of expressions yield different types of results.

Table 2-24. Characteristics of PL/M-86 Expressions

EXPRESSION	OPERATORS	RESULT
ARITHMETIC	+, −, *, /, MOD	NUMBER
RELATIONAL	>, <, =, >=, <=	"TRUE" - FFH "FALSE" - 0H
LOGICAL	AND, OR, XOR, NOT	8/16-BIT STRING

```
    /****SCALARS****/
DECLARE SWITCH          BYTE;
DECLARE COUNT           WORD,                   /*1 SCALAR*/
        INDEX           INTEGER;                /*1 SCALAR*/
DECLARE (NET, GROSS,  TOTAL)   REAL;            /*3 SCALARS*/

    /****ARRAYS****/
DECLARE MONTH (12)      BYTE;
DECLARE TERMINAL_LINE (80)          BYTE;

    /****STRUCTURE****/
DECLARE EMPLOYEE STRUCTURE
        (ID_NUMBER                  WORD,
        DEPARTMENT                  BYTE
        RATE                        REAL);

    /****ARRAY OF STRUCTURES****/
DECLARE INVENTORY_ITEM (100)        STRUCTURE
        (PART_NUMBER                WORD,
        ON_HAND                     WORD,
        RE_ORDER                    BYTE);

    /****ARRAY WITHIN STRUCTURE****/
DECLARE COUNTY_DATA                 STRUCTURE
        (NAME (20)                  BYTE,
        TEN_YR_RAINFALL(10)         BYTE,
        PER CAPITA_INCOME           REAL);
```

Figure 2-48. PL/M-86 Data Declarations

```
/*ARITHMETIC*/
A = 2; B = 3;
B = B+ 1;                   /*B CONTAINS 4*/
C = (A*B) -2;               /*C CONTAINS 6*/
C = ((A*B) + 3) MOD 3;      /*C CONTAINS 2*/

/*RELATIONAL*/
A =2; B= 3
C = B > A;                  /*C CONTAINS 0FFH*/
C = B < > A;                /*C CONTAINS 0FFH*/
C = B = (A+1);              /*C CONTAINS 0FFH*/

/*LOGICAL*/
A = 0011$0001B;             /*$ IS FOR READABILITY*/
B = 1000$0001B;
C = NOT B;                  /*C CONTAINS 0111$1110B*/
C = A AND B;                /*C CONTAINS 0000$0001B*/
C = A OR B;                 /*C CONTAINS 1011$0001B*/
C = B XOR A;                /*C CONTAINS 1011$0000B*/
C = (A AND B) OR 0F0H;      /*C CONTAINS 1111$0001B*/
```

Figure 2-49. Expressions in PL/M-86 Assignment Statements

Program Flow Statements

Simple PL/M-86 programs can be written with just DECLARE and assignment statements. Such programs, however, execute exactly the same sequence of statements every time they are run and would not prove very useful. PL/M-86 provides statements that change the flow of control through a program. These statements allow sections of the program to be executed selectively, repeated, skipped entirely, etc.

The IF statement (figure 2-50) selects one or the other of two statements for execution depending on the result of a relational expression. The IF statement is written:

```
IF relational-expression

  THEN statement1;

  ELSE statement2;
```

Statement1 is executed if the expression is "true"; statement2 is not executed in this case. If the relation is "false," statement1 is skipped and statement2 is executed. In determining the "truth" of an expression, the IF statement only examines the low-order bit of the result (1="true"). Therefore, arithmetic and logical expressions also may be used in an IF statement.

```
A = 3; B = 5;
IF A < B
  THEN MINIMUM = 1;      /*EXECUTED*/
  ELSE MINIMUM = 2;      /*SKIPPED*/

MORE_DATA = 0FFH;
IF NOT MORE_DATA
  THEN DONE = 1;         /*SKIPPED*/
  ELSE DONE = 0;         /*EXECUTED*/

/*NESTED IF STATEMENTS*/
CLOCK_ON = 1; HOUR=24; ALARM=OFF;
IF CLOCK_ON
  THEN IF HOUR = 24
    THEN IF ALARM = OFF
      THEN HOUR = 0; /*EXECUTED*/
```

Figure 2-50. PL/M-86 IF Statements

A DO block begins with a DO statement and ends with an END statement. All intervening statements are part of the block. A DO block can appear anywhere in a program that an executable statement can appear. There are four kinds of DO statements in PL/M-86: simple DO, DO CASE, interative DO, and DO WHILE.

A simple DO statement (figure 2-51) causes all the statements in the block to be treated as though they were a single statement. Simple DOs enable a single IF statement to cause multiple statements to be executed (the alternative would be to repeat the IF statement for every statement to be executed).

```
/*SIMPLE DO*/
A=5; B=9;
IF (A+2) < B THEN DO;
                X=X-1;      /*EXECUTED*/
                Y(X)=0;     /*EXECUTED*/
                END;
        ELSE    DO;
                X=X+1;      /*SKIPPED*/
                Y(X)=1;     /*SKIPPED*/
                END;


/*DO CASE*/
A = 2;
DO CASE (A);
    X = X+1;    /*SKIPPED*/
    X = X+2;    /*SKIPPED*/
    X = X+3;    /*EXECUTED*/
    X = X+4;    /*SKIPPED*/
    END;
```

Figure 2-51. PL/M-86 Simple DO and DO CASE

DO CASE (figure 2-51) causes one statement in the DO block to be selected and executed depending on the result of the expression (usually arithmetic) written immediately following DO CASE:

```
DO CASE arithmetic-expression;
```

If the expression yields 0, the first statement in the DO block is executed; if the expression yields 1, the second statement is executed, etc. A statement in the DO block may be null (consist of only a semicolon) to cause no action for selected cases. DO CASE provides a rapid and easily-understood way to respond to data like "transaction codes"

where a different action is required for each of many values a code might assume (an alternative would be an IF statement for every value the code could assume).

An iterative DO block (figures 2-52 and 2-53) is executed from 0 to an infinite number of times based on the relationship of an index variable to an expression that terminates execution. The general form is:

```
DO index = start-expr TO stop-expr BY step-expr;
```

The "BY step-expr" is optional, and the step is assumed to be 1 if not supplied (the typical case). When control first reaches the DO statement, start-expr is evaluated and is assigned to index. Then index is compared to stop-expr; if index exceeds stop-expr, control goes to the statement following the DO block, otherwise the block is executed. At the end of the block, the result of step-expr is added to index, and it is compared to stop-expr again, etc. (The iterative DO is quite flexible—this is a simplified explanation.) Iterative DOs are handy for "stepping through" an array. For example, an array of 10 elements could be zeroed by:

```
DO I = 0 TO 9;
   ARRAY(I) = 0;
   END;
```

In a DO WHILE (figures 2-52 and 2-54), the statements are executed repeatedly as long as the expression following WHILE evaluates to "true." DO WHILE often can be applied in situations where an interative DO will not work, or is clumsy, such as where repetition must be controlled by a non-integer value. Like an iterative DO, DO WHILE may be executed from 0 times to an infinite number of times.

```
/*ITERATIVE DO*/
DO I = 0 TO 5;
    ARRAY (I) = I;              /*EXECUTED 6 TIMES*/
    TOTAL = TOTAL+1;            /*EXECUTED 6 TIMES*/
    END;
/*I = 6 AT THIS POINT*/

/*DO WHILE*/
MORE = 0; SPACE_OK =1;
DO WHILE (MORE AND SPACE_OK);
    ITEMS = ITEMS + 1;          /*SKIPPED*/
    N_TRACKS =
    N_TRACKS + 10;              /*SKIPPED*/
    IF N_TRACKS >= 999          /*SKIPPED*/
        THEN SPACE_OK = 0;
    END;


/*DO WHILE*/
CODE = 'A';
DO WHILE (CODE = 'A');
    TEMP = TEMP * STEP;         /*EXECUTION STOPS*/
    IF TEMP > 98.6              /*AFTER TEMP*/
        THEN CODE = 'B';        /*EXCEEDS 98.6*/
        N_STEPS = N_STEPS + 1;
    END;
```

Figure 2-52. PL/M-86 Iterative DO and DO WHILE

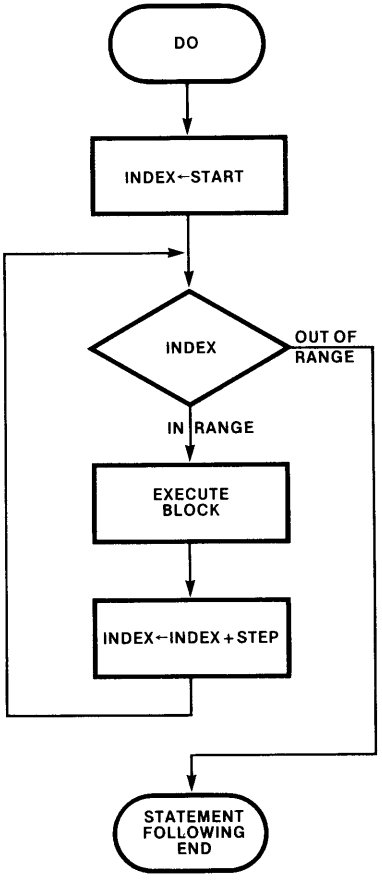

Figure 2-53. PL/M-86 Iterative DO Flowchart

A GOTO written in the form

```
GOTO target;
```

causes an unconditional transfer (branch) to another statement in the program. The statement receiving control would be written

```
target: statement;
```

where "target" is a label identifying the statement.

A CALL statement written in the form

```
CALL proc-name (parm-list);
```

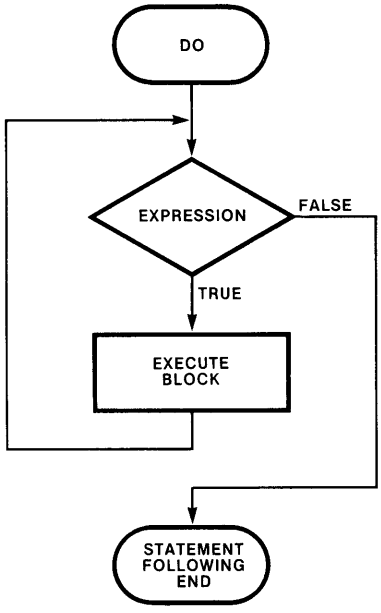

Figure 2-54. PL/M-86 DO WHILE Flowchart

activates a procedure defined earlier in the program. The variables listed in "parm-list" are passed to the procedure, the procedure is executed, and then control returns to the statement following the CALL. Thus, unlike a GOTO, a CALL brings control back to the point of departure.

Procedures

Procedures are "subprograms" that make it possible to simplify the design of complex programs and to share a single copy of a routine among programs. A procedure usually is designed to perform one function; i.e., to solve one part of the total problem with which the program is dealing. For example, a program to calculate paychecks could be broken down into separate procedures for calculating gross pay, income tax, Social Security and net pay. The organization of the "main" program then could be understood at a glance:

```
CALL GROSS_PAY;
CALL INCOME_TAX;
CALL SOCIAL_SECURITY;
CALL NET_PAY;
```

Furthermore, the income tax procedure could be divided into separate procedures for calculating state and federal taxes. Procedures, then, provide a mechanism by which a large, complex problem can be attacked with a "divide and conquer" strategy.

A procedure usually is defined early in a program, but it is only executed when it is referred to by name in a later PL/M-86 statement. A procedure can accept a list of variables, called parameters, that it will use in performing its function. These parameters may assume different values each time the procedure is executed.

PL/M-86 provides two classes of procedures, typed and untyped. A typed procedure returns a value to the statement that activates it and, in addition, may accept parameters from that statement. A typed procedure is activated whenever its name appears in a statement; the value it returns effectively takes the place of the procedure name in the statement. Typed procedures can be used in all kinds of PL/M-86 expressions. Untyped procedures may accept parameters, but do not return a value. Untyped procedures are activated by CALL statements. Figure 2-55 shows how simple typed and untyped procedures may be declared and then activated.

The statements forming the body of a procedure need not exist within the module that activates the procedure. The activating module can declare the procedure EXTERNAL, and the LINK-86 utility will connect the two modules.

PL/M-86 procedures can be written to handle interrupts. Procedures also may be declared REENTRANT, making them concurrently usable by different tasks in a multitasking system. PL/M-86 also has about 50 procedures built into the language, including facilities for:

- converting variables from one type to another
- shifting and rotating bits
- performing input and output
- manipulating strings
- activating the CPU $\overline{\text{LOCK}}$ signal.

```
/*DECLARATION OF A TYPED PROCEDURE THAT
  ACCEPTS TWO REAL PARAMETERS AND RETURNS A REAL VALUE*/
  AVG: PROCEDURE (X,Y) REAL;
       DECLARE (X,Y) REAL;
       RETURN (X+Y)/2.0;
       END AVG;

/*ACTIVATING A TYPED PROCEDURE*/
LOW = 2.0;
HIGH = 3.0;
TOTAL = TOTAL + AVG (LOW,HIGH);  /*2.5 IS ADDED TO TOTAL*/

/*DECLARATION OF AN UNTYPED PROCEDURE
  THAT ACCEPTS ONE PARAMETER*/
TEST: PROCEDURE (X);
  DECLARE X BYTE;
  IF X = 0H THEN
       COUNT = COUNT + 1;
  END TEST;

/*ACTIVATING AN UNTYPED PROCEDURE*/
CALL TEST (ALPHA);  /*COUNT IS INCREMENTED
                      IF ALPHA = 0*/
```

Figure 2-55. PL/M-86 Procedures

ASM-86

Programmers who are familiar with the CPU architecture can obtain complete access to all processor facilities with ASM-86. Since the execution unit on both the 8086 and the 8088 is identical, both processors use the same assembly language. Examples of processor features not accessible through PL/M-86 that can be utilized in ASM-86 programs include: software interrupts, the WAIT and ESC instructions and explicit control of the segment registers.

An ASM-86 program often can be written to execute faster and/or to use less memory than the same program written in PL/M-86. This is because the compiler has a limited "knowledge" of the entire program and must generate a generalized set of machine instructions that will work in all situations, but may not be optimal in a particular situation. For example, assume that the elements of an array are to be summed and the result placed in a variable in memory. The machine instructions generated by the PL/M-86 compiler would move the next array element to a register and then add the register to the sum variable in memory. An ASM-86 programmer, knowing that a register will be "safe" while the array is summed, could instead add all the array elements to a register and then move the register to the sum variable, saving one instruction execution per array element.

It is easier to write assembly language programs in ASM-86 than it is in many assembly languages. ASM-86 contains powerful data structuring facilities that are usually found only in high-level languages. ASM-86 also simplifies the programmer's "view" of the 8086/8088 machine instruction set. For example, although there are 28 different types of MOV machine instructions, the programmer always writes a single form of the instruction:

```
MOV destination-operand, source-operand
```

The assembler generates the correct machine-instruction form based on the attributes of the source and destination operands (attributes are covered later in this section). Finally, the ASM-86 assembler performs extensive checks on the consistency of operand definition versus operand use in instructions, catching many common types of clerical errors.

Statements

Compared to many assemblers, ASM-86 accepts a relaxed statement format (see figure 2-56). This helps to reduce clerical errors and allows programmers to format their programs for better readability. Variable and label names may be up to 31 characters long and are not restricted to alphabetic and numeric characters. In particular, the underscore (_) may be used to improve the readability of long names. Blanks may be inserted freely between identifiers (there are no "column" requirements), and statements also may span multiple lines.

All ASM-86 statements are classified as instructions or directives. A clear distinction must be made here between ASM-86 instructions and

```
; THIS STATEMENT CONTAINS A COMMENT ONLY

MOV       AX, [BX + 3]                    ; TYPICAL ASM-86 INSTRUCTION
    MOV AX,          [BX + 3]             ; BLANKS NOT SIGNIFICANT
MOV       AX,
&         [BX + 3]                        ; CONTINUED STATEMENTS

ZERO      EQU     0                       ; SIMPLE ASM-86 DIRECTIVE
CUR_PROJ  EQU     PROJECT [BX] [SI]       ; MORE COMPLEX DIRECTIVE
THE_STACK_STARTS_HERE  SEGMENT            ; LONG IDENTIFIER
TIGHT_LOOP:  JMP TIGHT_LOOP               ; LABELLED STATEMENT
MOV  ES: DATA_STRING [SI], AL             ; SEGMENT OVERRIDE PREFIX
WAIT:  LOCK XCHG  AX,SEMAPHORE            ; LABEL & LOCK PREFIX
```

Figure 2-56. ASM-86 Statements

8086/8088 machine instructions. The assembler generates machine instructions from ASM-86 instructions written by a programmer. Each ASM-86 instruction produces one machine instruction, but the form of the generated machine instruction will vary according to the operands written in the ASM-86 instruction. For example, writing

```
MOV BL,1
```

produces a byte-immediate-to-register MOV, while writing

```
MOV TERMINAL_NO,BX
```

produces a word-register-to-memory MOV. To the programmer, though, there is simply a MOV source-to-destination instruction.

ASM-86 instructions are written in the form:

```
(label:) (prefix) mnemonic (operand(s)) (;comment)
```

where parentheses denote optional fields (the parentheses are not actually written by programmers). The label field names the storage location containing the machine instruction so that it can be referred to symbolically as the target of a JMP instruction elsewhere in the program. Writing a prefix causes ASM-86 to generate one of the special prefix bytes (segment override, bus lock or repeat) immediately preceding the machine instruction. The mnemonic identifies the type of instruction (MOV for move, ADD for add, etc.) that is to be generated. Zero, one or two operands may be written next, separated by commas, according to the requirements of the instruction. Finally, writing a semicolon signifies that what follows is a comment. Comments do not affect the execution of a program, but they can greatly improve its clarity; all good ASM-86 programs are thoughtfully commented.

Writing a directive gives ASM-86 information to use in generating instructions, but does not itself produce a machine instruction. About 20 different directives are available in ASM-86. Directives are written like this:

```
(name) mnemonic (operand(s)) (;comment)
```

Some directives require a name to be present, while others prohibit a name. ASM-86 recognizes the directive from the mnemonic keyword written in the next field. Any operands required by the directive are written next, separated by commas. A comment may be written as the last field of a directive.

Some of the more commonly used directives define procedures (PROC), allocate storage for variables (DB, DW, DD) give a descriptive name to a number or an expression (EQU), define the bounds of segments (SEGMENT and ENDS), and force instructions and data to be aligned at word boundaries (EVEN).

Constants

Binary, decimal, octal and hexadecimal numeric constants (see figure 2-57) may be written in ASM-86 statements; the assembler can perform basic arithmetic operations on these as well. All numbers must, however, be integers and must be representable in 16 bits including a sign bit. Negative numbers are assembled in standard two's complement notation.

Character constants are enclosed in single quotes and may be up to 255 characters long when used

```
MOV       STRING [SI], 'A'      ; CHARACTER
MOV       STRING [SI], 41H      ; EQUIVALENT IN HEX
ADD       AX, 0C4H              ; HEX CONSTANT MUST START WITH NUMERAL
OCTAL_8   EQU  100              ; OCTAL
OCTAL_9   EQU  10Q              ; OCTAL ALTERNATE
ALL_ONES  EQU  11111111B        ; BINARY
MINUS_5   EQU  -5               ; DECIMAL
MINUS_6   EQU  -6D              ; DECIMAL ALTERNATE
```

Figure 2-57. ASM-86 Constants

to initialize storage. When used as immediate operands, character constants may be one or two bytes long to match the length of the destination operand.

Defining Data

Most ASM-86 programs begin by defining the variables with which they will work. Three directives, DB, DW and DD, are used to allocate and name data storage locations in ASM-86 (see figure 2-58). The directives are used to define storage in three different units: DB means "define byte," DW means "define word," and DD means "define doubleword." The operands of these directives tell the assembler how many storage units to allocate and what initial values, if any, with which to fill the locations.

```
A_SEG   SEGMENT
ALPHA   DB    ?          ; NOT INITIALIZED
BETA    DW    ?          ; NOT INITIALIZED
GAMMA   DD    ?          ; NOT INITIALIZED
DELTA   DB    ?          ; NOT INITIALIZED
EPSILON DW    5          ; CONTAINS 05H
A_SEG   ENDS

B_SEG   SEGMENT AT 55H ; SPECIFYING BASE ADDRESS
IOTA    DB    'HELLO'    ; CONTAINS 48 45 4C 4C 4F H
KAPPA   DW    'AB'       ; CONTAINS 42 41 H
LAMBDA  DD    B_SEG      ; CONTAINS 0000 5500 H
MU      DB    100 DUP 0  ; CONTAINS (100 X) 00H
B_SEG   ENDS
```

	ATTRIBUTES			OPERATORS	
VARIABLE	SEGMENT	OFFSET	TYPE	LENGTH	SIZE
ALPHA	A_SEG	0	1	1	1
BETA	A_SEG	1	2	1	2
GAMMA	A_SEG	3	4	1	4
DELTA	A_SEG	7	1	1	1
EPSILON	A_SEG	8	2	1	2
IOTA	B_SEG	0	1	5	5
KAPPA	B_SEG	5	2	1	2
LAMBDA	B_SEG	7	4	1	4
MU	B_SEG	11	1	100	100

Figure 2-58. ASM-86 Data Definitions

For every variable in an ASM-86 program, the assembler keeps track of three attributes: segment, offset and type. Segment identifies the segment that contains the variable (segment control is covered shortly). Offset is the distance in bytes of the variable from the beginning of its containing segment. Type identifies the variable's allocation unit (1 = byte, 2 = word, 4 = doubleword). When a variable is referenced in an instruction, ASM-86 uses these attributes to determine what form of the instruction to generate. If the variable's attributes conflict with its usage in an instruction, ASM-86 produces an error message. For example, attempting to add a variable defined as a word to a byte register is an error. There are cases where the assembler must be explicitly told an operand's type. For example, writing MOVE [BX],5 will produce an error message because the assembler does not know if [BX] refers to a byte, a word or a doubleword. The following operators can be used to provide this information: BYTE PTR, WORD PTR and DWORD PTR. In the previous example, a word could be moved to the location referenced by [BX] by writing MOVE WORD PTR [BX],5.

ASM-86 also provides two built-in operators, LENGTH and SIZE, that can be written in ASM-86 instructions along with attribute information. LENGTH causes the assembler to return the number of storage units (bytes, words or doublewords) occupied by an array. SIZE causes ASM-86 to return the total number of bytes occupied by a variable or an array. These operators and attributes make it possible to write generalized instruction sequences that need not be changed (only reassembled) if the attributes of the variables change (e.g., a byte array is changed to a word array). See figure 2-59 for an example of using the attributes and attribute operators.

Records

ASM-86 provides a means of symbolically defining individual bits and strings of bits within a byte or a word. Such a definition is called a record, and each named bit string (which may consist of a single bit) in a record is called a field. Records promote efficient use of storage while at the same time improving the readability of the program and reducing the likelihood of clerical errors. Defining a record does not allocate storage; rather, a record is a template that tells the assembler the name and location of each bit field within the byte or word. When a field name is written later in an instruction, ASM-86 uses the record to generate an immediate mask for instructions like TEST, AND, OR, etc., or an immediate count for shifts and rotates. See figure 2-60 for an example of using a record.

```
; SUM THE CONTENTS OF TABLE INTO AX
TABLE       DW        50  DUP(?)
; NOTE SAME INSTRUCTIONS WOULD WORK FOR
; TABLE     DB        25  DUP(?)
; TABLE     DW       118  DUP(?),  ETC.

            SUB       AX,AX              ; CLEAR SUM
            MOV       CX, LENGTH TABLE   ; LOOP TERMINATOR
            MOV       SI, SIZE TABLE     ;POINT SUBSCRIPT
                                         ; TO END OF TABLE
ADD_NEXT:   SUB       SI, TYPE TABLE     ; BACK UP ONE ELEMENT
            ADD       AX, TABLE [SI]     ; ADD ELEMENT
            LOOP      ADD_NEXT           ; UNTIL CX = 0
            ; AX CONTAINS SUM
```

Figure 2-59. Using ASM-86 Attributes and Attribute Operators

```
            EMP_BYTE  DB  ?              ; 1 BYTE, UNINITIALIZED
            ; BIT DEFINITIONS:
            ;    7-2     : YEARS EMPLOYED
                 1       : SEX (1 = FEMALE)
                 0       : STATUS (1 = EXEMPT)
            EMP_BITSRECORD               ;RECORD DEFINED HERE
            &          YRS_EMP : 6,
            &          SEX : 1,
            &          STATUS : 1
            .
            .
            .
            ; SELECT NONEXEMPT FEMALES EMPLOYED 10 + YEARS

            MOV        AL, EMP_BYTE      ; KEEP ORIGINAL INTACT
            TEST       AL, MASK SEX      ; FEMALE ?
            JZ         REJECT            ; NO, QUITE
            TEST       AL, MASK STATUS   ; NONEXEMPT?
            JNZ        REJECT            ;NO, QUIT
            SHR        AL, CL            ; ISOLATE YEARS
            CMP        AL, 11            ; >=10 YEARS?
            JL         REJECT            ; NO, QUIT
            ; PROCESS SELECTED EMPLOYEE
            .
            .
            .
REJECT:     ; PROCESS REJECTED EMPLOYEE
            .
            .
            .                            ; RECORD USED HERE
            MOV        CL, YRS_EMP       ; GET SHIFT COUNT
```

Figure 2-60. Using an ASM-86 RECORD Definition

Structures

An ASM-86 structure is a map, or template, that gives names and attributes (length, type, etc.) to a collection of fields. Each field in a structure is defined using DB, DW and DD directives; however, no storage is allocated to the structure. Instead, the structure becomes associated with a particular area of memory when a field name is referenced in an instruction along with a base value. The base value "locates" the structure; it may be a variable name or a base register (BX or BP). The structure may be associated with another area of memory by specifying a different base value. Figure 2-61 shows how a simple structure may be defined and used. Note that a structure field may itself be a structure, allowing much more complex organizations to be laid out.

Structures are particularly useful in situations where the same storage format is at multiple locations, where the location of a collection of variables is not known at assembly-time, and where the location of a collection of variables changes during execution. Applications include multiple buffers for a single file, list processing and stack addressing.

Addressing Modes

Figure 2-62 provides sample ASM-86 coding for each of the 8086/8088 addressing modes. The assembler interprets a bracketed reference to BX, BP, SI or DI as a base or index register to be used to construct the effective address of a memory operand. An unbracketed reference means the register itself is the operand.

The following cases illustrate typical ASM-86 coding for accessing arrays and structures, and show which addressing mode the assembler specifies in the machine instruction it generates:

- If ALPHA is an array, then ALPHA [SI] is the element indexed by SI, and ALPHA [SI + 1] is the following byte (indexed).
- If ALPHA is the base address of a structure and BETA is a field in the structure, then ALPHA.BETA selects the BETA field (direct).
- If register BX contains the base address of a structure and BETA is a field in the structure, then [BX].BETA refers to the BETA field (based).

```
EMPLOYEE        STRUC
    SSN         DB  9    DUP(?)
    RATE        DB  1    DUP(?)
    DEPT        DW  1    DUP(?)
    YR_HIRED    DB  1    DUP(?)
    EMPLOYEE    ENDS

MASTER          DB  12   DUP(?)
TXN             DB  12   DUP(?)

; CHANGE RATE IN MASTER TO VALUE IN TXN.
                MOV     AL, TXN.RATE
                MOV     MASTER.RATE, AL

; ASSUME BX POINTS TO AN AREA CONTAINING
;       DATA IN THE SAME FORMAT AS THE EMPLOYEE
;       STRUCTURE. ZERO THE SECOND DIGIT
;       OF SSN
                MOV     SI, 1   ; INDEX VALUE OF 2ND DIGIT
                MOV     [BX].SSN[SI],0
```

Figure 2-61. Using an ASM-86 Structure

```
ADD     AX, BX              ; REGISTER ← REGISTER
ADD     AL, 5               ; REGISTER ← IMMEDIATE
ADD     CX, ALPHA           ; REGISTER ← MEMORY (DIRECT)
ADD     ALPHA, 6            ; MEMORY (DIRECT) ← IMMEDIATE
ADD     ALPHA, DX           ; MEMORY (DIRECT) ← REGISTER
ADD     BL, [BX]            ; REGISTER ← MEMORY (REGISTER INDIRECT)
ADD     [SI], BH            ; MEMORY (REGISTER INDIRECT) ← IMMEDIATE
ADD     [PP].ALPHA, AH      ; MEMORY (BASED) ← REGISTER
ADD     CX, ALPHA [SI]      ; REGISTER ← MEMORY (INDEXED)
ADD     ALPHA [DI+2], 10    ; MEMORY (INDEXED) ← IMMEDIATE
ADD     [BX].ALPHA [SI], AL ; MEMORY (BASED INDEXED) ← REGISTER
ADD     SI, [BP+4] [DI]     ; REGISTER ← MEMORY (BASED INDEXED)
IN      AL, 30              ; DIRECT PORT
OUT     DX, AX              ; INDIRECT PORT
```

Figure 2-62. ASM-86 Addressing Mode Examples

- If register BX contains the address of an array, then [BX] [SI] refers to the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is an array, then [BX] .ALPHA [SI] selects the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is itself a structure, then [BX].ALPHA.BETA refers to the BETA field of the ALPHA substructure (based).
- If register BX points to a structure and the ALPHA field of the structure is an array and each element of ALPHA is a structure, then [BX].ALPHA[SI + 3].BETA refers to the field BETA in the element of ALPHA indexed by [SI + 3] (based indexed).

Note that DI may be used in place of SI in these cases and that BP may be substituted for BX. Without a segment override prefix, expressions containing BP refer to the current stack segment, and expressions containing BX refer to the current data segment.

Segment Control

An ASM-86 program is organized into a series of named segments. These are "logical" segments; they are eventually mapped into 8086/8088 memory segments, but this usually is not done until the program is located. A SEGMENT directive starts a segment, and an ENDS directive ends the segment (see figure 2-63). All data and instructions written between SEGMENT and ENDS are part of the named segment. In small programs, variables often are defined in one or two segment(s), stack space is allocated in another segment, and instructions are written in a third or fourth segment. It is perfectly possible, however, to write a complete program in one segment; if this is done, all the segment registers will contain the same base address; that is, the memory segments will completely overlap. Large programs may be divided into dozens of segments.

The first instructions in a program usually establish the correspondence between segment names and segment registers, and then load each segment register with the base address of its corresponding segment. The ASSUME directive tells the assembler what addresses will be in the segment registers at execution time. The assembler checks each memory instruction operand, determines which segment it is in and which segment register contains the address of that segment. If the assumed register is the register expected by the hardware for that instruction type, then the assembler generates the machine instruction normally. If, however, the hardware expects one segment register to be used, and the operand is *not* in the segment pointed to by that register, then the assembler automatically precedes the machine instruction with a segment override prefix byte. (If the segment cannot be overridden, the assembler produces an error message.) An example may clarify this. If register BP is used in an instruction, the 8086 and 8088 CPUs expect, as a default, that the memory operand will be located in the segment pointed to by SS—in the current

```
DATA_SEG      SEGMENT
   ; DATA DEFINITIONS GO HERE
DATA_SEG      ENDS

STACK_SEG   SEGMENT
   ; ALLOCATE 100 WORDS FOR A STACK AND
   ;    LABEL THE INITIAL TOS FOR LOADING SP.
         DW 100 DUP(?)
STACK TOP LABEL WORD
STACK_SEG     ENDS

CODE_SEG      SEGMENT
   ; GIVE ASSEMBLER INITIAL REGISTER-TO-SEGMENT
   ;   CORRESPONDENCE. NOTE THAT IN THIS
   ;   PROGRAM THE EXTRA SEGMENT INITIALLY
   ;   OVERLAPS THE DATA SEGMENT ENTIRELY.
ASSUME  CS: CODE_SEG,
&          DS: DATA_SEG,
&          ES: DATA_SEG,
&          SS: STACK_SEG

START:   ; THIS IS THE BEGINNING OF THE PROGRAM.
         ; LOC-86 WILL PLACE A JMP TO THIS
         ; LOCATION AT ADDRESS FFFF0H.

   ; LOAD THE SEGMENT REGISTERS. CS DOES NOT
   ;    HAVE TO BE LOADED BECAUSE SYSTEM
   ;    RESET SETS IT TO FFFFH, AND THE
   ;    LONG JMP INSTRUCTION AT THAT ADDRESS
   ;    UPDATES IT TO THE ADDRESS OF CODE_SEG.
   ;    SEGMENT REGISTERS ARE LOADED FROM AX
   ;    BECAUSE THERE IS NO IMMEDIATE-TO-
   ;    SEGMENT_REGISTER FORM OF THE MOV
   ;    INSTRUCTION.

                        MOV   AX, DATA_SEG
                        MOV   DS, AX
                        MOV   ES, AX
                        MOV   AX, STACK_SEG
                        MOV   SS, AX
; SET STACK POINTER TO INITIAL TOS.
                        MOV   SP, OFFSET STACK_TOP

; SEGMENTS ARE NOW ADDRESSABLE.
; MAIN PROGRAM CODE GOES HERE.
CODE_SEG                      ENDS

; NEXT STATEMENT ENDS ASSEMBLY AND TELLS
;    LOC-86 THE PROGRAMS STARTING ADDRESS.

                        END   START
```

Figure 2-63. Setting Up ASM-86 Segments

stack segment. A programmer may, however, choose to use BP to address a variable in the current data segment—the segment pointed to by DS. The ASSUME directive enables the assembler to detect this situation and to automatically generate the needed override prefix.

It also is possible for a programmer to explicitly code segment override prefixes rather than relying on the assembler. This may result in a somewhat better-documented program since attention is called to the override. The disadvantage of explicit segment overrides is that the assembler does not check whether the operand is in fact addressable through the overriding segment register.

ASM-86, in conjunction with the relocation and linkage facilities, provides much more sophisticated segment handling capabilities than have been described in this introduction. For example, different logical segments may be combined into the same physical segment, and segments may be assigned the same physical locations (allowing a "common" area to be accessed by different programs using different variable and label names).

Procedures

Procedures may be written in ASM-86 as well as in PL/M-86. In fact, procedures written in one language are callable from the other, provided that a few simple conventions are observed in the ASM-86 program. The purpose of ASM-86 procedures is the same as in PL/M-86: to simplify the design of complex programs and to make a single copy of a commonly-used routine accessible from anywhere in the program.

An ASM-86 program activates a procedure with a CALL instruction. The procedure terminates with a RET instruction, which transfers control to the instruction following the CALL. Parameters may be passed in registers or pushed onto the stack before calling the procedure. The RET instruction can discard stack parameters before returning to the caller.

Unlike PL/M-86 procedures, ASM-86 procedures are executable where they are coded, as well as by a CALL instruction. Therefore, ASM-86 procedures often are defined following the main program logic, rather than preceding it as in PL/M-86. Figure 2-64 shows how procedures may be defined and called in ASM-86. Section 2-10 contains examples of procedures that accept parameters on the stack.

LINK-86

Fundamentally, LINK-86 combines separate relocatable object modules into a single program. This process consists primarily of combining (logical) segments of the same name into single segments, adjusting relative addresses when segments are combined, and resolving external references.

A programmer can use a procedure that is actually contained in another module by naming the procedure in an ASM-86 EXTRN directive, or declaring the procedure to be EXTERNAL in PL/M-86. The procedure is defined or declared PUBLIC in the module where it actually resides, meaning that it can be used by other modules. When LINK-86 encounters such an external reference, it searches through the other modules in its input, trying to find the matching PUBLIC declaration. If it finds the referenced object, it links it to the reference, "satisfying" the external reference. If it cannot satisfy the reference, LINK-86 prints a diagnostic message. LINK-86 also checks PL/M-86 procedure calls and function references to insure that the parameters passed to a procedure are the type expected by the procedure.

LINK-86 gives the programmer, particularly the ASM-86 programmer, great control over segments (segments may be combined end to end, renamed, assigned the same locations, etc.). LINK-86 also produces a map that summarizes the link process and lists any unusual conditions encountered. While the output of LINK-86 is generally input to LOC-86, it also may again be input to LINK-86 to permit modules to be linked in incremental groups.

LOC-86

LOC-86 accepts the single relocatable object module produced by LINK-86 and binds the memory references in the module to actual memory addresses. Its output is an absolute object module ready for loading into the memory of an execution vehicle. LOC-86 also inserts a

```
FREQUENCY       DB      256 DUP (0)
.
.
USART_DATA      EQU     0FF0H           ; DATA PORT ADDRESS
USART_STAT      EQU     0FF2H           ; STATUS PORT ADDRESS
.
.
NEXT:           CALL    CHAR_IN
                CALL    COUNT_IT
                JMP     NEXT

CHAR_IN         PROC
    ; THIS PROCEDURE DOES NOT TAKE PARAMETERS.
    ;   IT SAMPLES THE USART STATUS PORT
    ;   UNTIL A CHARACTER IS READY, AND
    ;   THEN READS THE CHARACTER INTO AL
                MOV     DX, USART_STAT
    AGAIN:      IN      AL, DX          ; READ STATUS
                AND     AL, 2           ; CHARACTER PRESENT?
                JZ      AGAIN           ; NO, TRY AGAIN
                MOV     DX, USART_DATA
                IN      AL, DX          ; YES, READ CHARACTER
                RET
CHAR_IN         ENDP

COUNT_IT        PROC
    ; THIS PROCEDURE EXPECTS A CHARACTER IN AL.
    ;   IT INCREMENTS A COUNTER IN A FREQUENCY
    ;   TABLE BASED ON THE BINARY VALUE OF
    ;   THE CHARACTER.
                XOR     AH, AH          ; CLEAR HIGH BYTE
                MOV     SI, AL          ; INDEX INTO TABLE
                INC     FREQUENCY [S]; BUMP THE COUNTER
                RET
COUNT_IT        ENDP
```

Figure 2-64. ASM-86 Procedures

direct intersegment JMP instruction at location FFFF0H. The target of the JMP instruction is the logical beginning of the program. When the 8086 or 8088 is reset, this instruction is automatically executed to restart the system. LOC-86 produces a memory map of the absolute object module and a table showing the address of every symbol defined in the program.

LIB-86

LIB-86 is a valuable adjunct to the R & L programs. It is used to maintain relocatable object modules in special files called libraries. Libraries are a convenient way to make collections of modules available to LINK-86. When a module being linked refers to "external" data or instructions, LINK-86 can automatically search a series of libraries, find the referenced module, and include it in the program being created.

OH-86

OH-86 converts an absolute object module into Intel's standard hexadecimal format. This format is used by some PROM programmers and system loaders, such as the iSBC 957™ and SDK-86 loaders.

CONV-86

Users who have developed substantial, fully-tested assembly language programs for the 8080/8085 microprocessors may want to use CONV-86 to automatically convert large amounts of this code into ASM-86 source code (see figure 2-65). CONV-86 accepts an ASM-80 source program as input and produces an ASM-86 source program as output, plus a print file that documents the conversion and lists any diagnostic messages.

Some programs cannot be completely converted by CONV-86. Exceptions include:

- self-modifying code,
- software timing loops,
- 8085 RIM and SIM instructions,
- interrupt code, and
- macros.

By using the diagnostic messages produced by CONV-86, the converted ASM-86 source file can be manually edited to clean up any sections not converted. A converted program is typically 10-20% larger than the ASM-80 version and does not take full advantage of the 8086/8088 architecture. However, the development time saved by using CONV-86 can make it an attractive alternative to rewriting working programs from scratch.

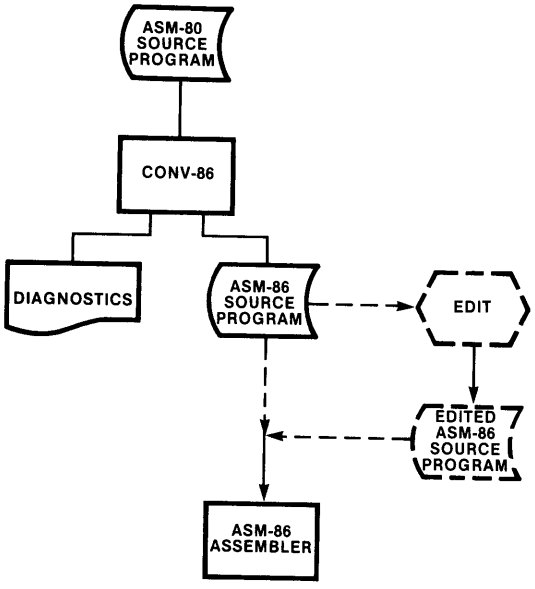

Figure 2-65. ASM-80/ASM-86 Conversion

Sample Programs

Figures 2-66 and 2-67 show how a simple program might be written in PL/M-86 and ASM-86. The program simulates a pair of rolling dice and executes on an Intel SDK-86 System Design Kit. The SDK-86 is an 8086-based computer with memory, parallel and serial I/O ports, a keypad and a display. The SDK-86 is implemented on a single PC board which includes a large prototype area for system expansion and experimentation. A ROM-based monitor program provides a user interface to the system; commands are entered through the keypad and monitor responses are written on the display. With the addition of a cable and software interface (called SDK-C86), the SDK-86 may be connected to an Intellec® Microcomputer Development System. In this mode, the user enters monitor commands from the Intellec keyboard and receives replies on the Intellec CRT display.

The dice program runs on an SDK-86 that is connected to an Intellec® Microcomputer Development System. The program displays two continuously changing digits in the upper left corner of the Intellec display. The digits are random numbers in the range 1-6. A roll is started by entering a monitor GO command. Pressing the INTR key on the SDK-86 keypad stops the roll.

There are two procedures in the PL/M-86 version of the dice program. The first is called CO for console output. This is an untyped PUBLIC procedure that is supplied on an SDK-C86 diskette. CO is written in PL/M-86 and outputs one character to the Intellec console. It is declared EXTERNAL in the dice program because it exists in another module. LINK-86 searches the SDK-C86 library for CO and includes it in the single relocatable object module it builds.

RANDOM is an internal typed procedure; it is contained in the dice module and returns a word value that is a random number between 1 and 6. RANDOM does not use any parameters and is activated in the parameter list passed to CO. When CO is called like this, first RANDOM is activated, then 30 is added to the number it returns and the sum is passed to CO.

```
PL/M-86 COMPILER    DICE

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
COMPILER INVOKED BY:  PLM86 :F1:DICE.P86 XREF

   1          DICE:  DO;
              /* THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE */

              /* GIVE NAMES TO CONSTANTS */
   2    1     DECLARE CLEAR$CRT1      LITERALLY '01BH';  /* INTELLEC  */
   3    1     DECLARE CLEAR$CRT2      LITERALLY '045H';  /*  CRT      */
   4    1     DECLARE HOME$CURSOR1    LITERALLY '01BH';  /*  CONTROL  */
   5    1     DECLARE HOME$CURSOR2    LITERALLY '048H';  /*  CODES    */
   6    1     DECLARE SPACE           LITERALLY '020H';  /*ASCII BLANK*/

              /* PROGRAM VARIABLES */
   7    1     DECLARE (RANDOM$NUMBER,SAVE)  WORD;

              /* CONSOLE OUTPUT PROCEDURE */
   8    1     CO:  PROCEDURE(X) EXTERNAL;
   9    2          DECLARE X    BYTE;
  10    2          END CO;

              /* RANDOM NUMBER GENERATOR PROCEDURE                  */
              /* ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:           */
              /*    "A GUIDE TO PL/M PROGRAMMING FOR                */
              /*     MICROCOMPUTER APPLICATIONS,"                   */
              /*    DANIEL D. MCCRACKEN,                            */
              /*    ADDISON-WESLEY, 1978                            */
  11    1     RANDOM:  PROCEDURE WORD;
  12    2        RANDOM$NUMBER = SAVE;        /*START WITH OLD NUMBER*/
  13    2        RANDOM$NUMBER = 2053 * RANDOM$NUMBER + 13849;
  14    2        SAVE = RANDOM$NUMBER;        /*SAVE FOR NEXT TIME*/
              /*FORCE 16-BIT NUMBER INTO RANGE 1-6*/
  15    2        RANDOM$NUMBER = RANDOM$NUMBER MOD 6  + 1;
  16    2        RETURN RANDOM$NUMBER;
  17    2        END RANDOM;

              /* MAIN ROUTINE */
              /* CLEAR THE SCREEN*/
  18    1     CALL CO(CLEAR$CRT1);
  19    1     CALL CO(CLEAR$CRT2);

              /* ROLL THE DICE UNTIL INTERRUPTED */
  20    1     DO WHILE 1;   /*"DO FOREVER"*/
              /*NOTE THAT ADDING 30 TO THE DIE VALUE */
              /*  CONVERTS IT TO ASCII.              */
  21    2        CALL CO(RANDOM + 030H);      /*1ST DIE*/
  22    2        CALL CO(SPACE);              /*BLANK*/
  23    2        CALL CO(RANDOM + 030H);      /*2ND DIE*/
              /* HOME THE CURSOR */
  24    2        CALL CO(HOME$CURSOR1);
  25    2        CALL CO(HOME$CURSOR2);
  26    2        END;

  27    1     END DICE;


CROSS-REFERENCE LISTING
-----------------------

  DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
 ----- ------ -----  --------------------------------

     2                CLEARCRT1              LITERALLY
                        18

     3                CLEARCRT2              LITERALLY
                        19

     8  0000H         CO                     PROCEDURE EXTERNAL(0) STACK=0000H
                        18   19   21   22   23   24   25

     1  0002H    71   DICE                   PROCEDURE STACK=0004H

     4                HOMECURSOR1            LITERALLY
                        24

     5                HOMECURSOR2            LITERALLY
                        25

    11  0049H    44   RANDOM                 PROCEDURE WORD STACK=0002H
                        21   23
```

Figure 2-66. Sample PL/M-86 Program

```
7  0000H     2  RANDOMNUMBER          WORD
                                        12   13   14   15   16

7  0002H     2  SAVE                  WORD
                                        12   14

6               SPACE                 LITERALLY
                                        22

8  0000H     1  X                     BYTE PARAMETER
                                         9

MODULE INFORMATION:

     CODE AREA SIZE     = 0075H    117D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0004H      4D
     MAXIMUM STACK SIZE = 0004H      4D
     51 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 2-66. Sample PL/M-86 Program (Cont'd.)

```
MCS-86 MACRO ASSEMBLER    DICE

ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:DICE.A86 XREF

LOC  OBJ                  LINE     SOURCE

                             1     ; THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE
                             2
                             3     ; CONSOLE OUTPUT PROCEDURE
                             4             EXTRN   CO:NEAR
                             5
                             6     ; SEGMENT GROUP DEFINITIONS NEEDED FOR PL/M-86 COMPATIBILITY
                             7     CGROUP  GROUP   CODE
                             8     DGROUP  GROUP   DATA,STACK
                             9
                            10     ; INFORM ASSEMBLER OF SEGMENT REGISTER CONTENTS.
                            11             ASSUME  CS:CGROUP,DS:DGROUP,SS:DGROUP,ES:NOTHING
                            12
                            13     ; ALLOCATE DATA
----                        14     DATA    SEGMENT PUBLIC  'DATA'
                            15     ; NOTE THAT THE FOLLOWING ARE PASSED ON THE STACK TO THE PL/M-86
                            16     ;  PROCEDURE 'CO'.  BY CONVENTION, A BYTE PARAMETER IS PASSED IN
                            17     ;  THE LOW-ORDER 8-BITS OF A WORD ON THE STACK.  HENCE, THESE ARE
                            18     ;  DEFINED AS WORD VALUES, THOUGH THEY OCCUPY 1 BYTE ONLY.
0000 1B00                   19     CLEAR_CRT1      DW      01BH    ; INTELLEC
0002 4500                   20     CLEAR_CRT2      DW      045H    ;   CRT
0004 1B00                   21     HOME_CURSOR1    DW      01BH    ;   CONTROL
0006 4800                   22     HOME_CURSOR2    DW      048H    ;   CODES
0008 2000                   23     SPACE           DW      020H    ; ASCII BLANK
000A ????                   24     SAVE            DW      ?       ; HOLDS LAST 16-BIT RANDOM NUMBER
----                        25     DATA    ENDS
                            26
                            27
                            28     ; ALLOCATE STACK SPACE
----                        29     STACK   SEGMENT STACK   'STACK'
0000 (20                    30             DW      20 DUP (?)
     ????
     )

0028                        31     ; LABEL INITIAL TOS: FOR LATER USE.
----                        32     STACK_TOP       LABEL   WORD
                            33     STACK   ENDS
                            34
                            35
                            36     ; PROGRAM CODE
----                        37     CODE    SEGMENT PUBLIC  'CODE'
                            38
                            39
                            40     ; RANDOM NUMBER GENERATOR PROCEDURE
                            41     ; ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:
                            42     ;   "A GUIDE TO PL/M PROGRAMMING FOR
                            43     ;   MICROCOMPUTER APPLICATIONS,"
                            44     ;   DANIEL D. MCCRACKEN
                            45     ;   ADDISON-WESLEY, 1978
0000                        46     RANDOM  PROC
0000 A10A00          R      47             MOV     AX,SAVE         ; NEW NUMBER =
```

Figure 2-67. ASM-86 Sample Program

```
MCS-86 MACRO ASSEMBLER    DICE

LOC  OBJ                  LINE     SOURCE

0003 B90508                 48             MOV     CX,2053          ;   OLD NUMBER * 2053
0006 F7E1                   49             MUL     CX               ;   + 13849
0008 051936                 50             ADD     AX,13849         ;
000B A30A00          R      51             MOV     SAVE,AX          ; SAVE FOR NEXT TIME
                            52             ; FORCE 16-BIT NUMBER INTO RANGE 1 - 6
                            53             ;   BY MODULO 6 DIVISION + 1
000E 2BD2                   54             SUB     DX,DX            ; CLEAR UPPER DIVIDEND
0010 B90600                 55             MOV     CX,6             ; SET DIVISOR
0013 F7F1                   56             DIV     CX               ; DIVIDE BY 6
0015 8BC2                   57             MOV     AX,DX            ; REMAINDER TO AX
0017 40                     58             INC     AX               ; ADD 1
0018 C3                     59             RET                      ; RESULT IN AX
                            60     RANDOM  ENDP
                            61
                            62
                            63     ; MAIN PROGRAM
                            64
                            65     ; LOAD SEGMENT REGISTERS
                            66     ; NOTE PROGRAM DOES NOT USE ES; CS IS INITIALIZED BY HARDWARE RESET;
                            67     ;  DATA & STACK ARE MEMBERS OF SAME GROUP, SO ARE TREATED AS A SINGLE
                            68     ;  MEMORY SEGMENT POINTED TO BY BOTH DS & SS.
0019 B8----          R      69     START:  MOV     AX,DGROUP
001C 8ED8                   70             MOV     DS,AX
001E 8ED0                   71             MOV     SS,AX
                            72
                            73     ; INITIALIZE STACK POINTER
0020 BC2800          R      74             MOV     SP,OFFSET DGROUP:STACK_TOP
                            75
                            76     ; CLEAR THE SCREEN
0023 FF360000        R      77             PUSH    CLEAR_CRT1
0027 E80000          E      78             CALL    CO
002A FF360200        R      79             PUSH    CLEAR_CRT2
002E E80000          E      80             CALL    CO
                            81
                            82     ; ROLL THE DICE UNTIL INTERRUPTED
0031 E8CCFF                 83     ROLL:   CALL    RANDOM           ; GET 1ST DIE IN AL
0034 0430                   84             ADD     AL,030H          ; CONVERT TO ASCII
0036 50                     85             PUSH    AX               ; PASS IT TO
0037 E80000          E      86             CALL    CO               ;   CONSOLE OUTPUT
003A FF360800        R      87             PUSH    SPACE            ; OUTPUT
003E E80000          E      88             CALL    CO               ;   A BLANK
0041 E8BCFF                 89             CALL    RANDOM           ; GET 2ND DIE IN AL
0044 0430                   90             ADD     AL,030H          ; CONVERT TO ASCII
0046 50                     91             PUSH    AX               ; PASS IT TO
0047 E80000          E      92             CALL    CO               ;   CONSOLE OUTPUT
                            93     ; HOME THE CURSOR
004A FF360400        R      94             PUSH    HOME_CURSOR1
004E E80000          E      95             CALL    CO
0051 FF360600        R      96             PUSH    HOME_CURSOR2
0055 E80000          E      97             CALL    CO
                            98     ; CONTINUE FOREVER
0058 EBD7                   99             JMP     ROLL
----                       100     CODE    ENDS
                           101

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME            TYPE     VALUE  ATTRIBUTES, XREFS

??SEG . . . .   SEGMENT          SIZE=0000H PARA PUBLIC
CGROUP. . . .   GROUP            CODE   7# 11
CLEAR_CRT1. .   V WORD   0000H   DATA  19# 77
CLEAR_CRT2. .   V WORD   0002H   DATA  20# 79
CO. . . . . .   L NEAR   0000H   EXTRN  4# 78 80 86 88 92 95 97
CODE. . . . .   SEGMENT          SIZE=005AH PARA PUBLIC 'CODE'  7# 37 100
DATA. . . . .   SEGMENT          SIZE=000CH PARA PUBLIC 'DATA'  8# 14 25
DGROUP. . . .   GROUP            DATA STACK   8# 11 11 69 74
HOME_CURSOR1.   V WORD   0004H   DATA  21# 94
HOME_CURSOR2.   V WORD   0006H   DATA  22# 96
RANDOM. . . .   L NEAR   0000H   CODE  46# 60 83 89
ROLL. . . . .   L NEAR   0031H   CODE  83# 99
SAVE. . . . .   V WORD   000AH   DATA  24# 47 51
SPACE . . . .   V WORD   0008H   DATA  23# 87
STACK . . . .   SEGMENT          SIZE=0028H PARA STACK 'STACK'
STACK_TOP . .   V WORD   0028H   STACK  32# 74
START . . . .   L NEAR   0019H   CODE  69# 104

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 2-67. ASM-86 Sample Program (Cont'd.)

The ASM-86 version of the dice program operates like the PL/M-86 version. Since the program uses the PL/M-86 CO procedure for writing data to the Intellec console, it adheres to certain conventions established by the PL/M-86 compiler. The program's logical segments (called CODE, DATA and STACK—the program does not use an extra segment) are organized into two groups called CGROUP and DGROUP. All the members of a group of logical segments are located in the same 64k byte physical memory segment. Physically, the program's DATA and STACK segments can be viewed as "subsegments" of DGROUP.

PL/M-86 procedures expect parameters to be passed on the stack, so the program pushes each character before calling CO. Note that the stack will be "cleaned up" by the PL/M-86 procedure before returning (i.e., the parameter will be removed from the stack by CO).

2.10 Programming Guidelines and Examples

This section addresses 8086/8088 programming from two different perspectives. A series of general guidelines is presented first. These guidelines apply to all types of systems and are intended to make software easier to write, and particularly, easier to maintain and enhance. The second part contains a number of specific programming examples. Written primarily in ASM-86, these examples illustrate how the instruction set and addressing modes may be utilized in various, commonly encountered programming situations.

Programming Guidelines

These guidelines encourage the development of 8086/8088 software that is adaptable to change. Some of the guidelines refer to specific processor features and others suggest approaches to general software design issues. PL/M-86 programmers need not be concerned with the discussions that deal with specific hardware topics; they should, however, give careful attention to the system design subjects. **Systems that are designed in accordance with these recommendations should be less costly to modify or extend. In addition, they should be better-positioned to take advantage of new hardware and software products that are constantly being introduced by Intel.**

Segments and Segment Registers

Segments should be considered as independent logical units whose physical locations in memory *happen* to be defined by the contents of the segment registers. Programs should be independent of the actual contents of the segment registers and of the physical locations of segments in memory. For example, a program should not take advantage of the "knowledge" that two segments are physically adjacent to each other in memory. The single exception to this fully-independent treatment of segments is that a program may set up more than one segment register to point to the same segment in memory, thereby obtaining addressability through more than one segment register. For example, if both DS and ES point to the same segment, a string located in that segment may be used as a source operand in one string instruction and as a destination string in another instruction (recall that a destination string must be located in the extra segment).

Any data aggregate or construct such as an array, a structure, a string or a stack should be restricted to 64k bytes in length and should be wholly contained in one segment (i.e., should not cross a segment boundary).

Segment registers should only contain values supplied by the relocation and linkage facilities. Segment register values may be moved to and from memory, pushed onto the stack and popped from the stack. Segment registers should never be used to hold temporary variables nor should they be altered in any other way.

As an additional guideline, code should *not* be written within six bytes of the end of physical memory (or the end of the code segment if this segment is dynamically relocatable). Failure to observe this guideline could result in an attempted opcode prefetch from non-existent memory, hanging the CPU if READY is not returned.

Self-Modifying Code

It is possible to write a program that deliberately changes some of its own machine instructions

during execution. While this technique may save a few bytes or machine cycles, it does so at the expense of program clarity. This is particularly true if the program is being examined at the machine instruction level; the machine instructions shown in the assembly listing may not match those found in memory or monitored from the bus. It also precludes executing the code from ROM. Also, because of the prefetch queue within the 8086 and 8088, code that is self-modified within six bytes of the current point of execution cannot be guaranteed to execute as intended. (This code may already have been fetched.) Finally, a self-modifying program may prove incompatible with future Intel products that assume that the content of a code segment remains constant during execution.

A corrollary to this requirement is that variable data should not be placed in a code segment. Constant data may be written in a code segment, but this is not recommended for two reasons. First, programs are simpler to understand if they are uniformly subdivided into segments of code, data and stack. Second, placing data in a code segment can restrict the segment's position independence. This is because, in general, the segment base address of a data item may be changed, but the offset (displacement) of the data item may not. This means that the entire segment must be moved as a unit to avoid changing the offset of the constant data. If the constant data were located in a data segment or an extra segment, individual procedures within the code segment could be moved independently.

Input/Output

Since I/O devices vary so widely in their capabilities and their interface designs, I/O software is inevitably device dependent. Substituting a hard disk for a floppy disk, for example, necessitates software changes even though the disks are functionally identical. I/O software can, however, be designed to minimize the effect of device changes on programs.

Figure 2-68 illustrates a design concept that structures an I/O system into a hierarchy of separately compiled/assembled modules. This approach isolates application modules that use the input/output devices from all physical characteristics of the hardware with which they ultimately communicate. An application module that reads a disk file, for example, should have no knowledge of where the file is located on the disk, what size the disk sectors are, etc. This allows these characteristics to change without affecting the application module. To an application module, the I/O system appears to be a series of file-oriented commands (e.g., Open, Close, Read, Write). An application module would typically issue a command by calling a file system procedure.

The file system processes I/O command requests, perhaps checking for gross errors, and calls a procedure in the I/O supervisor. The I/O supervisor is a bridge between the functional I/O request of the application module and the physical I/O performed by the lowest-level modules in the hierarchy. There should be separate modules in the supervisor for different types of devices and some device-dependent code may be unavoidable at this level. The I/O supervisor would typically perform overhead activities such as maintaining disk directories.

The modules that actually communicate with the I/O devices (or their controllers) are at the lowest level in the hierarchy. These modules contain the bulk of the system's device-dependent code that will have to be modified in the event that a device is changed.

The 8089 Input/Output Processor is specifically designed to encourage the development of modular, hierarchical I/O systems. The 8089 allows knowledge of device characteristics to be "hidden" from not only application programs, but also from the operating system that controls the CPU. The CPU's I/O supervisor can simply prepare a message in memory that describes the nature of the operation to be performed, and then activate the 8089. The 8089 independently performs all physical I/O and notifies the CPU when the operation has been completed.

Operating Systems

Operating systems also should be organized in a hierarchy similar to the concept illustrated in figure 2-69. Application modules should "see" only the upper level of the operating system. This level might provide services like sending messages between application modules, providing time delays, etc. An intermediate level might consist of housekeeping routines that dispatch tasks, alter

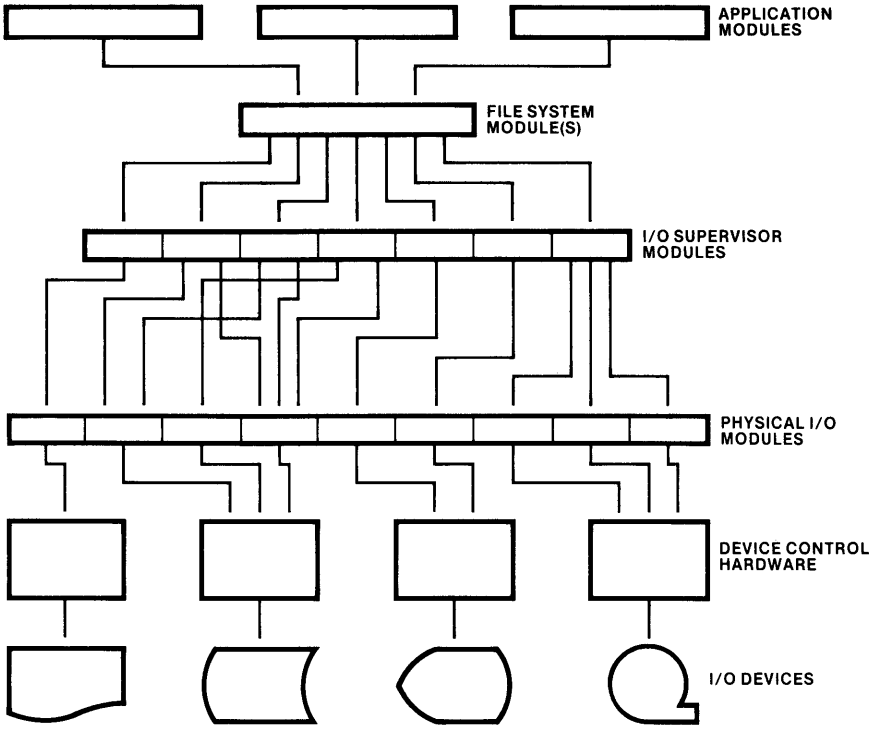

Figure 2-68. I/O System Hierarchy Concept

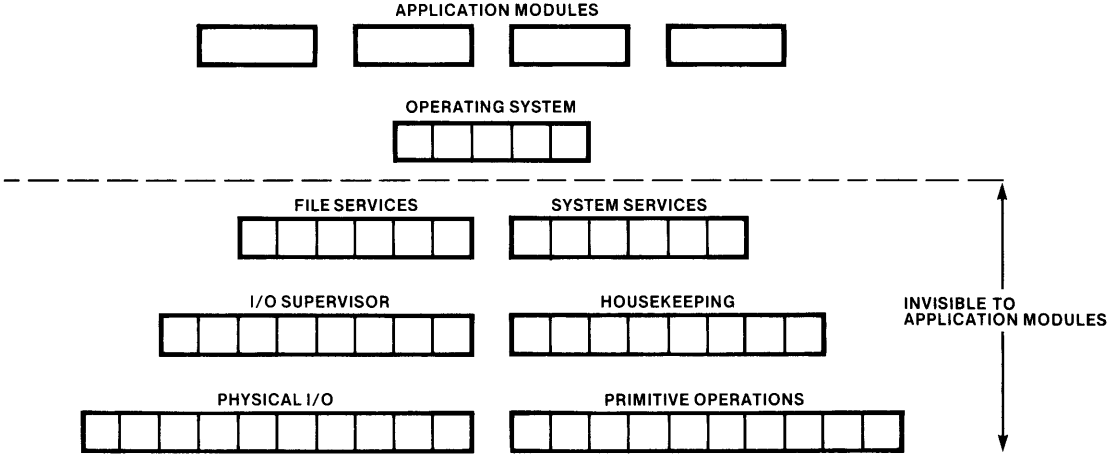

Figure 2-69. Operating System Hierarchy