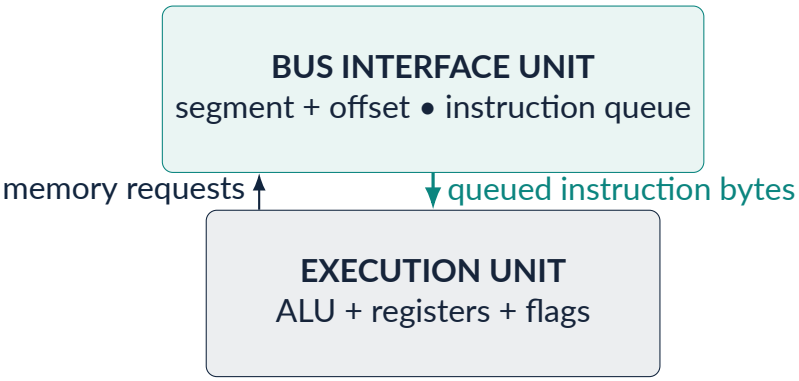
ZENITH STUDY HUB

Microprocessor & Micro-controller

UNIT II | INTEL 8086

Architecture • Addressing • Instructions
Interrupts • Assembly programming



OFFICIAL SYLLABUS BOUNDARY Savitribai Phule Pune University, Third Year Robotics & Automation Engineering, 2024 Pattern, Semester V. **PCC-302-ROA, Unit II (08 teaching hours).** The five required headings are: 8086 Architecture; Addressing Modes; Instruction Set; Interrupts; Programming Example 8086.

Prepared by: Om Deshmukh **Edition:** Complete Learning Edition 3.0
Date: 21 Nov 2026 Student-authored educational material; not an official SPPU publication. This document is a review candidate and has not been added to the live v35 site.

Guide map and learning contract

IN SIMPLE WORDS

A microprocessor is a machine that repeatedly fetches bytes, interprets instructions, and manipulates data. The 8086 becomes easier when you separate four questions: *where does an instruction come from? where does data come from? what operation happens? what changes control flow?*

EXACT SYLLABUS COVERAGE

Official Unit II topic	What this guide covers
8086 Architecture	BIU, EU, registers, flags, segmentation, physical address, bus banks, prefetch and operating modes
Addressing Modes of 8086	immediate, register, direct, indirect, based, indexed, based-indexed, displacement, effective address and segment defaults
Instruction Set of 8086	data movement, arithmetic, logical, branches, string, stack, flags and operand-size rules
Interrupts in 8086	INTR, NMI, INT, FLAGS/CS/IP stack save, vector table, IRET and numerical vector lookup
Programming example 8086	seven MASM-style original programs, dry runs, register/flag checks, practice exercises and corrections

Release and verification status. The earlier draft reported 74/74 static QA checks, but no full assembler/emulator execution. This expanded learning edition is undergoing layout, source mapping and independent arithmetic checks; do not label any program hardware-tested without running an appropriate toolchain. Publication to Drive or v35 production is not part of this package.

STUDY ORDER

1. Read each “in simple words” explanation, then redraw the adjacent block diagram.
2. Derive segment:offset addresses and effective-address examples by hand.
3. Learn flags before choosing signed versus unsigned jumps.

4. Trace each listing with inputs and outputs; do not just memorize mnemonics.
5. Answer the exam-style questions, then compare with the answer frameworks.

Contents

Guide map and learning contract 2

1 8086 architecture and working principle 4

2 Registers and the FLAGS word 10

3 8086 memory segmentation 12

4 8086 memory banking and bus timing 13

5 Addressing modes: finding operands 15

6 Instruction set: learn by behavior 20

7 Interrupts in the 8086 26

8 8086 programming conventions 31

9 Program 1: sum four bytes, keep the full 16-bit sum 32

10 Program 2: maximum of five unsigned values 33

11 Program 3: copy five bytes with REP MOVSB 34

12 Program 4: add two 16-bit words and retain carry 35

13 Program 5: even/odd and TEST 36

14 Program 6: count zero-valued bytes 36

15 Program 7: signed and unsigned branching 37

16 Solved numericals and exam examples 43

17 Worked addressing-mode drill 44

18 Instruction behavior: five important micro-cases 45

19 Algorithm visualizations and dry-run checkpoints 46

20 Source-trace cases for robotics 47

21 Extended original practice with answer checks 48

22 Exam-ready answer frameworks 49

23 Revision sheet and self-test 50

24 Sources, provenance and publication checklist 51

1 8086 architecture and working principle

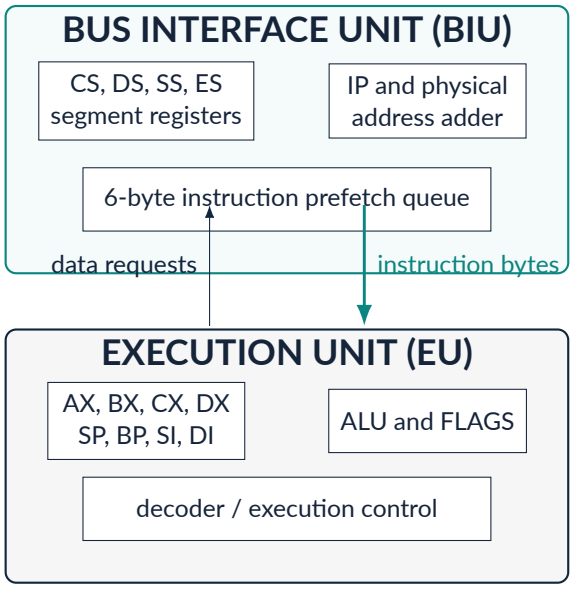
IN SIMPLE WORDS

The Intel 8086 is a 16-bit CPU with a 16-bit external data bus and a **20-bit physical address bus**. One physical byte is selected by a 20-bit address. Most programmer-visible registers contain 16 bits, so memory is addressed using a segment and an offset.

1.1 Characteristics at a glance

Characteristic	Meaning
16-bit data processing	AX, BX, CX, DX and ALU support 16-bit word operations; 8-bit halves AH/AL, etc. are addressable.
20 address lines	$2^{20} = 1,048,576$ bytes, exactly 1 MiB of physical address space.
16-bit data bus	An aligned 16-bit memory transfer can use both byte banks in one bus cycle.
BIU / EU organization	Instruction fetch may overlap execution through a six-byte queue.
Segmented addressing	Physical address is formed from a shifted 16-bit segment and 16-bit offset.
Operating mode	Minimum mode for simpler single-processor arrangements; maximum mode with companion bus-controller logic.

1.2 Functional architecture: what lives where?



HOW ONE INSTRUCTION RUNS

1. The BIU calculates an address and fetches instruction bytes into the queue when bus time permits.
2. The EU decodes the next queued instruction, reads registers, and may ask the BIU for a memory operand.
3. The EU computes results and changes relevant flags, registers or memory.
4. A taken branch or interrupt redirects the instruction stream; queued bytes from the old path cannot be reused.

Common trap: “16-bit CPU” does not mean 16 address lines. The original 8086 has 20 address lines and thus 1 MiB, not 64 KiB, of physical address space.

1.3 Instruction prefetch and limits

The BIU contains a **six-byte prefetch queue**, allowing overlapping fetch and execute activity in suitable circumstances. This is an early form of pipelining, not two independent processors. Memory accesses requested by the EU share the bus; jumps invalidate wrong-path prefetched bytes. Avoid promising a fixed speed-up: instruction mix, bus wait states, and branch frequency matter.

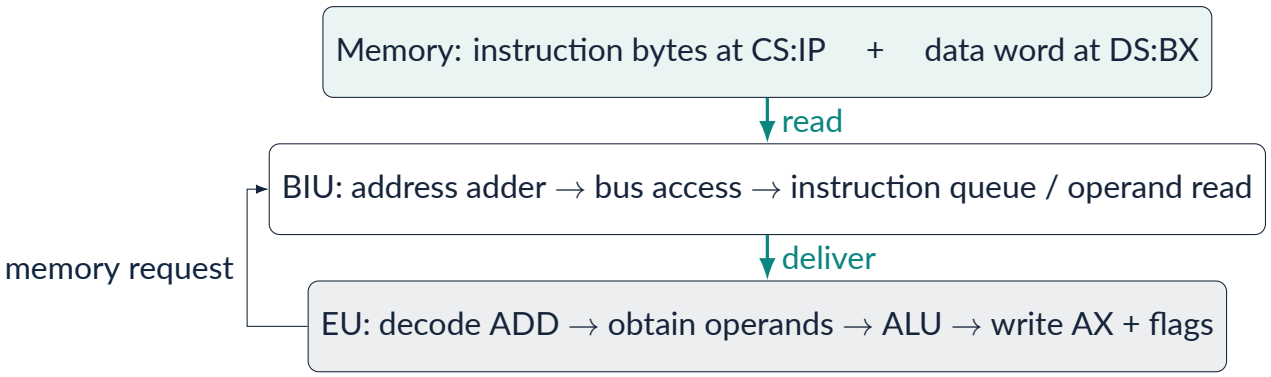
1.4 Minimum and maximum modes

	Minimum mode	Maximum mode
Typical arrangement	One processor with direct bus control from CPU	Multiprocessor/coprocessor arrangements
Bus controls	Selected bus status/read-write signals generated directly by CPU	Status decoded by external controller (such as 8288)
What to remember	Mode affects interface signaling, not the meaning of ADD or MOV	Same basic 8086 architecture, alternate bus-control organization

1.5 Architecture masterclass: see one complete instruction journey

IN SIMPLE WORDS
Think of the BIU as the delivery system and the EU as the workshop. The BIU brings instruction bytes and data from memory. The EU decides what they mean and performs calculations. Both are parts of ONE processor.

Suppose the CPU is executing `ADD AX,[BX]`. It first needs the instruction's encoded bytes from the code segment; next, it needs the operand stored in memory. Those two kinds of access do not have to use the same segment register. During ordinary instruction fetch, the BIU uses CS:IP. For the data reference `[BX]`, it normally uses DS:BX. The EU then performs the addition, updates AX and the defined status flags, and moves on.

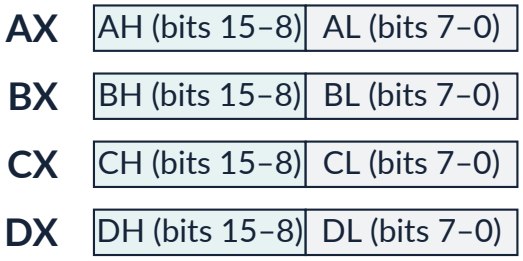


WHY THE QUEUE IMPROVES PERFORMANCE WITHOUT GUARANTEEING IT An 8086 queue holds at most six instruction bytes. During time when the external bus

is not occupied by an operand transfer, the BIU can obtain future instruction bytes. The EU may work on one instruction while the BIU obtains later ones. A branch changes the next instruction address: prefetched bytes from the abandoned path must be discarded, and the BIU starts again from the branch destination. If many instructions access memory or branch frequently, the speed advantage is reduced. This is an overlap of work, not simultaneous execution of two independent program instructions.

Exam question: BIU versus EU in four points. BIU contains segment registers, IP, address-generation logic, bus control and the prefetch queue. EU contains the ALU, general-purpose/pointer/index registers, FLAGS and instruction-decoding/execution control. BIU handles external instruction/-data transfers; EU executes operations. Their overlap can improve throughput, but bus contention and branches limit it.

1.6 Register masterclass: what every 16-bit register is for



AX=1234H means AH=12H, AL=34H.
Writing AL does not automatically replace AH.

AX frequently holds an accumulator value; some multiply/divide and I/O instructions use it implicitly. BX can act as a legal memory base. CX is the counter for LOOP and repetition prefixes; CL can also specify certain variable shift counts. DX participates with AX in extended arithmetic and can specify an I/O port. Their names indicate common uses, *not* that other legal arithmetic with those registers is forbidden. SP addresses the top of the stack relative to SS. BP often addresses stack-local data and defaults to SS for memory operands. SI and DI are index registers; string operations use SI for source and DI for destination. CS, DS, SS and ES identify segment bases; IP identifies an instruction offset. Do not confuse a register's bit width with the physical bus address width.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Register halves in an actual trace

AX starts as 1234H. After `MOV AL,56H`, AX becomes 1256H because AH remains 12H. If a later `MOV AH,00H` executes, AX becomes 0056H. Both MOV instructions preserve arithmetic flags.

1.7 The FLAGS word: draw the positions and explain the purpose

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

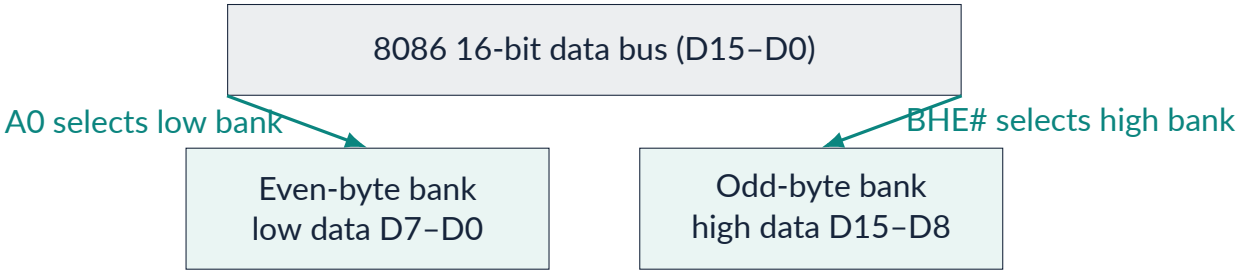
The important **status flags** are CF (unsigned carry or subtraction borrow), PF (even parity of low eight result bits), AF (carry across bit 3/4), ZF (zero result), SF (high result bit) and OF (signed overflow). The important **control flags** are TF (trap), IF (maskable interrupt enable) and DF (string direction). Some flag-word positions are reserved in the original 8086 and should not be treated as independent general-purpose bits.

Two common mistakes. (1) SF=1 alone does not prove signed overflow or a negative mathematical answer when the operation overflowed. (2) OF and CF can differ for the same addition because signed and unsigned interpretations have different ranges.

1.8 Memory bank selection: draw the low and high byte paths

IN SIMPLE WORDS

A byte is eight bits. A word is two bytes. The 8086 has sixteen data pins, but memory still has individually numbered BYTE addresses. The low and high banks cooperate to move a word at an even address.



Access	A0	BHE#	Interpretation
Byte at even address	0	1	Low bank only
Byte at odd address	1	0	High bank only
Word at even address	0	0	Both banks together, one transfer
Word at odd address	1, then 0	0, then 1	First high bank, then low bank: two transfers

A0 and BHE# are *active selection signals* for bank choice. The notation BHE# means active low, so "0" enables the high bank. For an odd-addressed word, the first and second memory bytes reside in different banks and two transfers are required. This explains an important performance detail without claiming a precise instruction timing for every system.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

One memory value, four distinct statements

A word 4A2BH is stored beginning at address 3001H. The lower address holds 2BH; the next address 3002H holds 4AH; the access is unaligned; and the 8086 normally needs two bus transfers to read the full word. Writing 4A2BH into one memory *byte* would be a size error.

1.9 Signals and external hardware: where architecture ends

The multiplexed AD0–AD15 pins transport address in the first bus phase and data later. External latches hold the address while the pins change role; ALE coordinates this action. RD# and WR# describe read/write transactions, and suitable buffering, address decoding, memory and peripherals complete the computer. In maximum mode bus-control signals are generated using an external controller such as the 8288. **A raw 8086 is not a standalone robot controller:** motor power stages and sensor interfaces are separate circuits.

> **Exam sketch recommendation.** Draw a CPU rectangle, an address latch receiving ALE, a decoder selecting RAM/ROM/I-O, and a bidirectional data-bus arrow. Label which signals originate from the CPU and where the motor-driver interface begins. A labelled block diagram is more useful than a memorized unexplained pin list.

2 Registers and the FLAGS word

IN SIMPLE WORDS

Registers are tiny, fast storage locations inside the processor. Different registers can hold numbers, offsets, loop counters, the stack pointer or segment values. Flags describe an operation's result or control certain CPU behavior.

Group	Registers	Common role
General	AX, BX, CX, DX	AX accumulator; BX base; CX loop count; DX data/port address. Each splits into two 8-bit halves (AH/AL etc.).
Pointer and index	SP, BP, SI, DI	Stack pointer, stack-frame base and memory/string indexing.
Segments	CS, DS, SS, ES	Code, data, stack and extra segment.
Instruction	IP	Offset of next instruction in CS.
Flags	CF PF AF ZF SF TF IF DF OF	Arithmetic status and execution-control bits.

2.1 Status flags versus control flags

Flag	Meaning	Useful test
CF	Unsigned carry / borrow	`FFH + 01H = 00H` in AL gives CF=1.
OF	Signed two's-complement overflow	`7FH + 01H = 80H` gives OF=1, CF=0.
ZF	Result is zero	`CMP AX,BX` sets ZF if equal.
SF	Sign bit of the result	Byte result 80H has SF=1.
PF	Even parity in low 8 result bits	03H has two set bits so PF=1.
AF	Carry/borrow between bit 3 and 4	Important in decimal-adjust operations.
TF	Trap (single-step) control	Debugging control.
IF	Maskable interrupt enable	CLI clears IF; STI sets IF.
DF	String processing direction	CLD increments SI/DI; STD decrements.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Signed and unsigned addition, same bit pattern

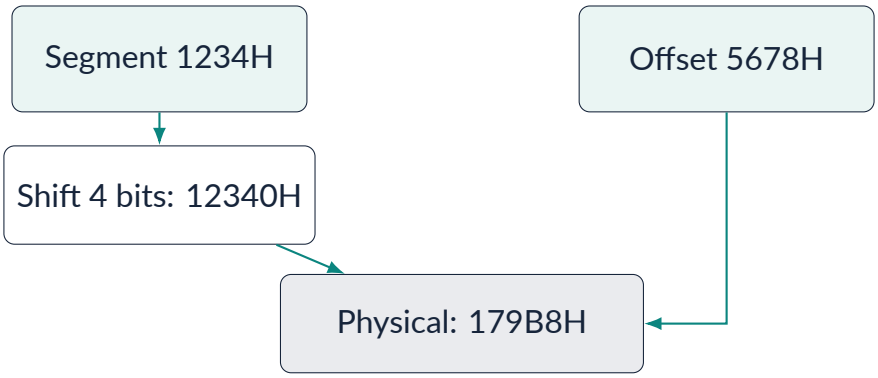
Take AL=7FH (+127 if signed) and add 01H. The bits become 80H. As *unsigned* arithmetic this is 127+1=128, with no carry beyond bit 7 (CF=0). As *signed* arithmetic +127+1 exceeds +127, so OF=1. Therefore use `JA/JB` for unsigned comparisons and `JG/JL` for signed comparisons after an appropriate comparison.

Flag rule: not all instructions update every flag. For example MOV and LEA do not update status flags; INC/DEC do not change CF. Read each instruction's defined effects rather than guessing from its name.

3 8086 memory segmentation

IN SIMPLE WORDS

Segment:offset is like a base address plus a displacement. Shift the segment value four binary positions left (multiply by 16), add the 16-bit offset, then keep the low 20 physical address bits. A different segment:offset pair may point to the same memory byte.



$$\text{Physical} = (16 \times \text{segment} + \text{offset}) \bmod 2^{20}$$

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Instruction address from CS:IP

CS=1234H, IP=5678H. Compute 1234H $\times$ 10H $=$ 12340H, then 12340H $+$ 5678H $=$ $\boxed{179B8H}$. This is the physical byte address of the next instruction, assuming the specified CS:IP.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

A data operand from DS:SI

DS=1A20H and SI=0F30H for an operand `MOV AL,[SI]`. Default segment DS gives 1A200H $+$ 0F30H $=$ $\boxed{1B130H}$.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Overlapping logical addresses

1000H:0020H and 1001H:0010H both give 10020H. One physical byte can have multiple logical names; segmentation alone is *not* memory protection.

Boundary case: At segment FFFFH and offset FFFFH, the arithmetic sum is 10FFEFH. The original 8086 exposes only 20 address bits, so the physical address wraps to 0FFEFH. Do not apply modern protected-mode address

rules.

3.1 Which segment is used?

Access	Default segment
Instruction fetch	CS:IP
Ordinary memory operand using BX/SI/DI or direct address	DS
Memory operand using BP as base	SS (unless overridden)
PUSH/POP/CALL/RET stack accesses	SS:SP
MOVSB source and destination	DS:SI to ES:DI (destination ES is fixed)

3.2 Stack: what happens to SP?

For an ordinary `PUSH AX`, SP decreases by 2 and then a 16-bit word is written at SS:SP. Because storage is little-endian, the low-order byte is at the lower address. A `POP BX` reads a word at SS:SP and then increases SP by 2.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Trace a PUSH

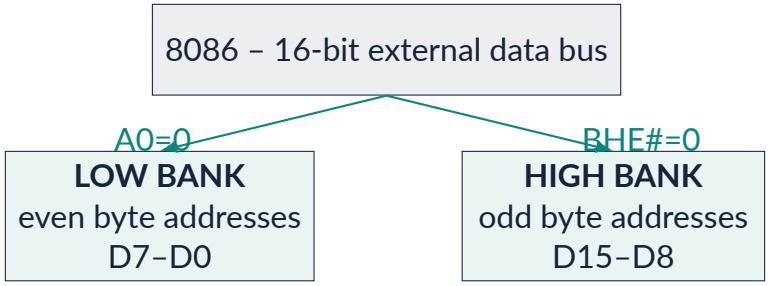
SS=2000H, SP=0100H, AX=1234H. After PUSH AX: SP=00FEH; physical address 200FEH holds 34H and 200FFH holds 12H. POP BX restores BX=1234H and SP=0100H.

4 8086 memory banking and bus timing

IN SIMPLE WORDS

A 16-bit word contains two adjacent bytes. The 8086 uses an even-addressed low bank (D7–D0) and odd-addressed high bank (D15–D8). An aligned word can use both banks together; an odd-addressed word needs

two bus cycles.



Read/write request	Bus behavior
Byte at even address	Low bank only, D7–D0.
Byte at odd address	High bank only, D15–D8.
Word at even address	Both banks in one normal data-transfer bus cycle, without added wait states.
Word at odd address	Two separate bank transfers at adjacent addresses.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

A word at even and odd addresses

For word 1234H at 2000H: memory[2000H]=34H and memory[2001H]=12H; an aligned 16-bit transfer can read both. At 2001H the low byte 34H is at 2001H (high bank) and high byte 12H at 2002H (low bank); two bus cycles are needed.

4.1 Multiplexed pins and bus phases

AD0–AD15 carry address information during the address phase and data during the data phase. ALE helps the system latch address information. Read/write and status signals distinguish transfer types. Wait states may be inserted for slower devices. **Do not** confuse a bus cycle with an entire instruction cycle: many instructions require multiple bus operations.

5 Addressing modes: finding operands

IN SIMPLE WORDS

An *addressing mode* tells the CPU where to find an operand. A register operand is inside the processor; an immediate operand is inside instruction bytes; a memory operand uses an effective address (EA) that the CPU calculates.

5.1 The modes you must recognize

Mode	Example (MASM style)	Meaning
Immediate	`MOV AL,25H`	25H is literal data.
Register	`MOV AX,BX`	Copy one register to another.
Direct memory	`MOV AL,[2500H]`	EA=2500H in DS.
Register indirect	`MOV AL,[BX]`	EA=BX.
Based + disp	`MOV AX,[BP+06H]`	EA=BP+6, default SS.
Indexed	`MOV AL,[SI]`	EA=SI, default DS.
Based indexed	`MOV AL,[BX+DI]`	EA=BX+DI, default DS.
Based indexed + disp	`MOV AX,[BX+SI+08H]`	EA=BX+SI+8.

5.2 Effective address (EA) versus physical address

$$\text{EA} = (\text{base} + \text{index} + \text{displacement}) \bmod 2^{16}$$

$$\text{physical} = (16 \times \text{segment} + \text{EA}) \bmod 2^{20}$$

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Based-indexed displacement

BX=1000H, SI=0020H, displacement=0008H and DS=3000H. Calculate in two

separate steps:

$$\text{EA} = 1000\text{H} + 0020\text{H} + 0008\text{H} = \boxed{1028\text{H}},$$
$$\text{physical} = 30000\text{H} + 1028\text{H} = \boxed{31028\text{H}}.$$

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

BP uses SS by default

BP=0040H, displacement=0006H, SS=2400H. For `MOV AX,[BP+06H]`, EA=0046H and physical address=24046H. If DS had the same offset it would reference a different byte; default segment matters.

5.3 Which combinations are legal on an original 8086?

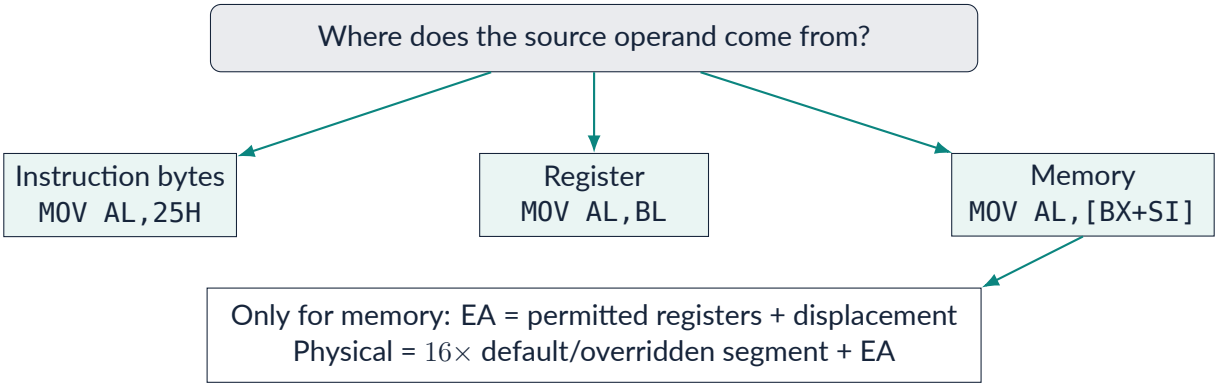
The memory addressing base/index combinations are `BX+SI`, `BX+DI`, `BP+SI`, `BP+DI`, or a permitted single base/index (BX, BP, SI, DI), with optional 8-bit or 16-bit displacement. A pure 16-bit direct displacement is also supported. Register AX is *not* a general memory base in the original 8086 addressing forms.

Illegal or ambiguous examples: `MOV AX,[AX]` is not an 8086 memory-addressing form. `MOV [2500H],[2600H]` is not a general MOV memory-to-memory operation (use a register or string instruction). When immediate data is stored into memory, make the byte/word size clear when it cannot be inferred.

5.4 Addressing modes: one decision tree instead of memorizing a list

IN SIMPLE WORDS

Before calculating anything, ask: does the instruction contain the number, name a register, or point into memory? If memory is involved, first find the effective address. Only then add the segment base.



On the original 8086, the legal memory base/index combinations have a special hardware meaning. Possible base registers are BX and BP; possible index registers are SI and DI; a permitted pair contains *one base plus one index*. You may also use a single eligible register or a direct 16-bit displacement. For example, `[BX+DI+08H]` is legal but `[SI+DI]` is not an original-8086 addressing form. More recent x86 processors allow forms the 8086 did not.

Critical 8086 restriction. `[BX+SI]` and `[BP+DI]` are legal. `[BX+BP]`, `[SI+DI]`, `[AX]` and scaled indexes such as `[SI*4]` are *not* original-8086 effective-address forms.

5.5 Segment selection: use the correct default every time

Instruction access	Default segment	Why
Fetch next instruction	CS	CS:IP selects code
`MOV AL,[BX]`	DS	BX selects data unless overridden
`MOV AX,[BP+04H]`	SS	BP means a stack-oriented memory reference
`PUSH AX`, `POP AX`	SS	Stack is addressed by SS:SP
`MOVSB` source	DS	Reads DS:SI
`MOVSB` destination	ES	Writes ES:DI

In memory operands containing BP, SS is the normal default segment even when an index register is also present: `[BP+SI]` uses SS unless a legal override explicitly changes it. The string destination segment for MOVSB is always ES on an original 8086; the source can be changed with a segment override.

5.6 Address worksheet: five worked problems, each with full units

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Problem A: direct displacement

Assume DS=2300H and instruction `MOV AL,[0050H]`. Because the addressing expression contains no base/index register, EA=0050H. Shift the data segment left by four bits: 23000H. Add the EA: **23050H**. This only identifies the address; the value placed in AL depends on that memory byte.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Problem B: BX+DI+displacement

Given DS=3000H, BX=0100H, DI=0020H and displacement 0018H: EA $= 0100\text{H} + 0020\text{H} + 0018\text{H} = 0138\text{H}$. Then physical $= 30000\text{H} + 0138\text{H} = \boxed{30138\text{H}}$. This is a legal based-indexed-displacement expression.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Problem C: BP+SI uses SS, not DS

Given BP=0FF0H, SI=0020H, SS=4000H and DS=2000H, the instruction `MOV AX,[BP+SI]` has EA=(0FF0H+0020H)=1010H. Default segment is SS, so physical byte addresses are 41010H and 41011H. Adding DS instead would wrongly point to 21010H.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Problem D: 16-bit effective-address wrap

Given BX=FFFFH, DI=0005H, DS=1000H and no displacement, the unbounded sum is 10004H. The 8086 EA is 16-bit, so EA=0004H, yielding physical 10004H. A full address calculation must account for both 16-bit EA wrap and 20-bit physical-address wrap as separate steps.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Problem E: aliasing and 20-bit physical wrap

Compare logical addresses 1234H:0020H and 1235H:0010H. Both give 12360H. Now compare FFFFH:0010H: the computed 21-bit value is 100000H, and the original 8086's 20 physical address lines select 00000H. Distinguish an alias of the same physical address from an out-of-range mathematical sum.

5.7 Find the byte versus find the value

A common numerical question stops at a calculated physical address. If the problem asks "what value appears in AL?" you also need the memory contents, plus the operand size and little-endian rules. For example, knowing DS=2000H and BX=0100H proves `MOV AL,[BX]` reads location 20100H, *not* whether that byte equals 7FH or FFH. The guide never invents contents of an unprovided memory cell.

5.8 Practice set with worked answers

1. DS=1111H, BX=0010H. Find the address read by `MOV AL,[BX]`. **Answer:** 11110H+0010H=11120H.
2. SS=3000H, BP=0010H, SI=0004H. Find `[BP+SI+02H]`.

   Answer: EA=0016H; physical=30016H.
3. Given BX=1234H and DI=0002H, is `[BX+DI]` legal? **Answer:** Yes; EA=1236H. The final physical address still needs a segment value.
4. Is `[SI+DI]` legal for the original 8086? **Answer:** No; it uses two index registers instead of one legal base/index combination.
5. With segment=FFFFH and offset=0020H, find the physical address. **Answer:** 100010H mod 100000H = 00010H.
6. If memory[20000H]=34H and memory[20001H]=12H, what word is read? **Answer:** 1234H (little-endian). Reading *only* the lower-address byte returns 34H.

Recommended exam-answer sequence. Write the instruction and the chosen segment; underline legal base/index terms; compute EA as a 16-bit offset; multiply segment by 10H; add EA and wrap to 20 bits if necessary; finally identify byte/word contents *only* when those bytes are given.

6 Instruction set: learn by behavior

IN SIMPLE WORDS

Every instruction has a job. Instead of memorizing an unstructured long list, group instructions by data flow, arithmetic, logic, control flow, stack and string processing. Write down the operand sizes and affected flags.

Category	Common 8086 mnemonics	What the group does
Transfer	MOV, LEA, XCHG, PUSH, POP	Copy values or addresses, exchange, save/restore stack values.
Arithmetic	ADD, ADC, SUB, SBB, INC, DEC, CMP, MUL, IMUL, DIV, IDIV	Compute and compare.
Logical/bit	AND, OR, XOR, NOT, TEST, SHL, SHR, SAR, ROL, ROR	Manipulate and test bits.
Control	JMP, CALL, RET, JZ, JNZ, JC, JNC, JAE, JB, LOOP	Alter execution flow using flags or counters.
String	MOVSB, MOVSW, CMPSB, SCASB, LODSB, STOSB, REP	Process arrays using SI, DI, CX and DF.
CPU / interrupts	INT, IRET, CLI, STI, CLD, STD, HLT, NOP	Interrupt processing and processor state control.

6.1 A few instructions that reveal the architecture

1. `MOV AX,BX` copies BX into AX without changing the status flags.
2. `ADD AX,BX` updates AX and the arithmetic status flags; the source BX is unchanged.
3. `CMP AL,BL` sets flags as though subtraction occurred but does not store the subtraction result.
4. `LEA SI,arr` computes the offset of arr, not the byte stored at arr.

5. `LOOP label` decrements CX and branches if CX is not zero, without changing arithmetic status flags.
6. `REP MOVSB` repeats byte copying from DS:SI to ES:DI while CX is nonzero. CLD/STD controls direction.

6.2 Signed versus unsigned branches

Comparison after CMP	Unsigned branch	Signed branch
a greater than b	JA (CF=0 and ZF=0)	JG (ZF=0 and SF=OF)
a greater/equal b	JAE (CF=0)	JGE (SF=OF)
a less than b	JB (CF=1)	JL (SF differs from OF)
a equal b	JE/JZ (ZF=1)	JE/JZ (ZF=1)

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Compare FFH and 01H

After `MOV AL,0FFH; CMP AL,01H`, the unsigned interpretation says 255>1, so JA is true. The signed interpretation says $-1 < +1$, so JL is true. They are different valid interpretations of the *same eight bits*.

6.3 Other instructions worth recognizing

`IN AL,DX` reads an I/O port number in DX into AL; `OUT DX,AL` writes AL to a port. PUSH/POP operate on 16-bit words. CALL/RET save and restore a return address. `INT n`/IRET transfer into and return from interrupt service. HLT stops instruction execution until a recognized event; it is not a DOS program-exit service.

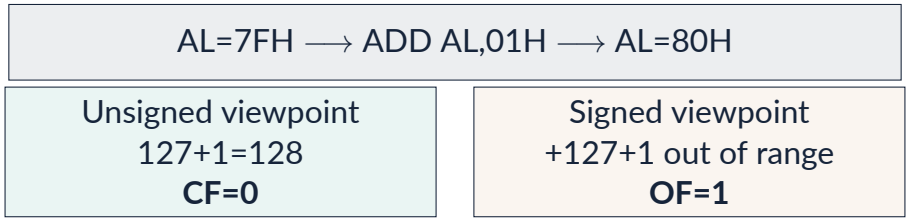
6.4 Instruction-set masterclass: learn the groups by changes in machine state

IN SIMPLE WORDS
Every instruction can be described with three questions: What data does it read? What destination does it change? What flags or control state does it affect? Once you can answer those, mnemonic memorization becomes much easier.

Group	Representative examples	What to pay attention to
Data transfer	MOV, XCHG, LEA, LDS, LES	Source versus destination; correct sizes; LEA loads an offset, not the memory contents.
Arithmetic	ADD, ADC, SUB, SBB, INC, DEC, CMP	Result size, CF versus OF, and whether a result is actually stored.
Logic and shifts	AND, OR, XOR, TEST, NOT, SHL, SHR, SAR, ROL, ROR	Bit masks, sign preservation in SAR, count syntax on the original 8086.
Program control	JMP, CALL, RET, Jcc, LOOP	Conditional flags, return address and branch target.
String and block	MOVSB, MOVSW, CMPSB, LODSB, STOSB, SCASB	DS:SI, ES:DI, CX, DF, and the appropriate REP-family prefix.
Interrupt and processor	INT, IRET, IN, OUT, CLI, STI, CLD, STD, HLT	Privilege/hardware assumptions and the interrupted register state.

6.5 Flags after ADD: compare two different number systems

Consider two eight-bit bytes: A=7FH and B=01H. The hardware adds the bit patterns and stores 80H. If A and B are *unsigned*, that means 127+1=128, which fits in 0..255; CF=0. If A and B represent signed two's-complement values, +127+1=+128 cannot fit in -128..+127; OF=1. It is the *same operation and the same stored bits*. Only the human interpretation and subsequent branch condition change.



The processor also updates other defined arithmetic flags. For an 8-bit result of 80H, ZF=0 and SF=1. CF reports carry beyond the chosen unsigned width; OF reports an invalid sign result for signed arithmetic. Mixing those tests makes maximum/minimum and range-check code wrong.

6.6 A second complete flag example: 80H + 80H

The byte sum is 100H, so the result stored in AL is 00H and CF=1. As signed values, -128 + -128 = -256, which is outside the signed-byte range; OF=1. ZF=1 because the stored result byte is zero, even though the *unbounded* mathematical sum is not zero. Notice why a zero flag alone is not enough to understand overflow.

Hardware distinction: The flags are set by the instruction and its operand width, not by whether a programmer intended the number to be signed. Signed and unsigned branches interpret those same flags differently.

6.7 CMP followed by the right jump: complete truth pattern

A comparison is essentially SUB for flags, but the destination operand is not overwritten. After `CMP AL, BL`, choose conditions according to the data model:

Meaning	Unsigned jump	Signed jump
Equal	JE (ZF=1)	JE (ZF=1)
Not equal	JNE (ZF=0)	JNE (ZF=0)
Left above/right	JA (CF=0, ZF=0)	JG (ZF=0, SF=OF)
Left at least right	JAE (CF=0)	JGE (SF=OF)
Left below right	JB (CF=1)	JL (SF≠OF)
Left at most right	JBE (CF=1 or ZF=1)	JLE (ZF=1 or SF≠OF)

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

FFH versus 01H with no guesswork

Let AL=FFH and BL=01H. After `CMP AL,BL`, the result bits from FFH-01H are FEH. On the *unsigned* scale, 255 is above 1, so JA succeeds. On the signed scale, FFH denotes -1, which is below +1, so JL succeeds. Neither jump changes the operands. This example is especially useful because it proves why one cannot substitute JG for JA.

6.8 ADD, ADC, SUB and SBB: how to handle an extra carry bit

`ADD` uses two operands. `ADC` adds a carry-in, making it ideal for high bytes or words of a multi-precision addition. `SUB` subtracts one operand. `SBB` subtracts a borrow-in. All of these need explicit width and input interpretation.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Add two unsigned 32-bit numbers using 16-bit arithmetic

Suppose the 32-bit numbers are split into low and high words: A=0001FFFFH, B=00000001H. Add the low words: FFFFH+0001H=0000H with CF=1. Then add the high words: 0001H+0000H+CF=0002H. The 32-bit result is 00020000H. In 8086-style code:

```
; A low=FFFFH, A high=0001H
; B low=0001H, B high=0000H
    MOV AX,0FFFFH
    ADD AX,0001H      ; AX=0000H, CF=1
    MOV DX,0001H      ; MOV preserves CF
    ADC DX,0000H      ; DX=0002H
; combined DX:AX = 00020000H
```

Why MOV DX is allowed between ADD and ADC. MOV does not change arithmetic flags. An unrelated ADD, CMP or TEST inserted between them would modify CF and destroy the carry chain.

6.9 Shifts and rotates: use an example with eight real bits

`SHL AL,1` moves all bits left and inserts zero into bit 0. `SHR AL,1` moves bits right and inserts zero into bit 7. `SAR AL,1` is arithmetic right shift: it replicates the original sign bit, preserving a negative two's-complement sign in common cases. Rotates (ROL/ROR) wrap bits around. For the original 8086, a variable

shift count uses CL; do not use arbitrary later-x86 immediate shift counts as if all were native 8086 encodings.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Why SAR differs from SHR

AL=F0H has bits 11110000. `SHR AL,1` gives 01111000=78H. Reset AL=F0H; `SAR AL,1` gives 11111000=F8H. If AL represents -16 signed, F8H represents -8: that is the intuitive arithmetic effect.

6.10 MUL and DIV: register assumptions matter

In an 8-bit unsigned `MUL r/m8`, the CPU multiplies AL by the operand and stores the 16-bit product in AX. In a 16-bit unsigned `MUL r/m16`, it multiplies AX by the operand and stores the 32-bit result across DX:AX. A byte `DIV r/m8` divides AX by the operand, placing quotient in AL and remainder in AH; a word `DIV r/m16` divides DX:AX by the operand, placing quotient in AX and remainder in DX. Division by zero or a quotient too large for its destination causes a divide-error interrupt, not a useful wrapped result.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

8-bit multiply, full product

Let AL=20H (=32) and BL=10H (=16). Execute `MUL BL`. Product is 512 decimal=0200H, so AX=0200H. Looking only at AL would incorrectly report 00H and lose the high byte. For `DIV`, always initialize the implicit dividend registers correctly before the instruction.

6.11 String instructions: what REP actually repeats

`MOVSB` copies one byte from DS:SI to ES:DI. REP repeats it CX times, changing the pointers according to DF. `MOVSW` copies words, advancing each pointer by two when DF=0. `CMPSB` compares source and destination memory bytes while moving pointers; conditional repetition uses REPE/REPNE when the instruction supports comparison termination. REP alone should not be treated as a magical “copy array” function without setting ES, DS, CX, SI, DI and direction.

Trace a three-byte forward copy. If DS:SI points to 10H,20H,30H; ES:DI points to a separate destination; CX=3; and DF=0: after `REP MOVSB`, destination has 10H,20H,30H; CX=0; SI and DI have each advanced by 3. If

DF=1, pointers move backwards and must be positioned at appropriate end addresses first.

7 Interrupts in the 8086

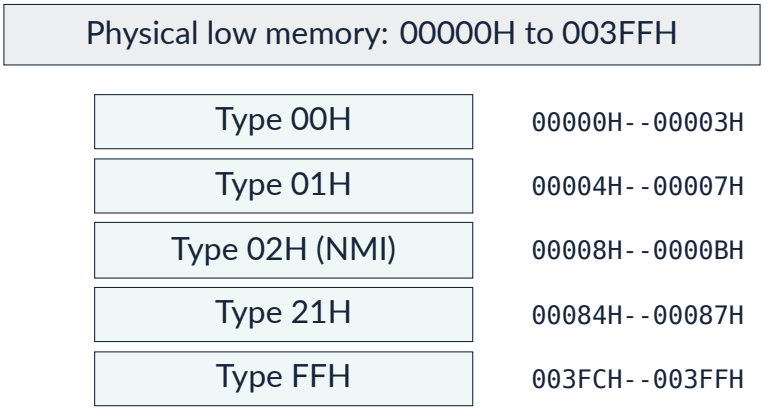
IN SIMPLE WORDS
An interrupt temporarily redirects the CPU to an interrupt-service routine (ISR). To return correctly, the processor must preserve enough of the interrupted execution state. The 8086 finds a new CS:IP through a vector table in low physical memory.

7.1 Interrupt types and triggers

Source	Mechanism	Important detail
External maskable	INTR pin, enabled by IF=1	External hardware supplies an interrupt type during acknowledge.
External non-maskable	NMI pin	Uses fixed type 2; not blocked by CLI/IF.
Software instruction	INT n	Instruction requests vector type n.
Processor-detected	Divide error, single-step, overflow (INTO)	Some conditions cause a specific interrupt type.

7.2 The interrupt vector table (IVT)

The **IVT occupies physical 00000H–003FFH**. It contains 256 entries, each exactly 4 bytes: two bytes for new IP (little-endian), then two bytes for new CS (little-endian). Entry n begins at $4n$.



WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Locate interrupt vector 21H

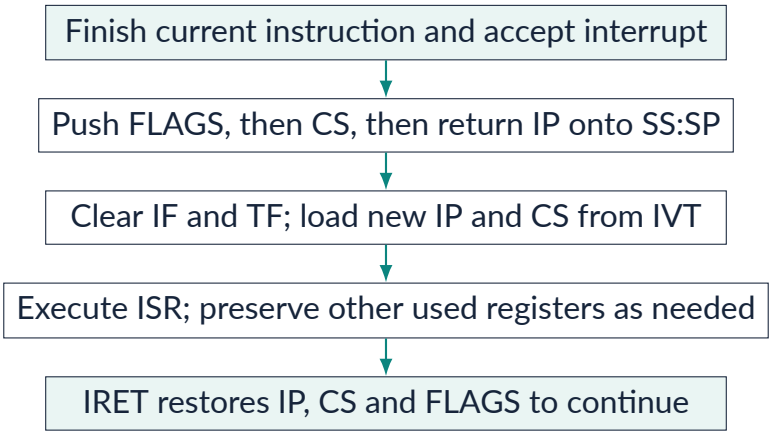
Type=21H (33 decimal). Start offset=$4 \times 33 = 132 = 84$H. Four bytes occupy **00084H–00087H**. Do not mistake these locations for the ISR code itself; they store the target pointer. In a DOS environment, INT 21H is assigned to DOS services; this assignment is *not* part of the CPU's intrinsic arithmetic.

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Locate NMI type 2

Type 02H starts at 4×2=00008H; the four-byte pointer ends at 0000BH.

7.3 What the CPU saves on an interrupt



INT, CALL and IRET are not interchangeable. CALL/RET handle a procedure return; an interrupt saves FLAGS as well as CS:IP and normally returns using IRET. A poorly written ISR can corrupt registers or the interrupted task.

7.4 INTR versus NMI

INTR depends on IF and uses an externally supplied vector type; NMI does not depend on IF and uses type 2. CLI prevents acceptance of maskable INTR requests, not NMI. Under standard 8086 interrupt entry both IF and TF are cleared, and IRET restores the saved FLAGS.

7.5 Interrupt masterclass: exactly what the vector holds

IN SIMPLE WORDS

An interrupt number is not a memory address and not the service program. It is an index into a table of four-byte pointers. Each pointer supplies a new offset (IP) and segment (CS).

For the original 8086, the interrupt vector table is fixed at physical addresses 00000H–003FFH. It contains 256 entries, numbered 00H–FFH. Each entry has four bytes stored in little-endian form:

4n	4n+1	4n+2	4n+3
IP low	**IP high**	**CS low**	**CS high**

To find type 21H: convert 21H to 33 decimal; multiply by four to obtain 132 decimal or 0084H. Thus bytes at 00084H and 00085H encode the new IP; bytes at 00086H and 00087H encode the new CS. These four bytes are the *destination pointer*. The ISR code itself can be elsewhere in memory.

7.6 Reconstruct an interrupt destination from four bytes

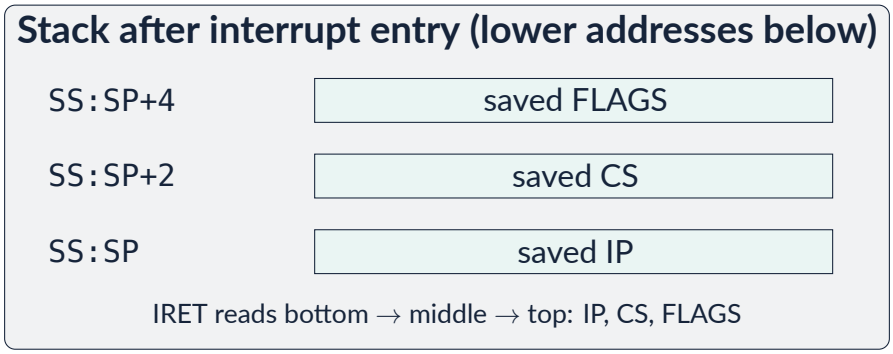
WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Full vector decoding exercise

Assume the physical bytes at 00084H, 00085H, 00086H and 00087H are 20H,01H,00H,F0H respectively. First read offset in little-endian: IP=0120H. Next read segment: CS=F000H. The CPU transfers to F000H:0120H, whose physical instruction location is F0000H+0120H=**F0120H**. The CPU does *not* execute the raw four table bytes as its interrupt handler.

7.7 A correct interrupt stack picture

When an interrupt is accepted, the 8086 saves enough state to return: FLAGS, CS and IP are pushed as words onto the descending stack. After completion of the hardware push sequence, SP points to the saved IP, with saved CS two bytes above and saved FLAGS another two bytes above. The handler returns using IRET, which pops IP, CS and FLAGS. The order of *pushing* and the reverse order of *popping* must not be confused.



WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Trace the stack pointer

If SP=0100H before the interrupt saves its three words, after saving FLAGS SP=00FEH, after saving CS SP=00FCH, and after saving IP SP=00FAH. The saved IP is the word at SS:00FAH, CS at SS:00FCH and FLAGS at SS:00FEH. IRET restores SP to 0100H after popping the three words, assuming the handler has balanced all its own pushes and pops.

7.8 Distinguish the four mechanisms

Source	How it is initiated	What to remember
Software INT n	Instruction executed by program	Uses explicit type n, and is not disabled merely by clearing IF.
External INTR	Level/request from suitable peripheral logic	Maskable using IF; interrupt type is supplied during acknowledge sequence.
External NMI	Dedicated nonmaskable input	Fixed type 2; clearing IF does not suppress it.
CPU-detected condition	Divide error / single step / INTO overflow case	Corresponding predefined type; distinguish from arbitrary user software services.

7.9 Service-routine design: what must be saved beyond CS:IP

The CPU automatically saves FLAGS, CS and IP. It does *not* automatically save every general-purpose register used by the interrupt handler. A handler that changes AX, BX, DS or another register needed by interrupted code must save and restore it according to its interface contract. For a simple pedagogical real-mode handler, the outline is:

```
my_isr PROC FAR
    PUSH AX
    PUSH DS
    ; establish intended DS if memory will be used
    ; service or acknowledge the actual device here
    ; avoid invented hardware register addresses
    POP DS
    POP AX
    IRET
my_isr ENDP
```

This is an *illustrative handler body*, not a standalone installed interrupt vector, not a complete DOS driver and not proof that a particular external device works. External hardware also needs an interrupt controller/acknowledge strategy suitable for the platform.

Serious ISR error. Using RET rather than IRET at the end of an interrupt leaves the FLAGS image on the stack and does not restore interrupt state correctly. Ending a handler without restoring registers can corrupt the interrupted program.

7.10 Interrupt practice, with explanations

1. Type 02H begins at which physical address? **Answer:** 00008H; four-byte vector occupies 00008H–0000BH. NMI uses this type.
2. Type 40H begins where? **Answer:** $4 \times 40\text{H} = 0100\text{H}$; bytes 00100H–00103H.
3. An IVT entry has bytes 78H,56H,34H,12H. What is the destination? **Answer:** 1234H:5678H, physical 179B8H.
4. Does CLI prevent a software `INT 21H`? **Answer:** No. CLI clears IF and thus blocks maskable external INTR, but an executed INT n still requests its software interrupt.
5. Does the 8086 automatically save AX during interrupt entry? **Answer:** No. A handler must preserve general-purpose registers it changes when required by its convention.

8 8086 programming conventions

IN SIMPLE WORDS
Assembly programs are easiest to verify when they name the assembler dialect, specify data sizes, initialize the data segment, state any operating-system assumption, and show the final memory result.

The following examples use **MASM/TASM-style real-mode 8086 assembly**. The shared wrapper below uses a DOS INT 21H / function 4CH exit *only for running as a DOS .EXE program*. It is not bare-metal firmware and should not be described as an 8086 hardware requirement.

```
.MODEL SMALL
.STACK 100H
.DATA
    ; declare inputs and result variables here
.CODE
```

```
main PROC
    MOV AX,@DATA
    MOV DS,AX
    ; program body here
    MOV AX,4C00H       ; DOS exit, if running on DOS
    INT 21H
main ENDP
END main
```

For each worked algorithm, paste the listed data declarations into the `.DATA` section and its body into the `main PROC` after DS initialization. Any example that uses string instructions also initializes ES. Stopping with NOP in a trace is not equivalent to returning to DOS. The checks accompanying this PDF validate logic arithmetically; no emulator claim is made.

8.1 Trace method

1. Write starting bytes and the declared numeric base (12 decimal versus 12H hex).
2. State all registers and memory locations initialized by the program.
3. For each loop, write the iteration, registers, flag-driven branch and memory effect.
4. State exactly what result size is stored, and what happens on overflow or empty input.
5. Check the result independently using ordinary arithmetic or a separate script.

9 Program 1: sum four bytes, keep the full 16-bit sum

Goal. Sum four unsigned bytes 12, 25, 38, 45 (decimal) into a 16-bit word, without silently discarding overflow when totals exceed 255.

```
; DATA declarations:
nums DB 12,25,38,45
sum  DW ?
; BODY after initializing DS:
    LEA SI,nums
    MOV CX,4
```

```
    XOR AX,AX
again:
    ADD AL,[SI]
    ADC AH,0          ; propagate carry into high byte
    INC SI
    LOOP again
    MOV sum,AX
```

Why ADC matters. ADD AL updates CF if the byte sum overflows. `ADC AH,0` increments the high byte when carry is set, preserving the full word result. This algorithm assumes a small enough number of bytes that the final 16-bit sum does not overflow.

Iteration	Input (decimal)	AX (decimal)	AX (hex)
1	12	12	000CH
2	25	37	0025H
3	38	75	004BH
4	45	120	0078H

Answer: The 16-bit result is AX=0078H (=120 decimal). In little-endian memory, `sum` stores 78H then 00H.

Challenge: for four inputs 250, 20, 10, 5 the correct total is 285=011DH; this listing retains it. An 8-bit-only accumulator would store 1DH and lose the high byte.

10 Program 2: maximum of five unsigned values

Goal. For `12H,63H,2FH,70H,4DH`, find the maximum using an unsigned conditional jump.

```
nums DB 12H,63H,2FH,70H,4DH
best DB ?
; BODY:
    LEA SI,nums
    MOV AL,[SI]       ; seed maximum with first byte
    INC SI
```

```
    MOV CX,4
scan:
    CMP AL,[SI]
    JAE keep          ; AL >= [SI] for unsigned data
    MOV AL,[SI]
keep:
    INC SI
    LOOP scan
    MOV best,AL
```

Dry run: 12H → 63H → 63H → 70H → 70H. CMP sets flags but does not overwrite AL. JAE uses CF=0 for unsigned comparison.

Answer: best=70H (112 decimal). The listing assumes a nonempty five-byte array; initializing from an empty array would be invalid.

Signedness: If the array contains signed two's-complement values, the unsigned JAE decision is not correct. A signed maximum uses the corresponding signed branch and a stated input interpretation.

11 Program 3: copy five bytes with REP MOVSB

Goal. Copy five consecutive bytes from src to a separate non-overlapping dst array.

```
src DB 10H,20H,30H,40H,50H
dst DB 5 DUP(?)
; BODY:
    MOV AX,@DATA
    MOV DS,AX
    MOV ES,AX          ; destination for MOVSB is ES:DI
    LEA SI,src
    LEA DI,dst
    MOV CX,5
    CLD                ; advance SI and DI
    REP MOVSB
```



Answer: After copying, destination bytes are 10H, 20H, 30H, 40H, 50H. CX=0, and SI and DI each advance by five.

Overlapping buffers: A simple forward copy may corrupt source bytes if destination overlaps a later source region. Choose a safe direction and CLD/STD based on the overlap pattern; in reusable code, restore DF as required by calling conventions.

12 Program 4: add two 16-bit words and retain carry

Goal. Add 1234H and 0112H, save the 16-bit result and separately save the carry flag as one byte.

```
a     DW 1234H
b     DW 0112H
sum   DW ?
carry DB ?
; BODY:
    MOV AX,a
    ADD AX,b
    MOV sum,AX          ; MOV does not change CF
    MOV carry,0         ; also does not change CF
    JNC no_carry        ; CF=0 means unsigned no carry
    MOV carry,1
no_carry:
    NOP
```

WORKED AND INDEPENDENTLY CHECKED EXAMPLE

Arithmetic and storage

1234H+0112H=1346H; 16-bit carry is zero. Memory at `sum` holds 46H at the lower address and 13H at the next address. `carry`=00H.

Answer: AX=1346H, carry=0. For FFFFH+0001H, AX=0000H and carry=1. The carry flag measures unsigned overflow; OF measures signed overflow.

13 Program 5: even/odd and TEST

Goal. Decide whether an unsigned integer stored in a byte is odd. The least significant bit is 1 for odd numbers.

```
value DB 37H
odd   DB ?
; BODY:
    MOV AL,value
    TEST AL,01H       ; compute AL AND 1 for flags only
    JZ even_num
    MOV odd,1
    JMP finished
even_num:
    MOV odd,0
finished:
    NOP
```

For 37H the binary form is 00110111, whose least significant bit is 1. ZF=0 after TEST, so execution follows the odd case. This is *not* the same as the CPU PF flag, which tests whether the low-order result byte has an even count of set bits.

Answer: odd=01H. Change the input to 06H and the result becomes 00H.

14 Program 6: count zero-valued bytes

Goal. Count zeros in a five-byte array without reading beyond its bounds.

```
nums  DB 00H,05H,00H,04H,00H
zeros DB ?
; BODY:
    LEA SI,nums
    MOV CX,5
    XOR BL,BL
check_next:
    CMP BYTE PTR [SI],0
    JNZ not_zero
    INC BL
not_zero:
    INC SI
    LOOP check_next
    MOV zeros,BL
```

Trace: zeros appear at positions 1, 3 and 5; BL finishes at 03H. The loop begins with CX=5 and executes exactly five times. `INC BL` does not affect CF, but does affect other status flags; here the JNZ decision is evaluated immediately after CMP, before INC BL.

Answer: `zeros`=03H. The result is a byte: arrays with over 255 zeros require a wider counter.

15 Program 7: signed and unsigned branching

Goal. Compare two byte values and see why the instruction is the same but the condition used afterward differs.

```
a DB 0FFH
b DB 01H
unsigned_above DB ?
signed_less    DB ?
; BODY:
    MOV AL,a
    CMP AL,b
    MOV unsigned_above,0  ; MOV preserves status flags
    JA yes_above          ; 255 > 1 (unsigned)
    JMP signed_test
yes_above:
    MOV unsigned_above,1
signed_test:
    CMP AL,b
    MOV signed_less,0
    JL yes_less           ; -1 < +1 (signed)
    JMP done
yes_less:
    MOV signed_less,1
done:
    NOP
```

Answer: `unsigned_above`=1 and `signed_less`=1. Same operands and CMP; different conditional interpretation.

15.1 Programming masterclass: read a program as a sequence of state changes

IN SIMPLE WORDS
A complete assembly answer consists of the algorithm, assembler assumptions, data declarations, actual instruction sequence, register trace, final stored bytes and the limitations. Simply writing mnemonics is not enough to prove understanding.

For the following MASM/TASM-style fragments, insert the data declarations under `.DATA` and the body after the data-segment initialization shown earlier. The symbols name storage locations; their numeric addresses are assigned by the assembler/linker. Program termination through DOS INT 21H is an operating-system service and must not be confused with the CPU's hardware behavior. **BEFORE WRITING ANY LOOP**

1. Decide the type: unsigned byte, signed byte, unsigned word, or signed word.
2. Decide the count: is it a fixed array, an externally supplied count or a sentinel-terminated list?
3. Pick the pointer: SI/DI are useful for arrays; BP is useful for stack-local operands.
4. Decide how to test a condition: CMP plus unsigned or signed jump; TEST for masked bits.
5. Check the result width: can a byte sum exceed FFH or a word sum exceed FFFFH?

15.2 Program 8: sum three unsigned 16-bit words into 32 bits

Learning objective: extend the carry-chain principle from the previous byte-sum program to words.

```
vals  DW 0FFFFH,0001H,0002H
outlo DW ?
outhi DW ?
; BODY after DS initialization:
    LEA SI,vals
    MOV CX,3
    XOR AX,AX          ; low 16 bits of sum
    XOR DX,DX          ; high 16 bits of sum
more_words:
```

```
    ADD AX,[SI]         ; add next 16-bit value
    ADC DX,0            ; propagate word carry
    ADD SI,2            ; next word is two bytes away
    LOOP more_words
    MOV outlo,AX
    MOV outhi,DX
```

Pass	Added word	DX:AX after pass	Decimal meaning
1	FFFFH	0000:FFFF	65535
2	0001H	0001:0000	65536
3	0002H	0001:0002	65538

Answer: The 32-bit result is DX:AX=00010002H (65538 decimal). In little-endian memory the low word is stored first by `outlo`; the high word is stored separately by `outhi`. The source array is unchanged.

Carry-placement rule. `ADC DX,0` must be after the low-word ADD and before another arithmetic instruction modifies CF. A 32-bit accumulator can also overflow if an unlimited number of words is added; the array length must match the chosen result width.

15.3 Program 9: maximum of four SIGNED bytes

Learning objective: compare a signed maximum with the unsigned maximum already taught.

```
values DB 0FFH,03H,0FEH,7FH  ; -1,3,-2,127
best   DB ?
; BODY after DS initialization:
    LEA SI,values
    MOV AL,[SI]
    INC SI
    MOV CX,3
signed_scan:
    CMP AL,[SI]
    JGE keep_signed    ; signed >=
    MOV AL,[SI]
keep_signed:
    INC SI
    LOOP signed_scan
```

```
    MOV best,AL
```

Trace. Start AL=FFH (-1). Compare to 03H (+3): -1 is smaller, update AL=03H. Compare to FEH (-2): +3 is larger, keep AL=03H. Compare to 7FH (+127): +3 is smaller, update AL=7FH. The final byte is 7FH, representing +127. If the same data were interpreted as unsigned, FFH (=255) would be the maximum.

Answer: Signed maximum = 7FH (+127). An unsigned maximum would be FFH (255). The assembly branch mnemonic is what distinguishes these interpretations.

15.4 Program 10: find a matching byte in a fixed array

Learning objective: combine CMP, a conditional branch, SI and CX, and explain the case when the value is absent.

```
values DB 11H,22H,33H,44H,55H
needle DB 33H
found  DB ?
; BODY after DS initialization:
    LEA SI,values
    MOV CX,5
    MOV found,0
search_next:
    MOV AL,[SI]
    CMP AL,needle
    JE match_here
    INC SI
    LOOP search_next
    JMP search_done
match_here:
    MOV found,1
search_done:
    NOP
```

Trace. First iteration compares 11H with 33H, not equal. Second compares 22H with 33H, not equal. Third compares 33H with 33H, ZF=1, so JE jumps to `match_here`, storing 1 in `found`. No byte after the matched position is read. When `needle`=77H, the loop checks all five entries and leaves found=0.

Address/result interpretation. This program returns only a found/not-found flag; it does not report an index. If the question asks for the position,

maintain a separate counter or compute SI relative to the array start with a clearly specified indexing convention.

15.5 Fault-finding exercise: identify five bugs

```
; INTENTION: max of FIVE unsigned bytes
    MOV AL,0
    LEA SI,values
    MOV CX,5
again_bad:
    CMP AL,[SI]
    JGE skip_bad       ; signed jump, data unsigned
    MOV AL,[SI]
skip_bad:
    INC SI
    LOOP again_bad
    MOV best,AL
```

What to correct. **(1)** If the task is unsigned, use an unsigned comparison such as JAE; **(2)** initializing AL to zero can work for *unsigned* bytes but cannot serve as a general signed-max seed; use the first element for a robust algorithm; **(3)** when seeding from the first element, advance SI once; **(4)** reduce CX to four remaining items; **(5)** verify the array contains five valid readable bytes and the `best` destination is declared as DB. Naming and initialization are part of correctness, not decoration.

15.6 Fully explained model solution: maximum array

1. **Data contract:** five unsigned bytes, at least one element, output one unsigned byte.
2. **Initialization:** load DS; point SI at the first byte; copy that byte into AL as the current maximum.
3. **Traversal:** move SI to the second byte and set CX=4, because one byte was consumed as the seed.
4. **Decision:** CMP AL,[SI]. If AL is at least the current input, JAE skips the update; otherwise load the larger value into AL.
5. **Advance:** INC SI, LOOP scan. LOOP decrements CX without replacing the arithmetic flags and stops at zero.
6. **Output:** store AL into the declared `best` byte and check it with an indepen-

dent maximum function.

Exam answer practice. Given 5 unsigned bytes 09H,8AH,12H,FFH,30H, the expected maximum is FFH. Show the progression 09H → 8AH → 8AH → FFH → FFH; explain the unsigned jump. For a signed array, FFH means -1 and the interpretation is different.

15.7 Complete hand-tracing worksheet for programming questions

Iteration	SI offset	CX before loop	AL / AX after work	Branch decision
Initial	start	length	start value	–
1	...	...	...	...
2	...	...	...	...
3	...	...	...	...
Final	end	0	stored result	exit

To validate a program independently, prepare at least three different test vectors: (a) normal mixed values; (b) an edge case such as FFH and carry; (c) repeated equal or zero values. Check both register results and the final memory bytes. If the code is tied to DOS or specific peripheral hardware, state that environment explicitly.

16 Solved numericals and exam examples

Problem	Derivation and answer
CS:IP = 4000H:0123H	40000H + 0123H = **40123H** physical.
DS:SI = 1A20H:0F30H	1A200H + 0F30H = **1B130H**.
BX=1000H, SI=0020H, disp=8	1000H + 20H + 8H = EA **1028H**.
DS=3000H, EA=1028H	30000H + 1028H = **31028H**.
Interrupt 0BH	4×0BH=2CH; IVT pointer starts at **0002CH**.
Interrupt 21H	4×21H=84H; IVT pointer spans **00084H–00087H**.
Word 1234H + 0112H	**1346H**, CF=0.
Byte FFH + 01H	**00H**, CF=1; ZF=1; OF=0.
Signed byte 7FH + 01H	**80H**, OF=1, CF=0.
Logical alias 1000:0020 / 1001:0010	Both physical addresses **10020H**.
Physical wrap FFFF:FFFF	10FFEFH low 20 bits = **0FFEFH**.

16.1 Correctly deriving an interrupt vector entry

A type number is an *index*. To find its four-byte pointer in the IVT, compute $4n$, read IP from two bytes and CS from the next two bytes (little-endian in each word). A vector for type 21H at 00084H is *not* automatically a procedure at 00084H.

16.2 Correctly interpreting an assembly result

When the code says `DB 25`, 25 is decimal in MASM unless another radix is specified. `DB 25H` is hexadecimal 25H (=37 decimal). When comparing or adding two bytes, record whether you interpret bit 7 as a sign and distinguish CF from OF. In an array algorithm, document input length, allowed overlap and whether the sum can exceed its storage width.

17 Worked addressing-mode drill

These problems combine the two calculations students often confuse: **effective address** (a 16-bit offset) and **physical address** (a 20-bit memory location). Use the actual default segment and assume no segment-override prefix.

17.1 Drill A: register-indirect operand

Given BX=2010H and DS=2100H, determine the byte read by `MOV AL,[BX]`. The effective address is 2010H. Its physical address is $21000\text{H} + 2010\text{H} = \boxed{23010\text{H}}$. This determines *where* the byte comes from, not its value. You cannot infer AL without knowing memory contents.

17.2 Drill B: BP-based operand

Given BP=0100H, SS=3000H and DS=2000H, find the physical address for `MOV AX,[BP+04H]`. EA=0104H. Because BP chooses SS by default, the physical location is **30104H**, *not* 20104H. Word data spans physical 30104H and 30105H.

17.3 Drill C: two index-related address terms

Given BX=0100H, DI=0200H, displacement=0030H and DS=2300H, find the location for `MOV AL,[BX+DI+30H]`. EA=0100H+0200H+0030H=**0330H**. Physical address=23000H+0330H=**23330H**.

17.4 Drill D: 16-bit effective-address wrap

Assume BX=FFFFH and SI=0002H for `MOV AL,[BX+SI]`. The computed EA is 10001H before retaining 16 bits, so EA=**0001H**. With DS=4000H, physical=**40001H**. The address calculation is modular; this is different from signed arithmetic overflow.

17.5 Drill E: distinguish code address and data address

Take CS=1000H, IP=0020H, DS=2200H, BX=0008H. The instruction fetch occurs at physical **10020H**; an ordinary `MOV AL,[BX]` data access occurs at **22008H**. BIU uses different segment registers depending on the requested access.

Solve systematically: identify opcode access versus data operand, choose CS/DS/SS/ES, compute EA from legal registers, then add the shifted segment. Do not silently assume an unknown memory value.

18 Instruction behavior: five important micro-cases

18.1 Case 1: CMP changes flags, not data

Suppose AL=10H and BL=20H. `CMP AL,BL` acts as a subtraction for flag calculation. Since 10H is below 20H unsigned, CF=1; AL remains 10H. A subsequent JB branches, while a subsequent JA does not.

18.2 Case 2: INC is not ADD for carry

If AL=FFH and CF was previously 1, `INC AL` produces AL=00H and updates several flags but leaves CF at its old value (1). By contrast, `ADD AL,1` computes CF from the addition and produces 1 in this case as well, but if CF had previously been 0, INC would leave it 0 whereas ADD would set it 1. This distinction matters when preserving a carry chain.

18.3 Case 3: a tiny arithmetic chain

Assume AL=F0H, then `ADD AL,30H` gives AL=20H, CF=1. Immediately performing `ADC AH,0` increments AH by 1. The two-byte pair AX therefore represents the full unsigned sum, provided the previous AH contents were the intended high part. This is the central technique in Program 1.

18.4 Case 4: direction flag in string operations

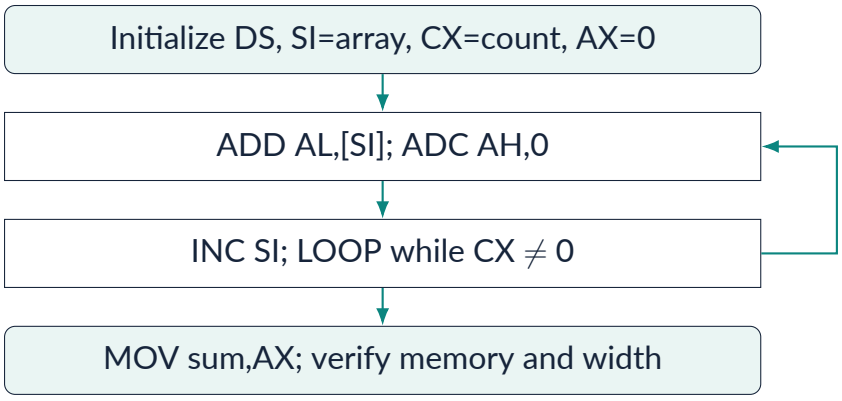
With DF=0 (CLD), MOVSB copies a byte and increments SI and DI. With DF=1 (STD), it decrements SI and DI. It does not itself decide whether forward or backward is safe for overlapping arrays; the programmer must choose the initial pointers and direction.

18.5 Case 5: a conditional branch reads flags, not a numeric type

The CPU does not attach a signed/unsigned label to 8-bit data in AL. After CMP, signed jumps interpret SF and OF, while unsigned jumps interpret CF and ZF. If you switch only the jump mnemonic, the same arithmetic may produce a different valid ordering.

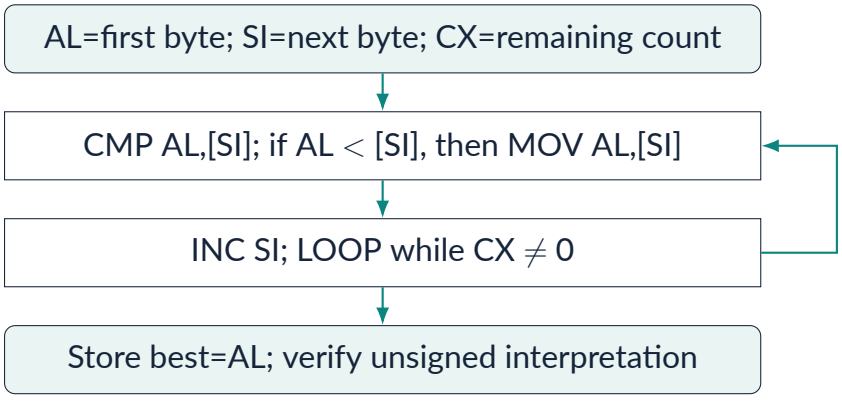
19 Algorithm visualizations and dry-run checkpoints

19.1 Summing a byte array



The loop uses two separate arithmetic instructions to preserve byte carry: `ADD AL,[SI]` updates CF, then `ADC AH,0` consumes that carry. Because INC SI and LOOP occur afterward, later changes to other flags do not interfere with the carry transfer.

19.2 Computing an unsigned array maximum



19.3 How to check a REP MOVSB trace

Prepare two non-overlapping arrays and initialize DS, ES, SI, DI, CX and DF. After REP MOVSB, verify all transferred bytes, CX=0, and SI/DI movement by exactly the count. If copying backwards with STD, the initial pointers must point at the ends of the areas. Do not copy a live file's memory without checking overlap and size.

20 Source-trace cases for robotics

The following are *educational examples of how 8086 instructions could form part of a control interface*. They are not complete hardware schematics, actual motor-driving code, or claims about a specific robot's firmware.

20.1 Encoder sample stored in RAM

An external interface can place an 8-bit count at a memory-mapped register. An 8086 program could load the byte into AL and compare it against a threshold. The address must be decoded by peripheral hardware and must not be invented from the processor's instruction-set table. Without known hardware addresses, write an algorithm rather than a fabricated working board program.

20.2 A loop over sensor values

If a sample buffer holds five unsigned bytes, the maximum program from this guide can calculate a peak reading. For a signed temperature deviation array containing negative two's-complement bytes, change the comparison rule and document the signed interpretation; otherwise the maximum may be wrong.

20.3 An asynchronous interrupt source

A sensor interface may assert INTR through suitable external logic. The 8086 saves execution state and uses an interrupt vector supplied during acknowledgment. The service routine must preserve registers it changes, read or acknowledge the device according to its actual hardware design and return through IRET. Do not confuse a hypothetical external vector with a DOS INT 21H service.

21 Extended original practice with answer checks

1. **Memory size.** Explain why 2^{20} physical addresses means 1 MiB, but a fixed 16-bit offset spans 64 KiB.
2. **Registers.** Identify the register used by LOOP, the default stack segment, and the extra segment in MOVSB destination.
3. **Register halves.** If AX=1234H, state AH and AL, then the value after MOV AL,56H.
4. **Segmentation.** Find physical addresses for 2000H:0100H, 2001H:00F0H, and decide whether they alias.
5. **EA modes.** With BP=0400H, SI=0030H, displacement=10H, SS=5000H, compute EA and physical address for [BP+SI+10H].
6. **Bank alignment.** State the two memory byte addresses occupied by word A2B3H stored at 1001H.
7. **Flags.** Compare the carry and signed overflow outcome of 80H + 80H in an 8-bit destination.
8. **Interrupts.** What is the byte address of the IVT pointer for type 40H? List the four locations it occupies.
9. **Control flow.** Explain why JAE is not generally a replacement for JGE.
10. **Program.** Modify Program 2 to scan seven unsigned values and justify changes to loop count.

Selected answer key (2) CX, SS, ES:DI. (3) AH=12H, AL=34H; after replacing AL, AX=1256H. (4) 20100H and 20100H: the two segment:offset pairs alias. (5) EA=0440H and physical address=50440H (BP default SS). (6) B3H at 1001H; A2H at 1002H; odd-addressed word transfer needs two bus cycles. (7) 80H+80H=00H with CF=1 and OF=1. (8) $4 \times 40\text{H} = 100\text{H}$; vector bytes 00100H–00103H. (9) JAE uses unsigned carry logic; JGE tests signed SF/OF relation. (10) seed first element, set CX to six remaining, and retain input-size checks.

22 Exam-ready answer frameworks

Q1. Explain the functional architecture of the 8086. Define 16-bit CPU and 20-bit address bus. Draw the labelled BIU/EU diagram. Describe segment registers, IP, queue and address generation in BIU; describe ALU, general registers, FLAGS and decoder in EU. Explain fetch/execute overlap and queue flush on a taken control transfer. Add the 1 MiB memory-size calculation.

Q2. Explain 8086 addressing modes with examples. Define operand and effective address. Give immediate, register, direct, register-indirect, based, indexed and based-indexed with displacement examples. Calculate EA and physical address for one example; note BP uses SS and BX/SI normally DS. Explain an illegal AX-based addressing form.

Q3. Explain the instruction set and its classifications. Group MOV/LEA, ADD/ADC/SUB/CMP, TEST/AND/OR/XOR, JMP/Jcc/LOOP, PUSH/POP/CALL/RET and MOVSB/REP. Explain flags and give signed versus unsigned comparison examples. State important operand-size restrictions.

Q4. Explain interrupts and the interrupt vector table in 8086. Define ISR. Distinguish INTR, NMI, software INT. Draw 256-by-4-byte IVT. Explain push FLAGS/CS/IP, clear IF/TF, load vector and IRET. Calculate a vector such as type 21H: pointer starts at 84H.

Q5. Write and explain an 8086 program. State assembler/dialect and input/output size. Initialize DS. Write a correctly counted loop and comment the flag-tested branch. Show sample input, a dry-run table and the final memory result. Discuss unsigned/signed interpretation and overflow behavior.

Paper-pattern caution. These are original examination-practice answer frameworks, not verified past-paper questions or a guaranteed 6+7 / 7+7 split. The official course states ESE 70 marks across five units.

23 Revision sheet and self-test

ONE-PAGE MEMORY MAP

Remember	Why
16-bit data / 20-bit address	1 MiB physical address space.
BIU = fetch, EU = execute	Six-byte instruction prefetch queue.
CS, DS, SS, ES	Code, data, stack and extra segment bases.
Physical = 16S+O	Keep low 20 bits for original 8086.
EA = base + index + disp	Only permitted 8086 base/index combinations.
CF versus OF	Unsigned carry versus signed overflow.
IVT = 00000H–003FFH	256 pointers of 4 bytes each; type n at 4n.
INT / IRET	Save and restore FLAGS, CS and IP.
REP MOVSB	Source DS:SI, destination ES:DI; CLD/STD direction.
JAE versus JGE	Unsigned versus signed comparison.

23.1 Self-test: solve without looking

1. Explain BIU and EU with a neat labelled diagram.
2. Why can a 16-bit processor address 1 MiB? Show the numerical step.
3. Draw the register groups, flags and segment registers of the 8086.
4. Calculate physical addresses for 1234H:5678H and FFFFH:FFFFH.
5. Differentiate byte and word memory access at even and odd addresses.
6. Given BX=2000H, SI=0010H, displacement=0018H and DS=3000H, find EA and physical address.
7. Why is `MOV AX,[AX]` invalid on the original 8086?
8. Explain the difference between CF and OF with examples.
9. What happens on INT 21H and how is its vector located?
10. Trace a signed comparison between FFH and 01H.

11. Write an array maximum program and explain its unsigned branch.
12. Write a byte-copy program with REP MOVSB and state source/destination segments.

23.2 Self-test answer checkpoints

(2) $2^{20} = 1,048,576$ bytes; (4) 179B8H and 0FFEFH; (6) EA=2028H and physical=32028H; (7) AX is not an allowed 8086 EA base; (8) CF unsigned carry, OF signed overflow; (9) 4×21H=84H, push state then load vector; (10) as unsigned 255>1, as signed $-1 < 1$. Other questions require diagrams or program listings, not a one-word answer.

24 Sources, provenance and publication checklist

Official syllabus. Savitribai Phule Pune University, *Third Year Robotics & Automation Engineering (2024 Pattern)*, course PCC-302-ROA, Unit II, printed syllabus page 16 in the supplied 2026–27 TE syllabus PDF. This is the authoritative list of the five required topics. Public Drive source previously reported: https://drive.google.com/file/d/1filXvrqbGbqBjZ-QXVcqEdUXWVMcktck/view.

Technical primary reference. Intel Corporation, *The 8086 Family User's Manual*, October 1979, chapters covering CPU organization, instructions, interrupts, memory addressing and hardware interfacing. A third-party university-hosted scan of the original publication is at https://www.inf.pucrs.br/~calazans/undergrad/orgcomp_EC/mat_microproc/intel-8086_family_Users_Manual.pdf. The technical reference is a separate source, not an SPPU publication. No unverified exact manual page numbers are asserted here.

Earlier Zenith material. Existing 16-page draft the *Zenith MM Unit II 8086 Study Guide Review Draft* PDF (text and historical QA report available). It was a student-prepared review draft with reported 74/74 static checks but not assembler/emulator-tested. This edition integrates deeper subject workshops, new original diagrams, program variants and review exercises; it does not claim third-party peer review.

Diagram and web-source policy. All diagrams contained in this edition are

newly drawn educational schematics, cross-checked conceptually against the Intel 8086 architecture reference. Wikimedia Commons hosts a CC0 8086 pinout drawing by Lamfe (2017; https://commons.wikimedia.org/wiki/File:Intel_8086_pinout.svg), and a CC BY-SA 4.0 Intel D8086 chip photograph by Thomas Nguyen (2016; https://commons.wikimedia.org/wiki/File:Intel_D8086_CS.jpg); their source and license pages were reviewed but their original bytes could not be downloaded in this execution environment, so these external assets are NOT embedded or falsely attributed.

PUBLICATION GATE – NOT YET LIVE Before any Drive/publication action: verify full topic coverage, execute a suitable assembler/emulator if available, review every page render, verify input/output traces, review copyright/attribution, capture PDF SHA-256, and update the Notes manifest/index. Keep v35 untouched and preserve its v34 rollback.

Before you rely on an assembly result

A worked dry run shows the intended algorithm, but an executable test requires the correct 8086 assembler syntax, an initialized stack and data segment, and an observable final memory result. For every listing, the reviewer should record:

1. The chosen assembler and the byte values assembled from source.
2. Entry-point assumptions, DS/SS initialization, input array length and address bounds.
3. Register and flag values after each branch or arithmetic instruction.
4. Expected versus observed output bytes, including carry or signed-overflow cases.
5. A deliberately difficult test: empty inputs where permitted, maximal bytes, odd addresses, and signed-versus-unsigned comparisons.

Example verification decision. A routine calculating the maximum unsigned byte can be checked with `12H, 63H, 24H, FFH, 38H`; expected output is FFH. For a signed maximum, the same byte FFH means -1, not 255. Specify which interpretation you intend before selecting the conditional jump. These are independent teaching test cases, not claims of emulator execution.