8086 Microprocessor Cheatsheet

Jishan Shaikh[*]

jishanshaikh9893@gmail.com

May 28, 2022

Abstract: This cheat sheet contains information about the 8086 microprocessors. Compared to standard textbooks, the main focus is conciseness, to-the-point descriptions, and readability. It is a direct result of scribe notes taken for the subject – microprocessors (3+1 credits). The main topics included in the document are 8086 overviews, internal architecture, register organization, modes of operation, addressing modes, interrupts, memory, and assembly language programs (Instruction types, Memory segmentation, and Memory models). The program bank contains an excellent collection of ASM program questions. A question bank is also provided for examinations/quizzes. This cheat sheet is published as an open document at www.github.com/jishanshaikh4/8086-cheatsheet/. You are more than welcome to use this!

1 8086 Overview

8086 is a 16-bit microprocessor. It has a 20-bit address bus that can access up to 2^{20} memory locations (1 MB). It can support up to 64K input/output ports. It provides 14, 16-bit internal registers. It has multiplexed address and data bus $AD_0 - AD_{15}$ and $A_{16} - A_{19}$. It requires a single-phase clock with a 33% duty cycle to provide an internal cycle. It has two operation modes - Minimum and Maximum mode. Its improvements over the 8085 microprocessors include pipelining, instruction queue, and segmentation. It can pre-fetch up to 6 instruction bytes from memory and queue them to speed up instruction execution (Pipelining). It usually requires a +5V power supply. It is packaged under a 40-pin dual inlined package.

2 Internal Architecture

8086 has two blocks, BIU (Bus Interface Unit) and EU (Execution Unit). The BIU performs all bus operations such as instruction fetching, reading, and writing operands to memory and calculating the address of memory operands. The instruction bytes are transferred to the instruction queue. EU executes instructions from the instruction system byte queue. Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism, called Pipelining. This results in efficient use of system bus and system performance. BIU contains an Instruction queue, Segment registers, Instruction pointer, and Address adder. EU contains Control circuitry, Instruction decoder, ALU, Pointer and index register, and Flag register.

[*]v2 (2022) with updated descriptions, proofread, and erratas. Compiled during CSE-313 (Microprocessors) and CSE-316 (Microprocessors Lab) at National Institute of Technology, Bhopal (India). **MIT License.**

.

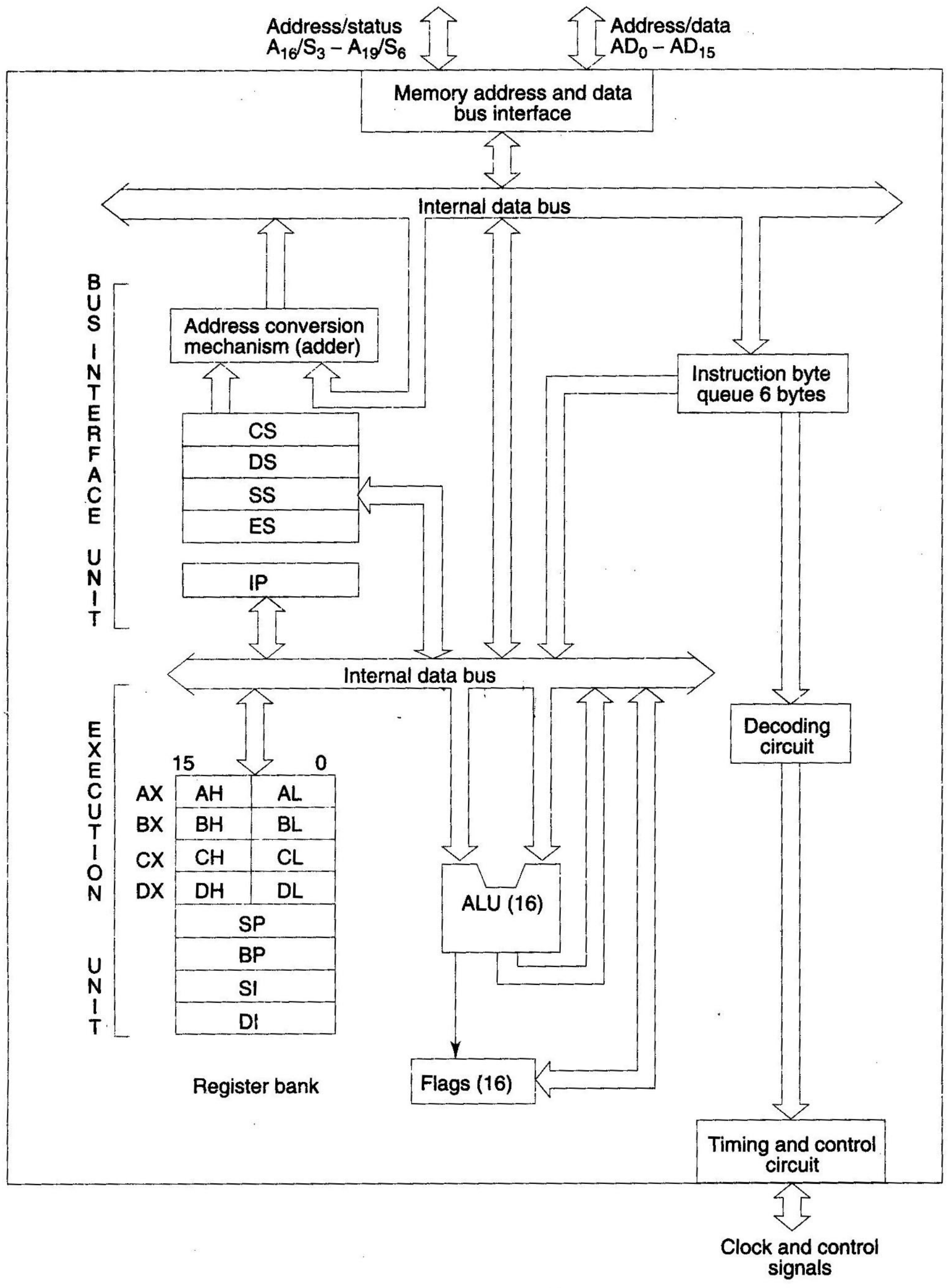


Figure 1: 8086 Architecture diagram.

Bus Interface Unit (BIU): It provides a full 16-bit bidirectional data bus and 20-bit address bus, and is fully responsible for performing all external bus operations. The functions of BIU include instruction fetch, instruction queuing, operand fetch, storage, address relocation, and bus control. It uses a mechanism known as an instruction stream queue to implement a pipeline architecture. This queue permits the prefetch of up to 6 bytes of instruction code. Prefetching instructions are held in its FIFO queue.

Since the data bus is of 16-bit size, BIU can fetch two instruction bytes in a single memory cycle. The time interval of which there doesn't any bus activity is known as *Idle state*, which may occur between bus cycles. The BIU also contains a dedicated adder, which is used to generate a 20-bit physical address that is the output of the address bus. It is formed by adding an appended 16-bit segment address and a 16-bit offset address. BIU is also responsible for the generation of bus control signals, such as those of memory read or write and I/O read or write.

Execution Unit (EU): It handles decoding and executing all instructions. The EU extracts

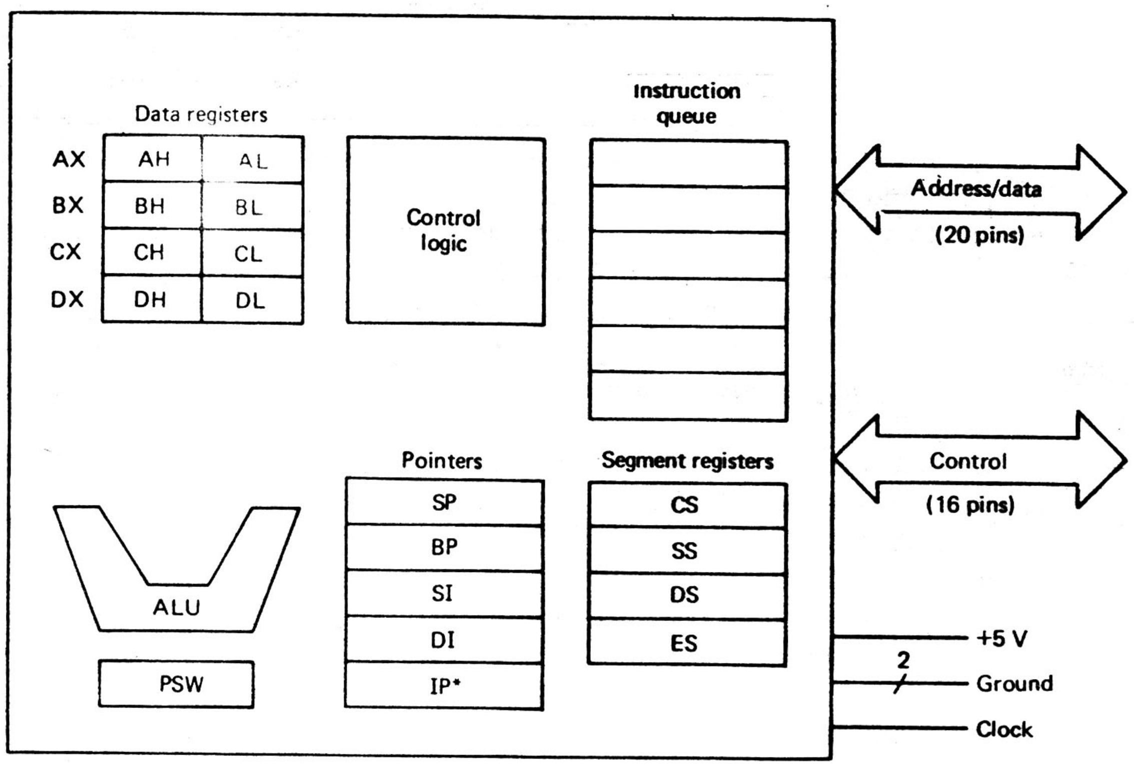


Figure 2: 8086 Internal bus configuration.

instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU, and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands. During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction. If the queue is empty, the EU has to wait for the next instruction byte to be fetched and shifted to the top of the queue. When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions. Whenever this happens, the BIU resets the queue and then begins to fetch instructions from this new location to refill the queue.

3 Register organization

The 8086 has four groups of user-accessible internal registers. They are instruction pointer, four data registers, four-pointer and index register, and four segment registers. 8086 has a total of 14, 16-bit registers including a 16-bit register called *status register*, with 9-bits implemented for status and control flags. Most of the registers contain data/instruction offsets within the 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data, and extra data. To specify where in 1 MB of processor memory these 4 segments are located, the processor uses **four segment registers:**

Code Segment (CS): It is a 16-bit register containing the address of a 64-bit segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack Segment (SS): It is a 16-bit register containing address of 64 KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base

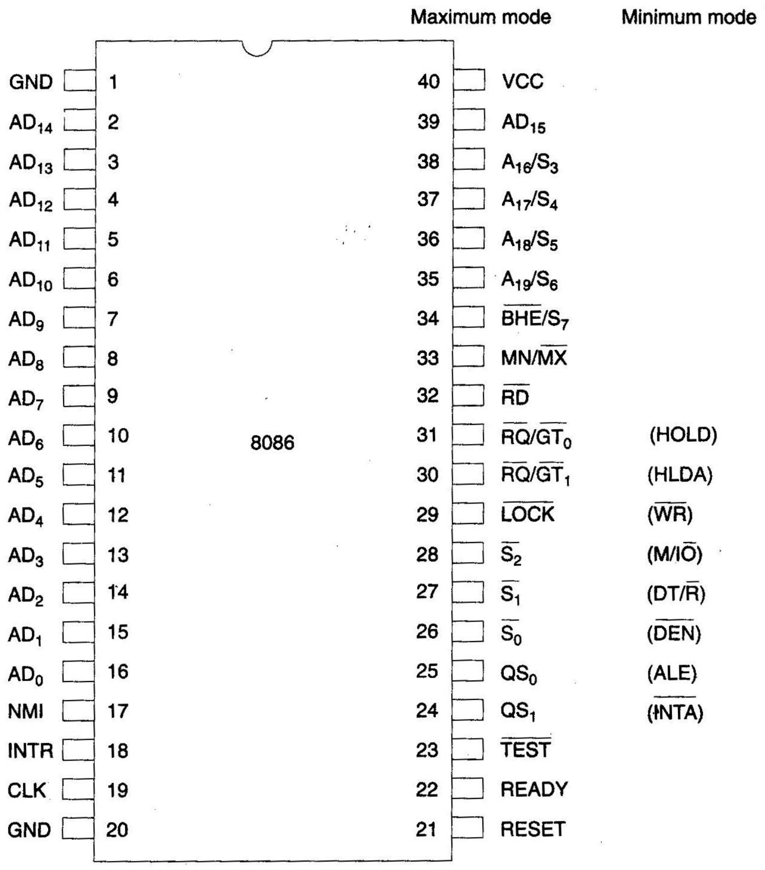


Figure 3: 8086 Pin configuration.

pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

Data Segment (DS): It is a 16-bit register containing address of 64 KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra Segment (ES): It is a 16-bit register containing address of 64 KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions.

It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS, DS or ES prefix. All general registers of the 8086 microprocessor can be used for arithmetic and logic operations. **The general registers** are:

Accumulator: register consists of two 8-bit registers AL and AH, which can be combined and used as a 16-bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

Base: register consists of two 8-bit registers BL and BH, which can be combined and used as a 16-bit register BX. BL in this case contains the low-order byte of the word and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count: register consists of two 8-bit registers CL and CH, which can be combined and used as a 16-bit register CX. When combined, the CL register contains the low-order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation.

Data: register consists of two 8-bit registers DL and DH, which can be combined and used as a 16-bit register DX. When combined, the DL register contains the low-order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains the high-order word of the initial or resulting number.

The following registers are both **general and index registers**:

Stack Pointer (SP): It is a 16-bit register pointing to the program stack.

Base Pointer (BP): It is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed, or register indirect addressing.

Source Index (SI): It is a 16-bit register used for indexed, based indexed, and register indirect addressing, as well as a source data address in string manipulation instructions.

Destination Index (DI): It is a 16-bit register used for indexed, based indexed, and register indirect addressing, as well as a destination data addressing in string manipulation instructions.

Other registers include **Instruction Pointer (IP)** and **Flag register** both of which are of 16-bit size. Only 9-bits of Flag register are used for setting flag bits (Other bits are don't care bits and useless in context), and are as follows:

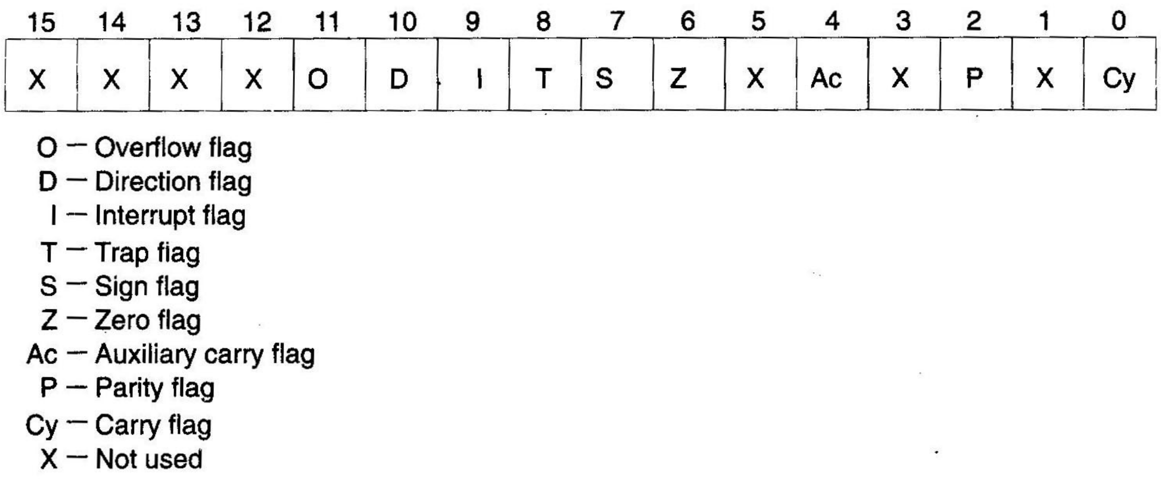


Figure 4: 8086 Flag register.

Overflow Flag (OF): Set if the result is a too large positive number or is a too small negative number to fit into the destination operand.

Destination Flag (DF): If set, then string manipulation instructions will auto-decrement index registers. If cleared, then the index registers will be auto-incremented.

Interrupt-enable Flag (IF): Setting this bit enables maskable interrupts.

Single-step Flag (TF): If set, then a single-step interrupt will occur after the next instruction.

Sign Flag (SF): Set if the most significant bit of the result is set.

Zero Flag (ZF): Set if the result is zero (not set).

Auxiliary carry Flag (AF): Set if there was a carry from or borrow to bits 0-3 in the AL register.

Parity Flag (PF): Set if the parity (the number of 1s) in the lower order byte of the result is even.

Carry Flag (CF): Set if there was a carry from or borrow to the most significant bit during the last result calculation.

4 Modes of operation: Minimum and Maximum mode

Minimum mode: The minimum mode is selected by applying logic 1 to the MN/MX# input pin. This is a single microprocessor configuration. See figures 5 and 6 of Memory read and write timing diagrams of 8086 in minimum mode.

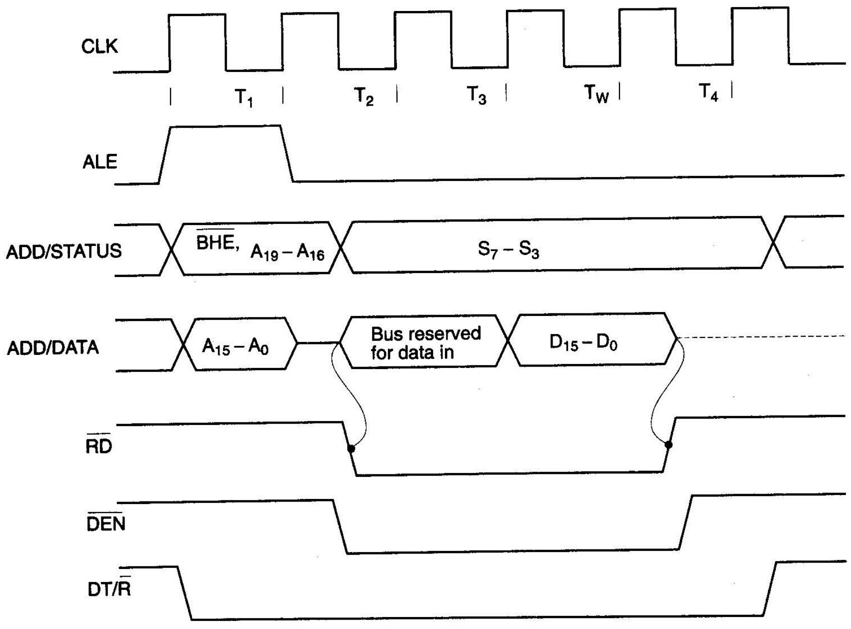


Figure 5: 8086 Memory read timing diagram in minimum mode.

Maximum mode: The maximum mode is selected by applying logic 0 to the MN/MX# input pin. This is a multi microprocessor configuration. See figures 7 and 8 of Memory read and write timing diagrams of 8086 in maximum mode.

5 Addressing Modes

The addressing modes supported by 8086 are-

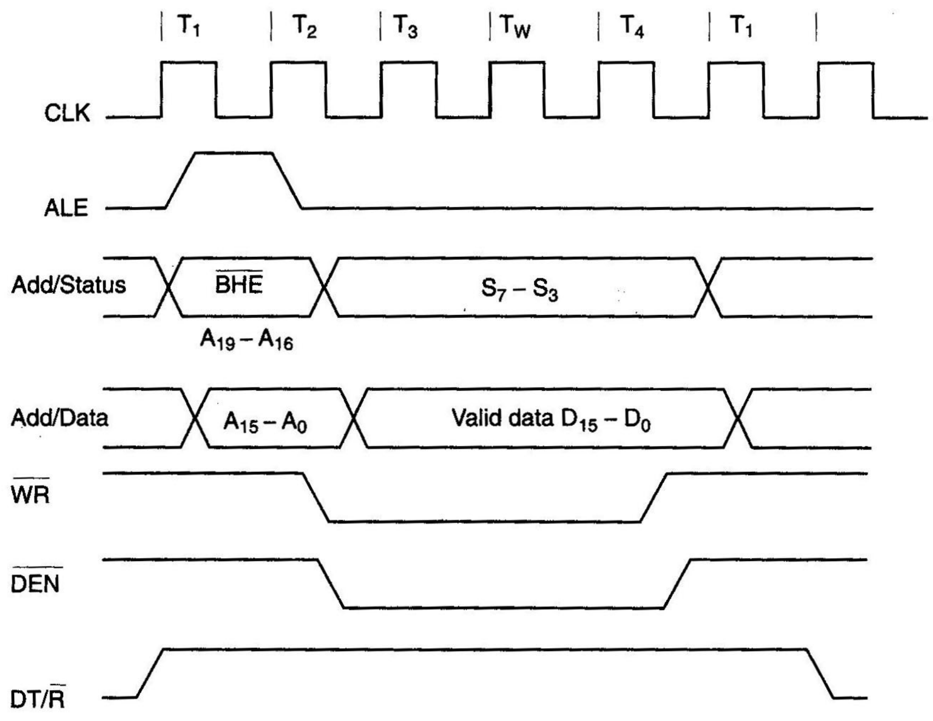


Figure 6: 8086 Memory write timing diagram in minimum mode.

Implied: The data value/data address is implicitly associated with the instruction.

Register: References the data in a register or in a register pair.

Immediate: The data is provided in the instruction.

Direct: The instruction operand satisfies the memory address where data is located (Memory direct).

Register Indirect: The instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX, and BP registers.

Based: 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), and the resulting value is a pointer to the location where data resides.

Indexed: 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), and the resulting value is a pointer to the location where data resides.

Based Indexed: The contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), and the resulting value is a pointer to the location where data resides.

Based Indexed with displacement: 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to the location where data resides.

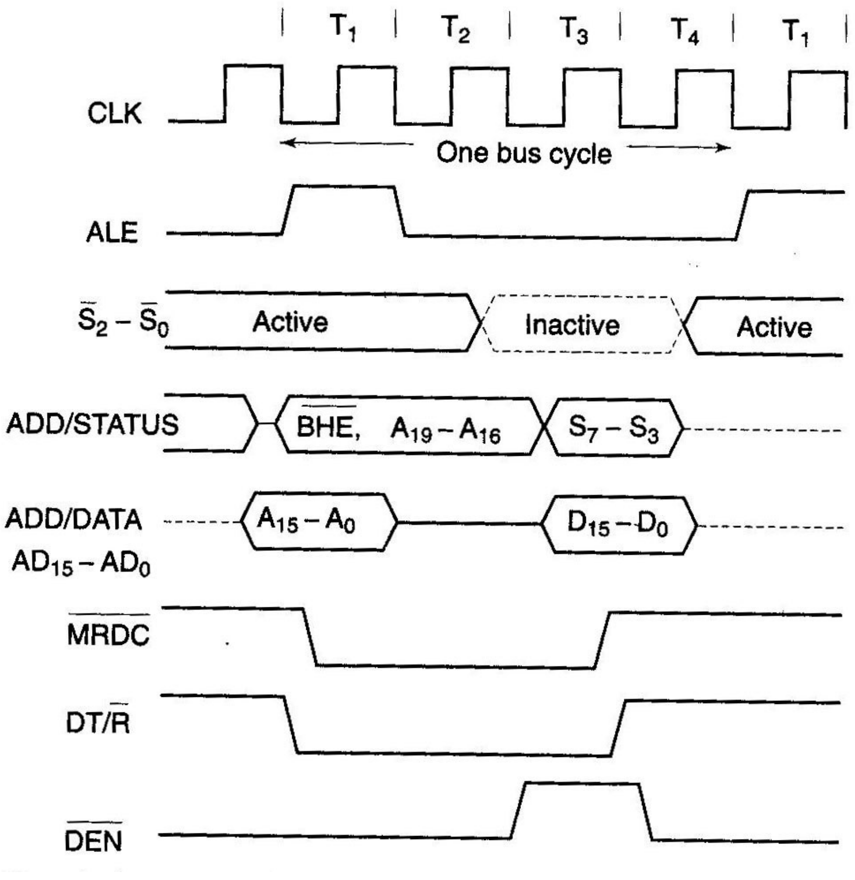


Figure 7: 8086 Memory read timing diagram in maximum mode.

6 Interrupts

Execution sequence for handling an interrupt: When an interrupt occurs, the processor stores a FLAGS register into a stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine, the address of which is stored in location 4 * interrupt_type_address. The interrupt processing routine should return with the IRET instruction.

8086 has the following interrupts-

INTR: It is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.

NMI: It is a non-maskable interrupt. The interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.

Software Interrupts: can be caused by:

INT instruction: Breakpoint interrupt. This is a type-3 interrupt.

INT interrupt_number instruction: Any one of interrupt from available 256 interrupts.

INTO instruction: Interrupt on overflow.

Single step interrupt: Generated if the TF flag is set. This is a type 1 interrupt. When the

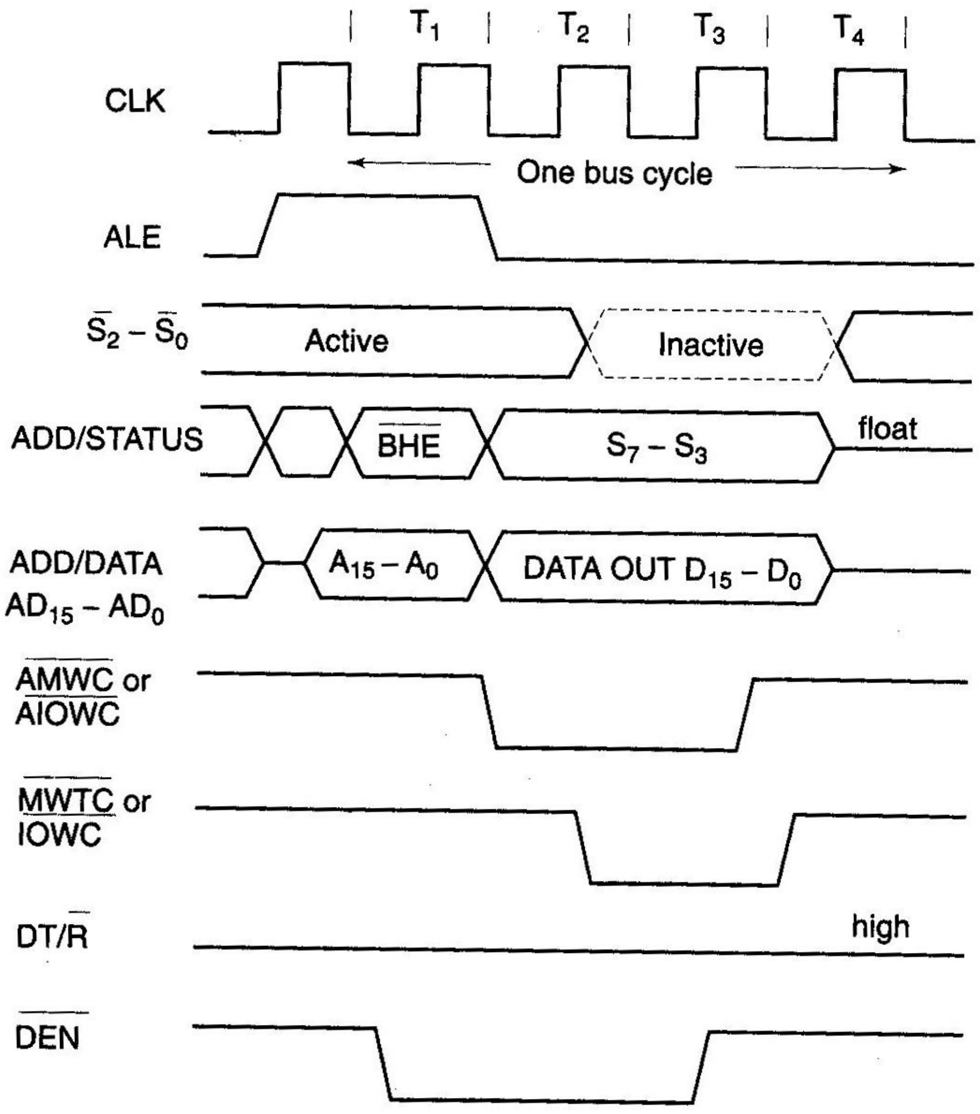


Figure 8: 8086 Memory write timing diagram in maximum mode.

CPU processes this interrupt, it clears the TF flag before calling the interrupt processing routine.

There exists some *processor exceptions* such as divide error (Type 0), Unused opcode (Type 6), and Escape opcode (Type 7). The processing of software interrupts is the same as that of hardware interrupts.

7 Memory

Program, data, and stack memories occupy the same memory space. As the most of the processor instructions use 16-bit pointers, the processor can effectively address only 64 KB of memory. To access memory outside 64 KB, the CPU uses special segment registers to specify where the code, stack and data 64 KB segments are positioned within 1 MB of memory. 16-bit pointers and data are stored as the address: low-order byte, address+1: high-order byte. 32-bit addresses are stored in "segment: offset" format as the address: low-order byte of segment, address+1: high-order byte of segment, address+2: low-order byte of offset, and address+3: high-order byte of offset. Physical memory address pointed by segment: offset pair is calculated as:

$$Physical\ address = (segment\_address * 16) + offset\_address$$

Program memory: Program can be located anywhere in memory. Jump and call instructions can be used for short jumps within currently selected 64 KB code segment, as well as for far jumps anywhere within 1 MB of memory. All conditional jump instructions can be used to jump within approximately +127 to -127 bytes from current instruction.

Data memory: The processor can access data in any one out of 4 available segments, which limits the size of accessible memory to 256 KB (if all four segments point to different 64 KB blocks). Accessing data from the Data, Code, Stack or Extra segments can be usually done by prefixing instructions with the DS:, CS:, SS: or ES: (some registers and instructions by default may use the ES or SS segments instead of DS segment). Word data can be located at odd or even byte boundaries. The processor uses two memory accesses to read a 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.

Stack memory: It can be placed anywhere in memory. The stack can be located at odd memory addresses, but it is not recommended for performance reasons.

0000h-02FFh are reserved memory locations for interrupt vectors. Each interrupt vector is a 32-bit pointer in format segment:offset. FFFF0h-FFFFFh - after reset, the processor always starts program execution at the FFFF0h address.

8 8086 Assembly Programming

8.1 8086 Instruction types (set)

Data flow Instructions: Transfer information between registers and memory locations or I/O ports. e.g. MOV, XCHG, LEA, PUSH, POP, PUSHF, POPF, IN, OUT.

Arithmetic Instructions: Perform arithmetic operations on binary or binary-coded-decimal (BCD) numbers. e.g. ADD, SUB, INC, DEC, ADC, SBB, NEG, CMP, MUL, IMUL, DIV, IDIV, CBW, CWD.

Bit manipulation Instructions: Perform shift, rotate, and logical operations on memory locations and registers. e.g. SHL, SHR, SAR, ROL, ROR, RCL, RCR, NOT, AND, OR, XOR, TEST.

Control transfer (flow) Instructions: Control sequence of program execution; include jumps and procedure transfers. e.g. JMP, JG, JL, JE, JNE, JGE, JLE, JNG, JNL, JC, JS, JA, JB, JAE, JBE, JNB, JNA, JO, JZ, JNZ, JP, JCXZ, LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ, CALL, RET.

String Instructions: Move, compare, and scan strings of information. e.g. MOVS, MOVSB, MOVSW, CMPS, CMPSB, CMPSW, SCAS, SCASB, SCASW, LODS, LODSB, LODSW, STOS, STOSB, STOSW.

Interrupt Instructions: Interrupt processor to service specific condition. e.g. INT, INTO, IRET.

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	((SP)) ← (S) (SP) ← (SP)-2	None
POP	Pop word off stack	POP D	(D) ← ((SP)) (SP) ← (SP)+2	Nonc

(a)

Operand (S or D)
Register
Seg-rcg (CS illegal)
Memory

(b)

Figure 9: PUSH and POP instructions.

Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	((SP)) ← (Flags) (SP) ← (SP)-2	None
POPF	Pop flags from stack	(Flags) ← ((SP)) (SP) ← (SP)+2	OF, DF, IF, TF, SF, ZF, AF, PF, CF

Figure 10: PUSHF and POPF instructions.

Processor Control Instructions: Set and clear status flags, and change the processor execution state, e.g. STC, STD, STI.

Miscellaneous Instructions: e.g. NOP, WAIT.

8.2 Memory Segmentation

A memory segment is a block of 2^{16} (64K) bytes. Each segment is identified by a segment number. Segment number is of 16-bits (0000-FFFF). A memory location is specified by an offset within a segment. Logical address to physical address is discussed in memory section.

Program segments: Program's code, data, and stack are loaded into different memory segments, namely code segment, data segment, and stack segment. At any time, only four memory segments are active. A program segment need not occupy the entire 64 KB. Data segment contains variable declarations and is declared by .DATA. Stack segment is used to store the stack, and is declared by .STACK size. Default size of stack is 1 KB. A code segment contains the program's instructions, and is declared by .CODE.

Memory Models: Following are memory models in 8086:

SMALL: Code in one segment, and data in one segment.

MEDIUM: Code in more than one segment, and data in one segment.

COMPACT: Code in one segment, and data in more than one segment.

LARGE: Code in more than one segment, and data in more than one segment, and no array larger than 64 KB.

HUGE: Code in more than one segment, and data in more than one segment, and arrays may be larger than 64 KB.

9 Most asked Assembly Language Programs (ALPs)

Roughly ordered in increasing order of difficulty. One program may ask for writing more than one assembly programs.

1. Write assembly language programs (ALPs) for addition and subtraction of n numbers (16 bits and 32 bits).
2. Write assembly language programs (ALPs) for multiplication and division of n numbers (16 bits and 32 bits).
3. Write assembly language programs (ALPs) to determine GCD (HCF) and LCM of two numbers (16-bits and 32 bits).
4. Write an assembly language program (ALP) to evaluate expressions.
5. Write assembly language programs (ALPs) for sorting and searching (Finding maximum and minimum numbers among n numbers).
6. Write assembly language programs (ALP) to perform Shift and rotate operations.
7. Write an assembly language program (ALP) to check whether given data is positive or negative using bit manipulation instructions.
8. Write an assembly language program (ALP) to count number of 0s and 1s in a given data.
9. Write an assembly language program (ALP) to check bit wise and nibble wise palindrome.
10. Write an assembly language program (ALP) for converting packed BCD to unpacked BCD.
11. Write an assembly language program (ALP) for BCD to ASCII conversion (Using arithmetic and logical instructions).
12. Write an assembly language program (ALP) for moving a block of data from one memory location to another.
13. Write an assembly language program (ALP) for reversing a string.
14. Write an assembly language program (ALP) for comparison of two strings.
15. Write an assembly language program (ALP) to find length of a string using string instructions.

16. Write an assembly language program (ALP) to change an already available ascending order byte string to descending order.

17. Write an assembly language program (ALP) to convert a given sixteen bit binary number to its GRAY equivalent.

18. Write an assembly language program (ALP) to find out transpose of a 3x3 matrix.

19. Write an assembly language program (ALP) to find nth Fibonacci number.

20. Write an assembly language (ALP) program for separation of odd and even numbers.

21. Write an assembly language program (ALP) to find out cube (or square) of an 8-bit hexadecimal number.

22. Write an assembly language program (ALP) for traffic light controller.

10 Question Bank

1. What is Pipelining and Segmentation? What were the drawbacks of 8085 that were overcame by them. What is the use of memory segmentation in 8086?

2. Which pins determine the mode of operation of 8086? What are the pins that specify various types of transfer in minimum and maximum mode?

3. What are the types of transfer of 8086? What is the use of Bus controller?

4. Explain in brief following terms: Bus, Memory module, I/O subsystem, Interface, Memory Interface, I/O Interface.

5. Explain the internal architecture of 8086 with diagram.

6. Describe the functions of EU and BIU.

7. Explain register organization of 8086. What the function of various registers and pointers in 8086?

8. Describe Flag register with its functions in 8086

9. How 20-bit address is generated in 8086?

10. Explain timing diagrams of memory read and memory write machine cycles in 8086.

11. Explain minimum and maximum mode of 8086? Which mode is used for multiprocessor configuration?

12. What are various flags of 8086?

13. What are operators and operands?

14. Enlist various instructions of 8086?

15. Describe following instructions: Data transfer instructions, Arithmetic instructions, Bit manipulation instructions, Program control instructions, Processor control instructions, string instructions.

16. What are addressing modes? Explain various addressing modes in 8086?

17. What are Procedures and MACROS? Write functions of them.
18. What are the differences between procedures and MACROS? What is the fundamental difference between a MACRO and a subroutine? Enlist various advantages and disadvantages of Macros.
19. Explain various machine language instruction formats supported by 8086.
20. What is Assembler and linker? What are the differences between them?
21. What are interrupts? Describe functions of an interrupt vector table. How many interrupts are there in 8085. Explain their maskability property and priority order between them.
22. What are the procedures when interrupt happens in 8086.
23. How are stack pointer and stack segment registers initialized when pushing first element on to a stack?
24. Describe interrupts. Classify vectored and non-vectored interrupts in 8086. Write a simple routine to demonstrate software interrupt procedure.

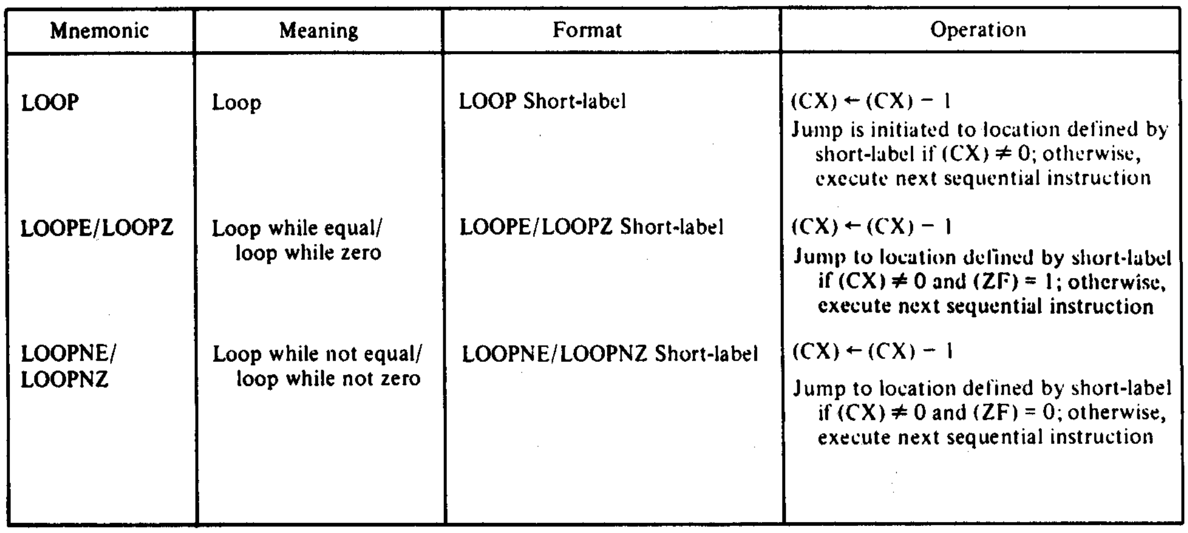
Mnemonic	Meaning	Format	Operation	Flags affected
Jcc	Conditional jump	Jcc Operand	If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed.	None

(a)

Mnemonic	Meaning	Condition
JA	above	CF = 0 and ZF = 0
JAE	above or equal	CF = 0
JB	below	CF = 1
JBE	below or equal	CF = 1 or ZF = 1
JC	carry	CF = 1
JCXZ	CX register is zero	(CF or ZF) = 0
JE	equal	ZF = 1
JG	greater	ZF = 0 and SF = OF
JGE	greater or equal	SF = OF
JL	less	(SF xor OF) = 1
JLE	less or equal	((SF xor OF) or ZF) = 1
JNA	not above	CF = 1 or ZF = 1
JNAE	not above nor equal	CF = 1
JNB	not below	CF = 0
JNBE	not below nor equal	CF = 0 and ZF = 0
JNC	not carry	CF = 0
JNE	not equal	ZF = 0
JNG	not greater	((SF xor OF) or ZF) = 1
JNGE	not greater nor equal	(SF xor OF) = 1
JNL	not less	SF = OF
JNLE	not less nor equal	ZF = 0 and SF = OF
JNO	not overflow	OF = 0
JNP	not parity	PF = 0
JNS	not sign	SF = 0
JNZ	not zero	ZF = 0
JO	overflow	OF = 1
JP	parity	PF = 1
JPE	parity even	PF = 1
JPO	parity odd	PF = 0
JS	sign	SF = 1
JZ	zero	ZF = 1

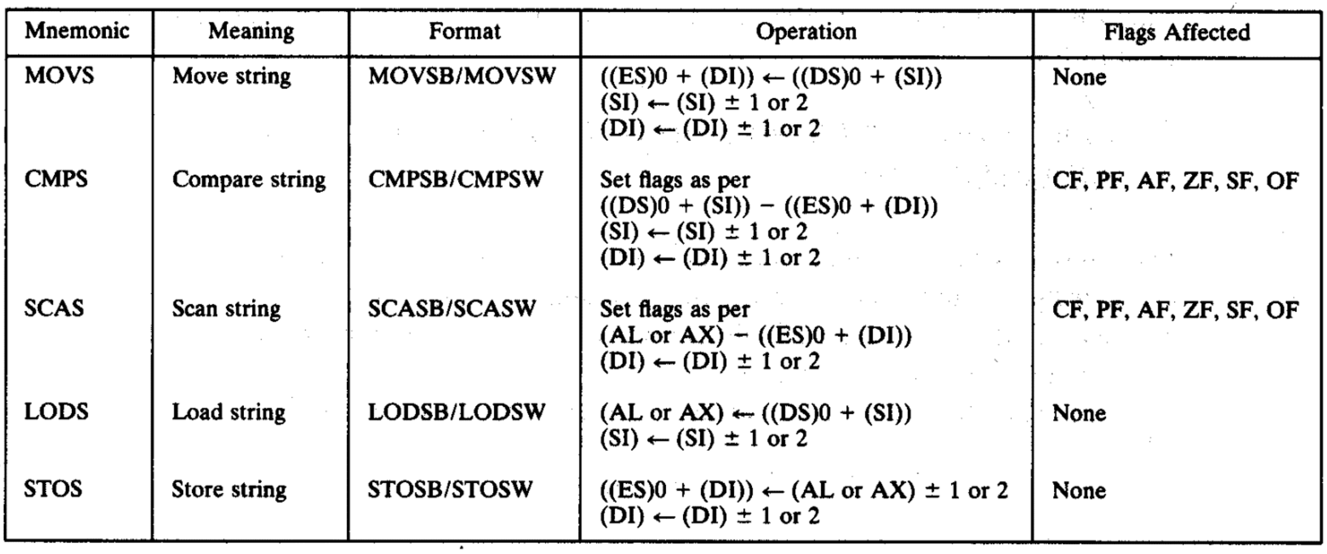
(b)

Figure 11: Jump instructions.



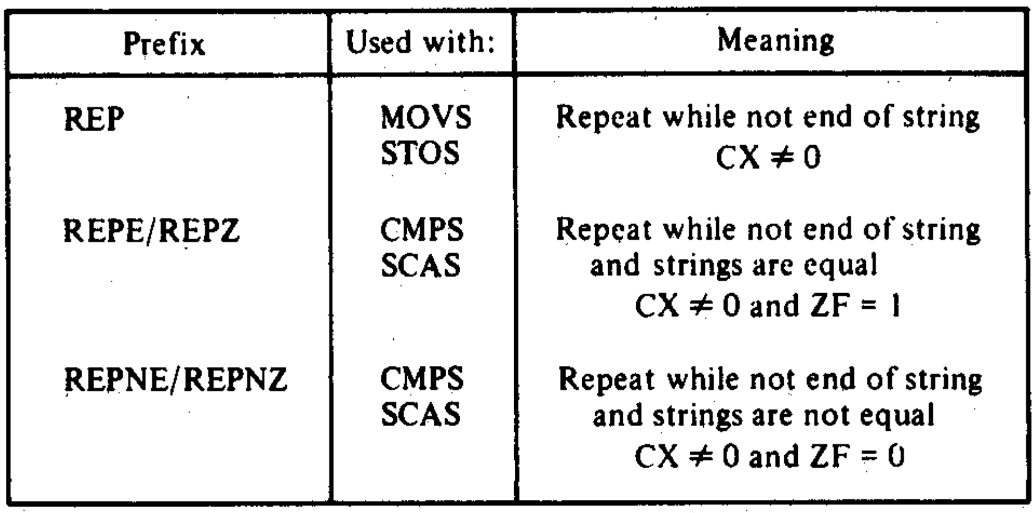
Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	(CX) ← (CX) − 1 Jump is initiated to location defined by short-label if (CX) ≠ 0; otherwise, execute next sequential instruction
LOOPE/LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 1; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 0; otherwise, execute next sequential instruction

Figure 12: Loop instructions.



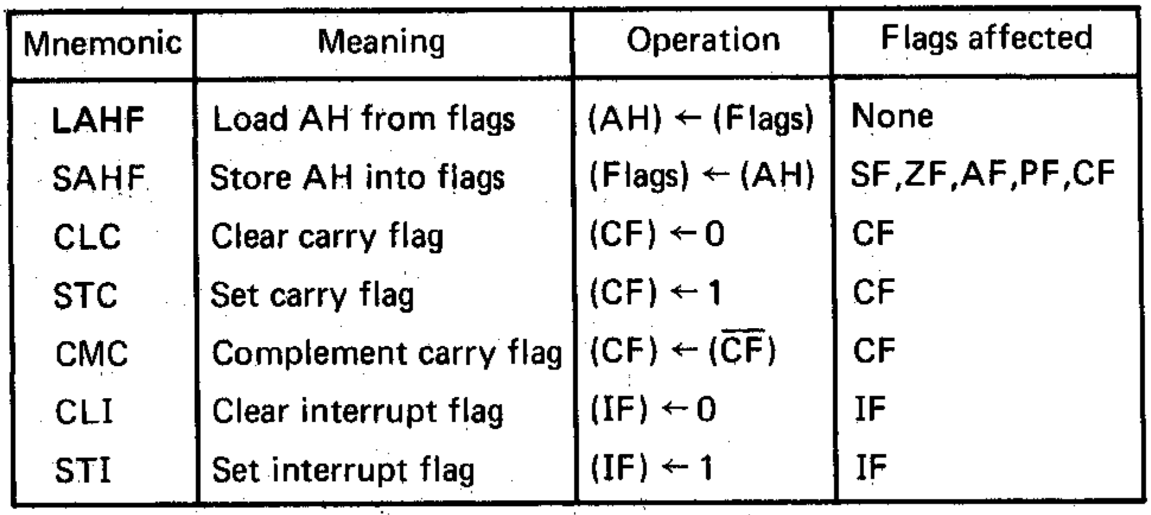
Mnemonic	Meaning	Format	Operation	Flags Affected
MOVS	Move string	MOVSB/MOVSW	((ES)0 + (DI)) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None
CMPS	Compare string	CMPSB/CMPSW	Set flags as per ((DS)0 + (SI)) − ((ES)0 + (DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/SCASW	Set flags as per (AL or AX) − ((ES)0 + (DI)) (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/LODSW	(AL or AX) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2	None
STOS	Store string	STOSB/STOSW	((ES)0 + (DI)) ← (AL or AX) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None

Figure 13: String handling instructions.



Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0

Figure 14: Repeat string instructions.



Mnemonic	Meaning	Operation	Flags affected
LAHF	Load AH from flags	(AH) ← (Flags)	None
SAHF	Store AH into flags	(Flags) ← (AH)	SF,ZF,AF,PF,CF
CLC	Clear carry flag	(CF) ← 0	CF
STC	Set carry flag	(CF) ← 1	CF
CMC	Complement carry flag	(CF) ← ($\overline{\text{CF}}$)	CF
CLI	Clear interrupt flag	(IF) ← 0	IF
STI	Set interrupt flag	(IF) ← 1	IF

Figure 15: Flag (Processor) control instructions.

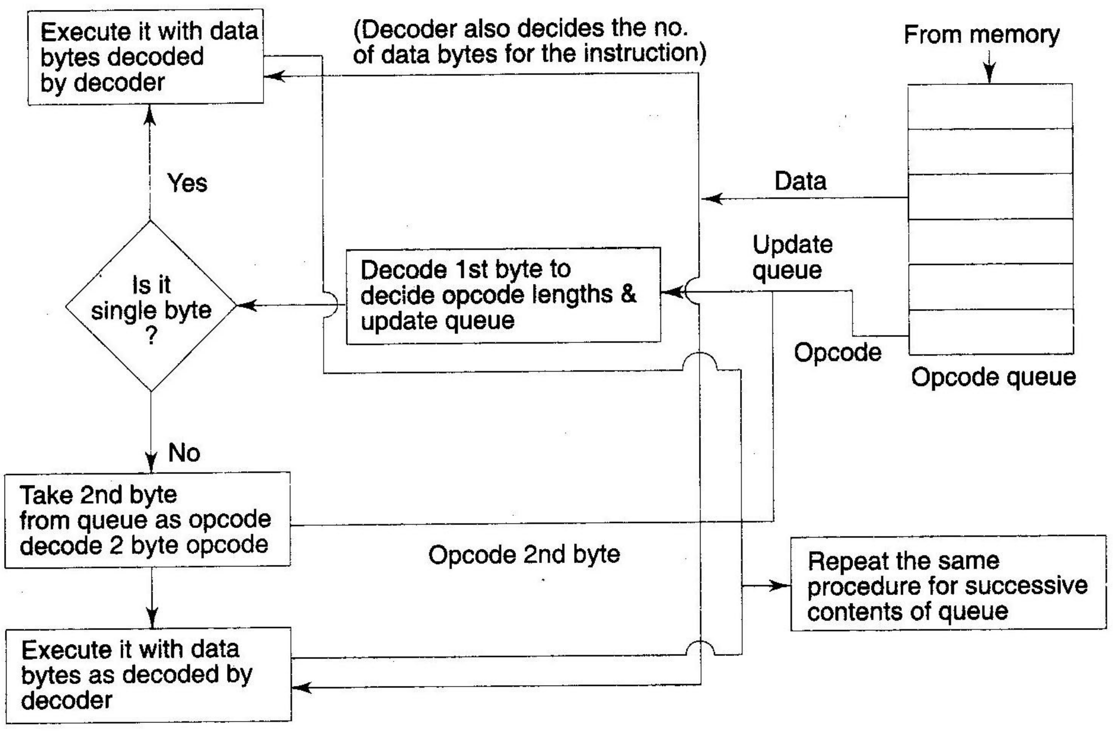

Execute it with data bytes decoded by decoder
(Decoder also decides the no. of data bytes for the instruction)
From memory
Yes
Data
Update queue
Is it single byte ?
Decode 1st byte to decide opcode lengths & update queue
Opcode
Opcode queue
No
Take 2nd byte from queue as opcode decode 2 byte opcode
Opcode 2nd byte
Repeat the same procedure for successive contents of queue
Execute it with data bytes as decoded by decoder

Figure 16: 8086 General instruction queue operation flow chart.