تقدم لجنة ElCoM الاكاديمية

ملاحظات لمادة:

لغة تجميع و معالجات دقيقة

من شرح:

د.عوني طرادات

جزيل الشكر للطالبة:

نتالي الكايد

2.1 Microarchitecture of the 8088/8086 Microprocessor

يعني لما تكون (BIU) مشغولة بال preperation فتكون (EU) مشغولة بتنفيذ التعليمات و المعالجة [الثنين بشتغلوا مع بعض]

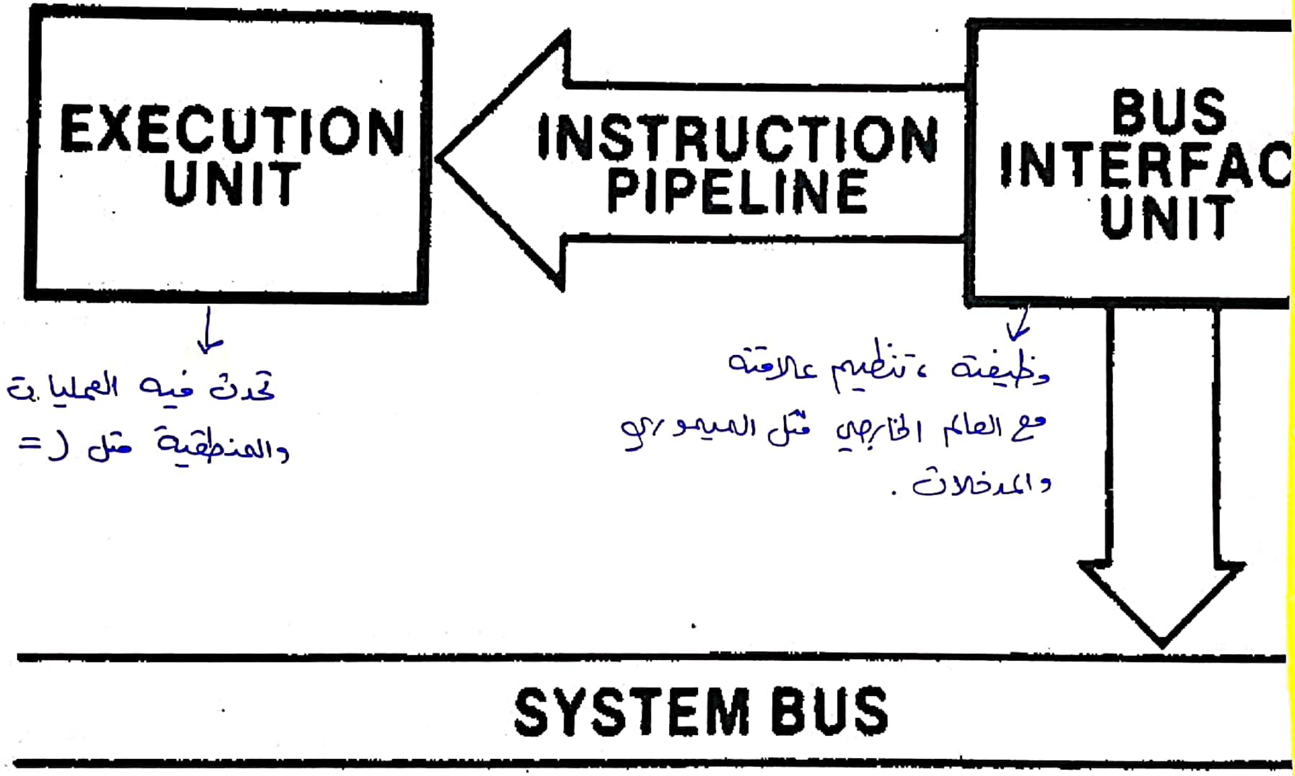
- 8088/8086 both employ ***parallel processing*** → يعني بوظف شغلتين مع بعض ، بقدر مثلاً يقرأ ويخزن مع بعض .
- 8088/8086 contain two processing unit – the ① ***bus interface unit* (BIU) and *execution unit*** ② ***(EU)***
- The bus interface unit is the path that 8088/8086 connects to external devices. → وظيفته تنظيم علاقته مع العالم الخارجي مثل الميموري ومدخلات الجهاز.
- The system bus includes an 8-bit (1 byte) bidirectional ① data bus for 8088 (16 bits for (2 byte) the 8086), a 20-bit ② address bus, and the signal needed to ③ control transfers over the bus.

→ system bus includes:
(1) address bus → width=20
(2) data bus → width=8 bit
(3) control bus →

8-bite → 1byte → يعني خلال دورة بقدر يحمل بس (1 byte)

16 - bite → 2 byte → يعني خلال دورة بقدر يحمل بس (2 byte)

2.1 Microarchitecture of the 8088/8086 Microprocessor



Pipeline architecture of the 8086/8088 microprocess

تحدث فيه العمليات الرياضية
والمنطقية مثل (= + - < > --)

وظيفته ، تنظيم علاقته
مع العالم الخارجي مثل الميموري
والمدخلات .

* يوجد داخل البروسيسر . مجموعة من
ال (Registers) ، وظيفتها تخزين المدخلات
والمخرجات بشكل مؤقت .

لذلك الريجستر عبارة عن
Small internal storage
تستقبل (2input) وتنتج (1 output) .

* يتكون الريجستر من مجموعة D-f.f
بحيث $Q(t+1) = D(t)$.
فهو يحتفظ بالداتا لمدة سايكل واحدة .

* لو عندي محل رقمه 14 موجود
بشارع ، بدي اشتري منه أغراض

بحيث أن رقم المحل بهاد الشارع ← address
والأغراض تعبر عن ← data
وعملية الشراء ← control

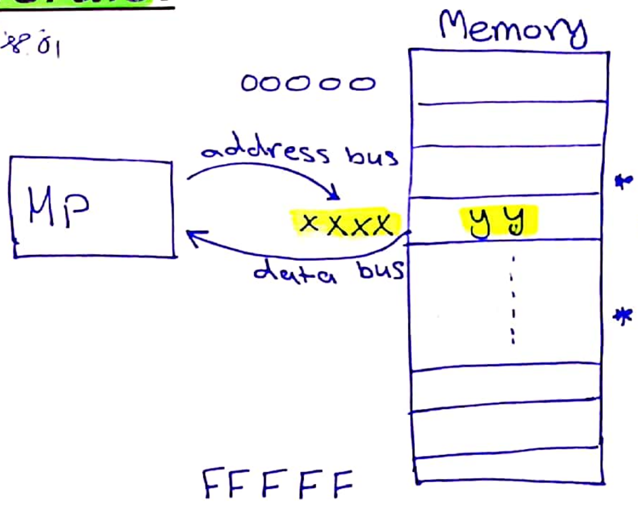
هكذا تعمل داخل البروسيسر ، بحيث
أن (address bus) يحدد الموقع
ليقوم (data bus) بأخذ الداتا
من الموقع ، ثم (control bus) يحدد
طبيعة العلاقة إمّا بالقراءة أو التخزين.

2.1 Microarchitecture of the 8088/8086 Microprocessor

- **BIU** is responsible for: (1) *Instruction fetching* جلب, (2) *memory reading/writing* and *inputting/* (3) *outputting data for peripherals.* الأجهزة الطرفية.
- **Components in BIU**
  - Segment register
  - The instruction pointer
  - Address generation adder
  - Bus control logic
  - Instruction queue
- **Components in EU**
  - Arithmetic logic unit, (ALU)
  - Status and control flags
  - General-purpose registers
  - Temporary-operand registers

الميموري عبارة عن list of locations

كل لوكيشن له address.



yy → data

XXXX → address

* address → address of the location (ID)

* dat → contant of the location

البروسيسر بده تجيب داتا من الميموري ، على سبيل المثال بدنا نجيب داتا من العنوان (XXXX) ، (address bus) تحدد العنوان على الميموري.

ثم بواسطة (data bus) تنتقل الداتا من الميموري إلى البروسيسر ، والداتا هي (yy).

ووظيفة (control bus) إنه تحدد العلاقة، وفي هذا المثال هي read.

2.1 Microarchitecture of the 8088/8086 Microprocessor

بما انه address طوله 20bit

يعني $1M = 1K * 1K = 2^{10} * 2^{10}$

وكمان 20 bit = 5 Hexa

عشان هيك بنلاحظ انه عناوين الميموري بدأت من (00000) وانتهت بـ (FFFFF). ⟵ هي نفسها عشرين صفر إلى عشرين واحد لكن حولناها.

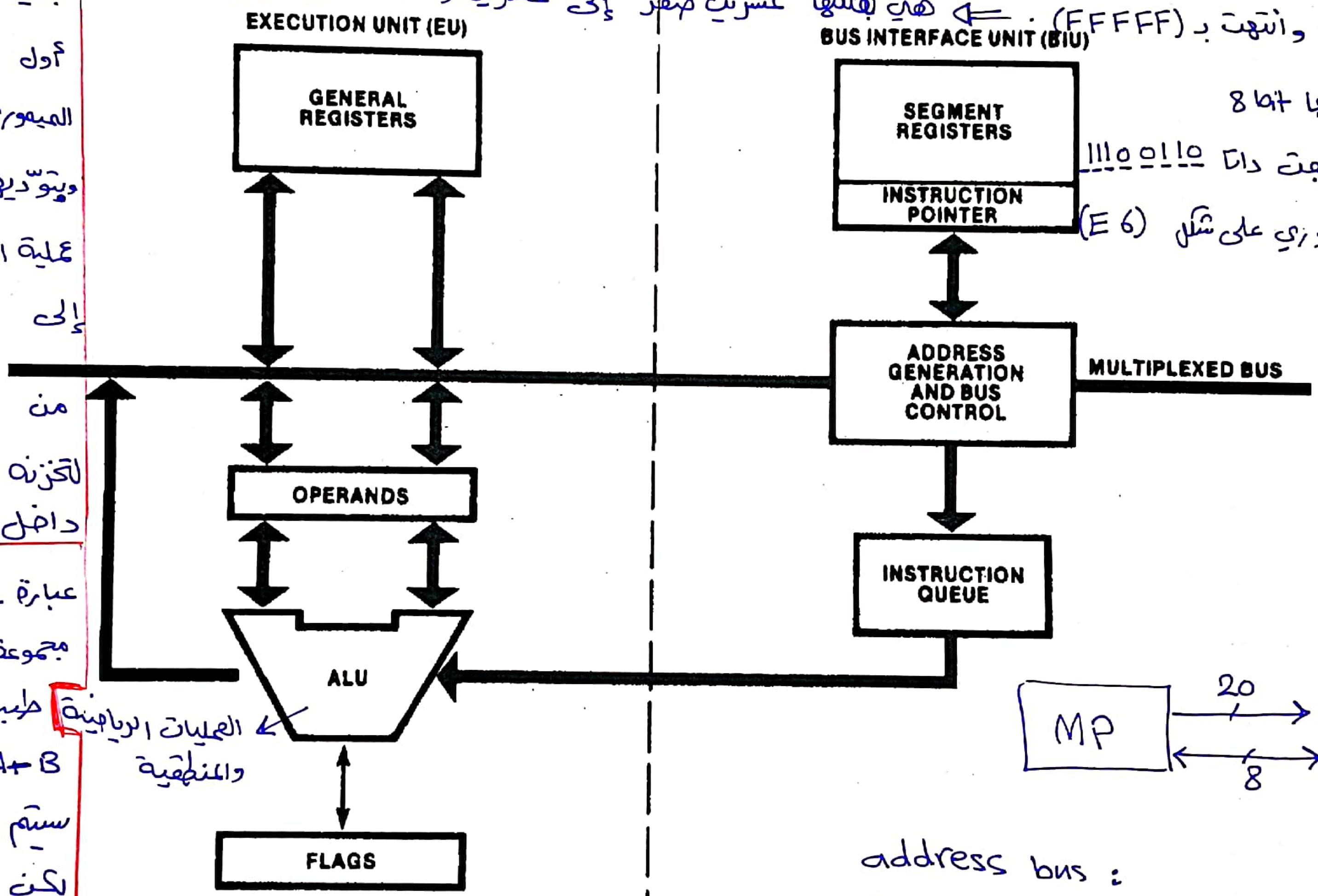
أما بالنسبة للداتا فطولها 8 bit

يعني لو البروسيسر بعث داتا 11100110 رح تتخزن داخل الميموري على شكل (E 6)

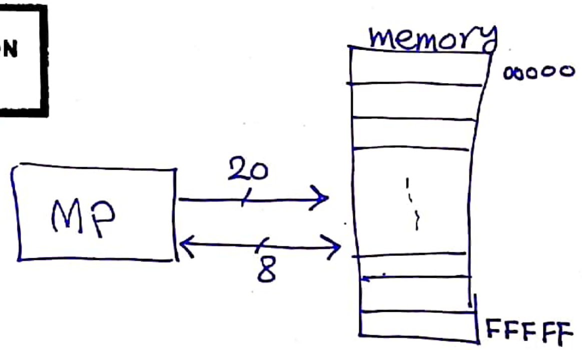
بدي أنفذ هاي العملية A*B=C أول اشي بتروح BIU على الميموري وبتحضر A* B =C وبتوديها على ALU فتقوم بتفعيل عملية الضرب (*) ، ثم تنتقل إلى (general registers) لتحضر كل من A و B وتحسب الناتج لتخزنه داخل C ، وتخزن الجواب داخل أحد الريجستر.

---

flag : عبارة عن ريجستر بحتوي مجموعة من الـ(bits) وظيفته تحدد طبيعة الناتج ، على سبيل المثال A+B ، كان الناتج يساوي -6 سيتم تخزين الرقم (-6) بأحد الريجستر لكن الـ(flag) سيمثل الاشارة السالبة فيعمل set للرقم 1 داخله وكما نعرف أن (1) تعني إشارة سالبة.



EU and BIU of the 8086/8088 microprocessors



address bus : يكون باتجاه الميموري

data bus : يكون باتجاهين لأنه معه خيارين هما القراءة من الميموري أو الكتابة عليها

2.2 Software Model of the 8088/8086 Microprocessor

* طبيعة الريجسترات الموجودة في BIU إنهم بخزنوا (address)

وطبيعة الريجسترات الموجودة في EU إنهم بخزنوا داتا .

- Software model describes: available registers, memory address space and I/O address space.
- 8088 microprocessor includes 13 16-bit internal registers.
  - The instruction pointer, → (BIU) IP (تخزن فيه address)
  - Four data registers, → AX, BX, CX, DX (تخزنوا داتا)
  - Two pointer register, → (EU) [BP, SP] → تخزنوا address.
  - Two index register, → SI, DI] → تخزنوا address.
  - Four segment registers, → BIU [CS, DS, SS, ES] → تخزنوا address.
- The status register, SR, with nine of its bits implemented for status and control flags.
- The memory address space is 1 Mbytes and the I/O address space is 64 Kbytes in length.

خاصّ بال codes

* الريجسترز نوعين :- (1) address reg
(2) data reg

* كل الريجسترز بخزنوا address ما عدا Ax, bx, cx, DX

موزعين منهم موجود بـ (BIU) ومنهم بـ (EU) .

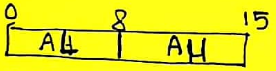
1 Mbytes = 64 K bytes

* لما نتعامل (address reg) ، كلهم بخزنوا (16 bit) / أما لما نتعامل مع (data reg) ، ممكن تخزنوا 8 bit أو 16 bit لأنه بنعرف إنه الداتا حجمهم 8bit فقط .

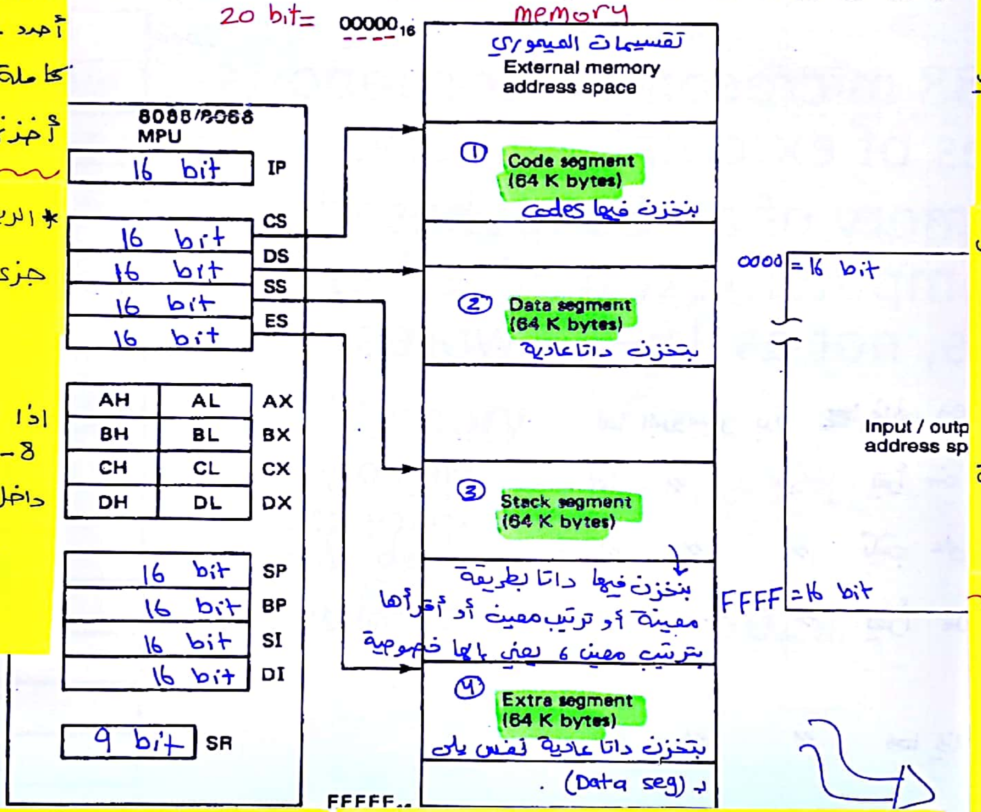
2.2 Software Model of the
/8086 Microprocessor

لما أحدد address لازم استعمل
(16 bit) كاملين ، لأنه منطقياً لما
أحدد عنوان فكان لازم أعطي كل التفاصيل
كاملة ، لكن الداتا عادي تقدر
أخزنها بـ (low part) حسب الحاجة .

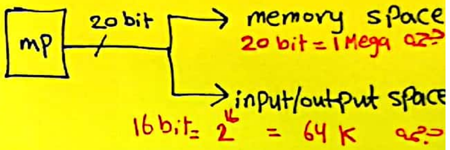
* الريجستر 16 bit يمكن قسمه لجزأين
جزء high وجزء low



اذا خزنت داتا خلال ال bits من
0-8 يعني AL وازا خزنت
داخل ال bits من 9-15 يعني AH



البروسيسر بتعامل مع عالمه الخارجي
مع شغلتين :



البروسيسر عندما يريد التعامل مع العالم الخارجي
بيستعمل (20 bit) مرة بيستخدمها للميموري
ومرات لل input/out ، حسب حاجته

بس مستحيل يستعمل ال(20 bit) لشغل
مع الشغلتين input/out و memory .

الميموري بتحتاج 20 bit كاملات لحتى
تتواصل مع البروسيسر ، لكن input/out
بحتاج فقط (16 bit) من ال (20 bit) الخاصات
بال (address bus) .

التقسيمات يلي بالميموري أربعة
حجم كل منها 64k ⟸ $64 * 4 = \boxed{256\ K}$
أقل من (1 Mega) .

فبنعمل segmentation ، مثلاً بنعمل
4 segement لكن مش كلهم فعّالين .

طيب كيف نحدد ال segment ؟
يعني لما البروسيسر بدو يتعامل مع واحد منهم كيف
بدو يحدده ؟

البروسيسر لازم يعرف طبيعة ال(seg) يلي بده يروح
يروح عليه ، مثلاً نعلّم كل (seg) بلون معين .

لذلك كل واحد من ال seg يدل عليه واحد من
الريجستر ، انظر إلى الأسهم على الرسمة

CS ⟶ code seg
DS ⟶ Data seg
SS ⟶ stack seg
ES ⟶ extra seg

طيب كل رجستر 16 واحنا حكينا انه مشان تتواصل مع الميموري
بدنا 20 ، طيب شو نعمل ؟؟

مشان نحل المشكلة بنستعمل رجسترين مثلاً offset و CS
كل واحد منهم 16 bit
$16+16=32$ ⟸ base : offset (16 bit, 16 bit) بنسميه logical address
بنمجهم مع بعض مشان يعطوا 20 bit ويصير اسمه
(Phsiycal address) .

من وين بنجيب offset ؟ عادةً بكونوا ip/sp/BP/si/di
مين همة ال base ؟ CS / DS / SS / ES

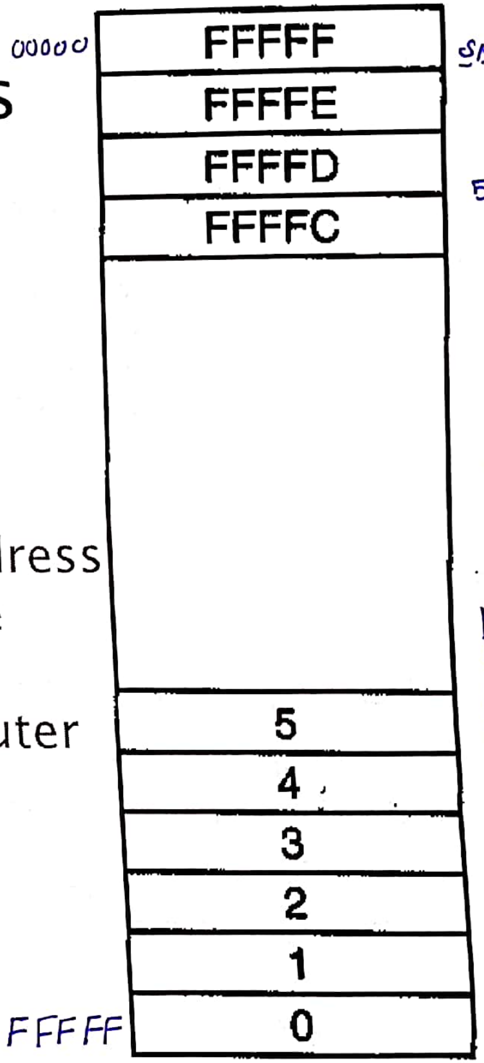
2.3 Memory Address Space and Data Organization

الميموري كاملة = 1 Mega
تحوي عدة lines كل واحد منهم = 1 byte = 8 bit في 8088

- The 8088 microcomputer supports 1 Mbytes of external memory.
- The memory of an 8088-based microcomputer is organized as 8-bit bytes, not as 16-bit words.

Base : offset

CS : ip لما البروسيسر بده يخاطب code seg بستخدم
DS : Si لما " " يقرأ من Data seg "
DS : Di " " " يكتب على " " "
SS : SP, أو Bp " " " يقرأ من stack seg "
ES : Si أو Di " " " يخاطب extra seg "

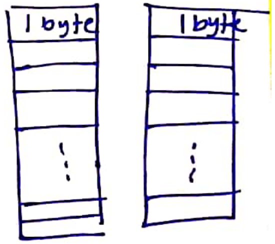


Memory address space of the 8088/8086 Microcomputer

قاموا بقسم الميموري من النص ليصبح كل جزء = 512 K

لأنه data bus بـ 8086 حجمها 16 bit.

فبقدر يقرأ 2byte خلال سايكل واحدة



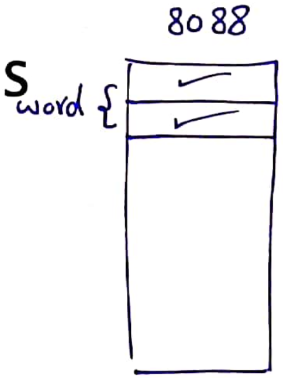
2.3 Memory Address Space and Data Organization

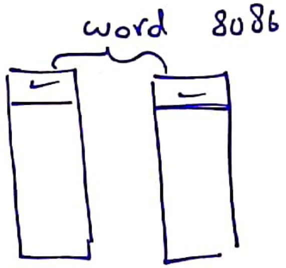
- The 8088 and 8086 can access any two consecutive bytes as word of data.

متتاليات

- Lower address byte and higher address byte.
- The two bytes represent the word

Address	Memory
...	...
00725	0101 0101
00724	0000 0010 = 5502_{16}
...	...

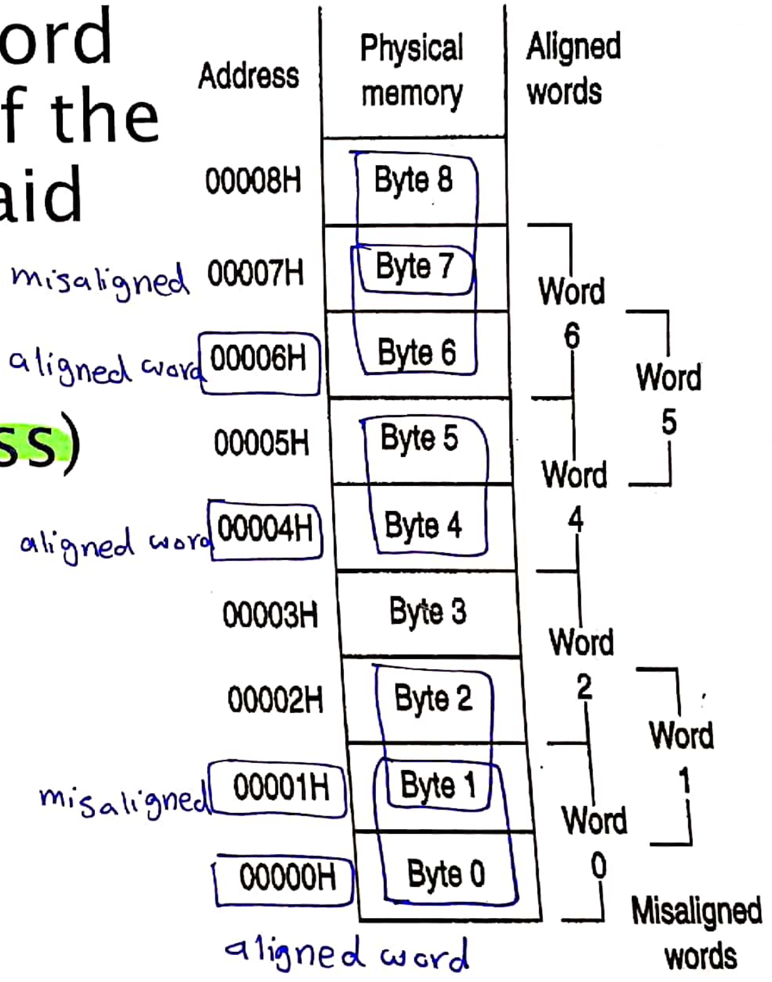




وعادي مش شرط يكونوا على نفس اللاين .

2.3 Memory Address Space and Data Organization

- Even- or odd-addressed word
  If the least significant bit of the address is 0, the word is said to be held at an even-addressed boundary.
- Aligned word (even-address) or misaligned word (odd-address).
- Words 0, 2, 4 & 6: aligned
- Words 1& 5: misaligned



* double word → متتاليات 4 byte يعني

byte 0,1,2,3 زي

aligned double word بنسميهم even دايما إنهم مبلشين من

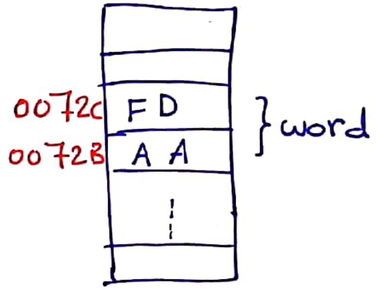
mis aligned double word بنسميهم odd مبلشين بـ byte 1,2,3,4 مثلاً

2.3 Memory Address Space and Data Organization

- EXAMPLE

  What is the data word shown below? Express the result in hexadecimal form. Is it stored at an even- or odd addressed word boundary? Is it an aligned or misaligned word of data?

Address	Memory
0072C	1111 1101
0072B	1010 1010



- Solution:

  $11111101_2 = FD_{16} = FD_H$

  $10101010_2 = AA_{16} = AA_H$

كيف بدي أعرف اذا موجب أو سالب ؟ بشوف الـ lower address

- Together the two bytes give the word

  $1111110110101010_2 = FDAA_{16} = FDAA_H$

وهي 0072B بحولها بتهيير 0000 0000 0111 0001 1011

00000000 01111 00010 1011

↓

بما انها (1) يعني odd

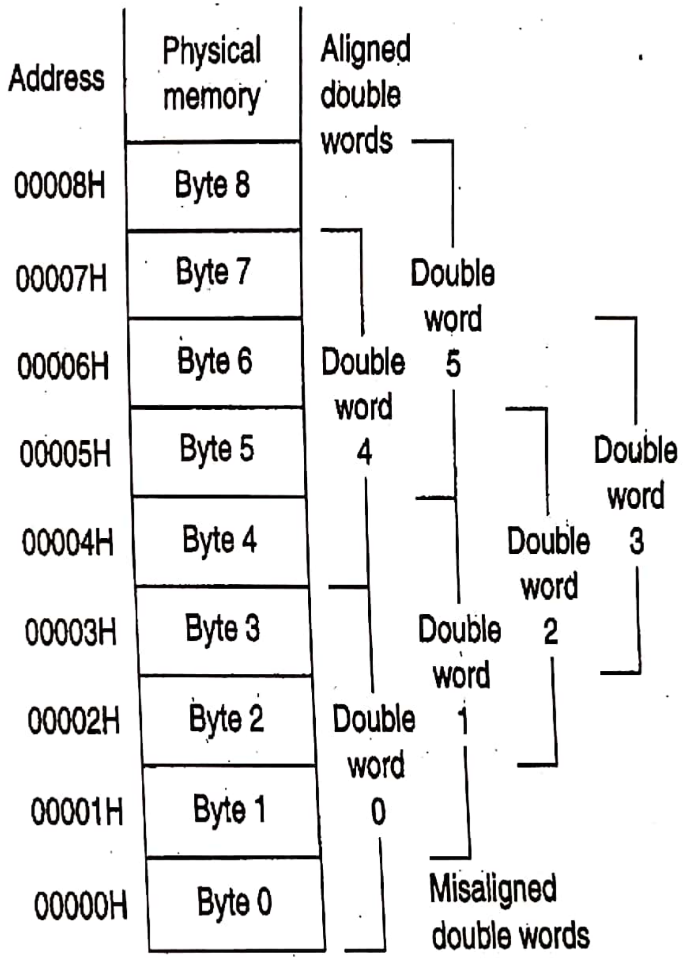
إذن بصير misaligned word

Expressing the address of the least significant byte in binary form gives $0072B_H = 0072B_{16} = 00000000011100101011_2$

- LSB = 1 ➔ the word is stored at odd-address boundary in memory.
- Therefore, it is misaligned word of data.

2.3 Memory Address Space and Data Organization

- A ***double word*** corresponds to four consecutive bytes of data stored in memory.
- Aligned double word is located at addresses of multiples of 4.
- Word 0 & 4: aligned only



2.3 Memory Address Space and Data Organization

- A pointer is a double word. The higher address word represents the ***segment base address*** while the lower address word represents the ***offset .***

3B 4C : 0065

higher lower

Address	Memory (binary)	Memory (hexadecimal)
00007_{16}	0011 1011	higher 3 B
00006_{16}	0100 1100	4 C
00005_{16}	0000 0000	0 0
00004_{16}	0110 0101	lower 6 5

Address	Memory (hexadecimal)
$0000B_{16}$	A0
$0000A_{16}$	00
00009_{16}	55
00008_{16}	FF

16 bit = 2 byte 12 bit = 2 byte

base : offset

higher lower

Pointer

32 bits [illegible]

4 bytes =

2 words =

Example: Segment base address = $3B4C_{16}$ = 0011101101001100_2

- Offset value = 0065_{16} = 0000000001100101_2

2.3 Memory Address Space and Data Organization

- EXAMPLE

  How should the pointer with content in segment base address equal to A000$_{16}$ and content in offset address equals 55FF$_{16}$ be stored at an even-address boundary starting at 00008$_{16}$? Is the double word aligned or misaligned?

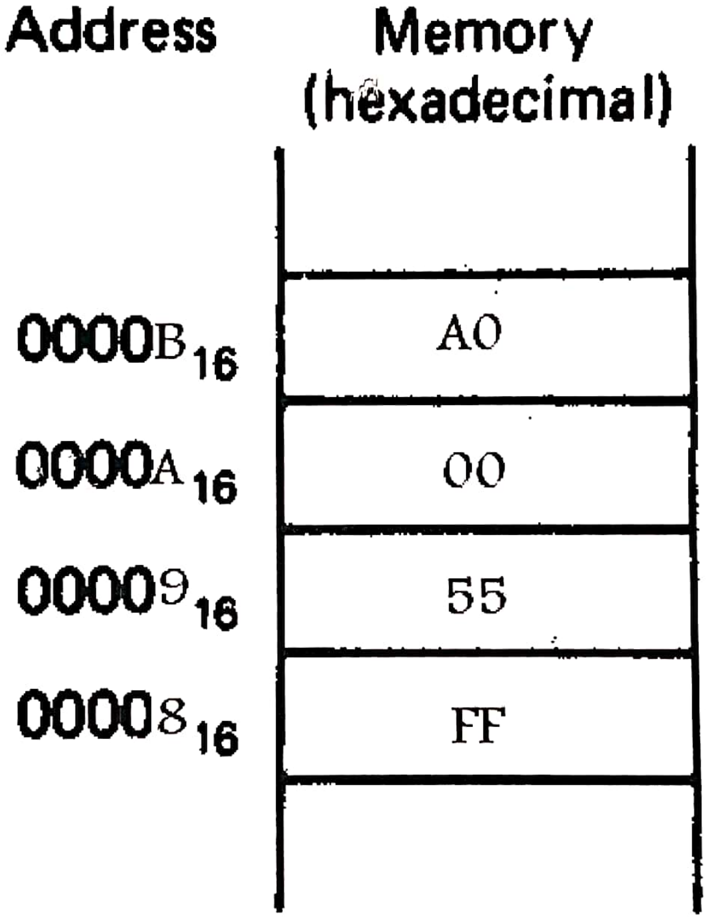
- Solution:

  Storage of the two-word pointer requires four consecutive byte locations in memory, starting at address 00008$_{16}$. The least significant byte of the offset is stored at address 00008$_{16}$ and is shown as FF$_{16}$ in the previous figure. The most significant byte of the offset, 55$_{16}$, is stored at address 00009$_{16}$. These two bytes are followed by the least significant byte of the segment base address, 00$_{16}$, at address 0000A$_{16}$, and its most significant byte, A0$_{16}$, at address 0000B$_{16}$. Since the double word is stored in memory starting at address 00008$_{16}$, it is aligned.

base : offset
A000 : 55FF

0000B	A0
0000A	00
00009	55
00008	FF

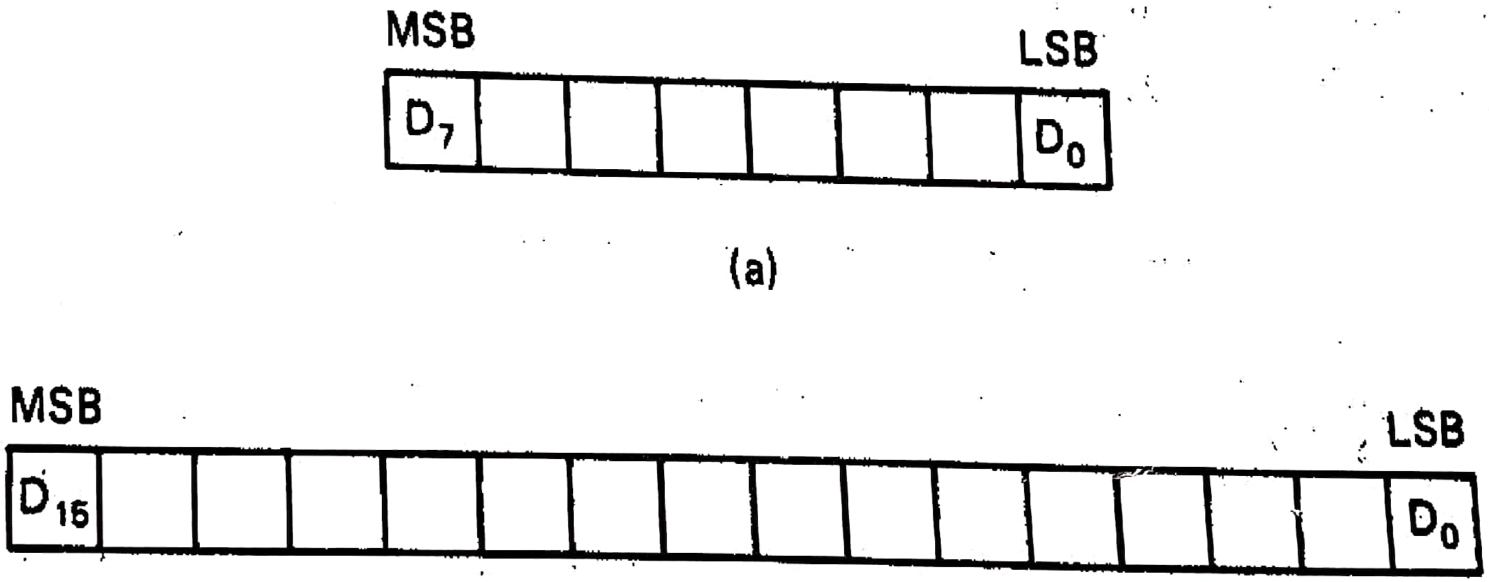
Previous Example Solution



Address	Memory (hexadecimal)
$0000B_{16}$	A0
$0000A_{16}$	00
00009_{16}	55
00008_{16}	FF

2.4 Data Types

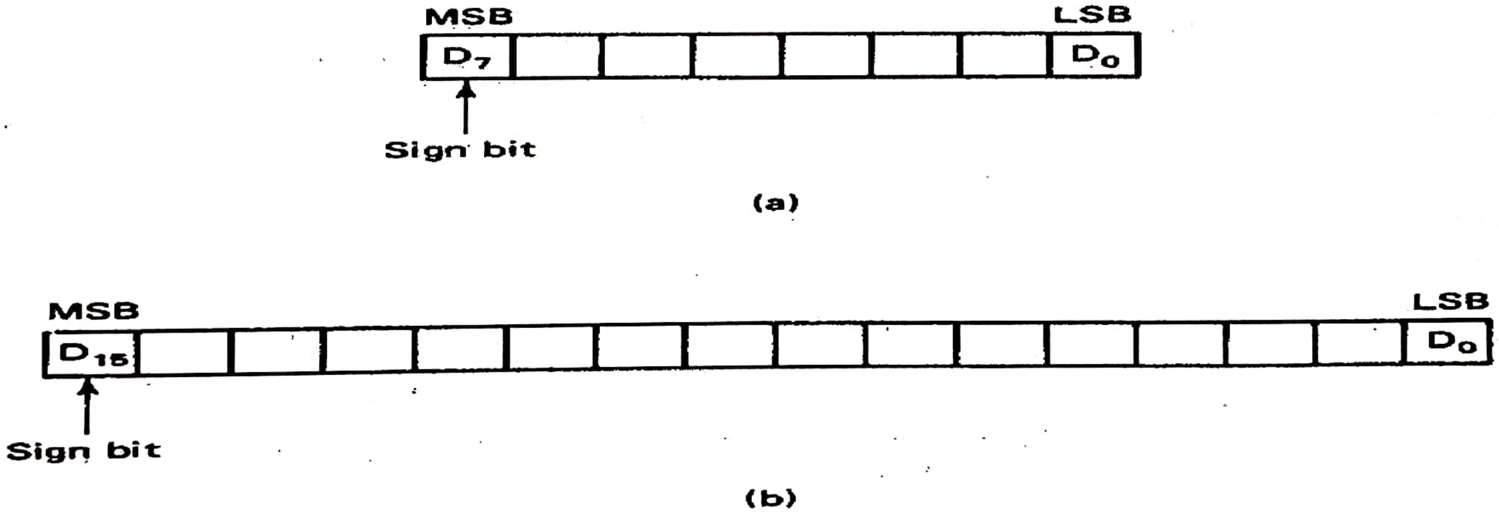
- Integer data type
  - Unsigned or signed integer
  - Byte-wide or word-wide integer



Unsigned byte and unsigned word integer

2.4 Data Types

- The most significant bit of a signed integer is a ***sign bit.*** A zero in this bit position identifies a positive number.
- The range of a signed byte integer is +127 ~ −128. The range of a signed word integer is +32767 ~ −32768.
- The 8088 always expresses negative numbers in 2’scomplement.



(a)

(b)

signed byte and signed word integer

2.4 Data Types

- EXAMPLE

A signed word integer equals $FEFF_{16}$. What decimal number does it represent?

- Solution:

$$FEFF_{16} = 1111111011111111_2$$

- The most significant bit is 1, the number is negative and is in 2's complement form.
- Converting to its binary equivalent by subtracting 1 from the least significant bit and then complement all bits give

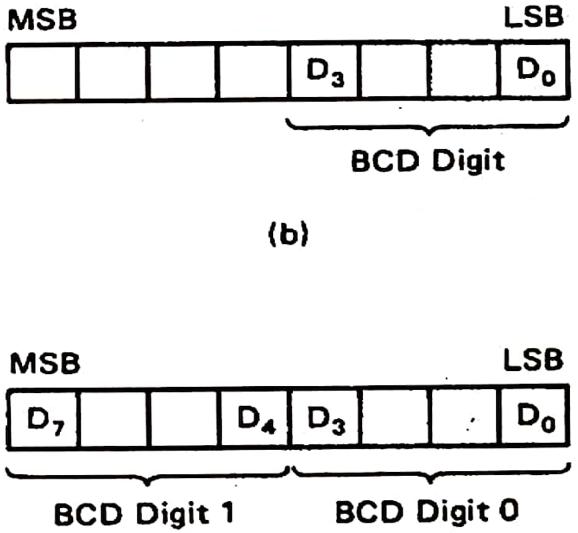
$$FEFF_{16} = -0000000100000001_2$$
$$= -257$$

2.4 Data Types

- The 8088 can also process data that is coded as ***binary-coded decimal (BCD) numbers.***
- BCD data can be stored in unpacked (upper 4 bits =0) or packed forms.

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

(a)



(b)

(c)

(a) BCD numbers (b) Unpacked BCD digit (c) Packed BCD digit

2.4 Data Types

- EXAMPLE

The packed BCD data stored at byte address 01000_{16} equals 10010001_2. What is the two digit decimal number?

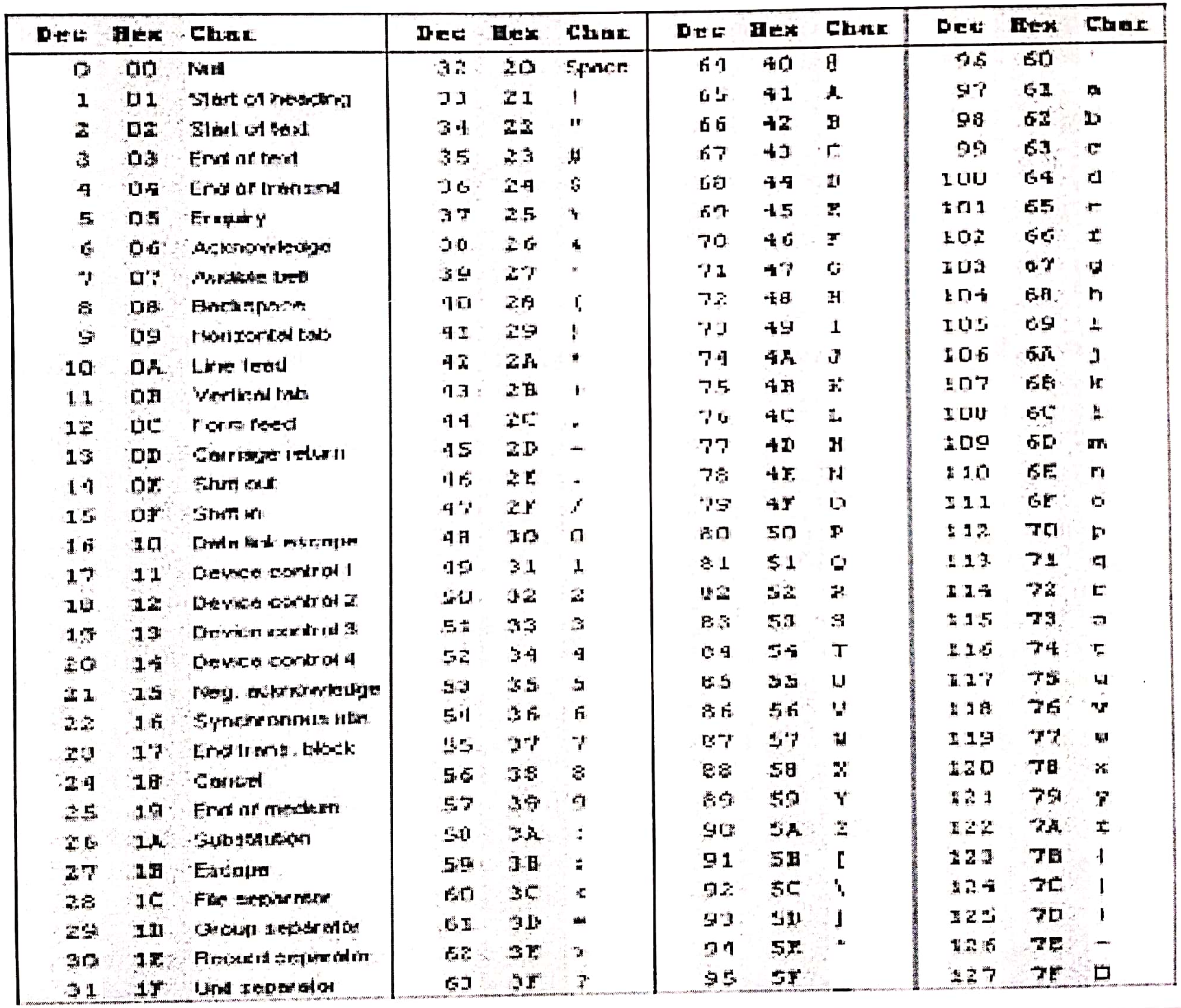
- Solution:

Writing the value 10010001_2 as separate BCD digits gives

$10010001_2 = 1001_{BCD}0001_{BCD} = 91_{10}$

2.4 Data Types

- The ASCII (American Standard Code for Information Interchange) digit
- The 8088 can process ASCII characters too.



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

2.4 Data Types

- EXAMPLE

Byte addresses 01100_{16} through 01104_{16} contain the ASCII data 01000001, 01010011, 01000011, 01001001, and 01001001, respectively. What do the data stand for?

- Solution:

Using the ASCII table, the data are converted to ASCII code:

(01100H) = 01000001_2 = A
(01101H) = 01010011_2 = S
(01102H) = 01000011_2 = C
(01103H) = 01001001_2 = I
(01104H) = 01001001_2 = I

2.5 Segment Registers and Memory Segmentation

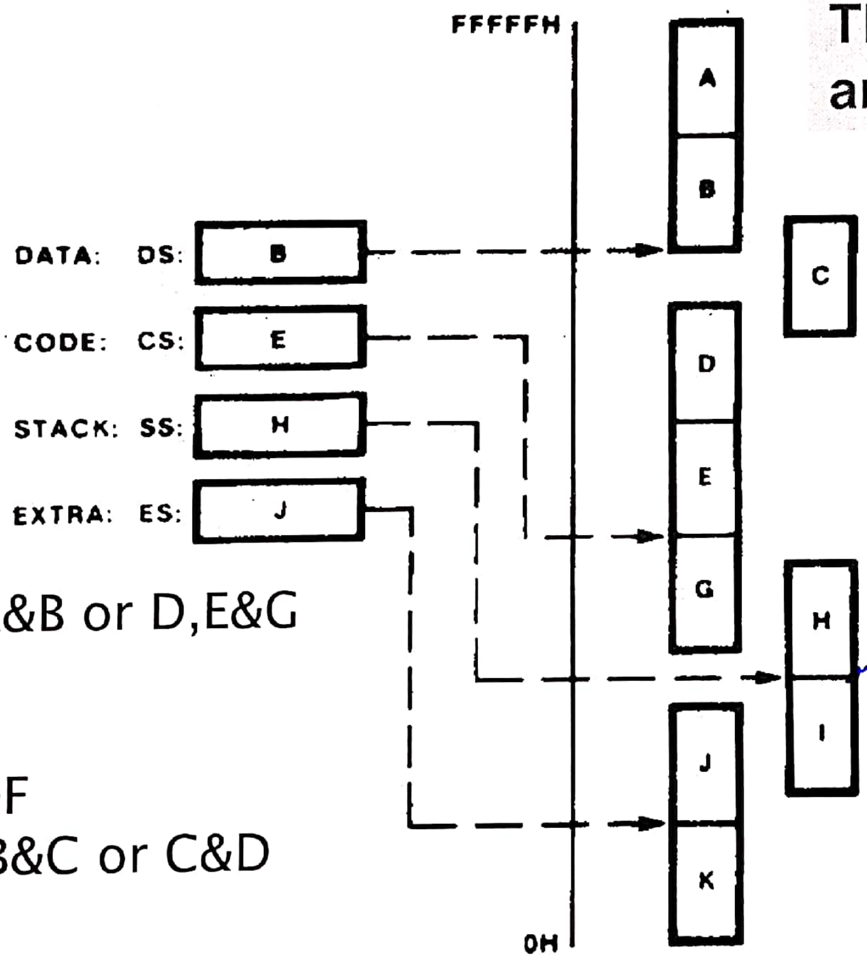
- A ***segment represents an independently addressable*** unit of memory consisting of 64K consecutive byte wide storage locations.
- Each segment is assigned a ***base address*** that identifies its starting point.
- Only four segments can be active at a time:
  - The code segment
  - The stack segment
  - The data segment
  - The extra segment
- The addresses of the active segments are stored in the four internal segment registers: CS, SS, DS, ES.

2.5 Segment Registers and Memory Segmentation

- Four segments give a maximum of 256Kbytes of active memory.
  - Code segment - 64K
  - Stack - 64K
  - Data storage - 128K
- The base address of a segment must reside on a **16-byte address boundary**.
- User accessible segments can be set up to be contiguous, adjacent, disjointed, or even overlapping.

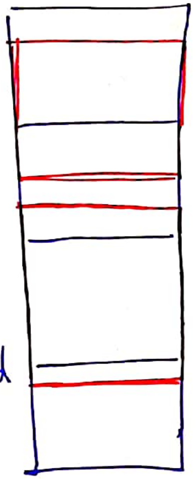
2.5 Segment Registers and Memory Segmentation

The segments registers are user accessible



- ☑ Contiguous—A&B or D,E&G or J & K
- ☑ Adjacent
- ☑ Disjointed—C&F
- ☑ Overlapping—B&C or C&D

بقدر أقسم ال seg
زي ما بدي ٤ الهم
يكون كل (seg) حجمه 64K
في حال كان تنين seg
متداخلين
رح يكون بينهم
over lapped
وحسب ما دخاطبهم البروسيسر
بتم اختيار واحد منهم
حتى لو متداخلين .



Contiguous, adjacent, disjointed, and overlapping segments

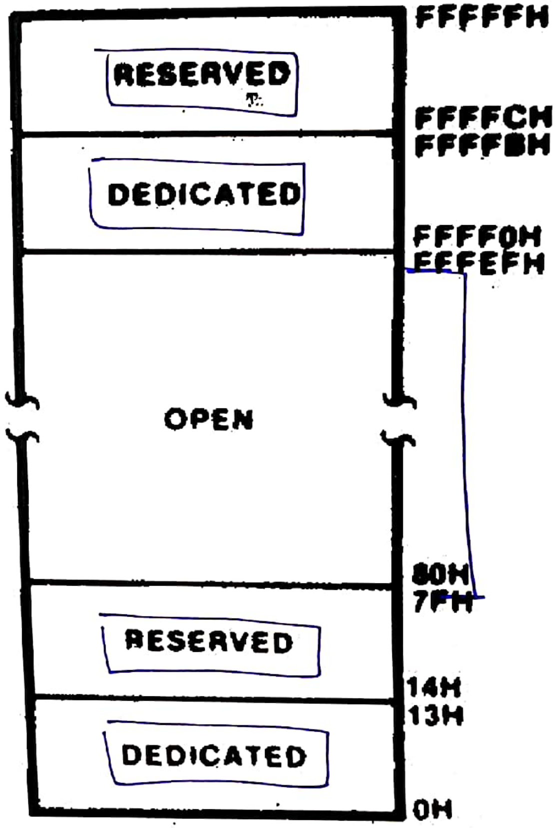
(/18)

2.6 Dedicated, Reserved, and General-Used Memory

- The **dedicated memory** (00000_{16} ~ 00013_{16}) are used for storage of the pointers to 8088's internal interrupt service routines and exceptions.
- The **reserved memory** (00014_{16} ~ $0007F_{16}$) are used for storage of the pointers to user-defined interrupts.
- The 128-byte dedicated and reserved memory can contain 32 interrupt pointers (double word 4-bytes) (128/4).
- The **general-use memory** (00080_{16} ~ $FFFEF_{16}$) stores data or instructions of the program.
- The **dedicated memory** ($FFFE0_{16}$ ~ $FFFEB_{16}$) are used for hardware reset jump instruction.
- The **reserved memory** ($FFFCF_{16}$ ~ $FFFFF_{16}$) are preserved for future use.

FFFEF

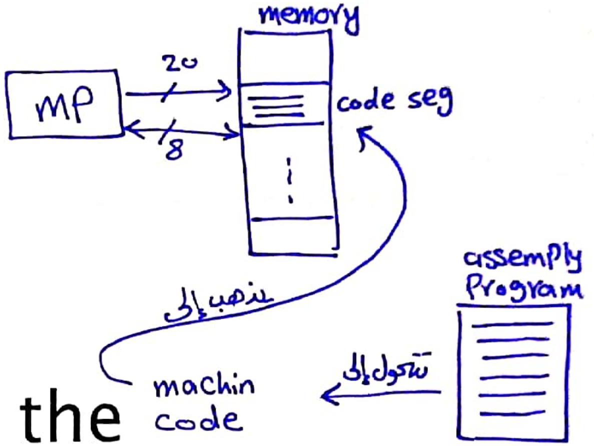
00086



حجم الميموري اميغا ومع ذلك مش كل الميموري متاحة للمستخدم ، اسمها dedicated and reserved

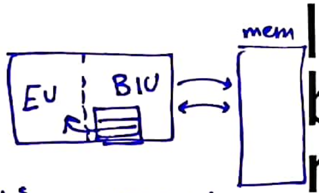
فالمستخدم مُتاح له فقط 00086-~~FFFEF~~ FFFEF

وظيفة البروسيسر حالياً ، يروح على الميموري يجيب الكود ويدخله على (BIU)
مشان نفسر ، مثلاً يعرف العملية ضرب ، جمع ، قسمة ---
يبعث خبر إلى (EU) مشان نفعّل العملية الرياضية المناسبة



2.7 Instruction Pointer

هلأ دور (EU) بجيب المتغير A والمتغير d وبجري عملية جمع بينهم وبخزن الناتج في B .
هلا (BIU) بتشتغل عالعملية التالية وتبعثها إلى (EU) وهكذا ---
* مين المسؤول عن التحرك إلى (next instruction) ؟ ip



- The *instruction pointer (IP)* identifies the location of the next word of instruction code to be fetched from the current code segment of memory.
- The offset in IP is combined with the current value in CS to generate the address of the instruction code. CS:IP forms 20-bit physical address of next word of instruction code.
- During normal operation, the 8088 fetches instructions from the code segment of memory, stores them in its instruction queue (why?), and executes them one after the other.

4 byte عبارة عن

الجواب ←

لما BIU بجيب أوامر
من الميموري ، بحطها
بمخزن مؤقت اسمه
(queue) ، مشان 8
تفلها (BIU) ideal
ولما تتعبى
ال (queue) كاملة
بتودي كل الأوامر
لعند (EU) .

* لما المستخدم يكتب (assemply Program) يتحول إلى (machine code) يتم تخزينه داخل (Code segment).

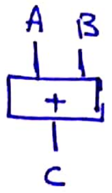
* B = A+d تكتب بـ (assemply lang) هيك بنكتب ⇐ ADD A,d,B (~~[illegible]~~).

* كل (Line) في (machine code) يمثل (instruction).

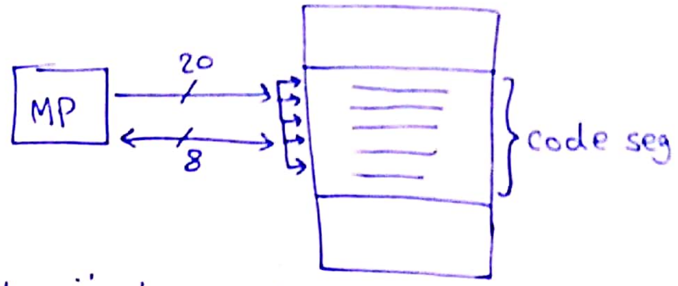
2.7 Instruction Pointer :

مهمته يأشر على
next instruction

- Instruction fetch sequence:
  - 8088/8086 fetches a word of instruction code from code segment in memory
    - Increments value in IP by 2 أو أكثر حسب length
    - Word placed in the instruction queue to wait for execution
    - 8088 prefetches up to 4 bytes of code
- Instruction execution sequence:
  - Instruction is read from output of instruction queue and executed
    - A,B Operands read from data memory, internal registers
    - Operation specified by the instruction performed on operands
    - c Result written back to either data memory or internal register



طبعاً لما تكون ال input مخزنة في internal reg أسهل نجيبها لحتى نعمل عملية رياضية بيهم، وأسرع من لما نجيبها من الميموري.



$ip_{new} = IP + C$

عبارة عن 2 byte مثلاً، تعتمد على (length) الخاص بـ (machine code).

*

* هدول (line segment) موجودين بنفس ال (Code seg) تبعتي كل (Line) له (ip) خاص به.

CS : ip
ثابت متغير

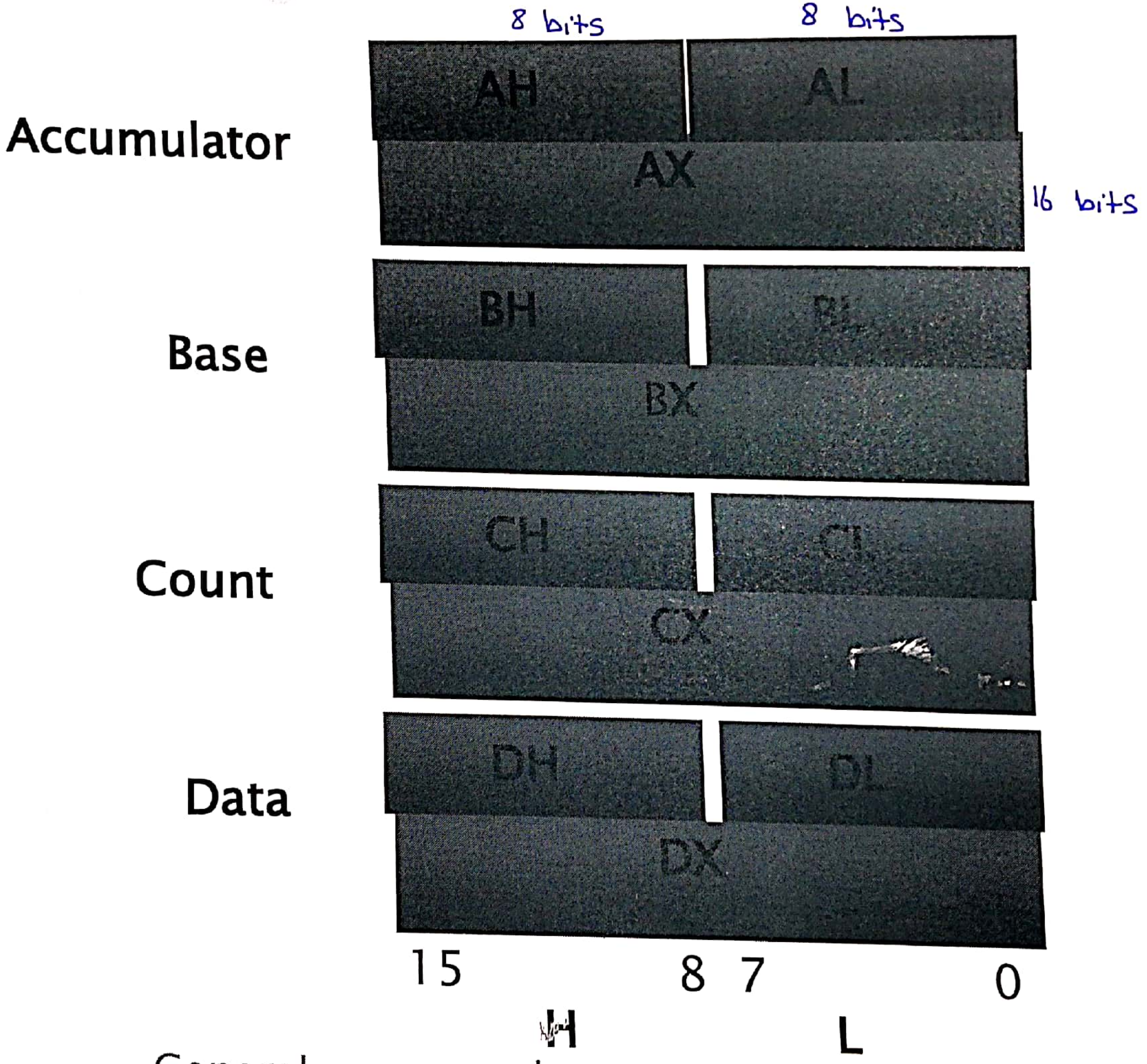
2.8 Data Registers

- The Data registers are used for temporary storage of frequently used intermediate results.
- The contents of the data registers can be read, loaded, or modified through software.
- The four data registers are:
  - Accumulator register, A
  - Base register, B
  - Counter register, C
  - Data register, D

general data regestor

- Each register can be accessed either as a whole (16 bits) for word data or as 8-bit data for byte-wide operation.

2.8 Data Registers



General-purpose data registers of 8088 microprocessor

متغير ثابت —

2.8 Data Registers

- Uses:
  - Hold data such as source or destination operands for most operations—ADD, AND, SHL (faster access)
  - Hold address pointer for accessing memory
- Some also have dedicated special uses
  - C—count for loop, repeat string, shift, and rotate operations
  - B—Table look-up translations, base address
  - D—indirect I/O and string I/O

2.8 Data Registers

مو حفظ

Register	Operations
AX	Word multiply, word divide, word I/O
AL	Byte multiply, byte divide, byte I/O, translate, decimal arithmetic
AH	Byte multiply, byte divide
BX	Translate
CX	String operations, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O

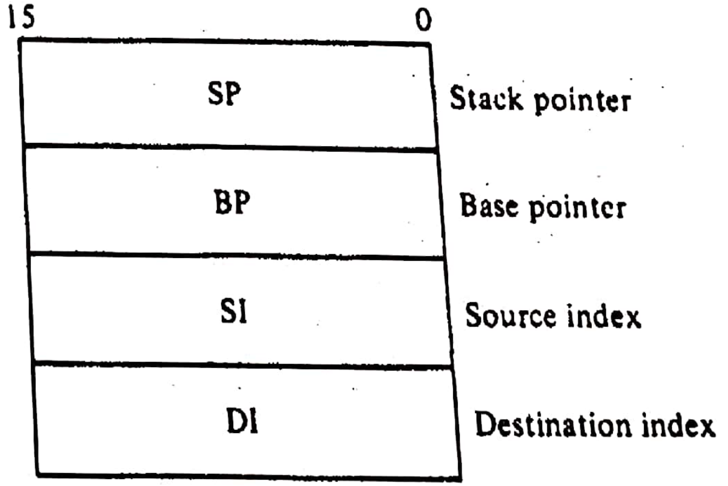
Dedicated register functions

2.9 Pointer and Index Registers

- The pointer registers and index registers are used to store offset addresses.
- Values held in the index registers are used to reference data relative to the data segment or extra segment.
- The pointer registers are used to store offset addresses of memory location relative to the stack segment register.
- Combining SP with the value in SS (SS:SP) results in a 20-bit address that points to the ***top of the stack*** (TOS).
- BP is used to access data within the stack segment of memory. It is commonly used to reference subroutine parameters.

2.9 Pointer and Index Registers

- The index register are used to hold offset addresses for instructions that access data in the data segment.
- The source index register (SI) is used for a source operand, and the destination index (DI) is used for a destination operand.
  - DS:SI—points to source operand in data segment
  - DS:DI—points to destination operand in data segment
- Also used to access information in the extra segment (ES)
- The four registers must always be used for 16-bit operations.


15 0

	SS : دايما مرتبط مع	SP	Stack pointer
	SS : 〃 〃 〃	BP	Base pointer
ES or	DS : 〃 〃 〃	SI	Source index
ES	DS : 〃 〃 〃	DI	Destination index

بشكل عام :
Stack Pointe
يستخدم ليؤشر على
top of the stack

2.10 Status Register :

هذا ريجستر رابع عشر ، يحمل مجموعة من ال (bits) ، كل (bit) عبارة عن (flag) أو (control bit) ما بخزن داتا ، بس بخزن (indication) عن طبيعة الناتج .

مثلاً لو كان الناتج سالب يقوم بتفعيل (bit) داخله لتبيّن أن الناتج سالب.

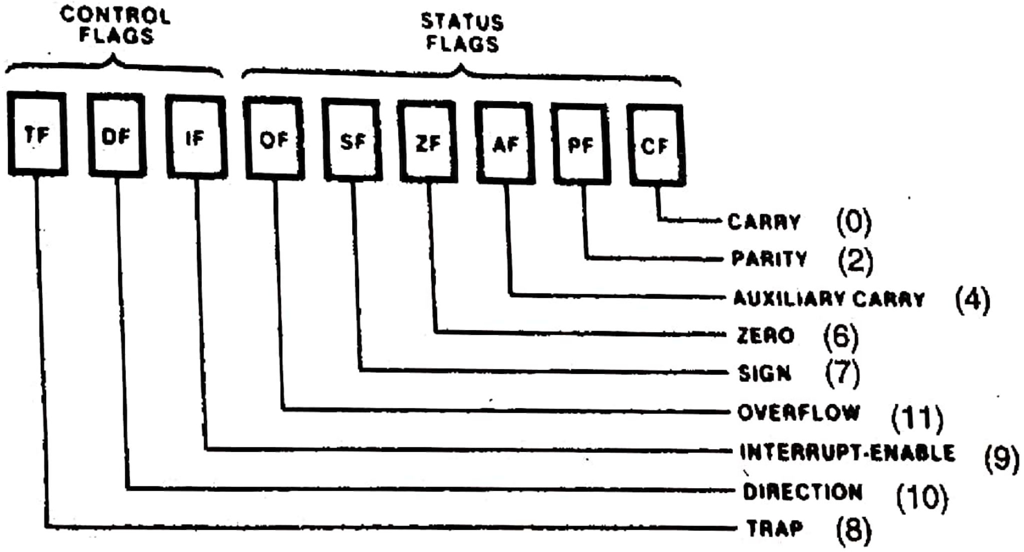
- The *status register*, also called the *flags register*, indicate conditions that are produced as the result of executing an instruction.
- Only nine bits of the register are implemented. **Six** of these bits represent ***status flags*** and the other **three** bits represent ***control flags***
- The 8088 provides instructions within its instruction set that are able to use these flags to alter the sequence in which the program is executed.

EICoM
ELECTRICAL COMPUTER MECHATRONICS

2.10 Status Register

- Status and control bits maintained in the flags register

9 bits =



لو أجريت A+B
وكان في over flow
رح تشغّل (of) داخل ال(flag)

- Generally Set and Tested Individually
- 9 1-bit flags in 8086; 7 are unused

2.10 Status Register

- Status flags indicate current processor status.

CF	Carry Flag	Arithmetic Carry/Borrow	
OF	Overflow Flag	Arithmetic Overflow	
ZF	Zero Flag	Zero Result; Equal Compare	
SF	Sign Flag	Negative Result; Non-Equal Compare	
PF	Parity Flag	Even Number of "1" bits	Odd parity
AF	Auxiliary Carry	Used with BCD Arithmetic	

2.10 Status Register

- Control flags influence the 8086 during execution phase

DF	Direction Flag	Auto-Increment/Decrement used for “string operations”
IF	Interrupt Flag	Enables Interrupts allows “fetch-execute” to be interrupted. Used to enable/disable external maskable interrupt requests
TF	Trap Flag	Allows Single-Step for debugging; causes interrupt after each op

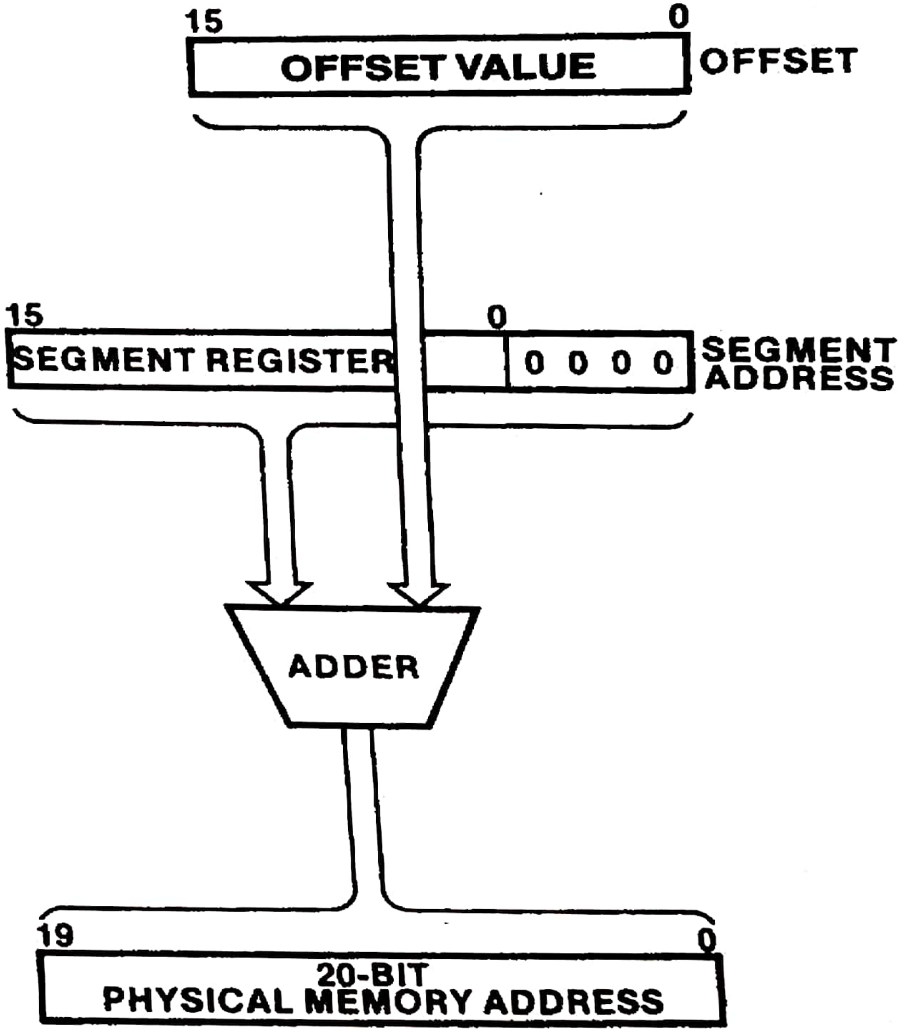
* يوجد داخل (BIU) اشي اسمه [adress generation and bus control unit]
ووظيفته - يحوّل من (digital add) لـ(Phisycal add) .

2.11 Generating a Memory Address

- A ***logical address in the 8088 microcomputer*** system is described by a segment base and an offset.
- The ***physical addresses that are used to access*** memory are 20 bits in length.
- The generation of the physical address involves combining a 16-bit offset value that is located in the instruction pointer, a base pointer, an index register, or a pointer register and a 16-bit segment base value that is located in one of the segment register.
- Segment base address (CS, DS, ES, SS) are 16 bit quantities.
- Offsets (IP, SI, DI, BX, DX, SP, BP, etc.) are 16 bit quantities.

2.11 Generating a Memory Address

- ☐ Physical Address: actual address used for accessing memory: 20-bits in length Formed by:
  - ☐ Shifting the value of the 16-bit segment base address left 4 bit positions
  - ☐ Filling the vacated four LSBs with 0s
  - ☐ Adding the 16-bit offset



Generating a physical address

* كيفية التحويل من logical إلى Phsiycal :-

بنجيب Base وبنعمله shift .بمقدار (1 Hexa decimal) وبعدين يجمعه الoffset-

. [4 Hexa decimal = 20 bit انه .بيعرف طبعاً]

Base ○
offset +
20 bit

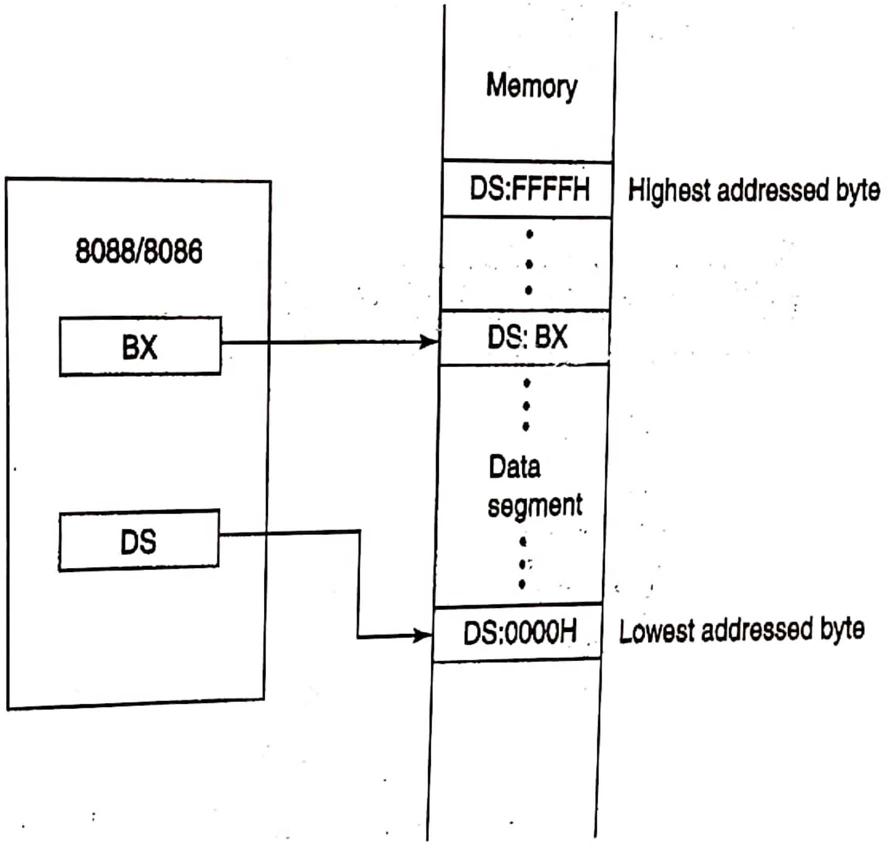
* مثال :-

Base = $F910_h$
offset = 1230_h

الحل :-

```
 F9100
 2230  +
 FA330
```

2.11 Generating a Memory Address



Boundary of a segment

- Four active segments CS, DS, ES, and SS

□Each 64-k bytes in size → maximum of 256K-bytes of active memory

□64K-bytes for code

□64K-bytes for stack

□128K-bytes for data (data and extra)

□Starting address of a data segment DS:0H → lowest addressed byte

□Ending address of a data segment DS:FFFFH → highest addressed byte

□Address of an element of data in a data segment

DS:BX → address of byte, word, or double word element of data in the data segment

2.11 Generating a Memory Address

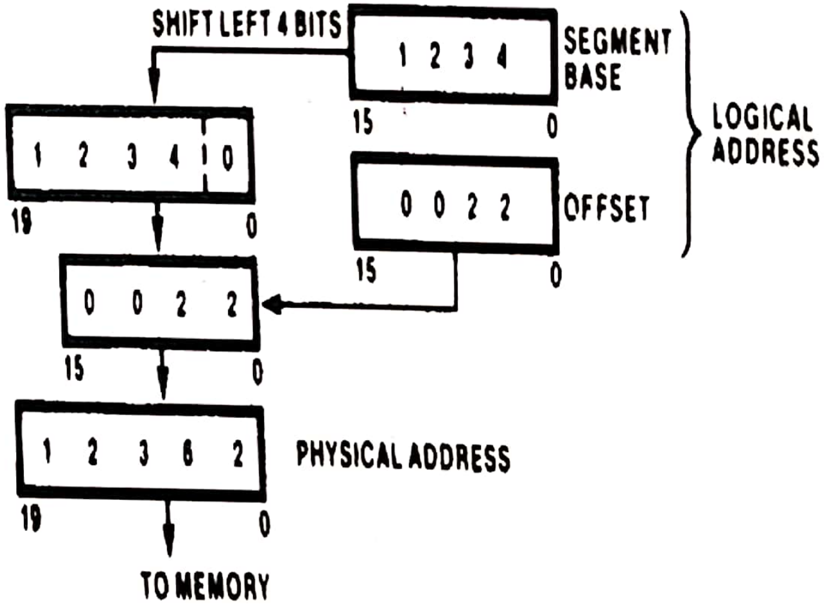
☐ **Example:**

Segment base address = 1234H
Offset = 0022H

$1234H = 0001\ 0010\ 0011\ 0100_2$
$0022H = 0000\ 0000\ 0010\ 0010_2$

Shifting base address,
$00010010001101000000_2 = 12340H$

Adding segment address and offset
$00010010001101000000_2 +$
$0000000000100010_2 =$
$= 00010010001101100010_2$
$= 12362H$



الحل :

$$\begin{array}{r} 12340 \\ 0022\ + \\ \hline 12362_H \end{array}$$

2.11 Generating a Memory Address

Base 0 → 002A0
offset + offset +
Phsyical → 002C3

offset = 002C3 −
002A0
00 23

0023
charبي،؟

- EXAMPLE

What would be the offset required to map to physical address location $002C3_{16}$ if the contents of the corresponding segment register are $002A_{16}$?

- Solution:

The offset value can be obtained by shifting the contents of the segment of the segment register left by four bit positions and then subtracting from the physical address. Shifting left give

$$002A0_{16}$$

Now subtracting, we get the value of the offset:

$$002C3_{16} - 002A0_{16} = 0023_{16}$$

2.11Generating a Memory Address

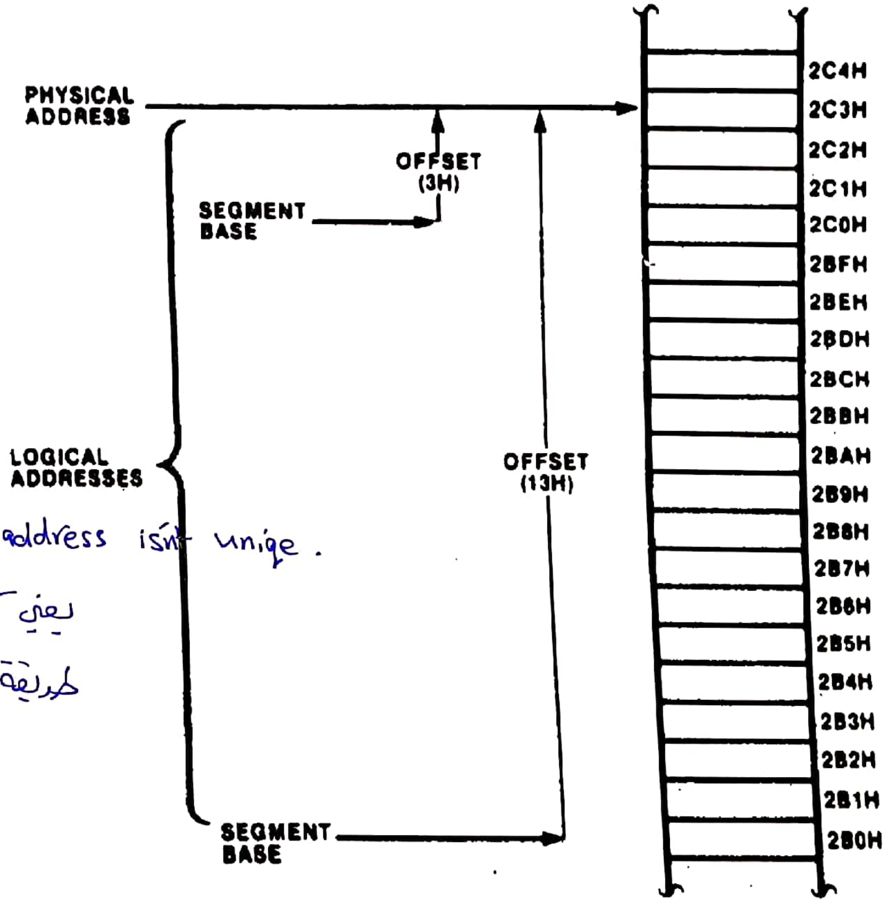
- Different logical addresses can be mapped to the same physical address location in memory.

☐Examples:

2BH:13H = 002B0H+0013H = 002C3H

2CH:3H = 002C0H + 0003H = 002C3H

* ■These logical addresses are called “aliases”



Physical address is unique but logical address isn't unige .

يعني كل لوكيشن له physical add واحد فقط لكن ممكن أوصله بأكثر من طريقة على سبيل المثال :

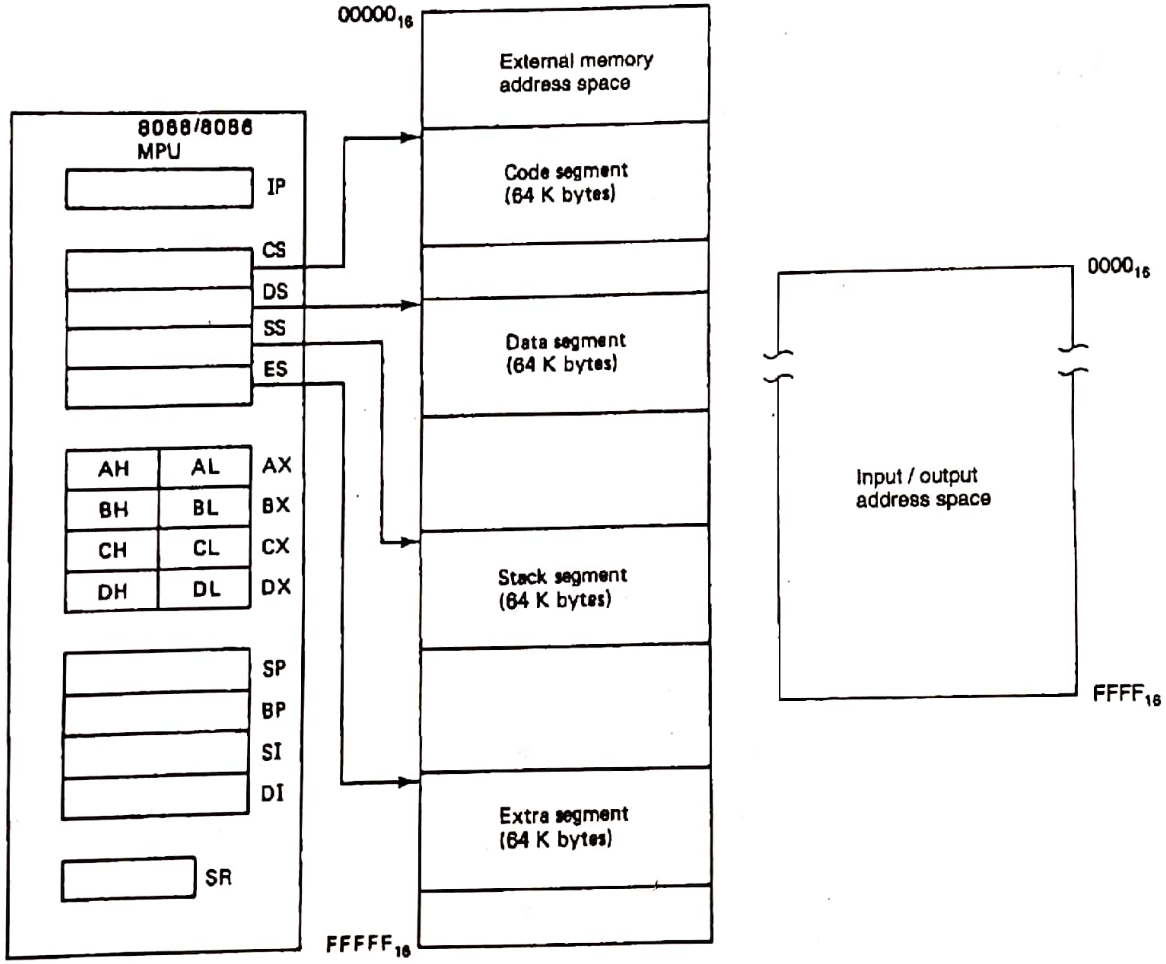
1+11 =12

2+10 =12

6+6 =12

⋮

2.2 Software Model of the 8088/8086 Microprocessor



Software model of the 8088/8086 microprocessors

53
22 −
21

2.12 The Stack

- The ***stack*** is implemented for temporary storage of information such as data or addresses.
- The stack is 64KBytes long and is organized from a software point of view as 32K words. 32 Kword بال
- The contents of the SP and BP registers are used as offsets into the stack segment memory while the segment base value is in the SS register.
- Push instructions (PUSH) and pop instructions (POP)
- ***Top of the stack (TOS) and bottom of the stack*** (BOS)
- The 8088 can push **word-wide data and address** information onto the stack from registers or memory.
- Many stacks can exist but only one is active at a time.

* Push data معناها ← write data

Pop data معناها ← read data

* SS ⇒ base segment reg

SS : SP أو BP

~~SS : SP [illegible] Push [illegible]~~

~~SS : BP [illegible] Pop " " "~~

* شو تفرق stack عن data seg ؟

التنتين بخزنوا داتا بس بال stack إلها خصوصية

بأن تخزن الداتا بترتيب معين وبسترجع الداتا

بترتيب معين.

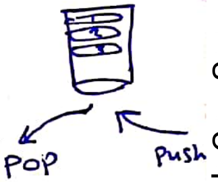
* كل Push بتقربني لل End
* كل Pop بتقربني لل Pottom

2.12 The Stack

* كل Push بصير sp=sp-2
* كل pop بصير sp=sp+2

* كل ما أخزن داتا جديدة التوب بنزل لتحت بمقدار 2

حتى أتذكر الشرح :-
مثال الوعاء المقلوب



النظام اسمه (LIFO)
Last in first out

Organization of stack:

☐SS:0000H → end of stack (lowest addressed word)

☐SS:FFFEH → bottom of stack (highest addressed word)

☐SS:SP → top of stack (last stack location to which data was pushed

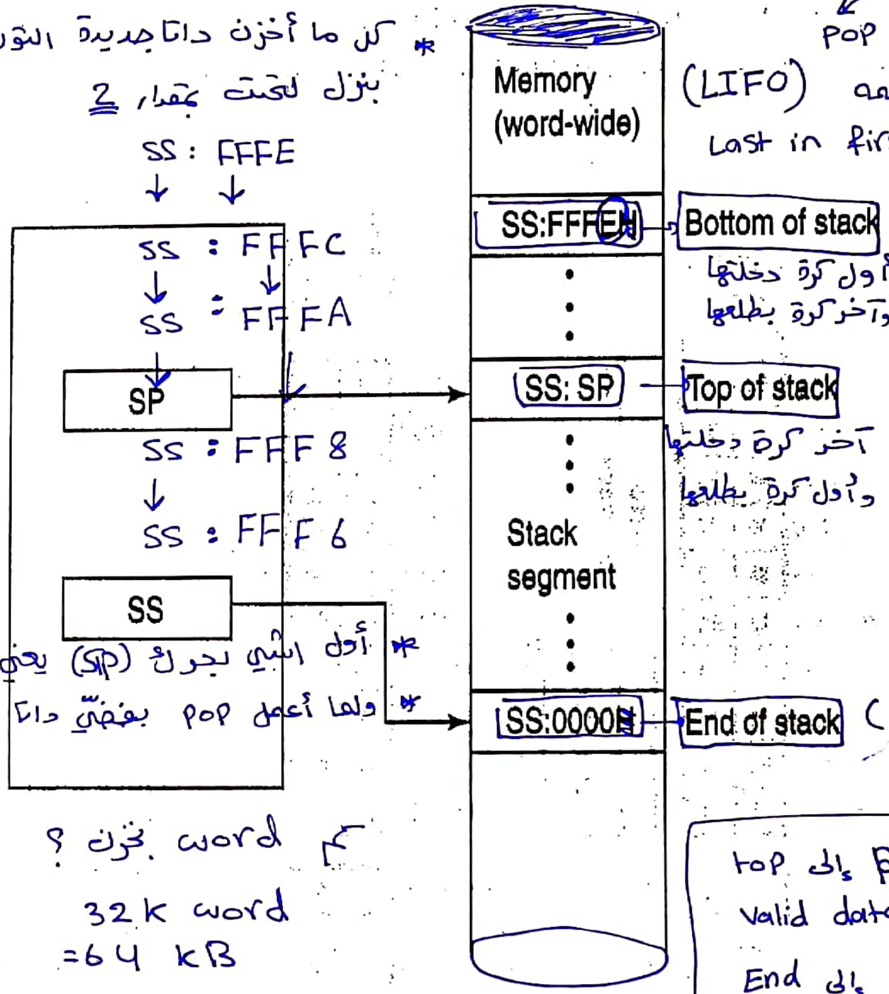
☐Stack grows down from higher to lower address

☐Used by call, push, pop, and return operations

☐Examples

PUSH SI → causes the current content of the SI register to be pushed onto the “top of the stack”

POP SI → causes the value at the “top of the stack” to be popped back into the SI register



Stack segment of memory

(BOS)
أول كرة دخلتها وآخر كرة بطلعها

آخر كرة دخلتها وأول كرة بطلعها

(EOS)

* أول اشي بحرك (SP) يعني بفضيله مكان وبعدين بدخل الداتا.
* ولما أعمل pop بفضي داتا ~~وبعدين~~ بالأول.

كم word نخزن ؟
32k word
=64 kB

من Bottom وإلى top تسمى valid data
ومن top وإلى End تكون فارغة من الداتا تسمى invalid data

لما تكونا فاضي top of stack = bottom of stack

Valid data = $\frac{Bot - top}{2} + 1$ هاد في حال طلبها بال (word) أما في حال طلبها بالبايت ما بقسم على 2 .

2.12 The Stack

□Stack status prior to execution of the instruction PUSH AX:

AX = 1234H

SS = 0105H

AEOS = SS:00 → 01050H = end of stack

SP = 0008H

ABOS = 0105_{16} + $FFFE_{16}$

= SS:FFFEH → 1104EH

ATOS = 0105_{16} + 0008_{16}

=SS:SP → 01058H = current top of stack

BBAAH = Last value pushed to stack

Addresses < 01058H = invalid stack data

Addresses >= 01058H = valid stack data

□In response to the execution of PUSH AX instruction:

1. SP → 0006H decremented by 2

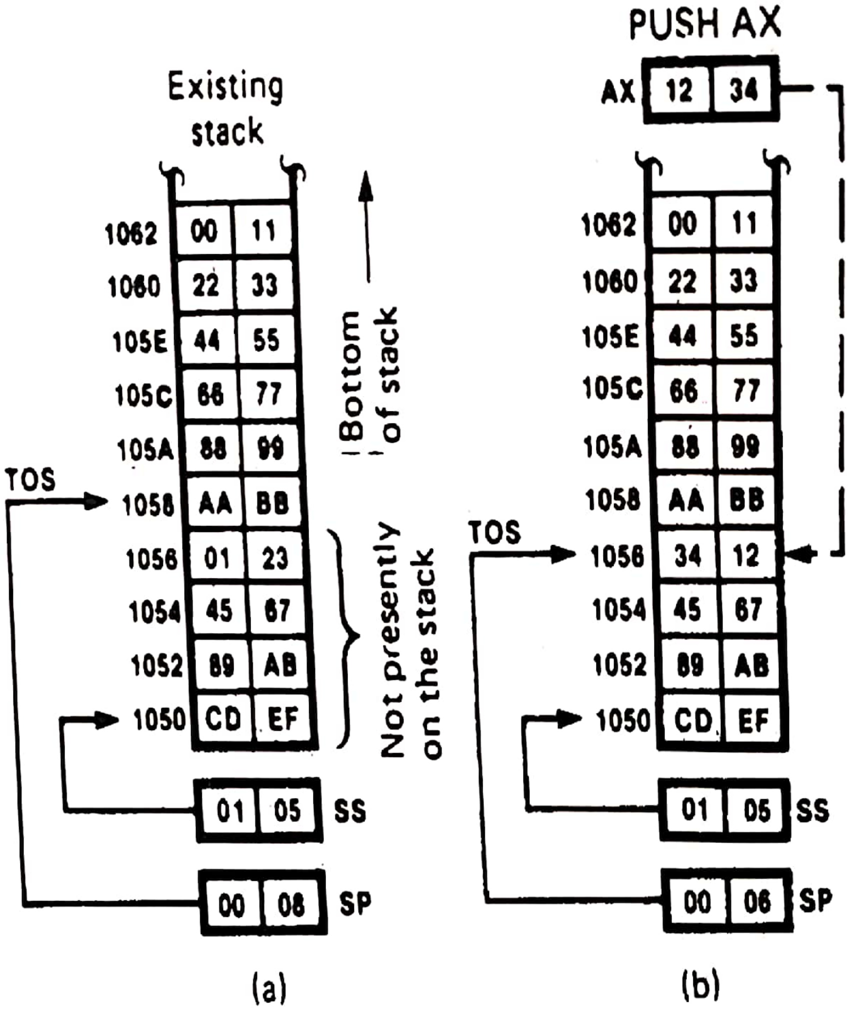
ATOP → 01056H

2. Memory write to stack segment

AL = 34H → 01056H

AH = 12H → 01057H

- How many free spaces ?
- When the stack get full ?



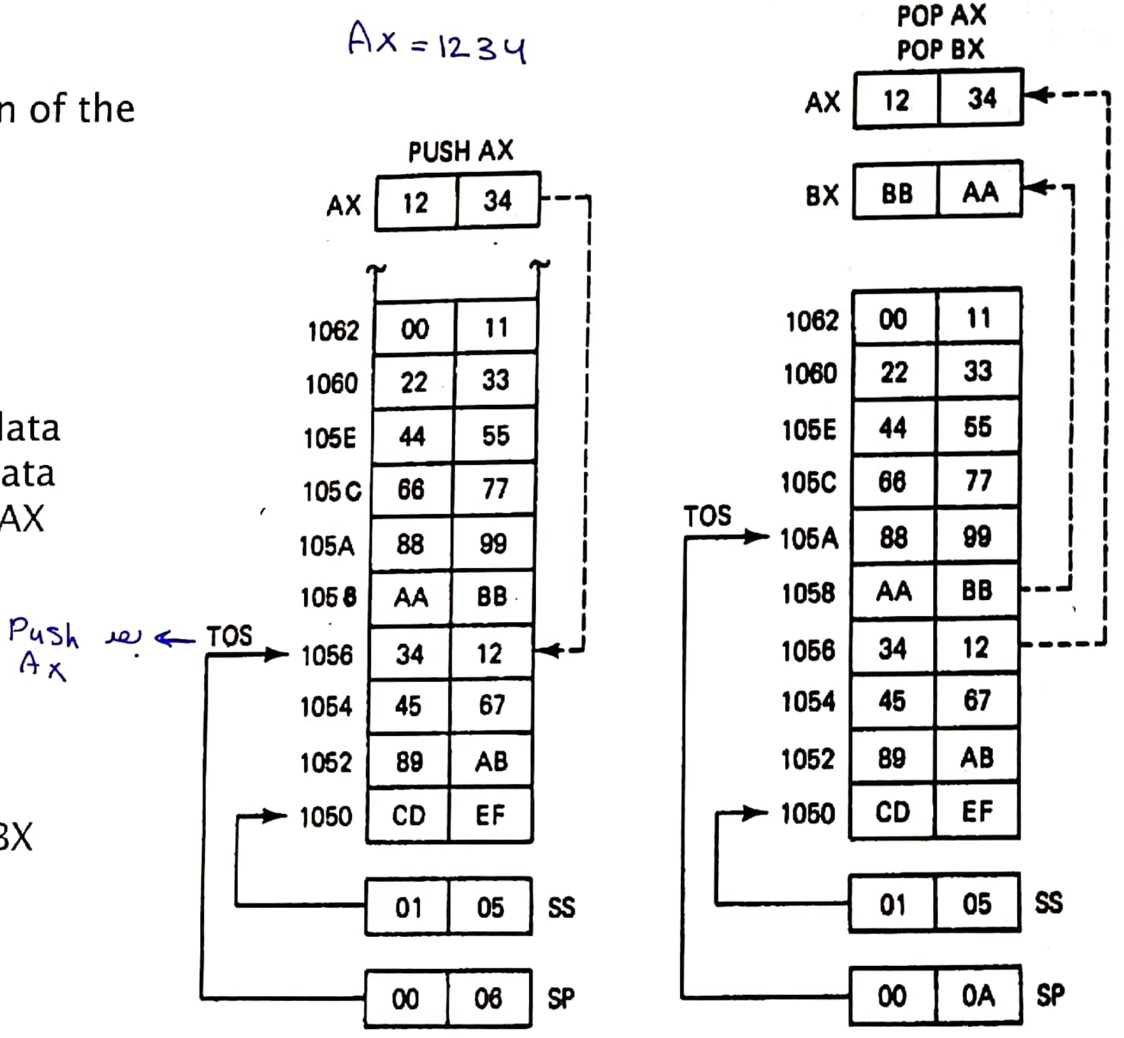
2.12 The Stack

- EXAMPLE: Pop operation

☐Status of the stack prior to execution of the instruction POP AX:
AX = XXXXH
SS = 0105H
SP = 0006H
ATOS = SS:SP → 01056H
1234H = Last value pushed to stack
Addresses < 01056H = invalid stack data
Addresses >= 01056H = valid stack data

☐In response to the execution of POP AX instruction
1. Memory read to AX
01056H = 34H → AL
01057H = 12H → AH
2. SP → 0008H incremented by 2
ATOP → 01058H

☐In response to the execution of POP BX instruction
1. Memory read to BX
01058H = AAH → BL
01059H = BBH → BH
2. SP → 000AH incremented by 2:
ATOP → 0105AH



(a) (b)

مثال :- given Ax = 1234
SP = 0008
SS = 0105

جد Pottom of the stack ؟

الحل :-

BOS => SS : FFFE
= 0105 : FFFE

تحويل من logical
الى Phesycal

111
= 01050
FFFE +
$\boxed{1104E}$ → Pottom of the stack.

جد end of the stack ؟

الحل :

EOS => SS : 0000
= 0105 : 0000
= 01050
0000 +
$\boxed{1050}$ → end of the stack

جد top of the stack ؟

الحل :-

TOS => SS : SP
= 0105 : 0008
= 01050
0008 +
$\boxed{1058}$ → top of the stack

هلأ بجيّ على الرسمة يلي بالسلايد ، محتوى tos = AA BB

* مطلوب نعمل (Push Ax) :

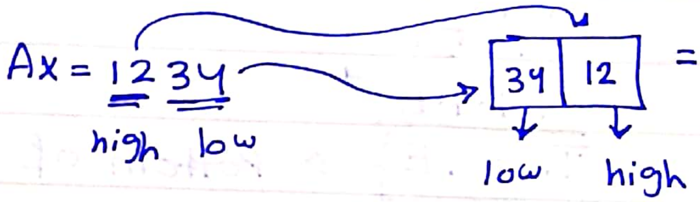
الحل :-

بفضّي مكان جديد وبعمل decrement

$SP = SP - 2$

$= 1058 - 2$

$= 1056$



* مطلوب نعمل (Pop Ax) ثم (Pop Bx) تكملة عالفرع السابق :-

الحل :-

top حالية = 1056 $\xrightarrow{\text{محتواه}}$ 1234

هلأ بناخد القيمة وبنخزنها داخل Ax .

$SP = SP + 2$

$= 1056 + 2$

$= 1058$ $\xrightarrow{\text{محتواه}}$ Ax | 12 | 34 |

ثم أعمل (Pop Px) ، يعني بناخد القيمة وبنخزنها في Px .

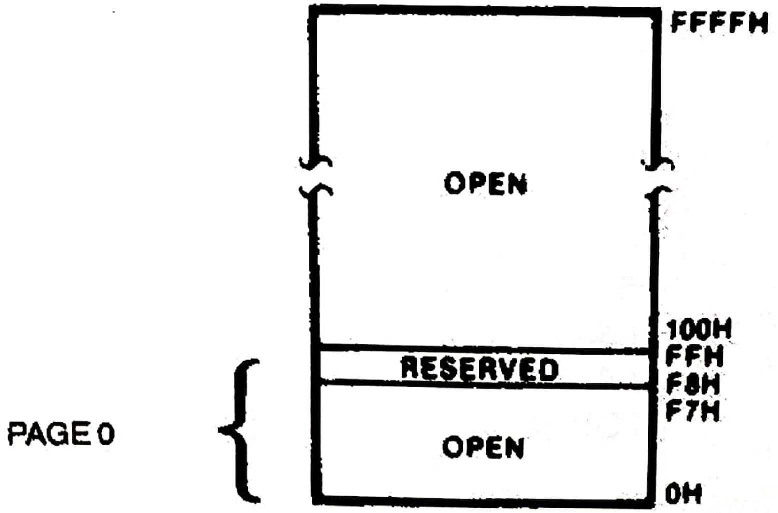
$SP = SP + 2$

$= 1058 + 2$

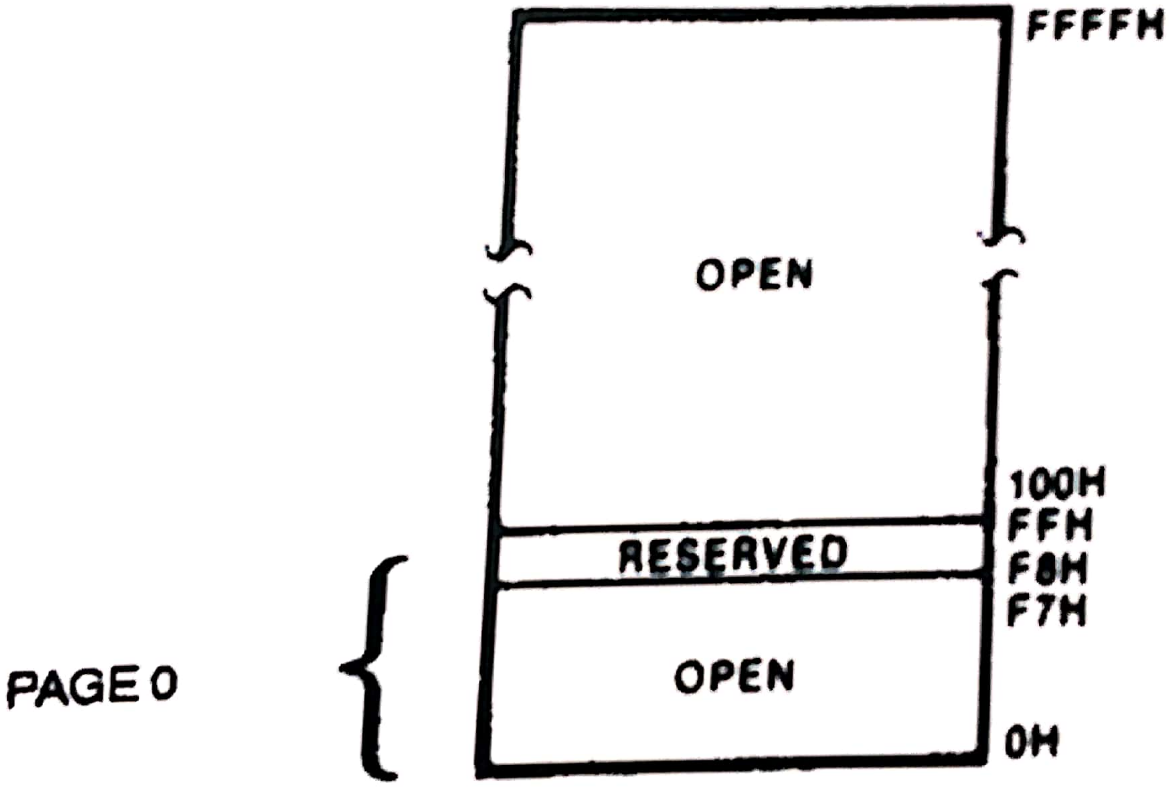
$= 105A$ $\xrightarrow{\text{محتواه}}$ Bx | BB | AA |

2.13 Input/Output Address Space

- The 8088 has separate memory and input/output (I/O) address space.
- The I/O address space is the place where I/O interfaces, such as printer and terminal ports, are implemented.
- The I/O address range is from 0000_{16} to $FFFF_{16}$. This represents 64KByte addresses.
- The I/O addresses are 16 bits long. Each of these addresses corresponds to one byte-wide I/O port.
- Certain I/O instructions can only perform operations to addresses 0000_{16} thru $00FF_{16}$ (***page 0***).
- Ports F8H through FF reserved



2.13 Input/Output Address Space



I/O address space

H.W. #2

- ☐ Solve the following problems from Chapter 2 from the course textbook:

3, 9, 14, 19, 26, 32, 34, 37, 45, 49, 55, 60, 65

If SS = C000, SP = FF00

1) How many data words currently in the stack

2) How the value EE11 will be pushed in the stack

Lecture Outline

- 3.1 Software: The Microcomputer Program
- 3.2 Assembly Language Programming Development on the PC
- 3.3 The Instruction Set (يوجد 6 instruction داخل EU)
- 3.4 The MOV Instruction → أحد هذه الinstruction وظيفته ينقل داتا من الميموري لوكيشن إلى الريجستر أو العكس ---
- 3.5 Addressing Modes

الحالات التي يكون عليها الـ inputs وoutputs لـ instruction معين.

زي Add يحتاج لتنين inputs وواحد output.

زي mutiply 〃 〃 〃 〃 〃

3.1 Software : The Microcomputer Program

*A = C+D

ADD C, D, A

C, D, A ⟹ operants

ADD ⟹ operation code

- A program is a sequence of commands that tell the microprocessor what to do.
- Each command is called “instruction”.
- An instruction can be divided into two parts:
  - Operation code (opcode) – one- to five-letter ***mnemonic***
- Operands: Identify whether the elements of data to be processed are in registers or memory.

* ADD Ax, Bx

معناه اجمع Ax مع Bx وخزّن
الناتج في Ax ، حيث تعتبر Ax input
و output بنفس الوقت.

*Opcode *Destination Operand *Source Operand

يوجد خاصية في 8086/8088 أنها
تأخذ 2operants ، واحد منهم يكون
destinati و source بنفس الوقت

AX + BX → AX معناه

ADD AX, BX

↳ source operand

- Source operand– location of one operand to be processed
- Destination operand—location of the other operand to be processed and the location of the result
- Format of an assembly statement:

LABEL: INSTRUCTION ; COMMENT

3.1 Software : The Microcomputer Program

- Label—address identifier for the statement
- Instruction—the operation to be performed
- Comment—documents the purpose of the statement
- Example:

START: MOV AX, BX ; COPY BX into AX

- Other examples:

INC SI ;Update pointer

ADD AX, BX

  - Few instructions have a label—usually marks a point to jump
  - Not all instructions need a comment
- What is the "MOV part of the instruction called?
- What is the BX part of the instruction called?
- What is the AX part of the instruction called?

3.1 Software : The Microcomputer Program

- Assembly language program
  - Assembly language program (.asm) file—known as "source code"
  - Converted to machine code by a process called "assembling"
  - Assembling performed by a software program — an "8088/8086 assembler"
  - "Machine (object) code" that can be run on a PC is output in the executable (.exe) file
  - "Source listing" output in (.lst) file—printed and used during execution and debugging of program
- DEBUG—part of "disk operating system (DOS)" of the PC
  - Permits programs to be assembled and disassembled
  - Line-by-line assembler
  - Also permits program to be run and tested
- MASM—Microsoft 80x86 macroassembler
  - Allows a complete program to be assembled in one step

3.1 Software : The Microcomputer Program

- Assembly source program

```
TITLE        BLOCK-MOVE PROGRAM
             PAGE ,132

COMMENT *This program moves a block of specified number of bytes
from one place to another place*

;Define constants used in this program

             N = 16 ;Bytes to be moved
             BLK1ADDR= 100H ;Source block offset address
             BLK2ADDR= 120H ;Destination block offset addr
             DATASEGADDR= 2000H ;Data segment start address

STACK_SEG              SEGMENT              STACK 'STACK'
   DB                     64 DUP(?)
   STACK_SEG              ENDS

CODE_SEG               SEGMENT              'CODE'
   BLOCK          PROC                 FAR
             ASSUME CS:CODE_SEG,SS:STACK_SEG
```

3.1 Software : The Microcomputer Program

- Assembly source program (continued)

```
;To return to DEBUG program put return address on the stack
                PUSH        DS
                MOV         AX, 0
                PUSH        AX
;Set up the data segment address
                MOV         AX, DATASEGADDR
                MOV         DS, AX
;Set up the source and destination offset addresses
                MOV         SI, BLK1ADDR
                MOV         DI, BLK2ADDR
;Set up the count of bytes to be moved
                MOV         CX, N
;Copy source block to destination block
NXTPT:          MOV         AH, [SI] ;Move a byte
                MOV         [DI], AH
                INC         SI ;Update pointers
                INC         DI
                DEC         CX ;Update byte counter
                JNZ         NXTPT ;Repeat for next byte
                RET         ;Return to DEBUG program
BLOCK                       ENDP
CODE_SEG                    ENDS
END             BLOCK       ;End of program
```

3.1 Software : The Microcomputer Program

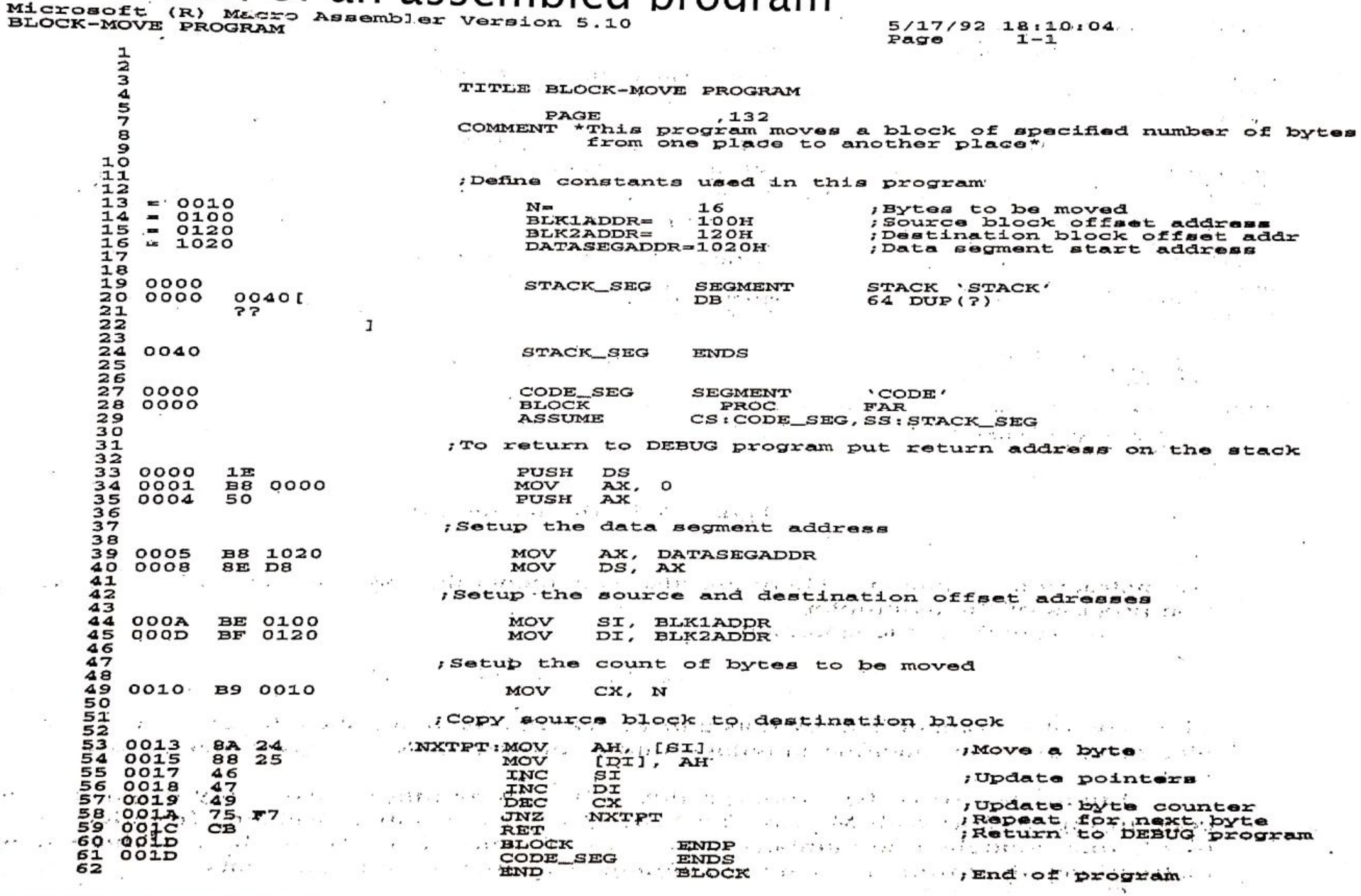
- Assembly language must be converted by an ***assembler to an equivalent machine language*** program for execution by the 8088.
- A directive is a statement that is used to control the translation process of the assembler.

  e.g. DB 64 DUP(?)

  - Defines and leaves un-initialized a block of 64 bytes in memory for use as a stack
- The machine language output produced by the assembler is called ***object code.***

3.1 Software : The Microcomputer Program

- Listing of an assembled program



```
Microsoft (R) Macro Assembler Version 5.10                                    5/17/92 18:10:04
BLOCK-MOVE PROGRAM                                                            Page     1-1

     1
     2
     3                                       TITLE BLOCK-MOVE PROGRAM
     4
     5                                             PAGE        ,132
     7                                       COMMENT *This program moves a block of specified number of bytes
     8                                                from one place to another place*
     9
    10
    11                                       ;Define constants used in this program
    12
    13  = 0010                                   N=              16              ;Bytes to be moved
    14  = 0100                                   BLK1ADDR=       100H            ;Source block offset address
    15  = 0120                                   BLK2ADDR=       120H            ;Destination block offset addr
    16  = 1020                                   DATASEGADDR=1020H               ;Data segment start address
    17
    18
    19  0000                                     STACK_SEG       SEGMENT         STACK 'STACK'
    20  0000     0040[                                           DB              64 DUP(?)
    21            ??
    22                        ]
    23
    24  0040                                     STACK_SEG       ENDS
    25
    26
    27  0000                                     CODE_SEG        SEGMENT         'CODE'
    28  0000                                     BLOCK             PROC          FAR
    29                                           ASSUME          CS:CODE_SEG,SS:STACK_SEG
    30
    31                                       ;To return to DEBUG program put return address on the stack
    32
    33  0000     1E                              PUSH    DS
    34  0001     B8 0000                         MOV     AX, 0
    35  0004     50                              PUSH    AX
    36
    37                                       ;Setup the data segment address
    38
    39  0005     B8 1020                         MOV     AX, DATASEGADDR
    40  0008     8E D8                           MOV     DS, AX
    41
    42                                       ;Setup the source and destination offset adresses
    43
    44  000A     BE 0100                         MOV     SI, BLK1ADDR
    45  000D     BF 0120                         MOV     DI, BLK2ADDR
    46
    47                                       ;Setup the count of bytes to be moved
    48
    49  0010     B9 0010                         MOV     CX, N
    50
    51                                       ;Copy source block to destination block
    52
    53  0013     8A 24                   NXTPT:MOV       AH, [SI]                        ;Move a byte
    54  0015     88 25                         MOV       [DI], AH
    55  0017     46                            INC       SI                              ;Update pointers
    56  0018     47                            INC       DI
    57  0019     49                            DEC       CX                              ;Update byte counter
    58  001A     75 F7                         JNZ       NXTPT                           ;Repeat for next byte
    59  001C     CB                            RET                                       ;Return to DEBUG program
    60  001D                                   BLOCK             ENDP
    61  001D                                   CODE_SEG          ENDS
    62                                         END               BLOCK                   ;End of program
```

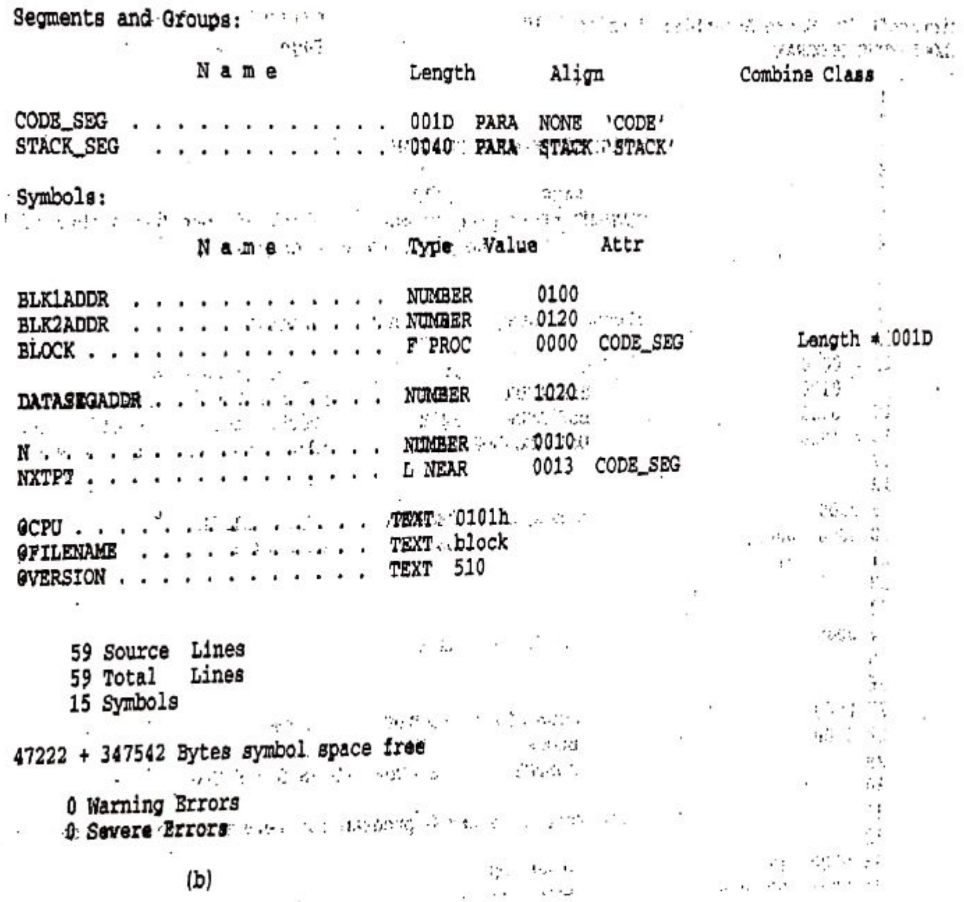
3.1 Software : The Microcomputer Program

- Listing of the assembled program

  e.g. 0013 8A 24 NXTPT: MOV AH, [SI] ; Move a byte
- Where:
  - 0013 = offset address (IP) of first byte of code in the CS
  - 8A24 = machine code of the instruction
  - NXTPT: = Label
  - MOV = instruction mnemonic
  - AH = destination operand
  - [SI] = source operand in memory

3.1 Software : The Microcomputer Program

- Listing of an assembled program



```
Segments and Groups:

                N a m e             Length     Align          Combine Class

CODE_SEG . . . . . . . . . . . . .  001D  PARA  NONE  'CODE'
STACK_SEG  . . . . . . . . . . . .  0040  PARA  STACK 'STACK'

Symbols:

                N a m e             Type   Value    Attr

BLK1ADDR . . . . . . . . . . . . .  NUMBER      0100
BLK2ADDR . . . . . . . . . . . . .  NUMBER      0120
BLOCK  . . . . . . . . . . . . . .  F PROC      0000  CODE_SEG          Length = 001D

DATASEGADDR  . . . . . . . . . . .  NUMBER      1020

N  . . . . . . . . . . . . . . . .  NUMBER      0010
NXTPT  . . . . . . . . . . . . . .  L NEAR      0013  CODE_SEG

@CPU . . . . . . . . . . . . . . .  TEXT  0101h
@FILENAME  . . . . . . . . . . . .  TEXT  block
@VERSION . . . . . . . . . . . . .  TEXT  510


     59 Source  Lines
     59 Total   Lines
     15 Symbols

47222 + 347542 Bytes symbol space free

     0 Warning Errors
     0 Severe Errors
```

(b)

- Other information provided in the listing
  - Size of code segment and stack
    - What is the size of the code segment?
    - At what offset address does it begin? End?
- Names, types, and values of constants and variables
  - At what line of the program is the symbol “N” define?
  - What value is it assigned?
  - What is the offset address of the instruction that uses N?
- # lines and symbols used in the program
  - Why is the value of N given as 0010?
- # errors that occurred during assembly

3.1 Software : The Microcomputer Program

- Assembly language versus high-level language
- It is easier to write program with high-level language.
- Program written in assembly language usually takes up less memory space and executes much faster.
- Device service routines are usually written in assembly language.
- Assembly language is used to write those parts of the application that must perform real-time operations, and high-level language is used to write those parts that are not time critical.

Memory Models (new slide)

Model	Data	Code	Definition
Tiny*	near		CS=DS=SS
Small	near**	near	DS=SS
Medium	near**	far	DS=SS, multiple code segments
Compact	far	near	single code segment, multiple data segments
Large	far	far	multiple code and data segments
Huge	huge	huge	multiple code and data segments; single array may be >64 KB

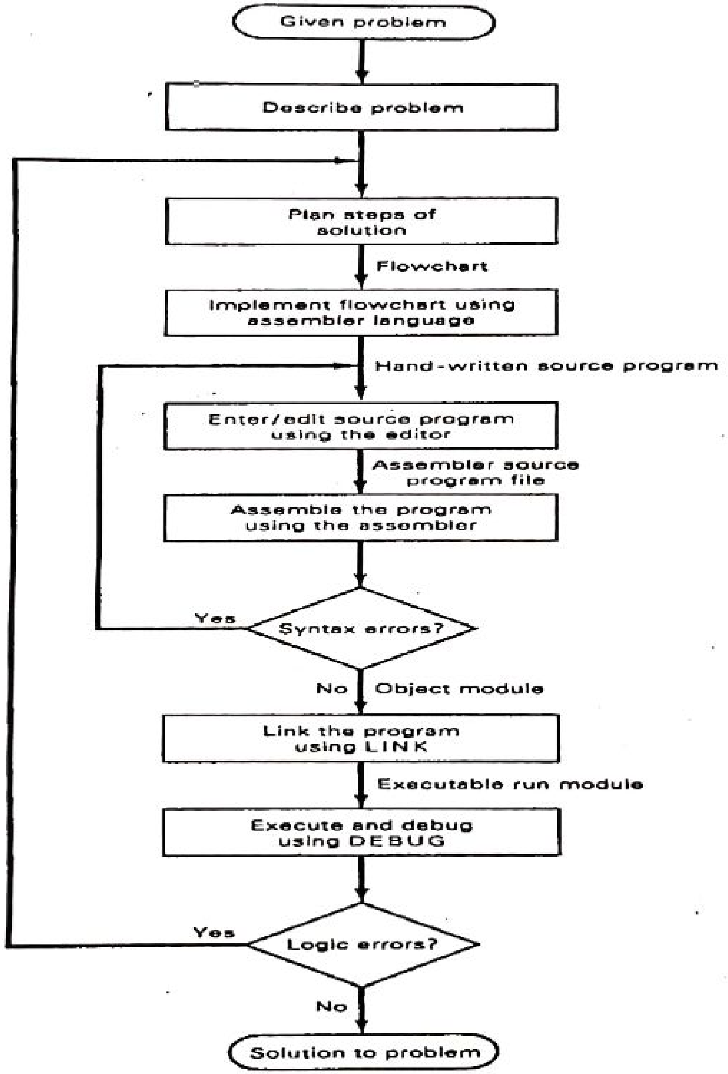
* In the Tiny model, all four segment registers point to the same segment.

** In all models with *near* data pointers, **SS** equals **DS**.

3.2 Assembly Language Programming Development on the PC

- Describing the problem
- Planning the solution
- Coding the solution with assembly language
- Creating the source program
- Assembling the source program into an object module
- Producing a run module
- Verifying the solution
- Programs and files involved in the program development cycle

3.2 Assembly Language Programming Development on the PC



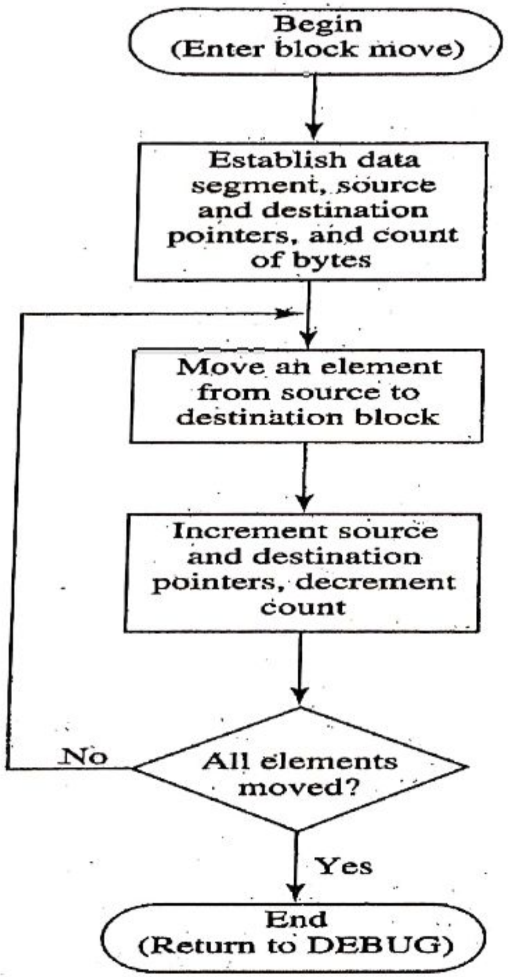
Program development cycle

3.2 Assembly Language Programming Development on the PC

- Describing the problem
  - Most applications are described with a written document called an ***application specification.***
- Planning the solution
  - A ***flowchart is an outline that both documents the*** operations that must be performed by software to implement the planned solution and shows the sequence in which they are performed.

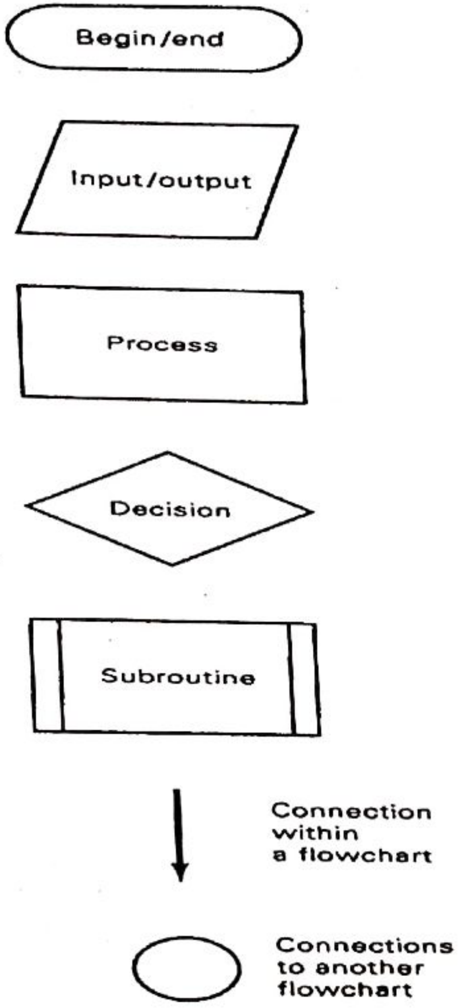
3.2 Assembly Language Programming Development on the PC

Flow chart of a block-move program



3.2 Assembly Language Programming Development on the PC

Commonly used flowchart symbols



3.2 Assembly Language Programming Development on the PC

- Coding the solution with assembly language
  - Two types of statements are used in the source program:
    - The ***assembly language instructions*** are used to tell the microprocessor what operations are to be performed to implement the application.
    - A ***directive*** is the instruction to the assembler program used to convert the assembly language program into machine code.

3.2 Assembly Language Programming Development on the PC

- Coding the solution with assembly language
  - The ***assembly language instructions***

[Example]

```
        MOV   AX, DATASEGMENT
        MOV   DS, AX
        MOV   SI, BLK1ADDR
        MOV   DI, BLK2ADDR
        MOV   CX, N
```

  - The ***directive***

[Example]

```
        BLOCK         PROC FAR
  or
        BLOCK         ENDP
```

3.2 Assembly Language Programming Development on the PC

- Creating the source program
  - The EDIT editor
  - The Notepad editor in Windows
  - The Microsoft PWB (Programmer's Work Bench)
- Assembling the source program into an object module
  - The Microsoft MASM assembler
  - The Microsoft PWB (Programmer's Work Bench)
    - Used for writing and compiling code
  - The ***assembler source file*** **and the** ***object module***

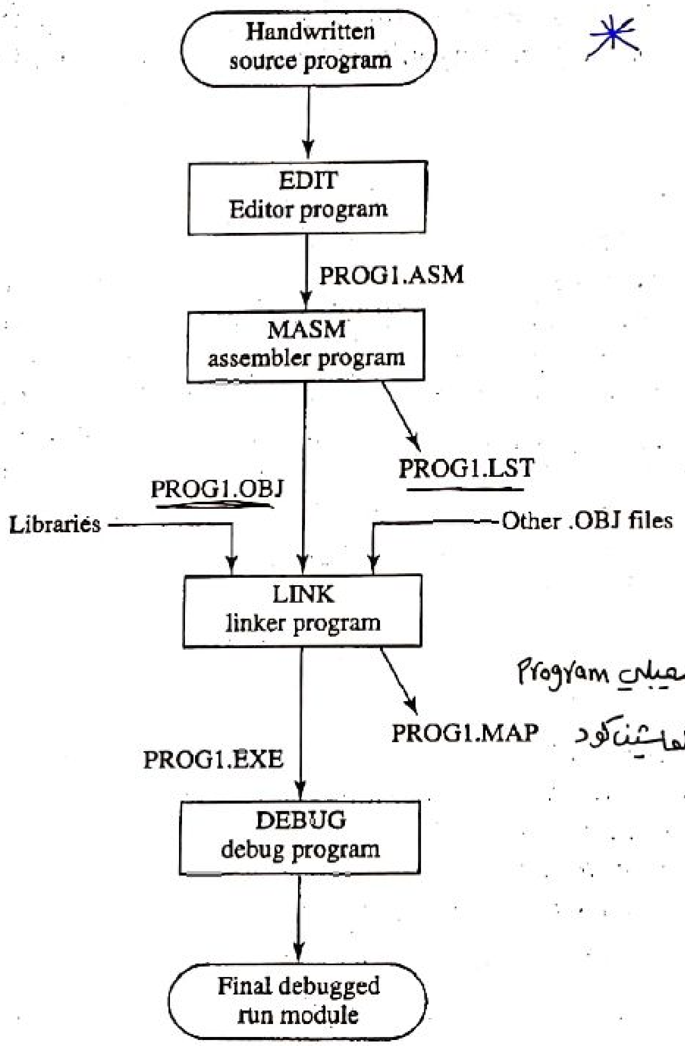
3.2 Assembly Language Programming Development on the PC

- Producing a run module
  - The object module must be processed by the LINK program to produce an executable ***run module.***
- Verifying the solution
- Programs and files involved in the program development cycle

PROG1.ASM	(Editor)
PROG1.OBJ	(Assembler)
PROG1.LST	(Assembler)
PROG1.EXE	(Linker)
PROG1.MAP	(Linker)

3.2 Assembly Language Programming Development on the PC

The development programs and users files



*

بتكتب كود البرامج يدوياً

↓

بتعمله editing

↓

بتخزنه على Program.ASM

↓

MASM

الها 2 ~~Programs~~ outputs

Program OBJ

Program.LST

↓

بتكوّنوا من الماشين كود الأسمبلي Program

وبختلفوا عن بعض بشكل عرض الماشين كود

↓

Linker Program

بعمل maping للبرنامج على الميموري

↓

debug

↓

run

3.3 The Instruction Set

- * The instruction set of a microprocessor defines the basic operations that a programmer can specify to the device to perform
- Instruction set groups عددهم 6
  - Data transfer instructions (moving data between registers and/or memory locations).
  - Arithmetic instructions (ADD,SUB,DIV,MUL,INC,DEC)
  - Logic instructions (AND,OR,XOR,ROL,NOT)
  - String manipulation instructions (REP,MOVS,LODS,STDS)
  - Control transfer instructions (JMP,RET,LOOP)
  - Processor control instructions (CLC,CMC,STC,HLT,)

بتحكم في بعض ال flags الموجودين

في (status reg) زي clear flag

--- زي complement flag

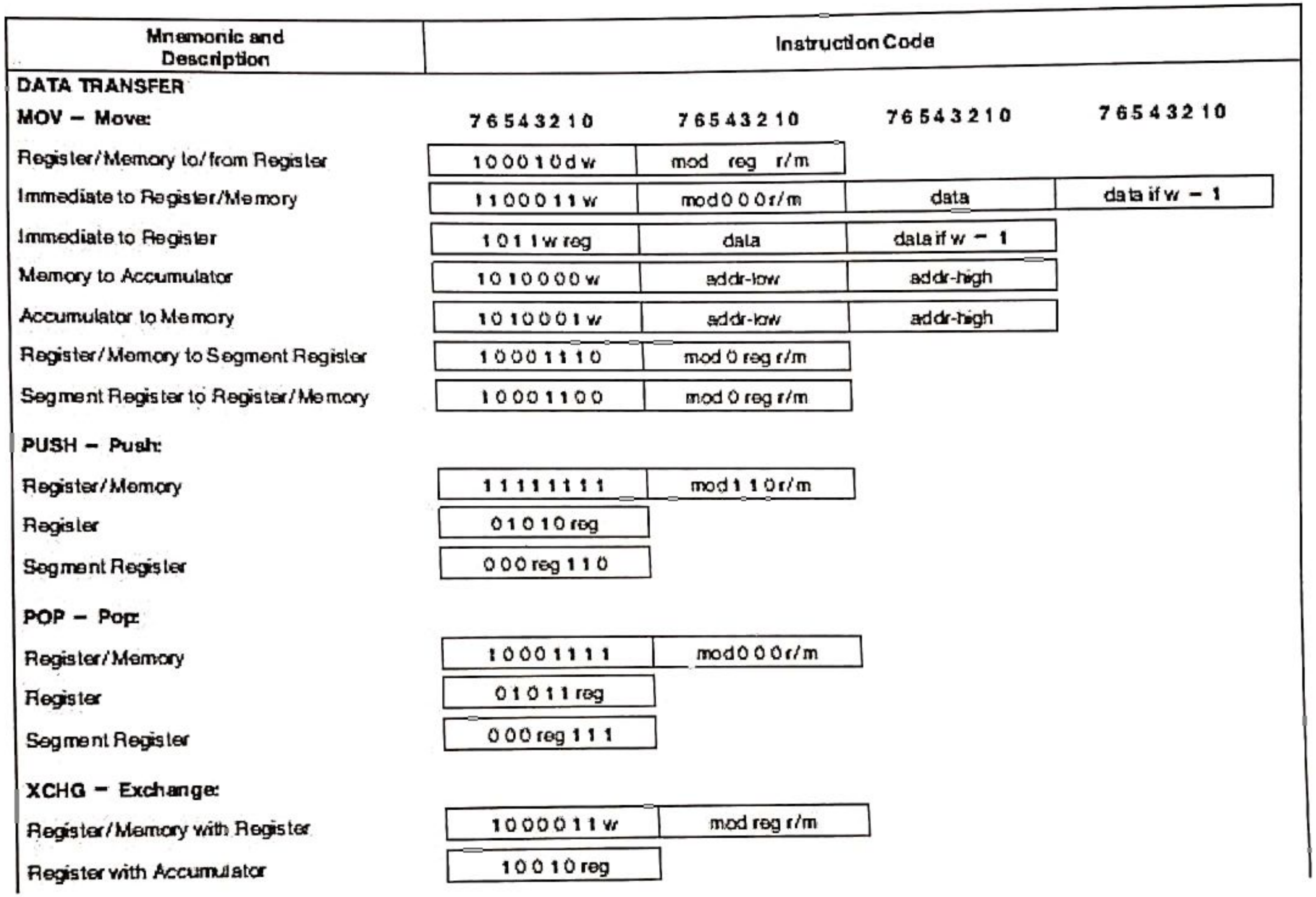
هو عبارة عن (Very semple instruction) مهمته نقل الداتا :- Mov instruction
بين الريجستر والميموري .
فينقل بين الريجستر والميموري أو العكس
أو ينقل بين الريجسترات
ولكن لا ينقل من الميموري إلى الميموري

3.3 The Instruction Set

- In assembly language each instruction is represented by a "mnemonic" that describes its operation and is called its "operation code (opcode)"
  - MOV = move → data transfer
  - ADD = add → arithmetic
  - JMP = unconditional jump → control transfer
- Operands: Identify whether the elements of data to be processed are in registers or memory
  - Source operand- location of one operand to be processed
  - Destination operand—location of the other operand to be processed and the location of the result

3.3 The Instruction Set

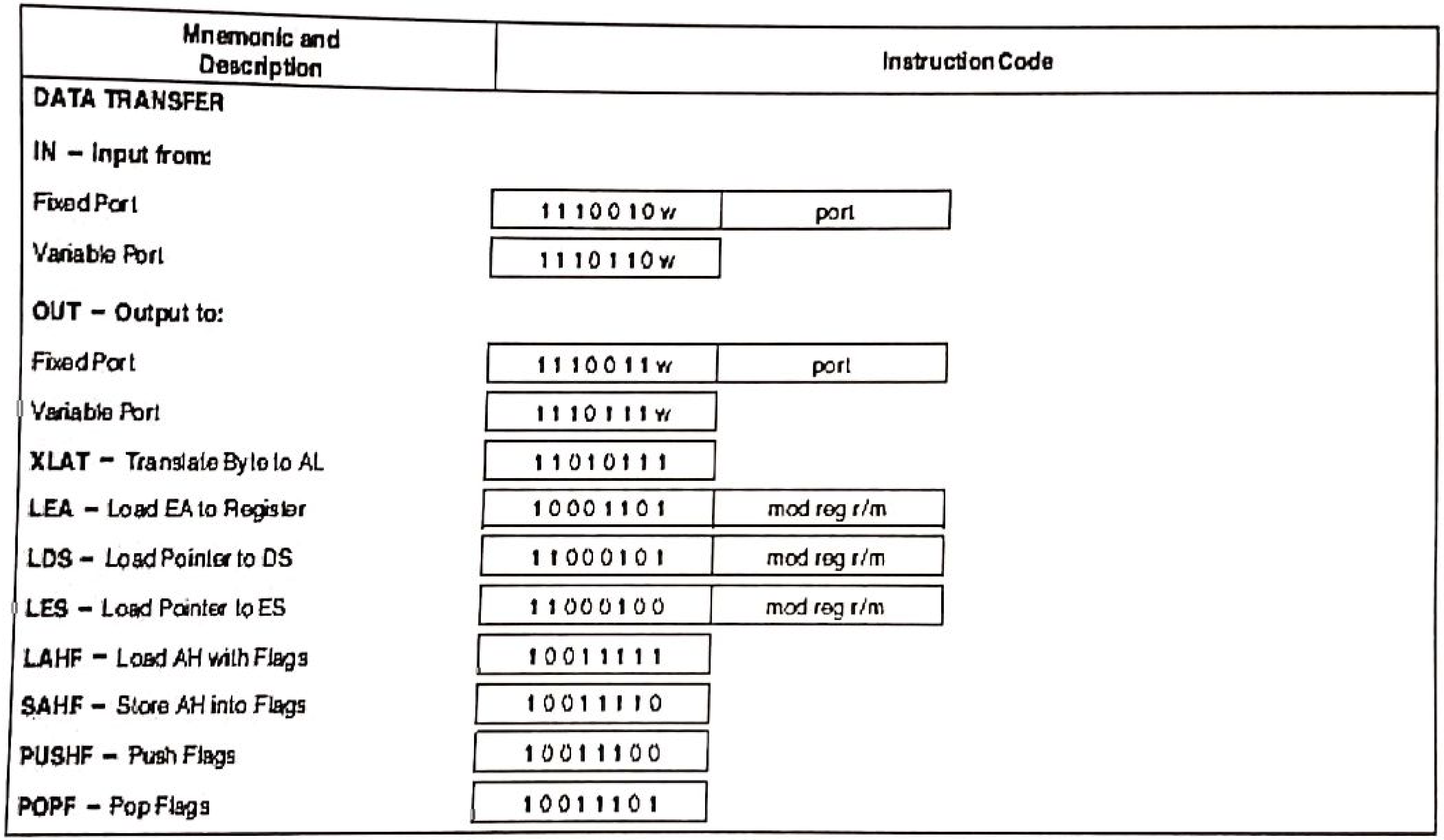
- Data transfer instructions



Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV – Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH – Push:				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
POP – Pop:				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
XCHG – Exchange:				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			

3.3 The Instruction Set

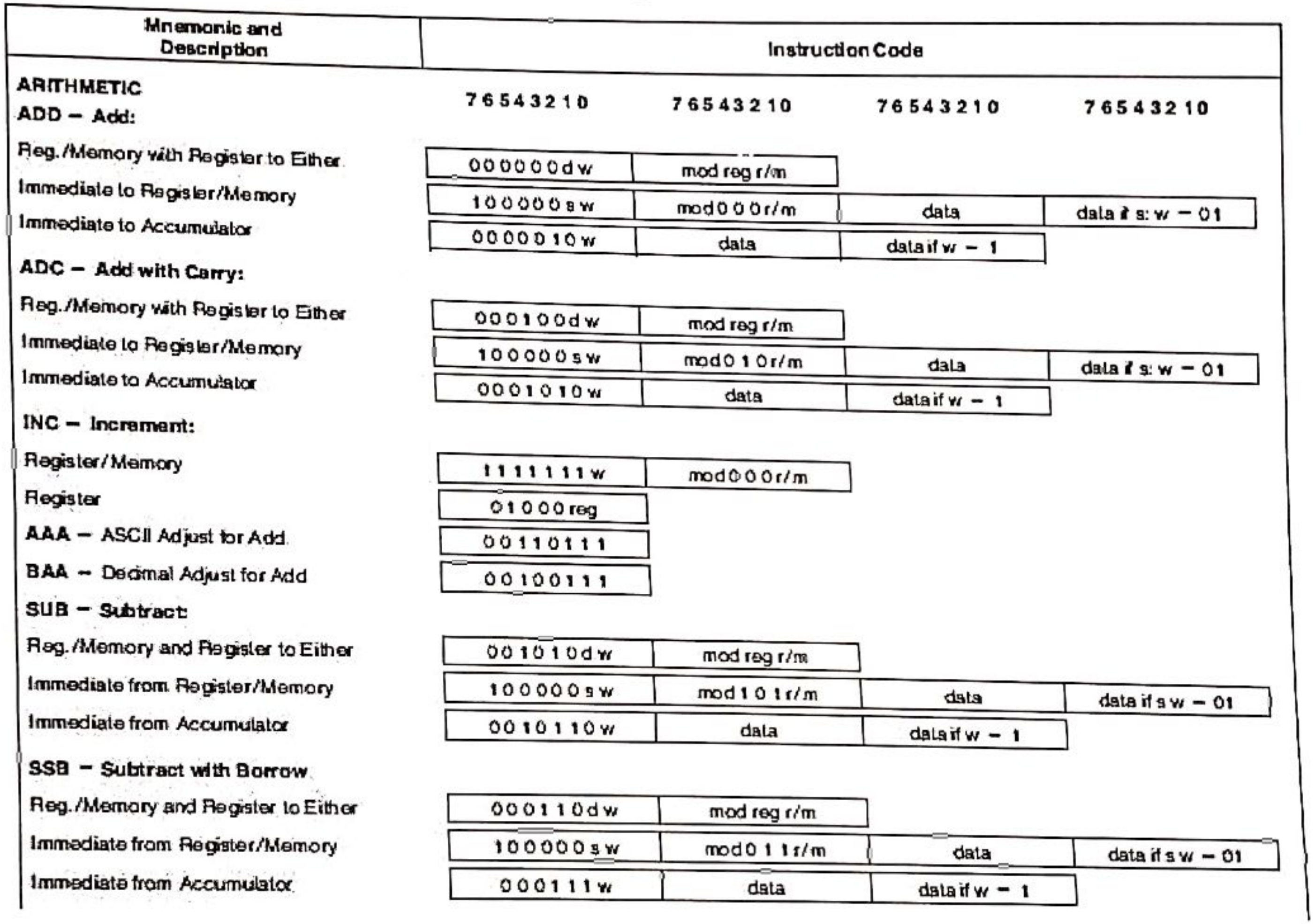
- Data transfer instructions



Mnemonic and Description	Instruction Code	
DATA TRANSFER		
IN – Input from:		
Fixed Port	1 1 1 0 0 1 0 w	port
Variable Port	1 1 1 0 1 1 0 w	
OUT – Output to:		
Fixed Port	1 1 1 0 0 1 1 w	port
Variable Port	1 1 1 0 1 1 1 w	
XLAT – Translate Byte to AL	1 1 0 1 0 1 1 1	
LEA – Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m
LDS – Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m
LES – Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m
LAHF – Load AH with Flags	1 0 0 1 1 1 1 1	
SAHF – Store AH into Flags	1 0 0 1 1 1 1 0	
PUSHF – Push Flags	1 0 0 1 1 1 0 0	
POPF – Pop Flags	1 0 0 1 1 1 0 1	

3.3 The Instruction Set

- Arithmetic instructions



Mnemonic and Description	Instruction Code			
ARITHMETIC	76543210	76543210	76543210	76543210
ADD − Add:				
Reg./Memory with Register to Either	000000dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 0 0 0 r/m	data	data if s: w − 01
Immediate to Accumulator	0000010w	data	data if w − 1	
ADC − Add with Carry:				
Reg./Memory with Register to Either	000100dw	mod reg r/m		
Immediate to Register/Memory	100000sw	mod 0 1 0 r/m	data	data if s: w − 01
Immediate to Accumulator	0001010w	data	data if w − 1	
INC − Increment:				
Register/Memory	1111111w	mod 0 0 0 r/m		
Register	01000 reg			
AAA − ASCII Adjust for Add	00110111			
BAA − Decimal Adjust for Add	00100111			
SUB − Subtract:				
Reg./Memory and Register to Either	001010dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 1 0 1 r/m	data	data if s w − 01
Immediate from Accumulator	0010110w	data	data if w − 1	
SSB − Subtract with Borrow				
Reg./Memory and Register to Either	000110dw	mod reg r/m		
Immediate from Register/Memory	100000sw	mod 0 1 1 r/m	data	data if s w − 01
Immediate from Accumulator	000111w	data	data if w − 1	

3.3 The Instruction Set

- Arithmetic instructions

Mnemonic and Description	Instruction Code			
ARITHMETIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
DEC – **Decrement:**				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG – Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP – **Compare:**				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS – ASCII Adjust for Subtract	0 0 1 1 1 1 1 1			
DAS – Decimal Adjust for Subtract	0 0 1 0 1 1 1 1			
MUL – Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL – Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM – ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV – Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV – Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD – ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW – Convert Byte to Word	1 0 0 1 1 0 0 0			
CWD – Convert Word to Double Word	1 0 0 1 1 0 0 1			

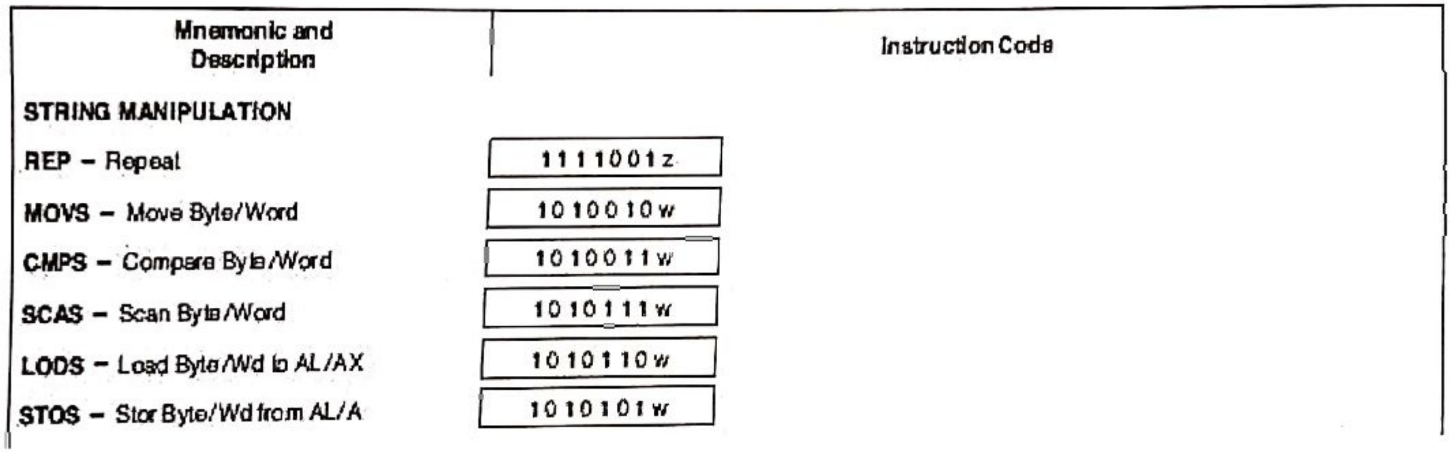
3.3 The Instruction Set

- Logic instructions

Mnemonic and Description	Instruction Code			
LOGIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
NOT – Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL – Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR – Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR – Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL – Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR – Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL – Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR – Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND – And:				
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w – 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w – 1	
TEST – And Function to Flags, No Result:				
Register/Memory and Register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w – 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w – 1	
OR – Or:				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w – 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w – 1	
XOR – Exclusive or:				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w – 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w – 1	

3.3 The Instruction Set

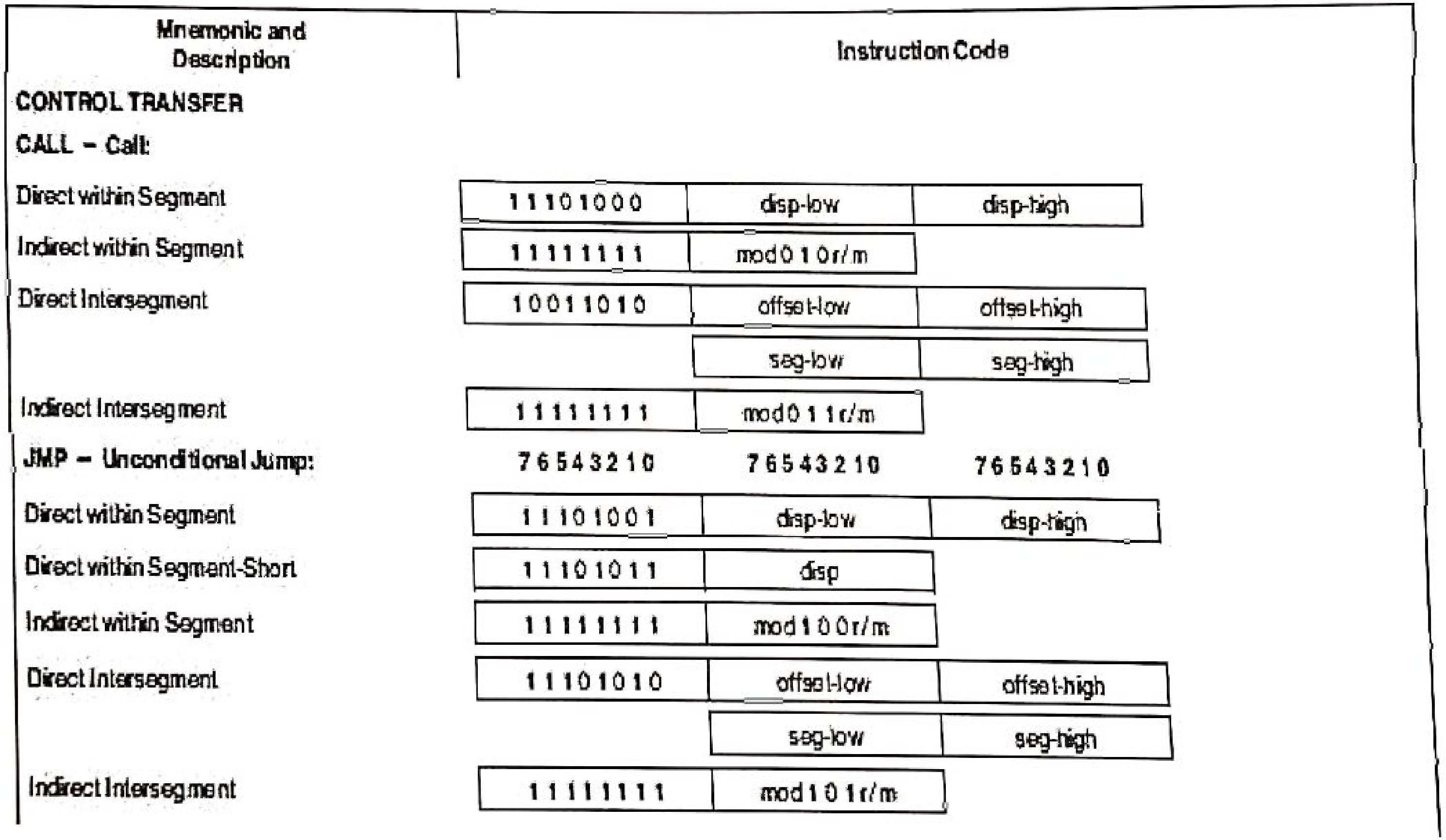
- String manipulation instructions



Mnemonic and Description	Instruction Code
STRING MANIPULATION	
REP – Repeat	1111001z
MOVS – Move Byte/Word	1010010w
CMPS – Compare Byte/Word	1010011w
SCAS – Scan Byte/Word	1010111w
LODS – Load Byte/Wd to AL/AX	1010110w
STOS – Stor Byte/Wd from AL/A	1010101w

3.3 The Instruction Set

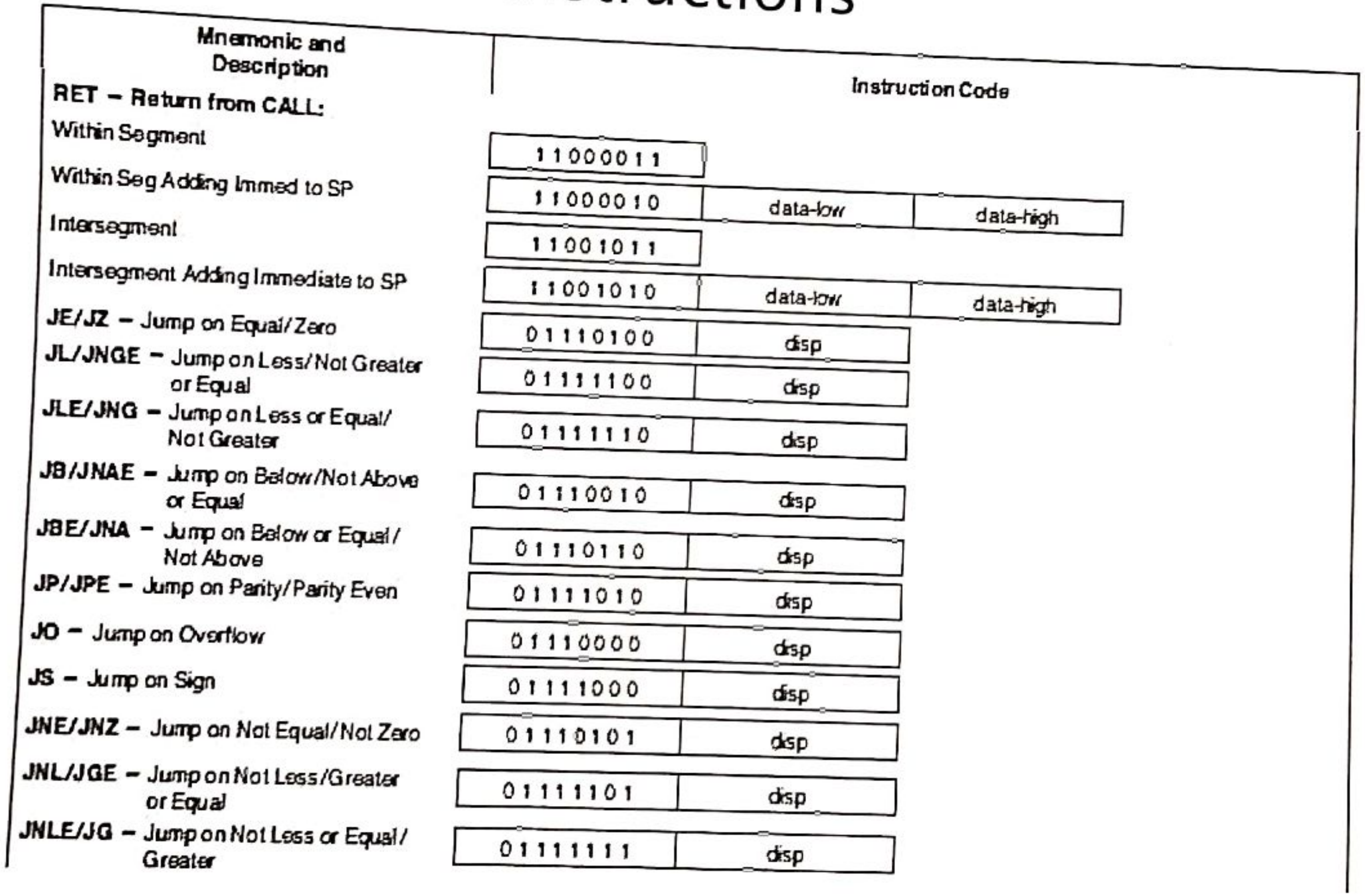
- Control transfer instructions



Mnemonic and Description	Instruction Code		
CONTROL TRANSFER			
CALL – Call:			
Direct within Segment	11101000	disp-low	disp-high
Indirect within Segment	11111111	mod 0 1 0 r/m	
Direct Intersegment	10011010	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	11111111	mod 0 1 1 r/m	
JMP – Unconditional Jump:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within Segment	11101001	disp-low	disp-high
Direct within Segment-Short	11101011	disp	
Indirect within Segment	11111111	mod 1 0 0 r/m	
Direct Intersegment	11101010	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	11111111	mod 1 0 1 r/m	

3.3 The Instruction Set

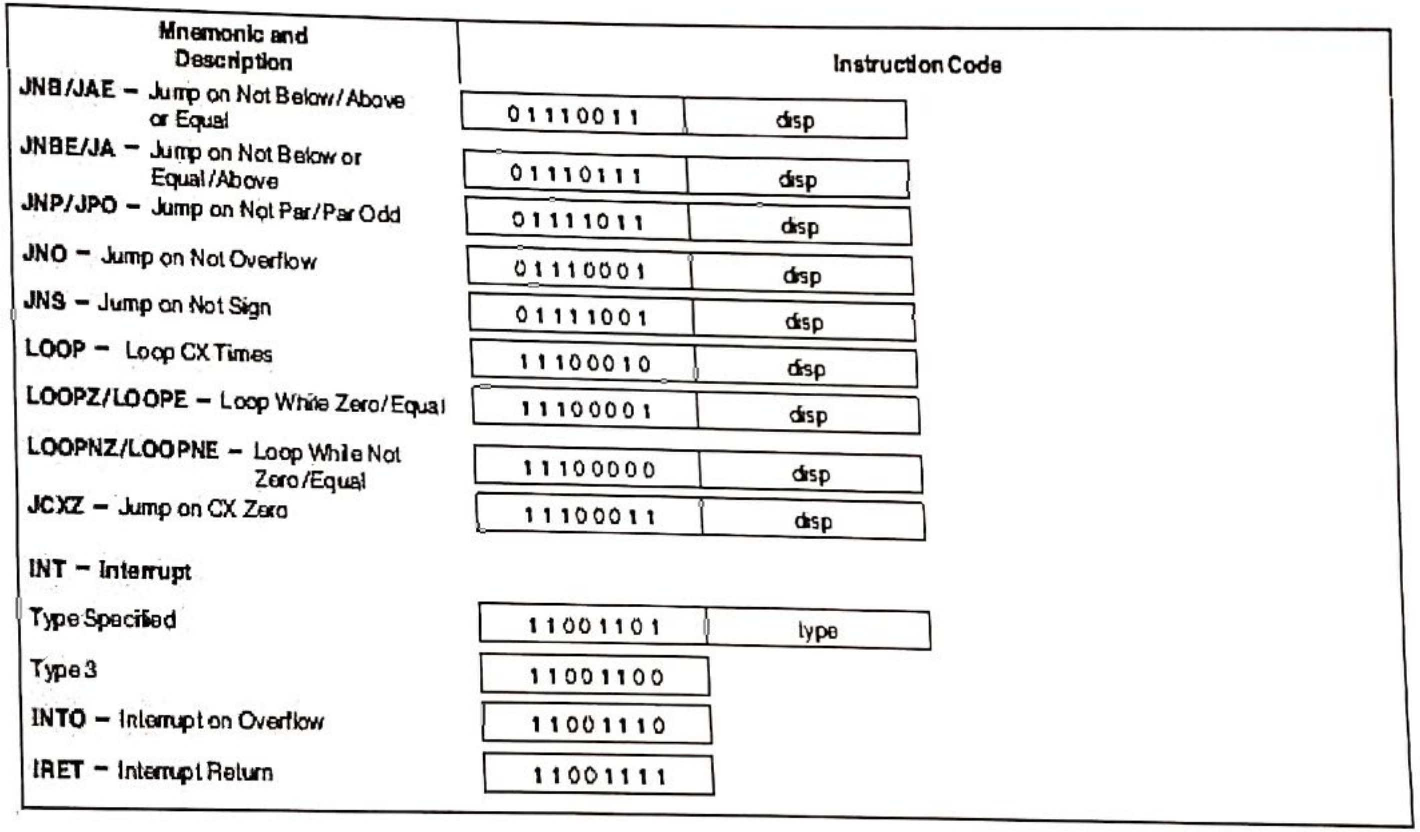
- Control transfer instructions



Mnemonic and Description	Instruction Code		
RET – Return from CALL:			
Within Segment	11000011		
Within Seg Adding Immed to SP	11000010	data-low	data-high
Intersegment	11001011		
Intersegment Adding Immediate to SP	11001010	data-low	data-high
JE/JZ – Jump on Equal/Zero	01110100	disp	
JL/JNGE – Jump on Less/Not Greater or Equal	01111100	disp	
JLE/JNG – Jump on Less or Equal/ Not Greater	01111110	disp	
JB/JNAE – Jump on Below/Not Above or Equal	01110010	disp	
JBE/JNA – Jump on Below or Equal/ Not Above	01110110	disp	
JP/JPE – Jump on Parity/Parity Even	01111010	disp	
JO – Jump on Overflow	01110000	disp	
JS – Jump on Sign	01111000	disp	
JNE/JNZ – Jump on Not Equal/Not Zero	01110101	disp	
JNL/JGE – Jump on Not Less/Greater or Equal	01111101	disp	
JNLE/JG – Jump on Not Less or Equal/ Greater	01111111	disp	

3.3 The Instruction Set

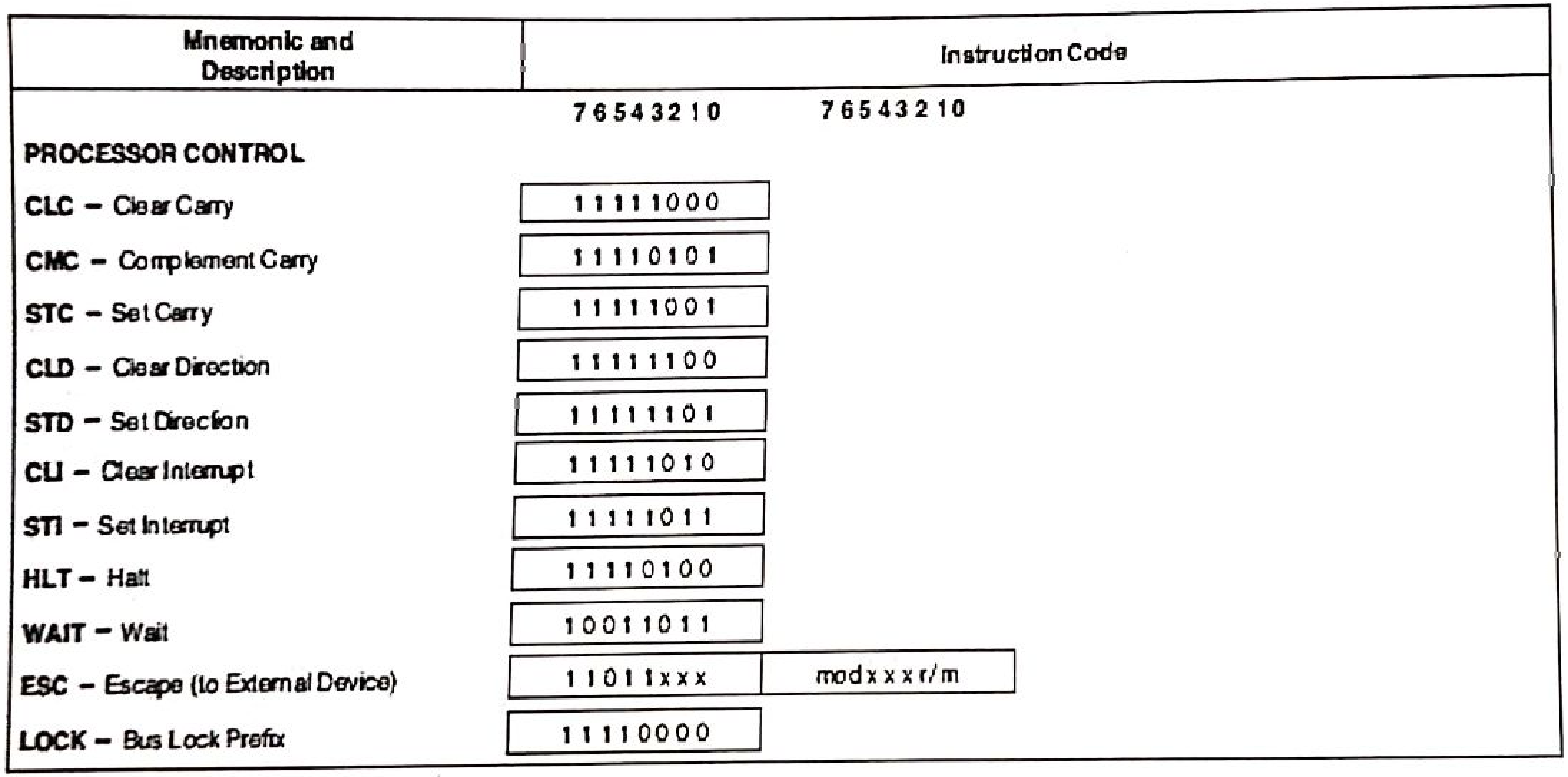
- Control transfer instructions



Mnemonic and Description	Instruction Code	
JNB/JAE – Jump on Not Below/Above or Equal	01110011	disp
JNBE/JA – Jump on Not Below or Equal/Above	01110111	disp
JNP/JPO – Jump on Not Par/Par Odd	01111011	disp
JNO – Jump on Not Overflow	01110001	disp
JNS – Jump on Not Sign	01111001	disp
LOOP – Loop CX Times	11100010	disp
LOOPZ/LOOPE – Loop While Zero/Equal	11100001	disp
LOOPNZ/LOOPNE – Loop While Not Zero/Equal	11100000	disp
JCXZ – Jump on CX Zero	11100011	disp
INT – **Interrupt**		
Type Specified	11001101	type
Type 3	11001100	
INTO – Interrupt on Overflow	11001110	
IRET – Interrupt Return	11001111	

3.3 The Instruction Set

- Process control instructions



Mnemonic and Description	Instruction Code	
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
PROCESSOR CONTROL		
CLC – Clear Carry	1 1 1 1 1 0 0 0	
CMC – Complement Carry	1 1 1 1 0 1 0 1	
STC – Set Carry	1 1 1 1 1 0 0 1	
CLD – Clear Direction	1 1 1 1 1 1 0 0	
STD – Set Direction	1 1 1 1 1 1 0 1	
CLI – Clear Interrupt	1 1 1 1 1 0 1 0	
STI – Set Interrupt	1 1 1 1 1 0 1 1	
HLT – Halt	1 1 1 1 0 1 0 0	
WAIT – Wait	1 0 0 1 1 0 1 1	
ESC – Escape (to External Device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK – Bus Lock Prefix	1 1 1 1 0 0 0 0	

typicall instruction عبارة
بأخذ ≧2 operand

3.4 The MOV Instruction

- The move (MOV) instruction is used to transfer a byte or a word of data from a source operand to a destination operand.
- e.g. MOV DX, CS
  MOV [SUM], AX

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D,S	(S) → (D)	None

op code

transfaring data from source to ~~desti~~ distination

(a)

Mov instr لأنه مش arthmetic ولا logical فما بينتج عنها result .

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

- Note that the MOV instruction cannot transfer data directly between external memory.

[1] Register $\xrightarrow{\text{مثال}}$ Mov BX, AX :- source أحد الحالات التي بكون فيها ال ←
[2] memory $\xrightarrow{\text{مثال}}$ Mov CX, [1000] ⇐ يعني الداتا بتكون بأحد جدول الحالات
[3] Immediate $\xrightarrow{\text{مثال}}$ Mov DX, 1010

على الصفحة التالية

Allowed operands for MOV instruction

destination (لها حالتان) : ① register

② memory

3.4 The MOV Instruction

يعني اعمل كوبي من CS وتتخزن في DX .

- MOV DX, CS

Mov AX , 2020 → تعني انقل 2020 وخزنها في AX | 20 | 20 |

دايماً هاي تدل على الميموري .

* Mov Ax , [2020] → تعني اذهب للميموري على العنوان 2020 وخذ منه الداتا أي xxyy وانقلها على ~~[illegible]~~ الريجيستر AX .

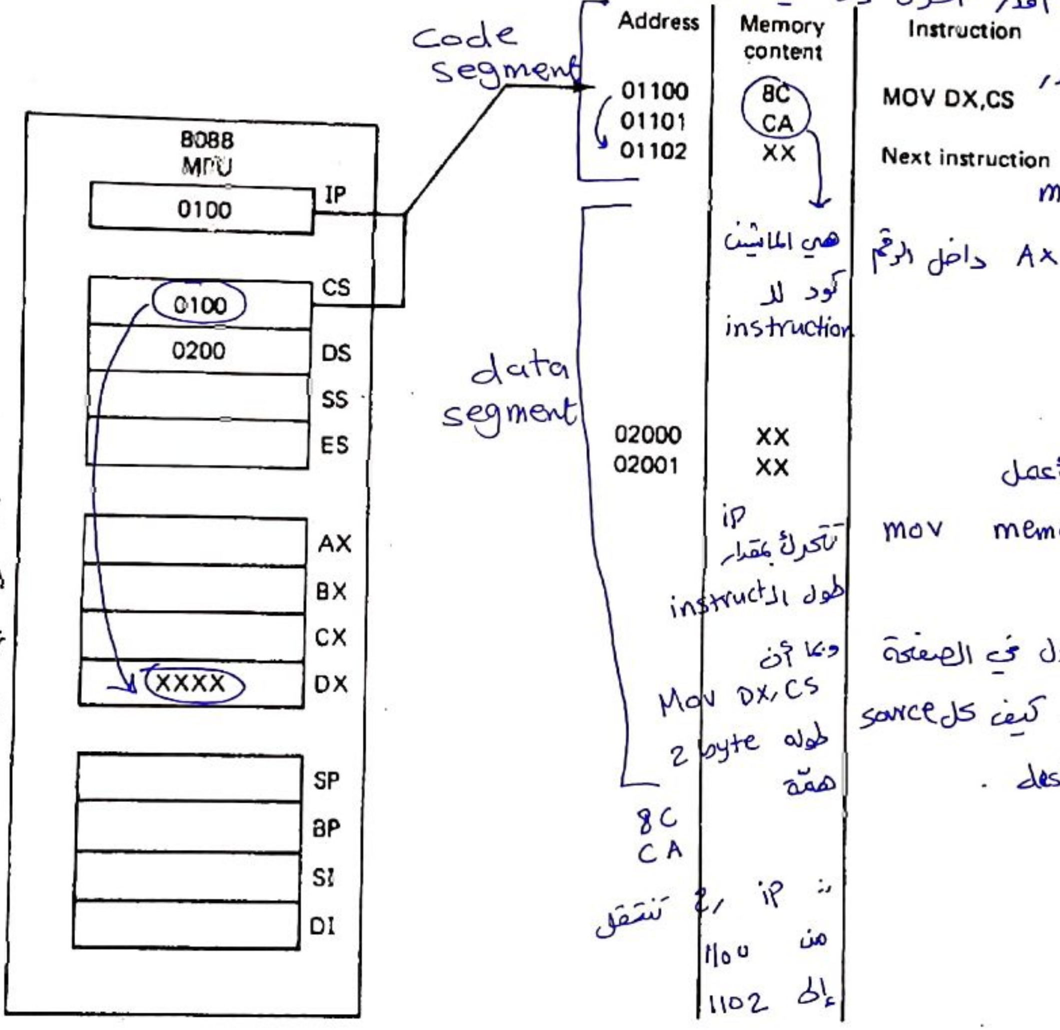
ليش immediate لسيش من ضمن الحالات؟

لأنه لا يمكن أقدر أخزن داتا في number

مثلاً مستحيل أقدر أعمل هيك

mov 1000, Ax

ما بقدر أخزن Ax داخل الرقم 10000 .



هي الماشين كود للـ instruction

ip تتحرك بمقدار طول الـ instruct

وبما أن Mov DX, CS طوله 2 byte همّة 8C CA

∴ ip رح تنتقل من 1100 إلى 1102

كمان لا يمكن أعمل mov memory , memory

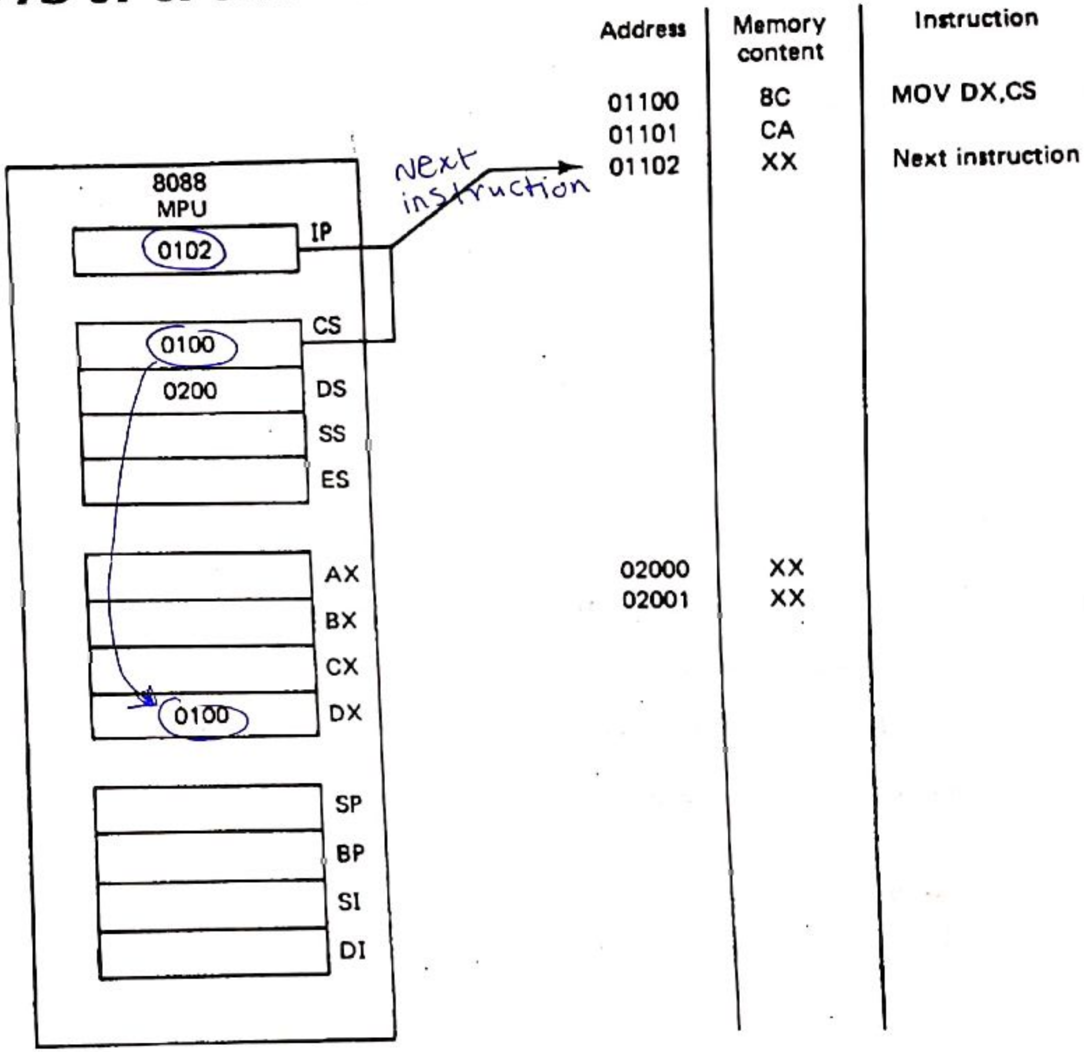
انظر إلى الجدول في الصفحة السابقة ولاحظ كيف كل source تقابلها destination .

Before execution

(c)

3.4 The MOV Instruction

- MOV DX, CS



(d)

After execution

3.5 Addressing Modes

:- هيّة الحالات اللي بكون عليهم الـ source أو الـ operand destination

operand destination → memory, ~~imm~~, reg

source → imm, memory, reg

- Addressing mode is a method of specifying an operand & categorized into three types:
  - ① Register operand addressing mode
  - ② Immediate operand addressing mode
  - ③ Memory operand addressing mode
    - Direct addressing mode
    - Register indirect addressing mode
    - Based addressing mode
    - Indexed addressing mode
    - Based-indexed addressing mode

الميموري مقسومة لخمس أنواع

[ليست للحفظ].

* mov Ax ,Bx → ماهو address mode for souce ?

الجواب ريجستر

ماهو address mod for destination?

الجواب رجستر

* Mov Bx , 2000 → ماهو address mod for source?

الجواب imm

ماهو // للـ destination؟

الجواب reg

* Mov Ax , [1000]

address mod for destination → reg

address mod for source → memory

3.5 Addressing Modes

- Register operand addressing mode:

  The operand to be accessed is specified as residing in an internal register of 8088.

  e.g. MOV $\underbrace{\text{AX}}_{\text{reg}}$, $\underbrace{\text{BX}}_{\text{reg}}$
- Only the data registers can be accessed as bytes or words
  - Ex. AL,AH → bytes
  - AX → word
- Index and pointer registers as words
  - Ex. SI → word pointer
- Segment registers only as w
  - Ex. DS → word pointer

Register	Operand sizes	
	Byte (Reg 8)	Word (Reg 16)
Accumulator	AL, AH	AX
Base	BL, BH	BX
Count	CL, CH	CX
Data	DL, DH	DX
Stack pointer	—	SP
Base pointer	—	BP
Source index	—	SI
Destination index	—	DI
Code segment	—	CS
Data segment	—	DS
Stack segment	—	SS
Extra segment	—	ES

3.5 Addressing Modes

Register operand addressing mode

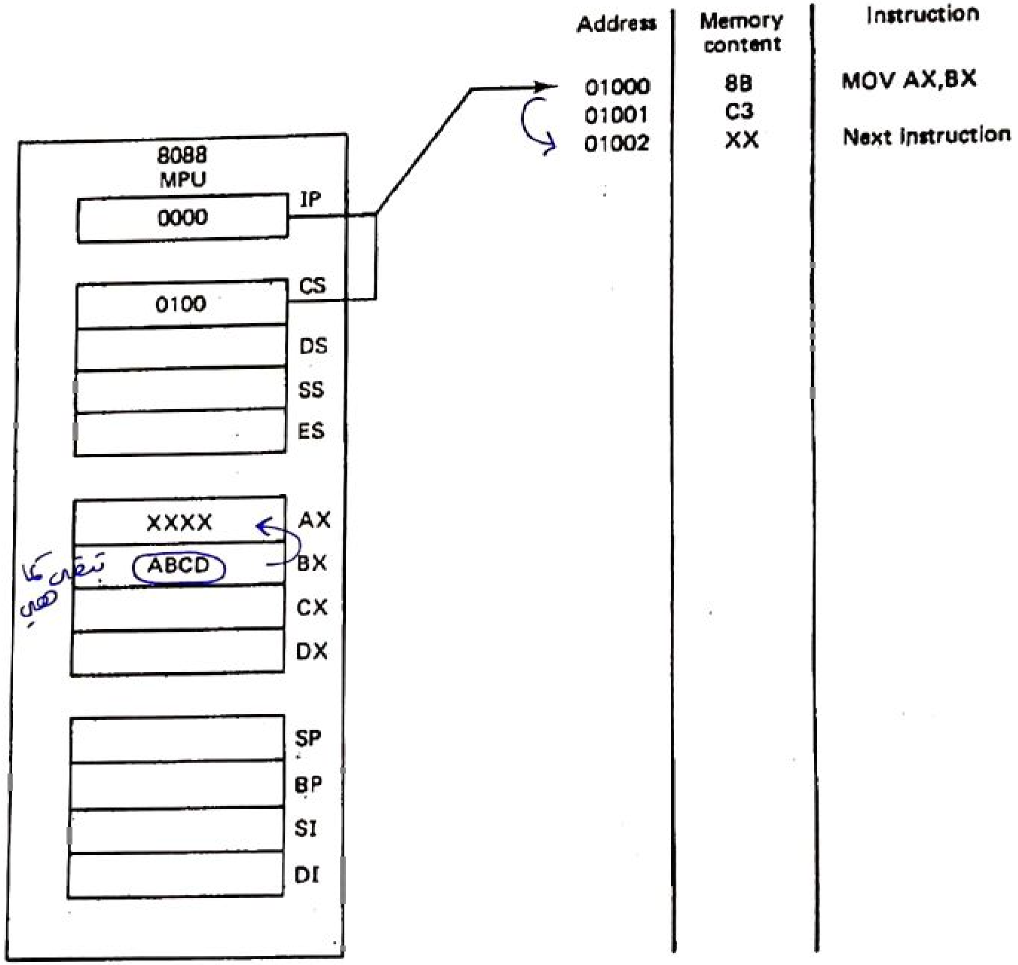
- Example

MOV AX,BX

Source = BX → word data
Destination = AX → word data
Operation: (BX) → (AX)

- State before fetch and execution

CS:IP = 0100:0000 = 01000H
Move instruction code = 8BC3H
(01000H) = 8BH
(01001H) = C3H
(BX) = ABCDH
(AX) = XXXX → don't care state



Address	Memory content	Instruction
01000	8B	MOV AX,BX
01001	C3	
01002	XX	Next instruction

Before execution (a)

3.5 Addressing Modes

- Register operand addressing mode

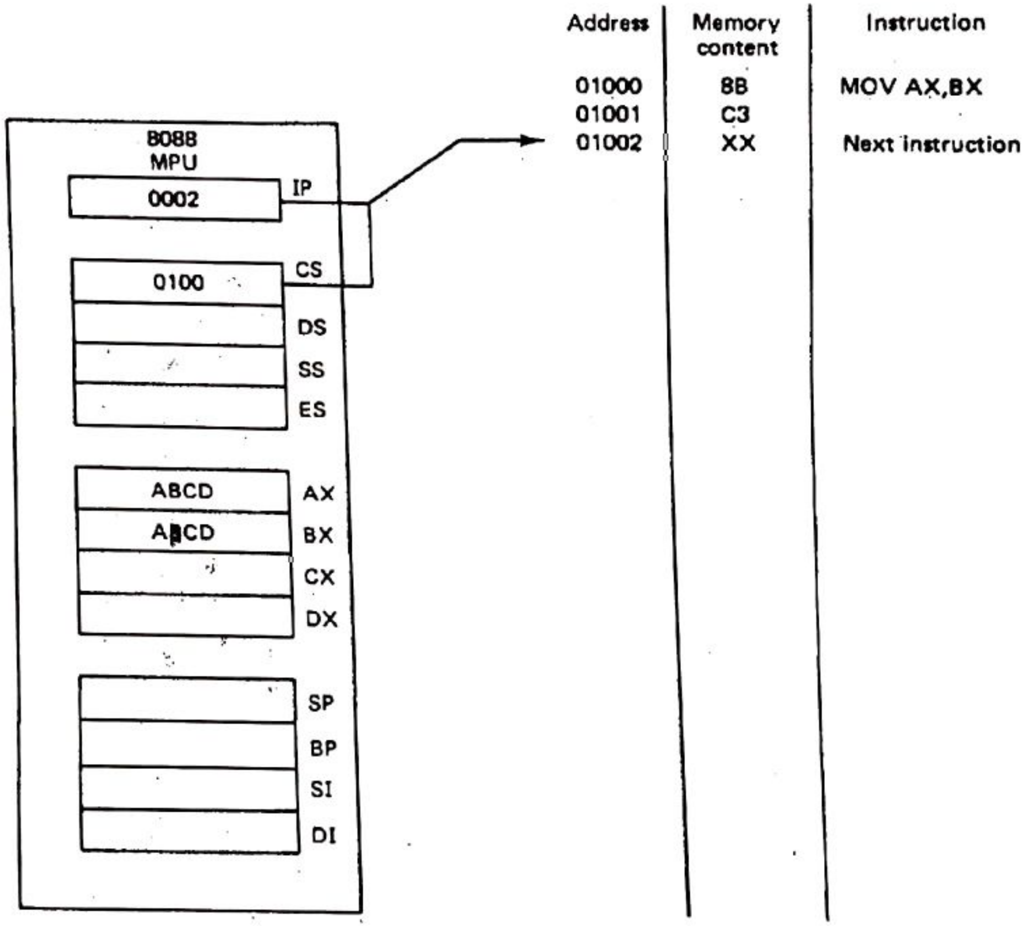
- Example (continued)
- State after execution

CS:IP = 0100:0002 = 01002H

01002H → points to next sequential instruction

(BX) = ABCDH

(AX) = ABCDH → Value in BX copied into AX



Address	Memory content	Instruction
01000	8B	MOV AX,BX
01001	C3	
01002	XX	Next instruction

(b)

After execution

3.5 Addressing Modes

- Immediate operand addressing mode
  - Operand is coded as part of the instruction
  - Applies only to the source operand
  - Destination operand uses register addressing mode
- Types
  - Imm8 = 8-bit immediate operand
  - Imm16 = 16-bit immediate operand
  - General instruction structure and operation

MOV Rx,ImmX

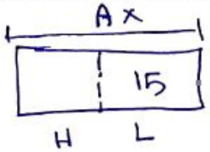
ImmX → (Rx)

Before execution

* ما بصير أعمل mov إلا بين 2 size متساويات.
فما بقدر أعمل mov BL , Ax
فإذا كان السورس بايت لازم يكون destination كمان byte
فالصحيح هو mov BL, AL

3.5 Addressing Modes

- Immediate operand addressing mode



- Example

MOV AL,15H

Source = Imm8 → immediate byte data

Destination = AL → Byte of data

- Operation: (Imm8) → (AL)

- State before fetch and execution

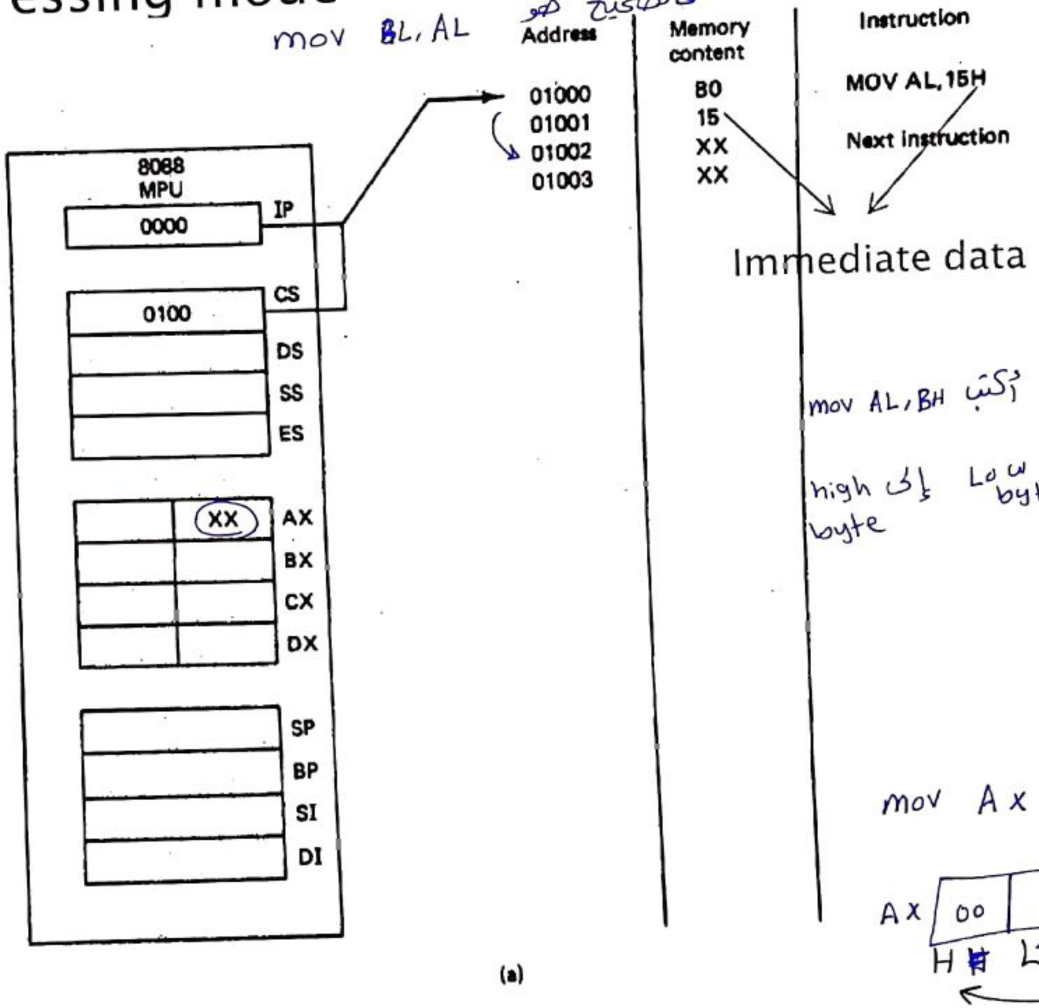
CS:IP = 0100:0000 = 01000H

Move instruction code = B015H

(01000H) = B0H

(01001H) = 15H → Immediate data

What about MOV AL,1515H ?



وكمان بصير أكتب mov AL, BH
لأنه من Low byte إلى high byte

Before execution

مثال :-

mov A x , 15

A x 00 15
H L

هي بالأصل
0015
H L

3.5 Addressing Modes

- Immediate operand addressing mode

- Example (continued)
- State after execution

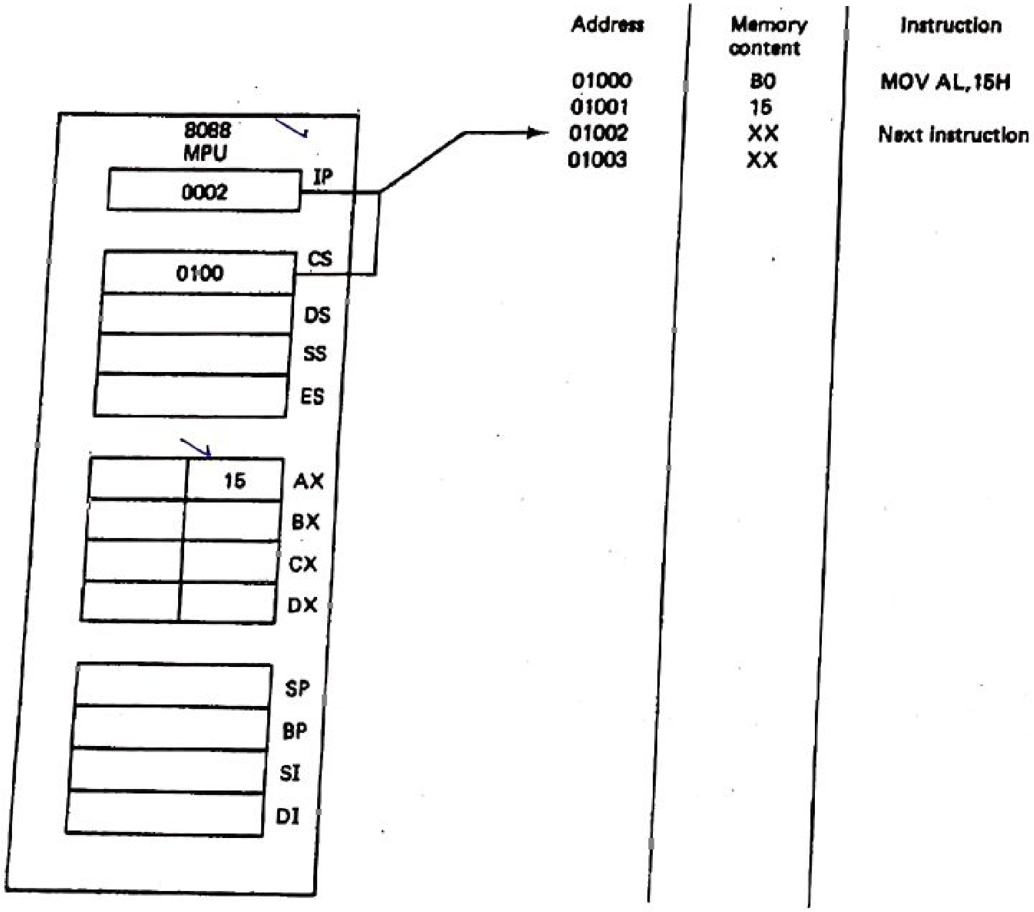
(AH) = XX → don't care state

What about MOV AL,1515H ?

هاي الحالة not Valied

لأنه ما في مساحة أخزن

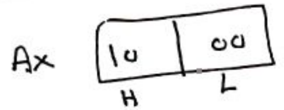
1515 ب AL



Address	Memory content	Instruction
01000	B0	MOV AL,15H
01001	15	
01002	XX	Next instruction
01003	XX	

(b)

After execution



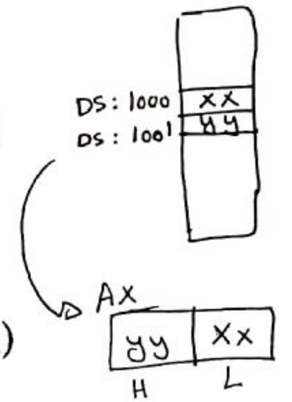
⇐ Mov Ax , 1000 ↳ imm

3.5 Addressing Modes

Mov Ax , [1000] ↳ memory ⇒

⇐ بروح على الأدريس يلي offset تبعه 1000 وبجيب الداتا منه

1000 ← هاد اسمه offset أو effective address



- * Memory addressing modes

  To reference an operand in memory, the 8088 must calculate the physical address (PA) of the operand and then initiate a read or write operation to this storage location.

Physical Address (PA) = Segment Base Address (SBA) + Effective Address (EA)

[offset]

PA = SBA : EA

PA = Segment base : Base + Index + Displacement

Base

offset

$$PA = \left\{\begin{matrix}CS\\SS\\DS\\ES\end{matrix}\right\} : \left\{\begin{matrix}BX\\ \text{أو}\\ BP\end{matrix}\right\} \overset{\text{أو}}{+} \left\{\begin{matrix}SI\\ \text{أو}\\ DI\end{matrix}\right\} \overset{\text{أو}}{+} \left\{\begin{matrix}\text{8-bit displacement}\\\text{16-bit displacement}\end{matrix}\right\}$$

ما بصير يجتمعوا Si مع Di وBx مع BP و 16 مع 8 bit

EA = Base + Index + Displacement

بس بصير يكون Bx مع Si أو BP مع Si مع 8 bit ---

Physical and effective address computation for memory operands

Where:

SBA = Segment base address

EA = Effective address (offset)

- **Components of a effective address**
  - **Base → base registers BX or BP**
  - **Index → index register SI or DI**
  - **Displacement → 8 or 16-bit displacement**
  - **Not all elements are used in all computations—results in a variety of addressing modes**

مثال:-

Mov AX , [SI + 10] عادي ✓

Mov Ax , [Bx + SI] عادي ✓

Mov Ax, [SI] عادي ✓

اذا جوّا [ ] ما انذكر اسم base address

بكون هو DS تلقائياً ، لأنه احنا بدنا نجيب داتا من الميموري.

DS ↓ default base address

مثال :- Mov Ax , [BX+SI]

كيفا أحسب Phesycal add ؟

بما انه مش ذاكر اسم base add

فتلقائياً هي DS .

Memory addressing modes

Effective address (EA) DS : [IB+SI]

BX / BP   Si / Di   8 bit /16 bit

DS0

IB +
SI

Memory addressing mode	Example	Base	Index	Disp.
Direct	MOV CX,[1234] بس displasment			✓
Indirect	MOV AX,[SI]	✓	OR ✓	
Based أو بهاي الطريقة	MOV [BX]+ 1234H,AL MOV [BX+1234] .	✓		✓
Indexed	MOV AL,[SI]+1234H		✓	✓
Based-Index	MOV AH, [BX][SI]+1234H	✓	✓	✓

PA = SBA + EA

offset * EA = Base + Index + Displacement

وال Base reg

اذا مش محدد نفرضه DS

3.5 Addressing Modes

- Memory addressing modes – Direct addressing mode

PA = Segment base: Direct address

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} \text{Direct address} \end{Bmatrix}$$

The default segment register is DS

Computation of a direct memory address

e.g. MOV AX, [1234H]

direct هاد

- Similar to immediate addressing in that information coded directly into the instruction
- Immediate information is the effective address called the direct address
- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[DA] → immediate 8-bit or 16 bit displacement

[DA]: Displacement Address

- Segment base address is DS by default

  PA = DS:[DA]

- Segment override prefix (SEG) is required to enable use of another segment register

  PA = SEG:ES:[DA]

3.5 Addressing Modes

▸ Memory addressing modes - Direct addressing mode

- Example

MOV CX,[1234H]

- State before fetch and execution
- Instruction

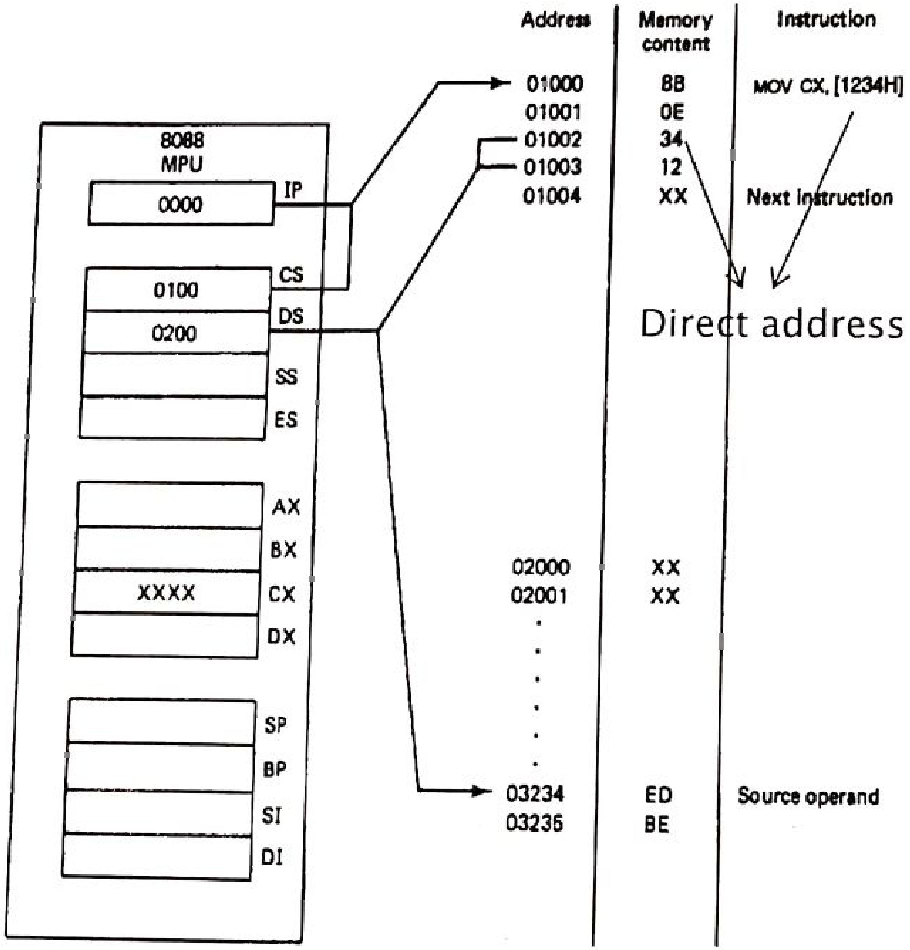
CS = 0100H
IP = 0000H
CS:IP = 0100:0000H = 01000H
(01000H,01001H) = Opcode = 8B0E
(01003H,01002) = DA = 1234H

- Source operand—direct addressing

DS = 0200H
DA = 1234H
PA = DS:DA = 0200H:1234H
= 02000H+1234H = 03234H
(03235H,03234H) = BEEDH

- Destination operand--register addressing

(CX) = XXXX → don't care state



Before execution (a)

3.5 Addressing Modes

- Memory addressing modes – Direct addressing mode

- Example (continued)
- State after execution

Instruction

CS:IP = 0100:0004 = 01004H

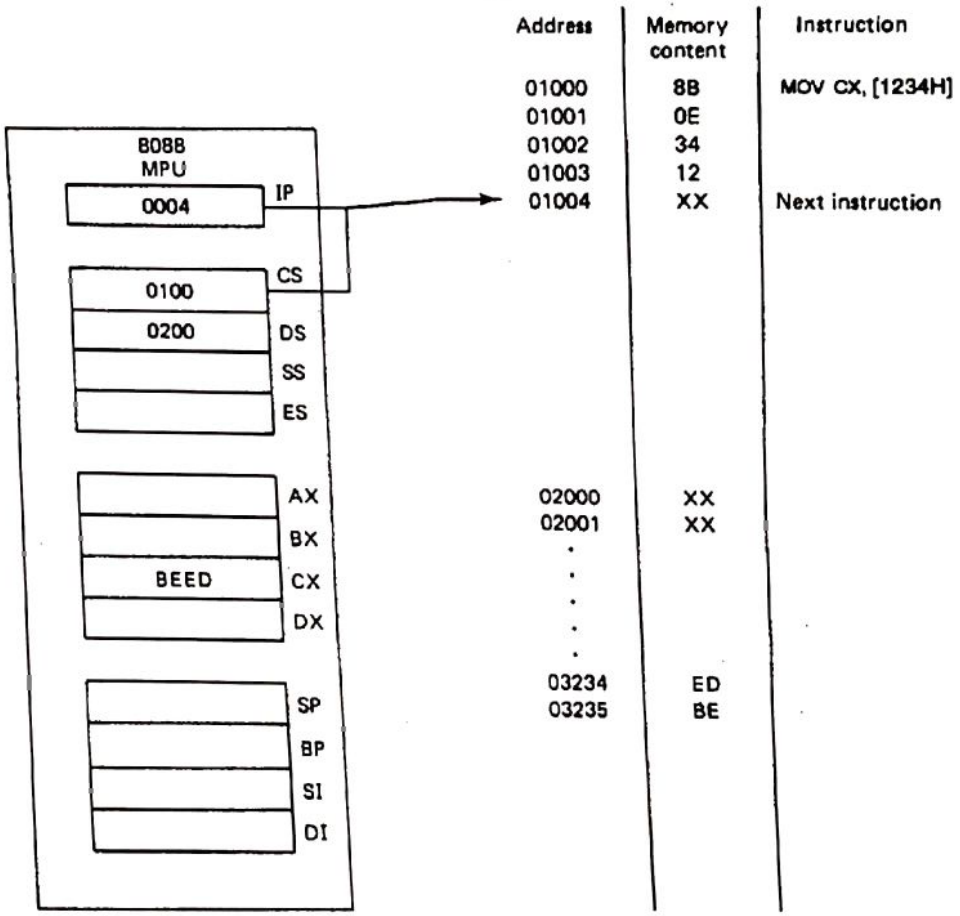
01004H → points to next Sequential instruction

- Source operand

(03235H,03234H) = BEEDH → unchanged

- Destination operand

(CX) = BEEDH



Address	Memory content	Instruction
01000	8B	MOV CX, [1234H]
01001	0E	
01002	34	
01003	12	
01004	XX	Next instruction
02000	XX	
02001	XX	
.		
.		
.		
.		
.		
03234	ED	
03235	BE	

(b)

After execution

3.5 Addressing Modes

- Memory addressing modes – Register indirect addressing mode

PA = Segment Base : Direct Address

$$PA = \left\{\begin{matrix}CS\\DS\\SS\\ES\end{matrix}\right\} : \left\{\begin{matrix}BX\\BP\\SI\\DI\end{matrix}\right\}$$

The default segment register is DS

Computation of an indirect memory address

e.g. MOV AX, [SI]

- Similar to direct addressing in that the affective address is combined with the contents of DS to obtain the physical address
- Effective address resides in either a base or index register
- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[Rx] → 16-bit offset

- Segment base address is DS by default for BX, SI, and DI
- Segment base address is SS by default for BP

PA = DS:[Rx]

- Segment override prefix (SEG) is required to enable use of another segment register

PA = SEG:ES:[Rx]

3.5 Addressing Modes

- Memory addressing modes – Register indirect addressing mode

- Example

PA = $02000_{16} + 1234_{16} = 03234_{16}$

MOV AX,[SI]

- State before fetch and execution Instruction

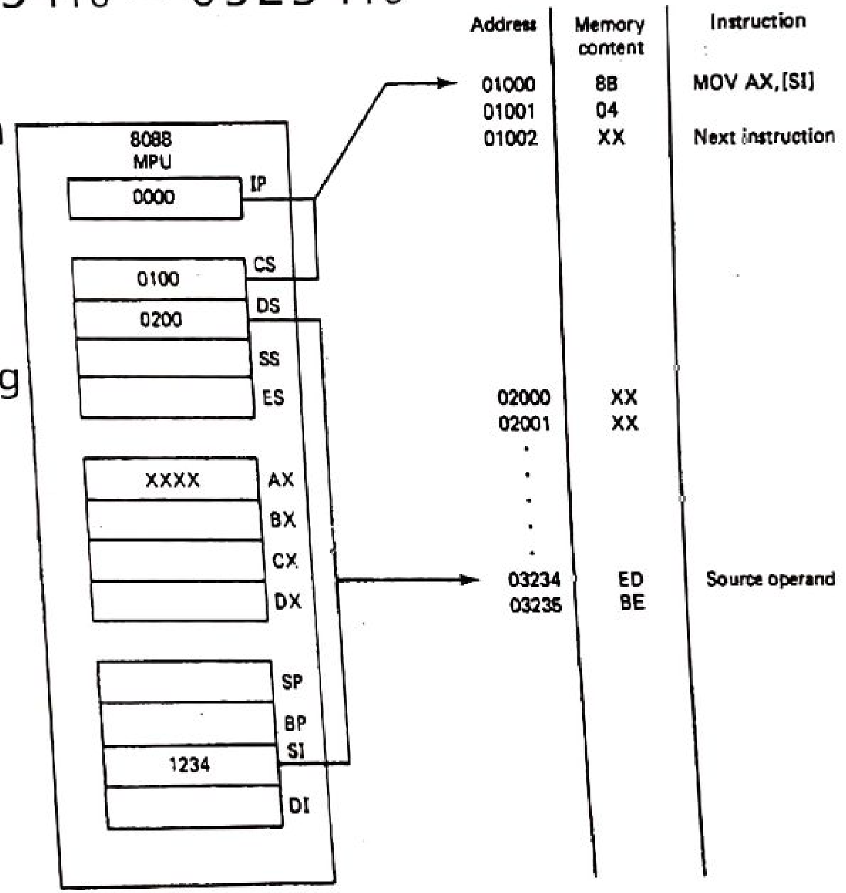
CS = 0100H
IP = 0000H
CS:IP = 0100:0000H = 01000H
(01000H,01001H) = Opcode = 8B04H

- Source operand-register indirect addressing

DS = 0200H
SI = 1234H
PA = DS:SI = 0200H:1234H
= 02000H + 1234H
= 03234H
(03235H,03234H) = BEEDH

- Destination operand-register operand addressing

(AX) = XXXX → don't care state



Before execution(a)

3.5 Addressing Modes

- Memory addressing modes - Register indirect addressing mode

$$PA = 0200_{16} + 1234_{16} = 03234_{16}$$

- Example (continued)
  - State after execution Instruction

CS:IP = 0100:0002 = 01002H

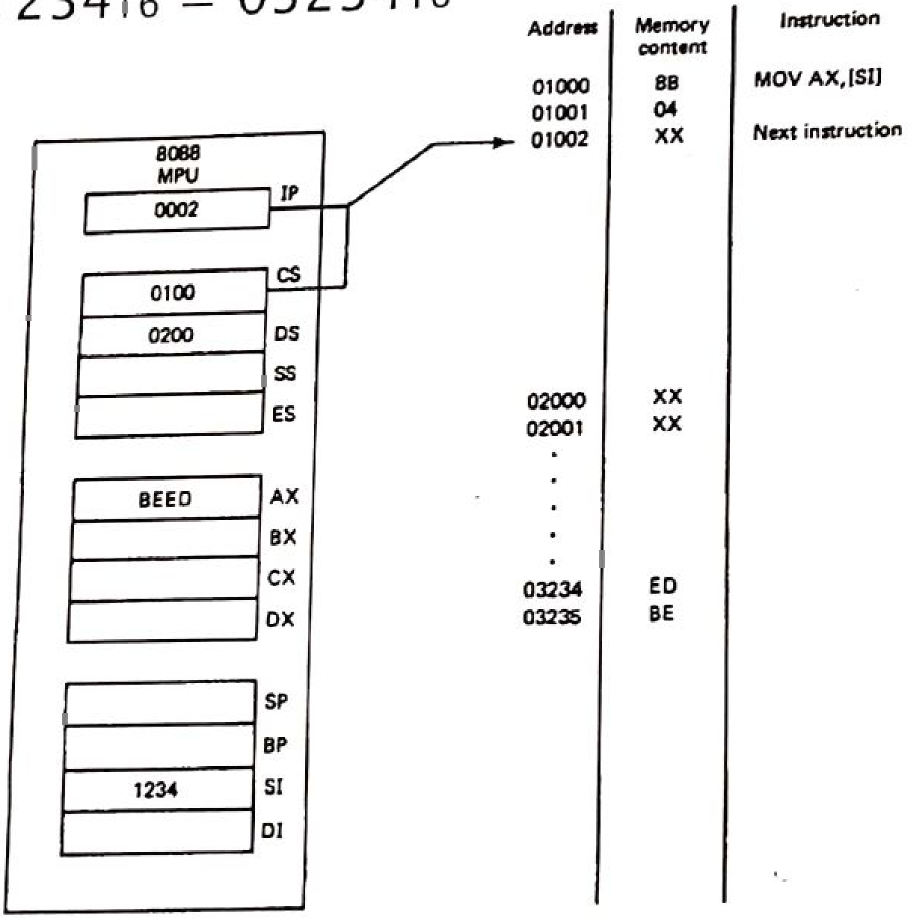
01002H → points to next sequential instruction

- Source operand

(03235H,03234H) = BEEDH → unchanged

- Destination operand

(AX) = BEEDH

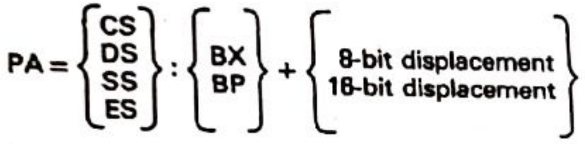


(b)

After execution

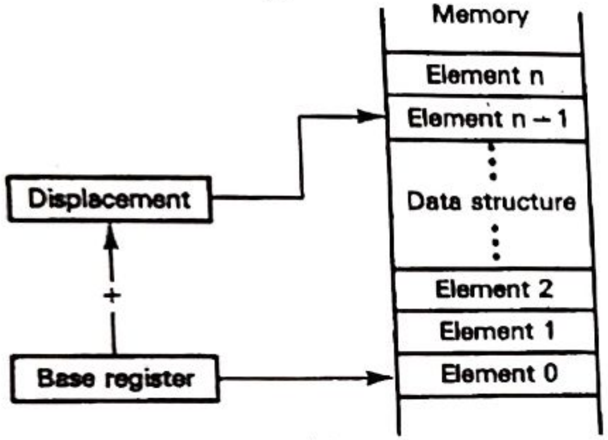
3.5 Addressing Modes

- Memory addressing modes – Based addressing mode



$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} \text{8-bit displacement} \\ \text{16-bit displacement} \end{Bmatrix}$$

(a)



(b)

e.g. MOV [BX]+1234H, AL

- Effective address formed from contents of a base register and a displacement
- Base register is either BX or BP (stack)
  - Direct/indirect displacement is 8-bit or 16bit
- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA:[BX or BP] + DA

- Accessing a data structure
  - Based addressing makes it easy to access elements of data in an array
  - Address in base register points to start of the array
  - Displacement selects the element within the array
  - Value of the displacement is simply changed to access another element in the array
  - Program changes value in base register to select another array

3.5 Addressing Modes

- Memory addressing modes - Based addressing mode

$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$

- Example

MOV [BX] +1234H,AL

- State before fetch and execution Instruction

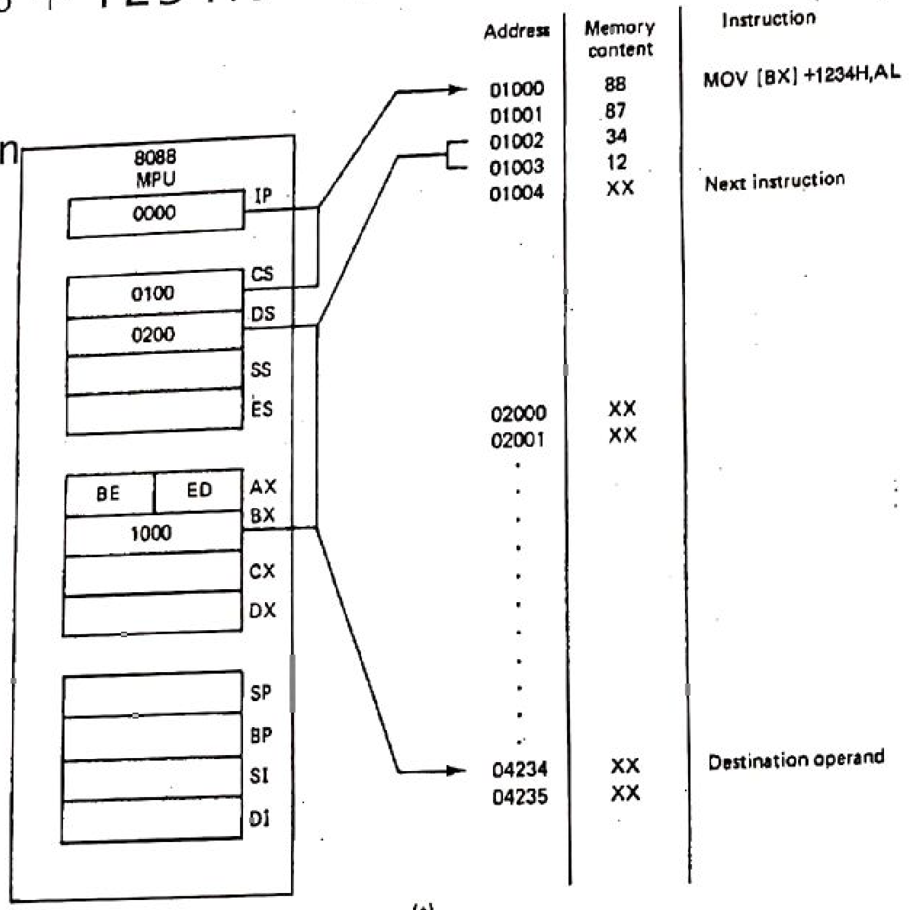
CS = 0100H, IP = 0000H
CS:IP = 0100:0000H = 01000H
(01000H,01001H) = Opcode = 8887H
(01002H,01003H) = Direct displacement = 1234H

- Destination operand—based addressing

DS = 0200H, BX = 1000H, DA = 1234H
PA = DS:DS+DA = 0200H:1000H+1234H
= 02000H+1000H+1234H
= 04234H
(04234H) = XXH

- Source operand—register operand addressing

(AL) = ED



(a)

Before execution

3.5 Addressing Modes

- Memory addressing modes – Based addressing mode

$$PA = 02000_{16} + 1000_{16} + 1234_{16} = 04234_{16}$$

- Example (continued)
- State after execution Instruction

CS:IP = 0100:0004 = 01004H

01004H → points to next sequential instruction

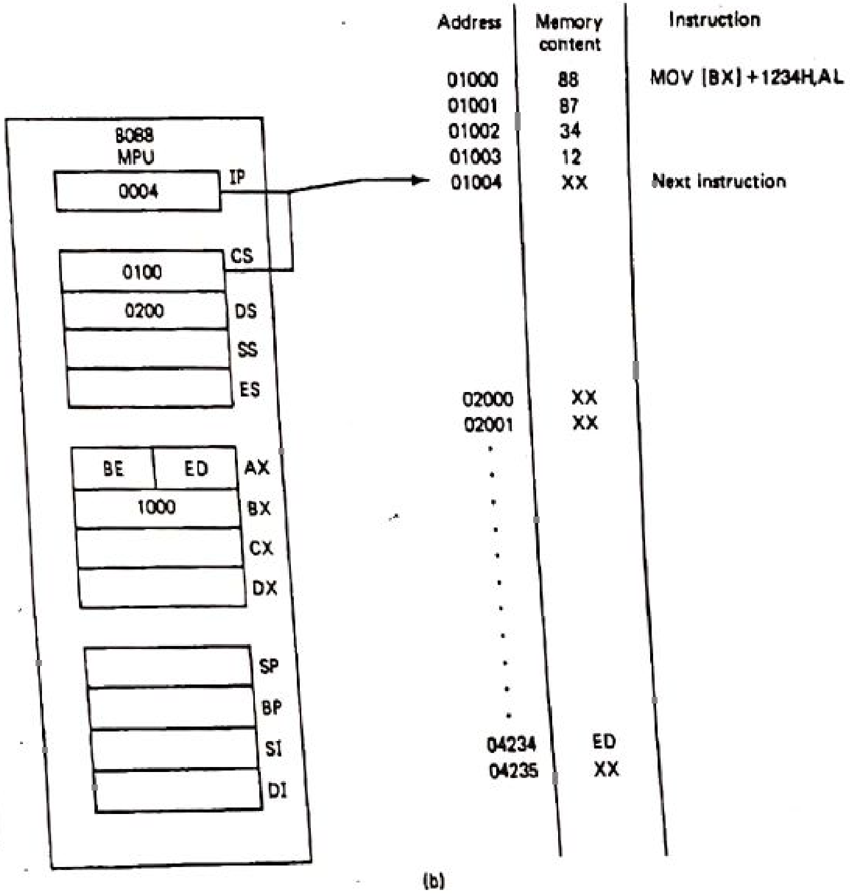
- Destination operand

(04234H) = EDH

- Source operand

(AL) = EDH → unchanged

Note: *if BP is used instead of BX, the calculation of PA is performed using SS instead of DS.*

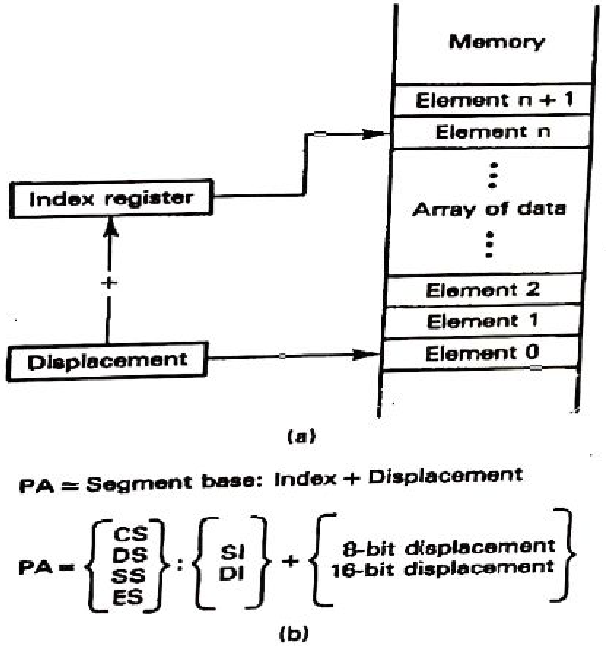


After execution

3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

PA = Segment Base : Index + Displacement



Computation of an indexed address

e.g. MOV AL, [SI]+1234H

- Similar to based addressing, it makes accessing elements of data in an array easy
- Displacement points to the beginning of array in memory
- Index register selects element within the array
- Program simply changes the value of the displacement to access another array
- Program changes (re-computes) value in index register to select another element in the array
- Effective address formed from direct displacement and contents of an index register
- Direct displacement is 8-bit or 16-bit
- Index register is either SI → source operand or DI → destination operand
- Physical address computation

PA = SBA:EA → 20-bit address

PA = SBA: DA + [SI or DI]

3.5 Addressing Modes

- Memory addressing modes – Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

- **Example**

MOV AL,[SI] +1234H

- State before fetch and execution Instruction

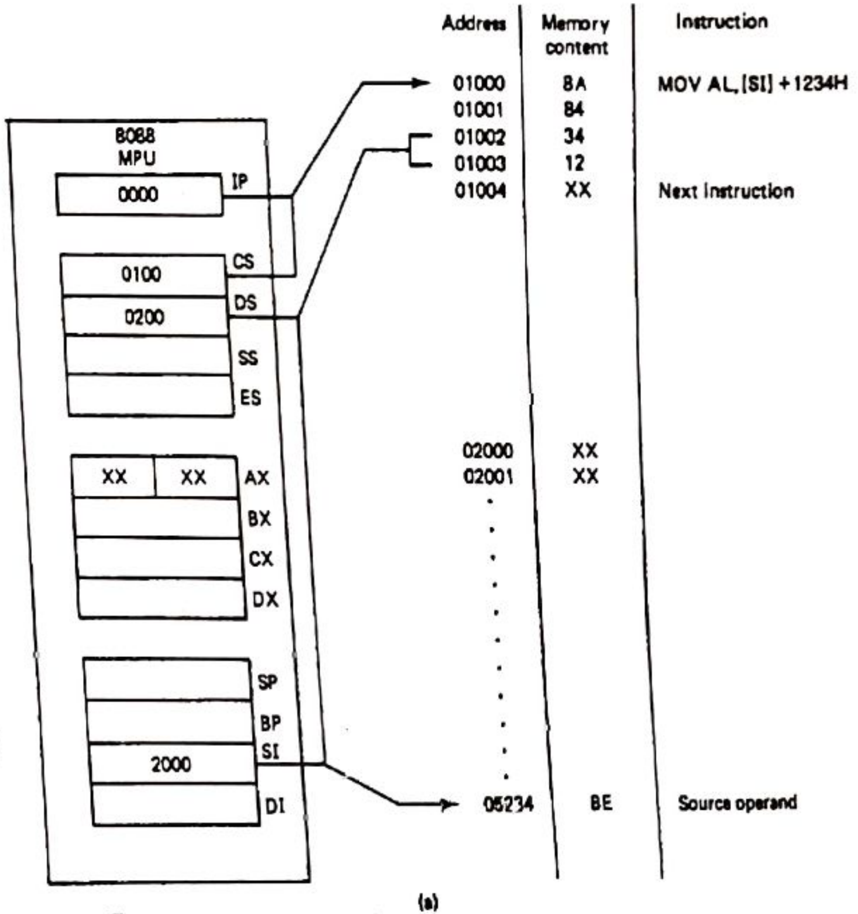
CS = 0100H
IP = 0000H
CS:IP = 0100:0000H = 01000H
(01000H,01001H) = Opcode = 8A84H
(01002H,01003H) = Direct displacement = 1234H

- Source operand—indexed addressing

DS = 0200H
SI = 2000H
DA = 1234H
PA = DS:SI+DA = 0200H:2000H+1234H
= 02000H+2000H+1234H
= 05234H
(05234H) = BEH

- Destination operand—register operand addressing

(AL) = XX → don't care state



Address	Memory content	Instruction
01000	8A	MOV AL,[SI] +1234H
01001	84	
01002	34	
01003	12	
01004	XX	Next Instruction
02000	XX	
02001	XX	
05234	BE	Source operand

Before execution

3.5 Addressing Modes

- Memory addressing modes - Indexed addressing mode

$$PA = 02000_{16} + 2000_{16} + 1234_{16} = 05234_{16}$$

- Example (continued)
- State after execution Instruction

CS:IP = 0100:0004 = 01004H

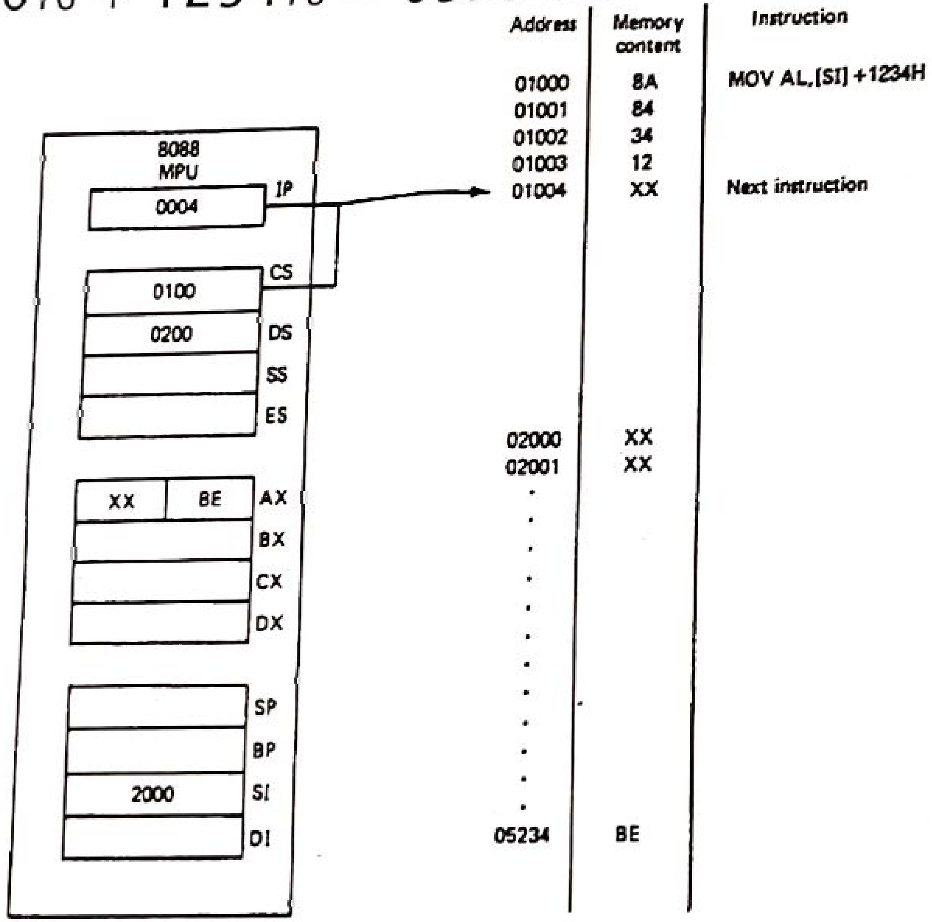
01004H → points to next sequential instruction

- Source operand

(05234H) = BEH → unchanged

- Destination operand

  (AL) = BEH

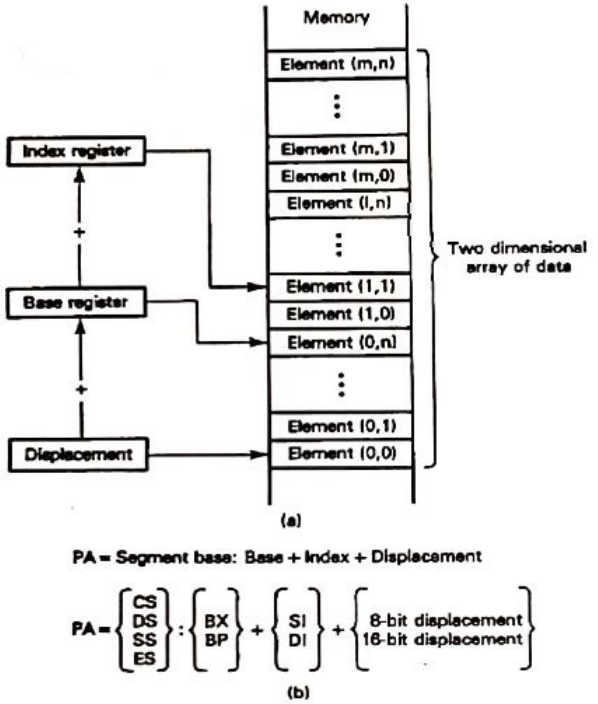


(b)

After execution

3.5 Addressing Modes

Memory addressing modes – Based-indexed addressing mode



Computation of an indexed address

e.g. MOV AH, [BX][SI]+1234H

- Combines the functions of based and indexed addressing modes
- Enables easy access to two-dimensional arrays of data
- Displacement points to the beginning of array in memory
- Base register selects the row *(m) of elements*
- Index register selects element in a column *(n)*
- Program simply changes the value of the displacement to access another array
- Program changes (re-computes) value in base register to select another row of elements
- Program changes (re-computes) the value of the index register to select the element in another column
- Effective address formed from direct displacement and contents of a base register and an index register
- Direct displacement is 8-bit or 16bit
- Base register either BX or BP (stack)
- Index register is either SI → source operand or DI → destination operand
- Physical address computation

PA = SBA:EA → 20-bit address
PA = SBA:DA + [BX or BP] + [SI or DI]

EA = Bx + SI + 1234

=1000 + 2000 + 1234

= $\underbrace{4234}_{\text{offset}}$

بما أن base seg مش موجودة نفرضها DS

= 0200

3.5 Addressing Modes

- Memory addressing modes – Based-indexed addressing mode

  PA = $0200_{16} + 1000_{16} + 2000_{16} + 1234_{16} = 06234_{16}$

* • **Example** يعني اقل byte مش word

  MOV AH, [BX][SI] +1234H

- State before fetch and execution Instruction

CS = 0100H, IP = 0000H

CS:IP = 0100:0000H = 01000H

(01000H,01001H) = Opcode = 8AA0H

(01002H,01003H) = Direct displacement = 1234H

- Source operand-based-indexed addressing

DA = 1234H, DS = 0200H, BX = 1000H,

SI = 2000H

PA = DS:DA +BX +SI

= 0200H:1234H + 1000H + 2000H

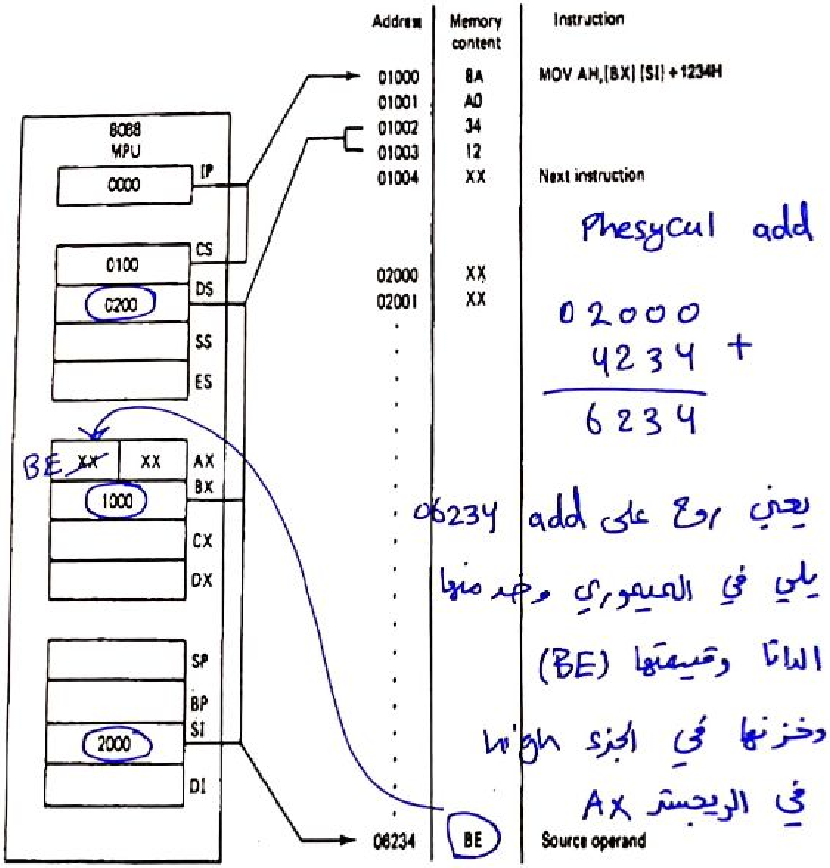
= 02000H+1234H +1000H + 2000H

= 06234H

(06234H) = BEH

- Destination operand—register operand addressing

(AH) = XX → don't care state



Phesycal add

$$\begin{array}{r} 02000 \\ 4234 + \\ \hline 6234 \end{array}$$

يعني روح على add 06234 يلي في الميموري وخد منها الداتا وقيمتها (BE) وخزنها في الجزء high في الريجستر AX

(a)

Before execution

3.5 Addressing Modes

- Memory addressing modes – Based-indexed addressing mode

  PA = $0200_{16} + 1000_{16} + 2000_{16} + 1234_{16} = 06234_{16}$

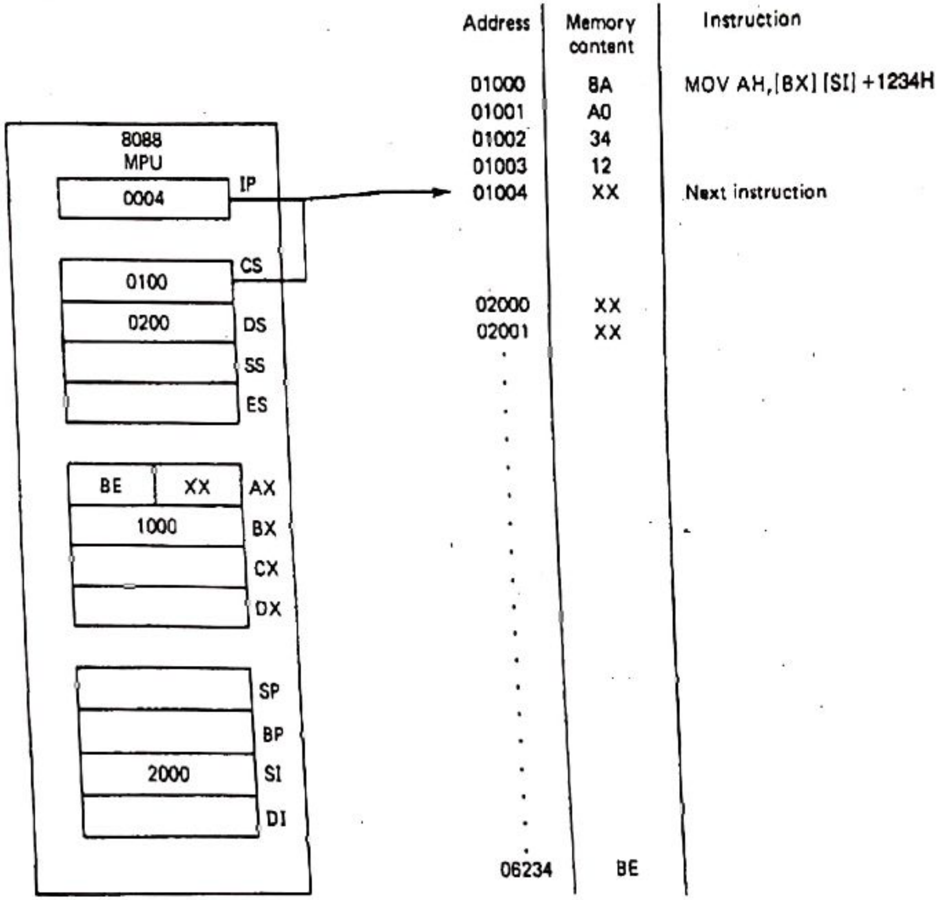
- Example (continued)
  - State after execution Instruction

    CS:IP = 0100:0004 = 01004H

    01004H → points to next sequential instruction
  - Source operand

    (06234H) = BEH → unchanged
  - Destination operand

    (AH) = BEH



(b)

After execution

H.W. #3

- Solve the following problems from Chapter 3 from the course textbook:

  5, 10, 23, 25, 26, 29, 31

4.1 Converting Assembly Language Instructions to Machine Code

مثال:- Mov Ax, Bx
لا يحتاج إلا 2 بايت .

مثال:- Mov [1000] ,2000
عدد البايت في الماشين
كود = 6
٢ بايت ثابتات
٢ بايت للميموري مود
٢ بايت الـ imm مود

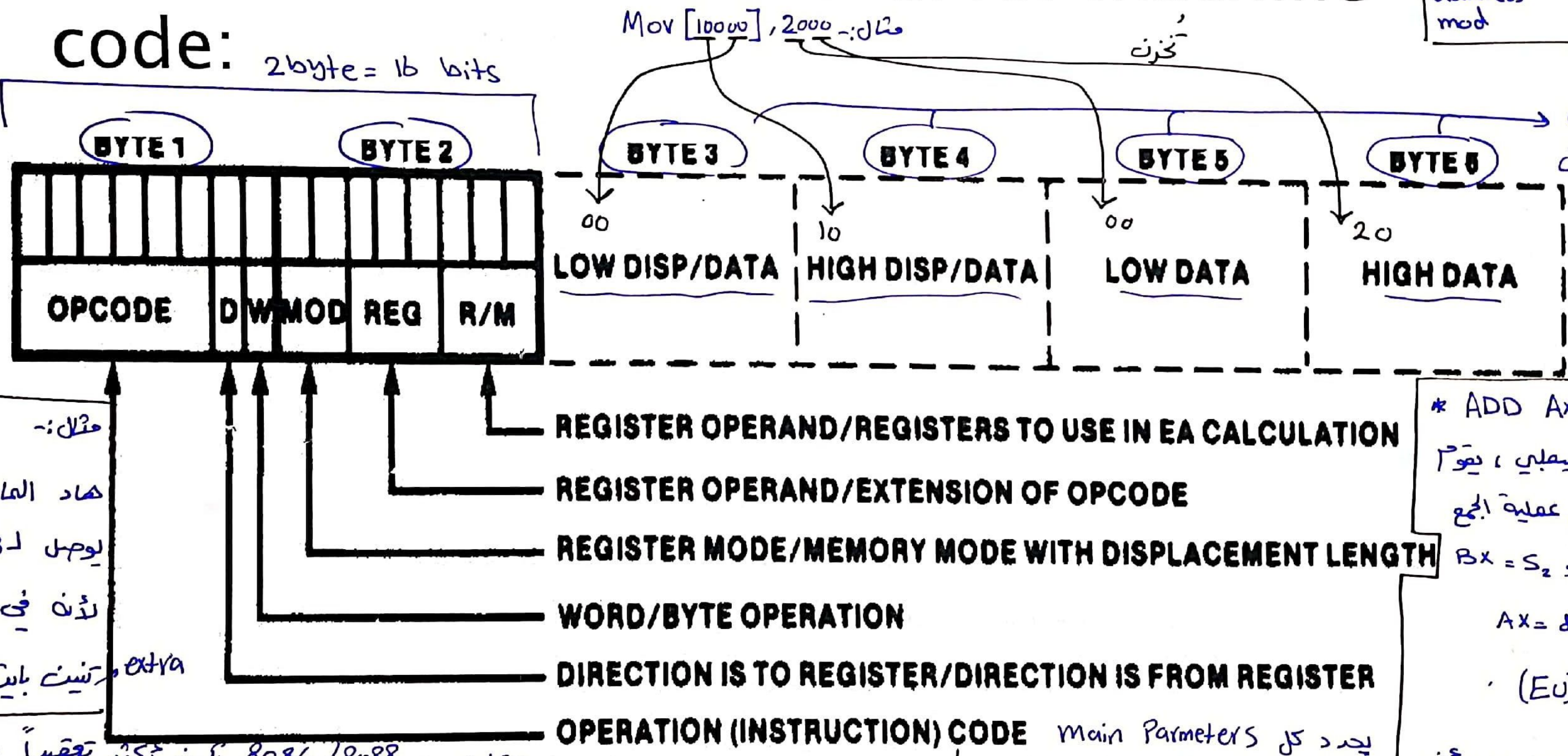
مثال:- Mov Ax ,[1000]
عدد البايت = 4
لأنه في 2 بايت ثابتات
وفي 2 بايت مشان memory address mod

- General instruction format for machine code:

2byte = 16 bits

general machine format

2byte تحتوي-
أساسيات



REGISTER OPERAND/REGISTERS TO USE IN EA CALCULATION

REGISTER OPERAND/EXTENSION OF OPCODE

REGISTER MODE/MEMORY MODE WITH DISPLACEMENT LENGTH

WORD/BYTE OPERATION

DIRECTION IS TO REGISTER/DIRECTION IS FROM REGISTER

OPERATION (INSTRUCTION) CODE

يحدد كل main Parmeters
التي يحتاجهم .

يُضافوا إلى الماشين
كود في الحالات
المتعلقة بـ address modes
زي ميموري و imm

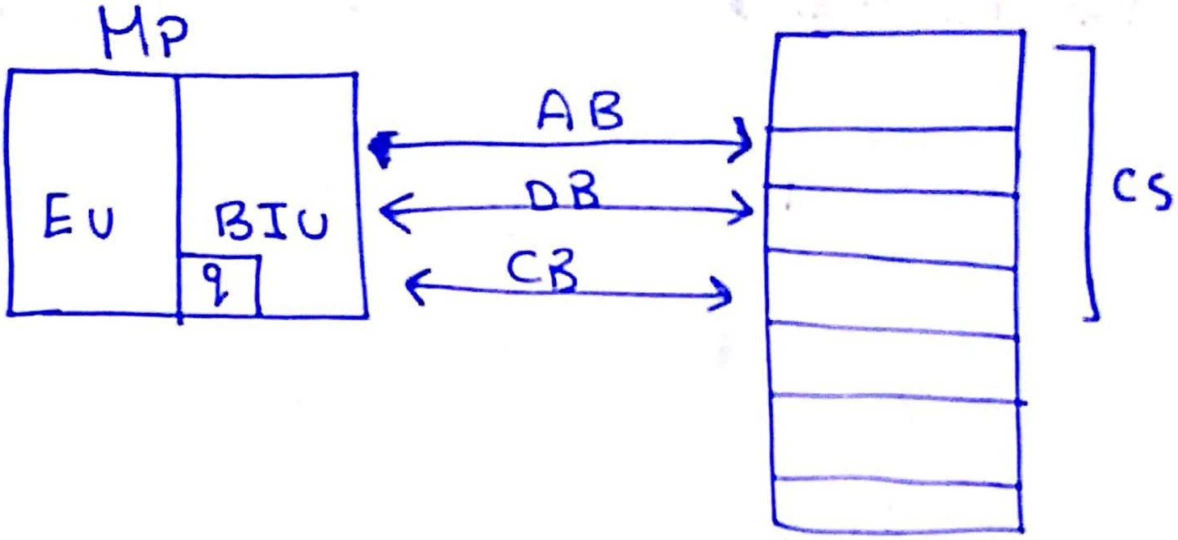
مثال:- Mov Ax ,2000
هاد الماشين كود بالتأكيد ٢,
لوصل لـ byte3 و byte4
لأنه في (Imm data) وفي 2000
extra ترتيب بايت. فعدد البايت ≃ 4

* ADD Ax , Bx
هنا كود بلغة الأسمبلي ، يقوم
البروسيسر بتفعيل عملية الجمع
وتحديد Ax= S_1 و Bx = S_2
وتحديد Ax= destination
كل ذلك يحصل في (EU) .

فهنا الـ (insruction) يجب أن
يتحول إلى مجموعة من الـ bits
(0,1) تكون لها دلالات ، من خلال
هذه الدلالات على البروسيسر أن

عند التعامل مع 8086 /8088 يكون أكثر تعقيداً والسبب :-
(١) عدد الريجستر أكثر وهو 13 ريجستر ويمكن تقسيمهم إلى H ، L .
(٢) عندما نقسم إلى source و destination كل منهم له فروع ، والميموري لها 5 أقسام [address mod] .
(٣) عدد الـ operation يصل إلى 64 .

قد يصل طول الماشين كود
في 8086/8088 إلى 6 byte
= 48 bit ، لكن في معظم
الأحيان 2byte .

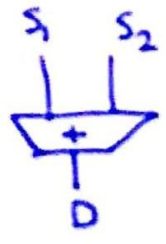


* لنفرض أن البروسيسر بعت address بده يعمل instruction fetch حيث أن ال insructions تكون على شكل (0/1) في code seg

فينتقل ال instruction عبر الداتا bus إلى BIU ، حيث يحتوي quieu يخزن فيه 4 byte بشكل مؤقت .

* لنفرض أن ال instruction التي جبناها من CS كانت ADD Ax, Bx ، لازم EU تفهم طبيعة هذا ال instruction ، فتفهم أن operation=ADD وأن S_1 = Bx ، S_2 = Ax ، D = Ax ، size = word

فهذا ال insruction يحتوي 5 عناصر التي ذكرناها .

* فيقوم EU بتفعيل (Adder) وصيغ شكله  ، ويربط

S_1 ب Bx ويربط S_2 ب Ax ويربط D ب AX .

2

رسمة Machin code بشكل عام

Operation code	S1	S2	D	Size

لو طلب مني أصمم machin code لآلة بتنفذ هاي الinstruction :-

ADD
Sub
mul
Div
OR
and
inv
XOR

لنفرض انه هاي الآلة ما بتدعم الميموري (ما بتجيب داتا أو بتودّيها على الميموري) بس بتحتوي 4 ريجسترز و size تبعها يا B يا W .

هاد الجدول اسمه Look up tafel

Operations		Reg		Size	
ADD	000	A1	00	Byte	0
sub	001	A2	01	word	1
mul	010	A3	10		
Div	011	A4	11		
OR	100				
and	101				
inv	110				
XoR	111				

بما انه عندي 8 insruction كم bit نحتاج لحتى أقدر أميز بين 8 خيارات؟

$2^3 = 8$

∴ 3 bits

وبما أن الريجسترز عددهم 4 فأنا نحتاج 2bits حتى أميز بينهم

وال size عبارة عن خيارين فنحتاج 1bits للتمييز بينهم

$\boxed{3}$

$\underbrace{ADD}_{op}\ \underbrace{A_{1x}}_{S_2},\ \underbrace{A_{2x}}_{S_1}$

لنفرض أن عندي هذا ال instruction

وبدنا نطلع الماشين كود له .

$ADD \xrightarrow[\text{من الجدول}]{\text{تقابلها}} 000$

$A_1 \xrightarrow{S_2} 00$

$A_2 \xrightarrow{S_1} 01$

$A_1 \xrightarrow{D} 00$

opcode	S_1	S_2	D	size
$\underbrace{000}_{3bits}$	$\underbrace{01}_{2}$	$\underbrace{00}_{2}$	$\underbrace{00}_{2}$	$\underbrace{1}_{1}$

لأنه word A_x

2 bytes > 10 bits =

فننقل الماشين كود

$(000\ 01\ 00\ 00\ 1)_b$

$(0\,2\,1)_H$

* عملية ال ~~decoding~~ decoding هي عملية هاردوير (enable / disenable)

الماشين كود لل Sub :- Sub A_1x ، A_2x *

000	01	00	00	1

$(000\ 01\ 00001)_b$

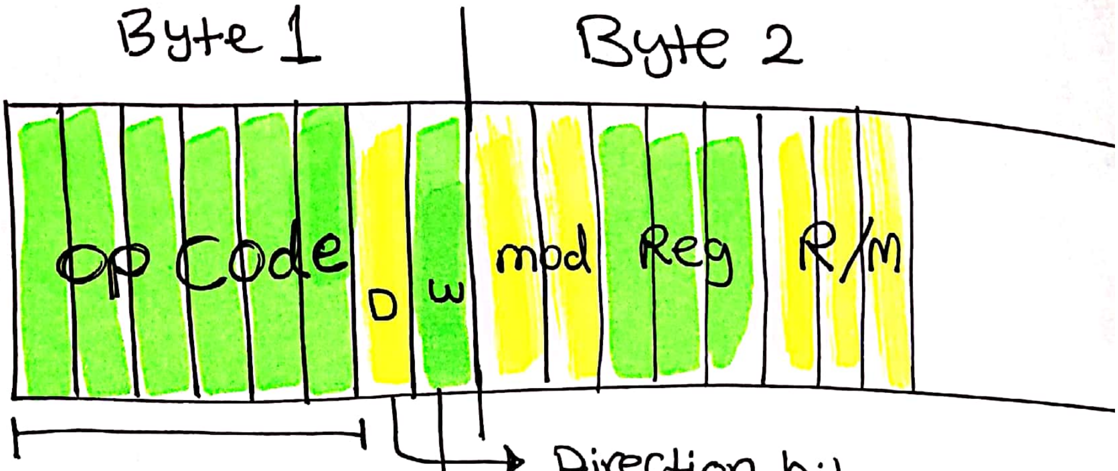
→ (0A1)

* الكومبايلر بحوّل ADD A_x, A_2x إلى ماشين كود (0000100001)

* ويقوم الـ Linker بتخزينه في الميموري.

* البروسيسر يعمل fetsh للماشين كود عن طريق BIU.

* BIU و EU. بعملوا decoding للـinstruction وتقوم EU بتفعيل الفنكشن حسب حاجتها له.



6 bits

اسمهم op code

وظيفتهم يعبروا عن

ال operation

زي add, sub, ...

يصل عدد الخيارات = 2^6 = 64

لذلك بحتاج 6 bits حتى أقدر أميز بينهم.

D → Direction bit

to reg or from reg

W → تعبر عن ال size

مثال :-

Mov AL, BL → byte

Mov Ax, Bx → word

- Register direction bit
  - 1 – the register opera
  - 0 – the register opera
- Data size bit (W–bit)
  - 1 – 16–bit data size
  - 0 – 8–bit data size

مثال :-

Mov [1000], Ax

الريجستر موجود sources

إذن قيمة D = 0

مثال :-

Mov Ax , [1000]

الريجستر موجود كـ destination

إذن قيمة D = 1

مثال :-

Mov Ax, Bx

هاي حالة خاصة

رح نعتمد هاد الحل :

D = 0 (يعني الريجستر رح نعتبرها source)

مثال :- Mov Bx, Ax

W = 1

مثال :- Mov Bh , AL

W = 0

4.1 *Encoding (1):* MOV BL, AL

- EXAMPLE

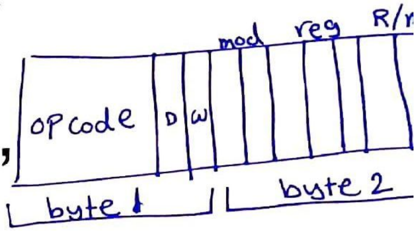
  Encode the instruction in machine code

  MOV BL, AL

طريقة الحل :-

* هل في memory operand ؟ لا

* // // imm operand ؟ لا ، لذلك ما رح نحتاج إلا 2byte



- Solution:

  Byte 1: OPCODE = 100010 (for MOV),

  D = 0 (source), W = 0 (8-bit)

- This leads to BYTE 1 = $10001000_2 = 88_{16}$
- In byte 2 the source operand, specified by REG, is AL

  REG = 000, MOD = 11, **R/M = 011**

- Therefore, BYTE 2 = $11000011_2 = C3_{16}$

  MOV BL, AL = $88C3_{16}$

نعبيهم بالترتيب حسب الجداول .

١ بنجيب op code بما انه (Mov reg/reg) = $\boxed{100010}$

٢ بنجيب D بما انه reg في ال ٢ ، d
بنعتمد طريقة معينة للحل وهي = $\boxed{D=0}$

٣ بنجيب w بما انه BL، AL يعني بايت ، $\boxed{W=0}$

هيك كملنا أول بايت

٤ بنجيب Mod ، بما انه ما في ميموري فإن $\boxed{Mod = 11}$

٥ بنجيب reg ، بما انها AL ← $\boxed{Reg = 000}$

٦ بنجيب R/M ، بما انها BL وما في ميموري ← $\boxed{R/M = 011}$

بصير بالنهاية هيك ⇐ $(1000100011´000011)_b$ ⇐ $(88C3)_H$ (

MOV WORD PTR [BP][DI]+1234H, 0ABCDH

حالة استثنائية

لأنه ما في
رجيستر.

Mnemonic and Description	Instruction Code					
DATA TRANSFER						
MOV – Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m	Disp. L	Disp. H		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m reg	Disp. L	Disp. H	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1			
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high			
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high			

1 1 0 0 0 1 1

هاي الواحد للـ D
مشبتة هون لأنه مارجستر.

$\boxed{D = 1}$

$\boxed{W = 1}$ → لأنه operation word

Byte(1) → (11000111) = C7
Byte(2) → (10 000 011) = 83
Byte(3) → 34
Byte(4) → 12
Byte(5) → CD
Byte(6) → AB

MOD = 11		
R/M	**W = 0**	**W = 1**
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	**W=0**	**W=1**	**R/M**	**MOD=00**	**MOD=01**	**MOD=10**
000	AL	AX	000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16
001	CL	CX	001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16
010	DL	DX	010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16
011	BL	BX	011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16
100	AH	SP	100	(SI)	(SI)+D8	(SI)+D16
101	CH	BP	101	(DI)	(DI)+D8	(DI)+D16
110	DH	SI	110	DIRECT ADDRESS	(BP)+D8	(BP)+D16
111	BH	DI	111	(BX)	(BX)+D8	(BX)+D16

MOV [BP][DI]+1234H, DS

~~حلة~~ حالة خاصة لأنه الريجستر DS

MOD = 11		
R/M	**W = 0**	**W = 1**
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register	0 SR
ES	00
CS	01
SS	10
DS	0 11

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV – Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 SR r/m	Disp LO	Disp HI

D W

Byte (1) = 1000 1100 = ~~8D~~ 8C
Byte (2) = 10 011 011 = ~~[illegible]~~ 9D
Byte (3) = 34 = 34
Byte (4) = 12 = 12

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	**W=0**	**W=1**	**R/M**	**MOD=00**	**MOD=01**	**MOD=10**
000	AL	AX	000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16
001	CL	CX	001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16
010	DL	DX	010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16
011	BL	BX	011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16
100	AH	SP	100	(SI)	(SI)+D8	(SI)+D16
101	CH	BP	101	(DI)	(DI)+D8	(DI)+D16
110	DH	SI	110	DIRECT ADDRESS	(BP)+D8	(BP)+D16
111	BH	DI	111	(BX)	(BX)+D8	(BX)+D16

Example :-   320E 3412 ⟵ أو بهذا الشكل

32
0E
34
12

Find the instruction(s) ?

الحل :-

* نحل أول (2 byte) لل hexa

320E ⟹ 0011 0010 0000 1110

7 bit لل op code   D   w   mod   reg   R/M

برجع للجداول فبلاقي أن 001100 تقابل XOR

〃 〃 〃 〃 D = 1 يعني Reg موجود في destination

〃 〃 〃 〃 w = 0 يعني (L) Reg

〃 〃 〃 〃 R = 001 يعني CL

مبدئياً تكون عندي هذا الشكل XOR CL ,

〃 〃 〃 〃 Mod = 00 و R/M = 110

يعني ↓ أرقام ⇒ وقيمتهم 1234 ← جبتهم من الكود في السؤال

المصيدري عبارة عن

لازم ننتبه عالترتيب لما ننقل 3412 ⇐ 12 34

L H ⟶ H L

∴ الشكل النهائي

XOR CL , [1234]

Lecture Outline

- 5.1 Data-Transfer Instructions
- 5.2 Arithmetic Instructions
- 5.3 Logic Instructions
- 5.4 Shift Instructions
- 5.5 Rotate Instructions

5.1 Data-Transfer Instructions

بتخليني أقدر أنقل داتا ميموري إلى الريجستر أو من الريجستر إلى الميموري.
فلو بدي أنقل من ميموري إلى ميموري ، لازم أنقل من ميموري إلى ريجستر ثم أنقل من الريجستر إلى لوكيشن آخر في الميموري.

- The data-transfer functions provide the ability to move data either between its internal registers or between an internal register and a storage location in memory.
- The data-transfer functions include
  - MOV (Move byte or word)
  - XCHG (Exchange byte or word)
  - XLAT (Translate byte)
  - LEA (Load effective address)
  - LDS (Load data segment)
  - LES (Load extra segment)

5.1 Data-Transfer Instructions -

Move Instruction

- The MOVE Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D,S	(S) → (D)	None

(a)

Reg ← أو memory

→ Reg أو memory أو Imm

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Memory	Seg-reg

(b)

Allowed operands for MOV instruction

- Used to move (copy) data between:
  - Registers
  - Register and memory
  - Immediate operand to a register or memory
- General format:

MOV D,S

- Operation: Copies the content of the source to the destination

(S) → (D)

- Source contents unchanged
- Flags unaffected
- Allowed operands

Register

Memory

Accumulator (AH,AL,AX)

Immediate operand (Source only)

Segment register (Seg-reg)

- Examples:

MOV [SUM],AX

(AL) → (address SUM)

(AH) → (address SUM+1)

5.1 Data-Transfer Instructions – Move Instruction

- The MOVE Instruction

  e.g. MOV DX, CS
  MOV [SUM], AX

- Note that the MOV instruction cannot transfer data directly between external memory.

5.1 Data-Transfer Instructions –

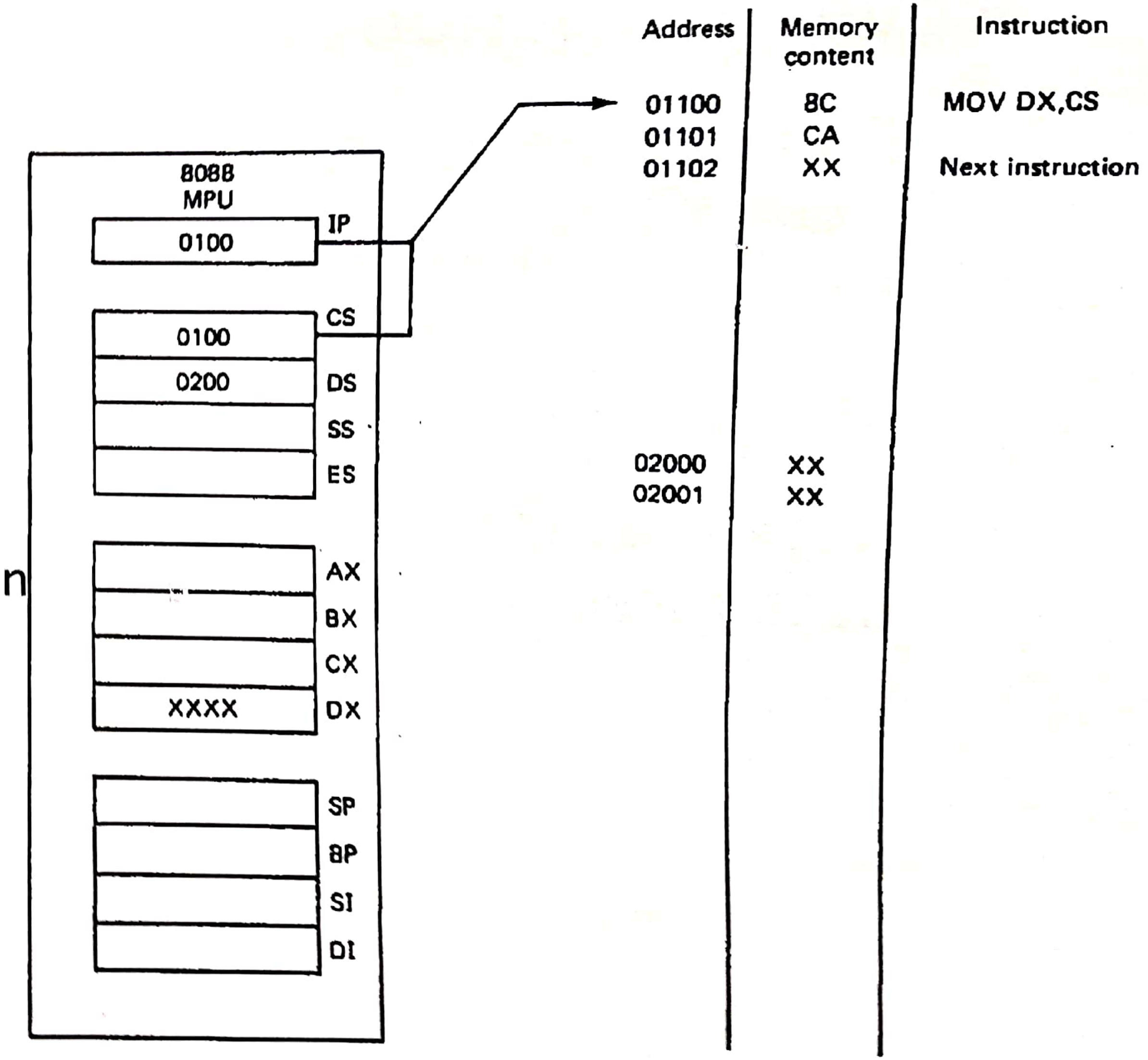
Move Instruction

MOV DX, CS

Source = CS → word data
Destination = DX → word data
Operation: (CS) → (DX)

- State before fetch and execution

CS:IP = 0100:0100 = 01100H
Move instruction code = 8CCAH
(01100H) = 8CH
(01101H) = CAH
(CS) = 0100H
(DX) = XXXX → don't care state



Address	Memory content	Instruction
01100	8C	MOV DX,CS
01101	CA	
01102	XX	Next instruction
02000	XX	
02001	XX	

Before execution

(c)

5.1 Data-Transfer Instructions -

Move Instruction

MOV DX, CS

- State after execution

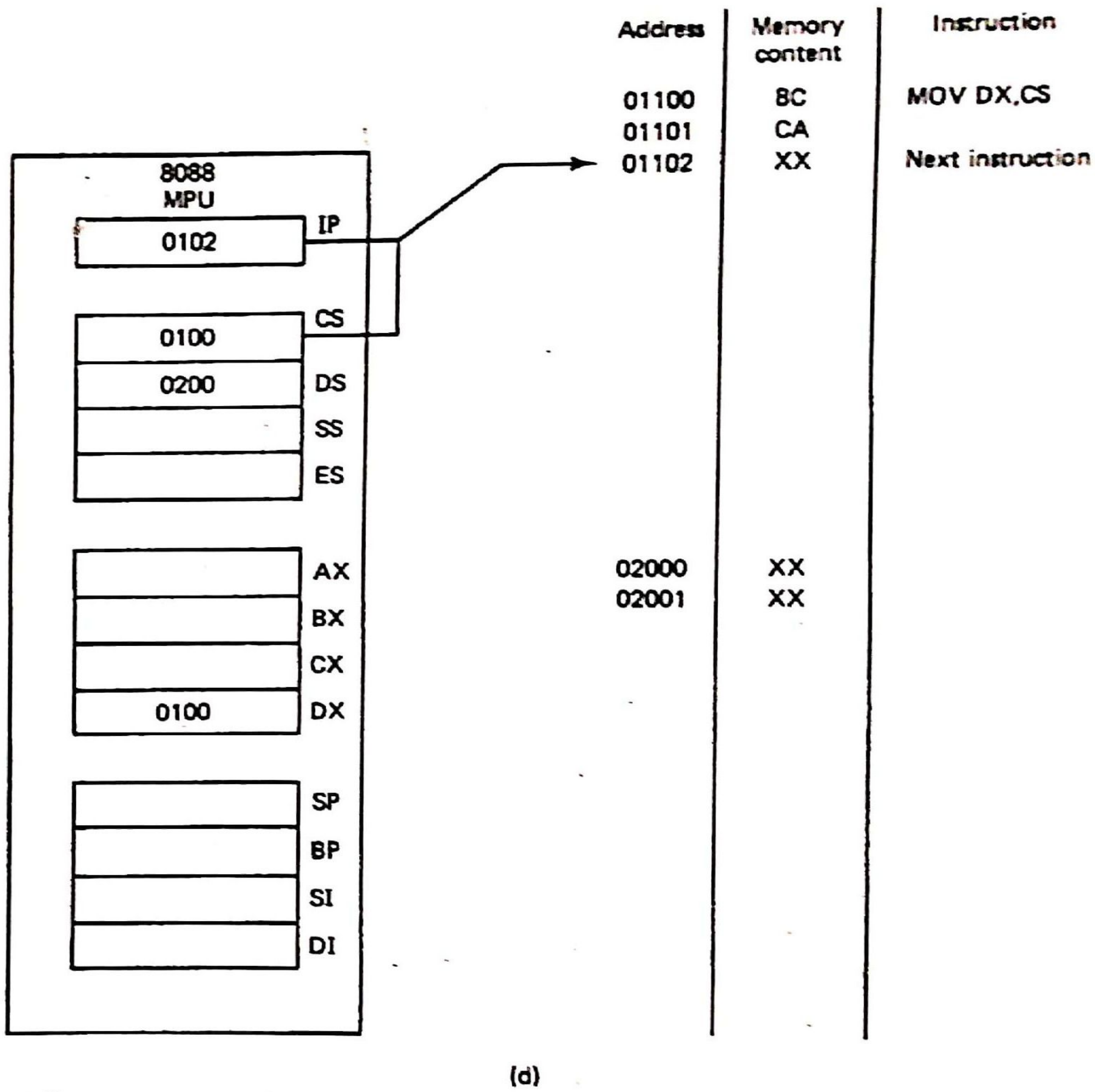
CS:IP = 0100:0102 = 01102H

01002H → points to next sequential instruction

(CS) = 0100H

(DX) = 0100H → Value in CS copied into DX

Value in CS unchanged



Address	Memory content	Instruction
01100	8C	MOV DX,CS
01101	CA	
01102	XX	Next instruction
02000	XX	
02001	XX	

After execution

5.1 Data-Transfer Instructions -

Move Instruction

- EXAMPLE

  What is the effect of executing the instruction

  MOV CX, [SOURCE_MEM]

  Where SOURCE_MEM equal to 20_{16} is a memory location offset relative to the current data segment starting at $1A00_{16}$.

- Solution:

$$((DS)0+20_{16}) \rightarrow (CL)$$

$$((DS)0+20_{16}+1_{16}) \rightarrow (CH)$$

Therefore CL is loaded with the contents held at memory address

$$1A000_{16} + 20_{16} = 1A020_{16}$$

and CH is loaded with the contents of memory address

$$1A000_{16} + 20_{16} + 1_{16} = 1A021_{16}$$

5.1 Data-Transfer Instructions -

Move Instruction

- EXAMPLE

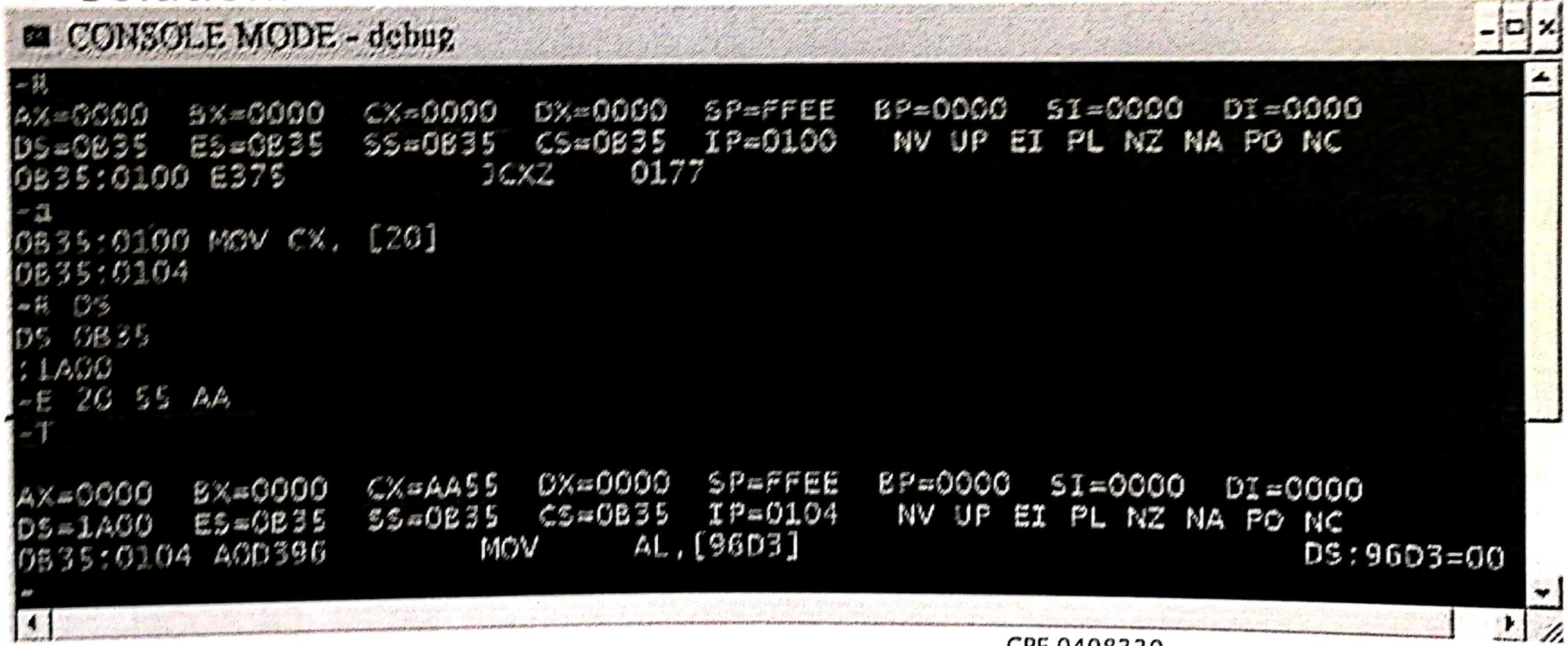
Use the DEBUG to verify

MOV CX,[20]

DS = 1A00, (DS:20) = AA55H

(1A00:20) → (CX)

- Solution:



```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100   NV UP EI PL NZ NA PO NC
0B35:0100 E375          JCXZ    0177
-a
0B35:0100 MOV CX, [20]
0B35:0104
-R DS
DS 0B35
:1A00
-E 20 55 AA
-T

AX=0000  BX=0000  CX=AA55  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1A00  ES=0B35  SS=0B35  CS=0B35  IP=0104   NV UP EI PL NZ NA PO NC
0B35:0104 A0D396        MOV     AL,[96D3]                    DS:96D3=00
-
```

5.1 Data-Transfer Instructions -

Move Instruction

- Example—Initialization of internal registers with immediate data and address information
- DS, ES, and SS registers initialized from immediate data via AX

IMM16 → (AX)
(AX) → (DS) & (ES) = 2000H
IMM16 → (AX)
(AX) → (SS) = 3000H

- Data registers initialized

IMM16 → (AX) =0000H
(AX) → (BX) =0000H
IMM16 → (CX) = 000AH and (DX) = 0100H

- Index register initialized from immediate operations

IMM16 → (SI) = 0200H and (DI) = 0300H

DS,ES to 2000H

SS to 3000H

AX, BX to 0H

CX to 0A

SI to 200

DI to 300

```
MOV  AX,2000H
MOV  DS, AX
MOV  ES, AX
MOV  AX,3000H
MOV  SS,AX
MOV  AX,0H
MOV  BX,AX
MOV  CX,0AH
MOV  DX,100H
MOV  SI,200H
MOV  DI,300H
```

5.1 Data-Transfer Instructions -

Exchange Instruction

مثال:-

XCHG Ax,Bx

يعني خذ الداتا الموجودة في Ax وضعها في Bx وخذ الداتا الموجودة في Bx وضعها في Ax

* لو عملنا Zoomin داخل XCHG ، نلاحظ أننا نستعمل 3 mov أول واحد نأخذ داتا من BX ونضعها في مكان مؤقت ثم تنقل داتا من Ax ---

هذا الجدول يحدد طبيعة الـOperation.

Mnemonic	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG D,S	(D) ↔ (S)	None

(a)

ممكن يكون ← Destination | Source → ممكن يكون

① ريجستر ② ميموري | ① ريجستر ② ميموري

Destination	Source
Accumulator	Reg16
Memory	Register
Register	Register
Register	Memory

(b)

Allowed operands for XCHG instruction

- Used to exchange the data between two data registers or a data register and memory
- General format:

XCHG D,S

- Operation: Swaps the content of the source and destination
- Both source and destination change

(S) → (D)

(D) → (S)

- Flags unaffected
- Special accumulator destination version executes faster
- Examples:

XCHG AX,DX

(Original value in AX) → (DX)

(Original value in DX) → (AX)

5.1 Data-Transfer Instructions –

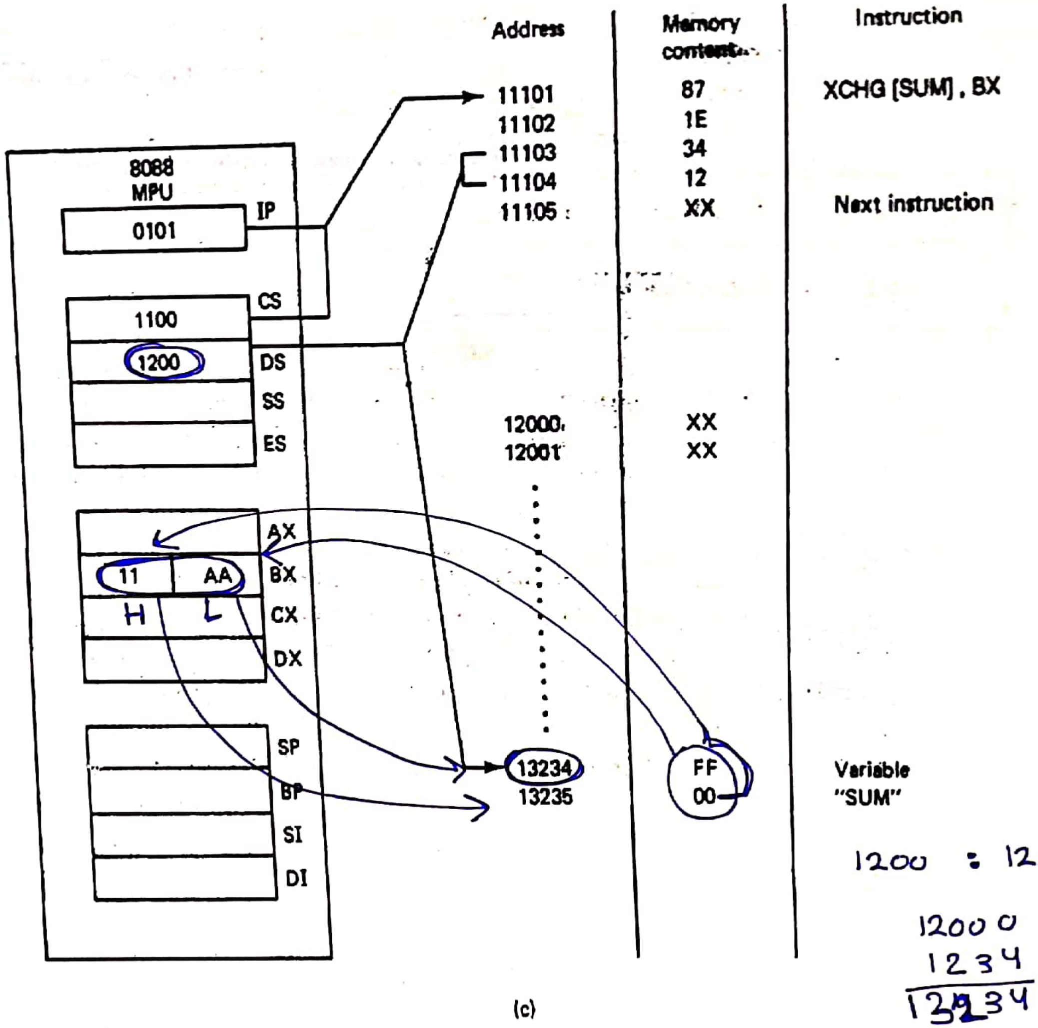
Exchange Instruction

XCHG [SUM],BX
Note: SUM = 1234

Source = BX → word data
Destination = memory offset
SUM → word data
Operation: (SUM) → (BX)
(BX) → (SUM)
What is the general logical address of the destination operand?

- State before fetch and execution

CS:IP = 1100:0101 = 11101H
Move instruction code = 871E3412H
(01104H,01103H) = 1234H = SUM
(DS) = 1200H
(BX) = 11AA
(DS:SUM) = (1200:1234) = 00FFH



Before execution

* 1200 : 1234

12000
1234 +
13234

* نريد أن ننقل الداتا من الريجستر Bx ونضعها في الميموري التي عنوانها 1234 ، والعكس .

* نحسب ك address : .k أنه مش محدد الـ base فهو تلقائياً Ds

وبما أن BX محتواه 11 AA يعني 2byte

فباخذ (00FF)^{12} كاملة .

* الماشين كود مقداره (4 byte) .

5.1 Data-Transfer Instructions – Exchange Instruction

XCHG [SUM],BX

- State after execution

CS:IP = 1100:0105 = 11105H

11005H → points to next sequential instruction

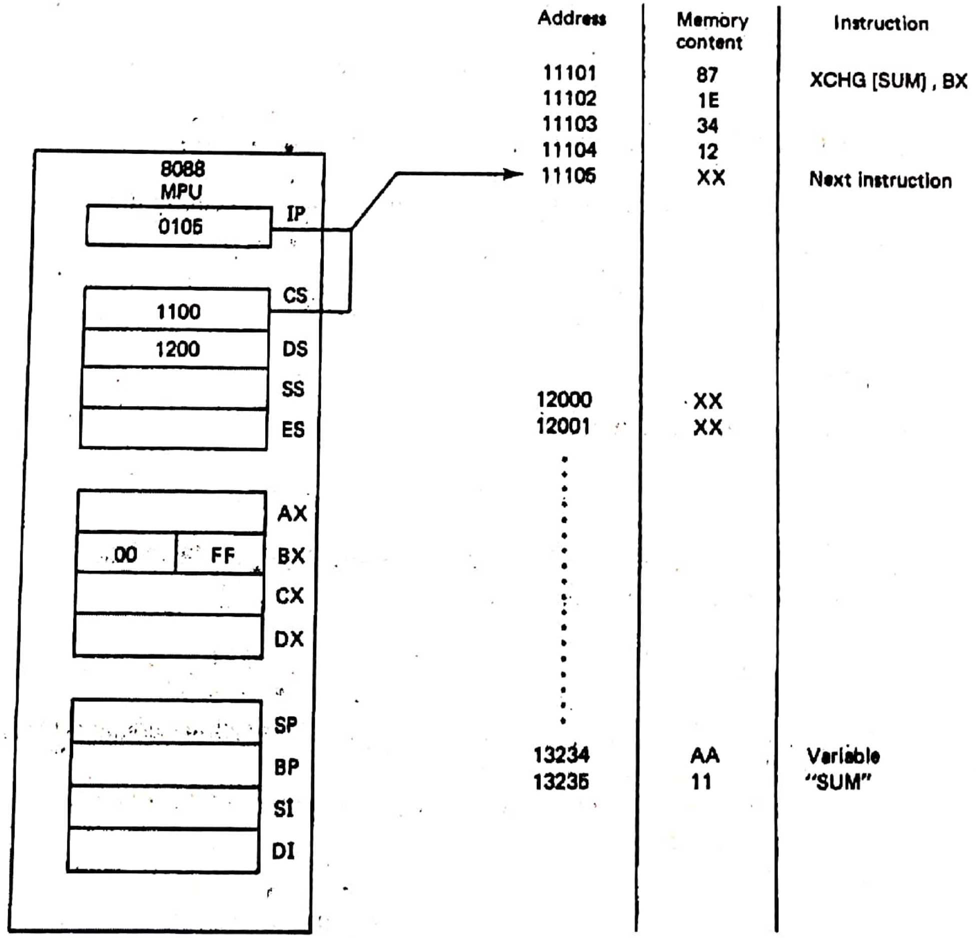
- Register updated

(BX) = 00FFH

- Memory updated

(1200:1234) = AAH

(1200:1235) = 11H



Address	Memory content	Instruction
11101	87	XCHG [SUM], BX
11102	1E	
11103	34	
11104	12	
11105	XX	Next instruction
12000	XX	
12001	XX	
13234	AA	Variable "SUM"
13235	11	

(d)

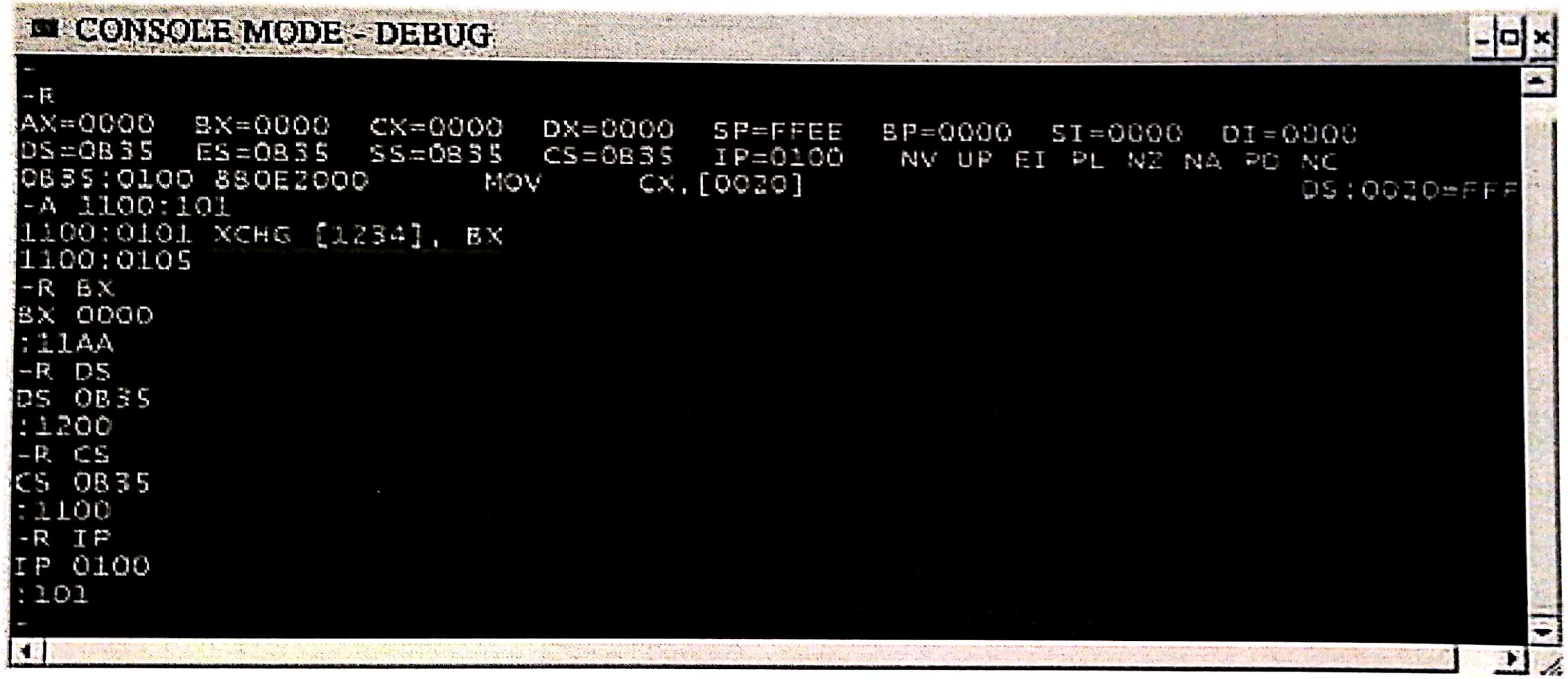
After execution

5.1 Data–Transfer Instructions – Exchange Instruction

- EXAMPLE

  Use the DEBUG to verify the previous example.

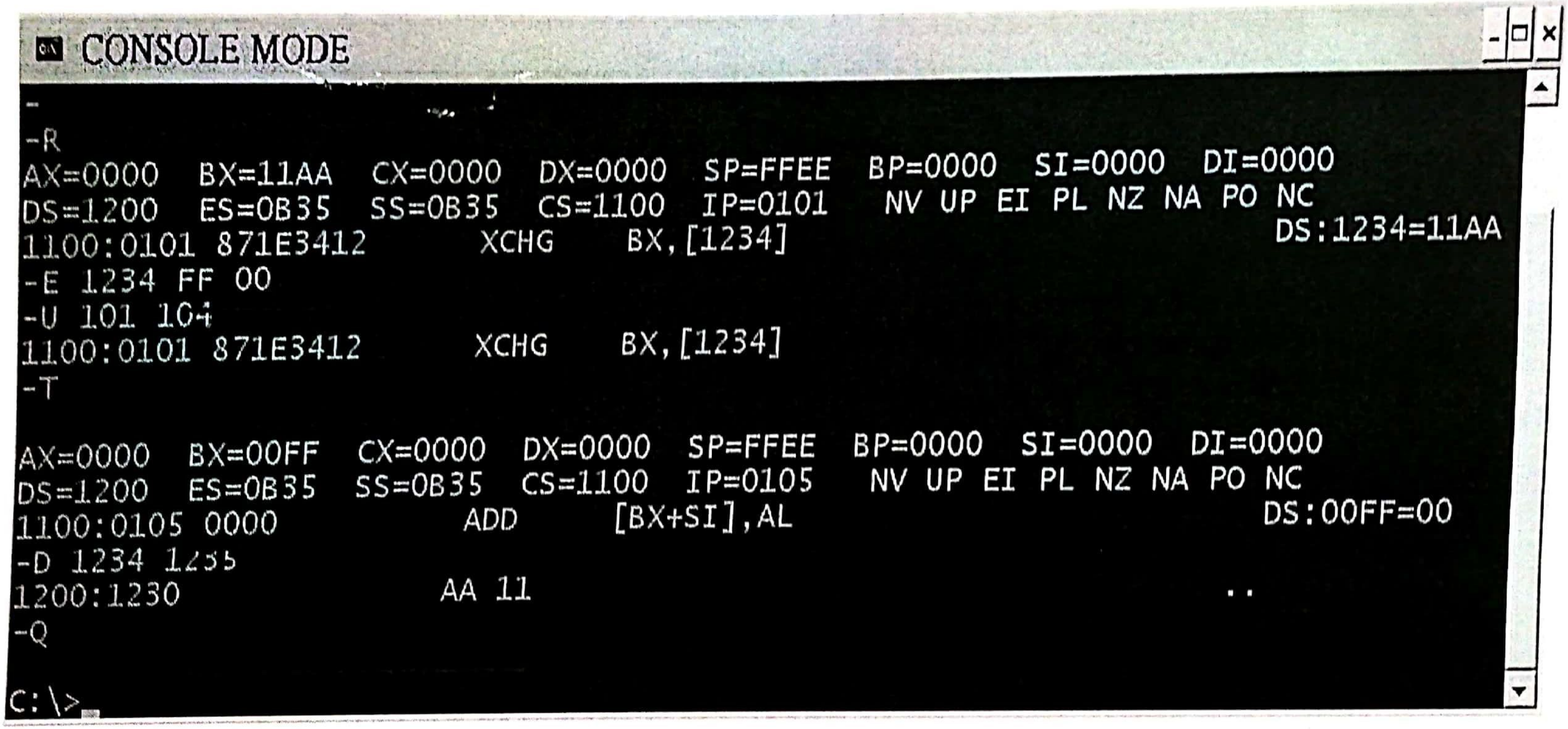
- Solution:



```
-
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0100   NV UP EI PL NZ NA PO NC
0B35:0100 8B0E2000       MOV     CX,[0020]                       DS:0020=FFF
-A 1100:101
1100:0101 XCHG [1234], BX
1100:0105
-R BX
BX 0000
:11AA
-R DS
DS 0B35
:1200
-R CS
CS 0B35
:1100
-R IP
IP 0100
:101
-
```

5.1 Data-Transfer Instructions – Exchange Instruction

- **Solution (cont'd):**



```
-
-R
AX=0000  BX=11AA  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1200  ES=0B35  SS=0B35  CS=1100  IP=0101   NV UP EI PL NZ NA PO NC
1100:0101 871E3412      XCHG    BX,[1234]                     DS:1234=11AA
-E 1234 FF 00
-U 101 104
1100:0101 871E3412      XCHG    BX,[1234]
-T

AX=0000  BX=00FF  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1200  ES=0B35  SS=0B35  CS=1100  IP=0105   NV UP EI PL NZ NA PO NC
1100:0105 0000          ADD     [BX+SI],AL                    DS:00FF=00
-D 1234 1235
1200:1230             AA 11                                ..
-Q

C:\>
```

5.1 Data-Transfer Instructions – Translate Instruction

‣ The XLAT Instruction :

ما معه operation ، بيعمل عملية واحدة دائمًا .

Mnemonic	Meaning	Format	Operation	Flags affected
XLAT	Translate	XLAT	((AL)+(BX)+(DS)0) → (AL)	None

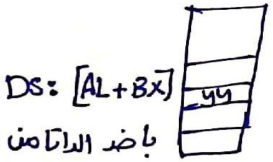
متكافئين . (Mov AL, [AL + Bx] → هاد هيك بنحله :

- Translate instruction
  - Used to look up a byte-wide value in a table in memory and copy that value in the AL register
  - General format:

XLAT

  - Operation: Copies the content of the element pointed to in the source table in memory to the AL register

((AL)+(BX) +(DS)0) → (AL)



باخد الداتا من الميموري وبنقلها على رجيستر AL .

Where:
(DS)0 = Points to the active data segment
(BX) = Offset to the first element in the table
(AL) = Displacement to the element of the table that is to be accessed*
*8-bit value limits table size to 256 elements

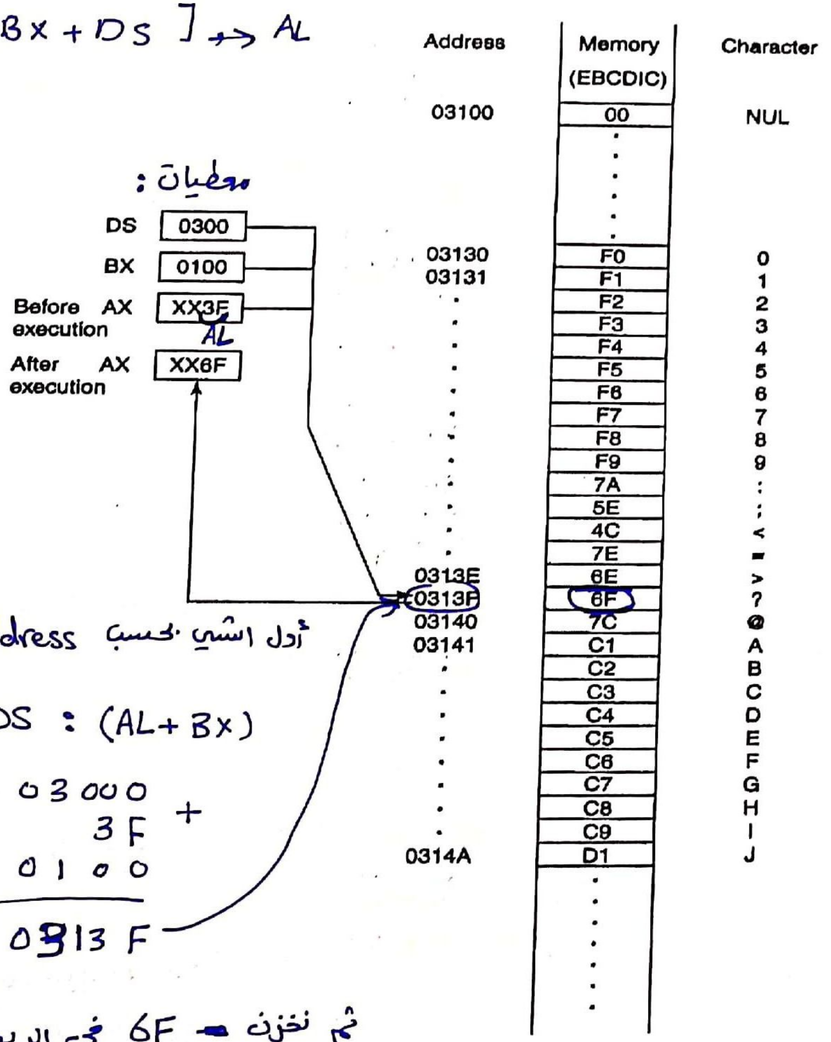
5.1 Data-Transfer Instructions – Translate Instruction

XLAT [AL + BX + DS] → AL

- Application: ASCII to EBCDIC Translation
- Fixed EBCDIC table coded into memory starting at offset in BX
- Individual EBCDIC codes placed in table at displacement (AL) equal to the value of their equivalent ASCII character
- A = 41H in ASCII, A = C1H in EBCDIC
- Place the value C1H in memory at address (41H+(BX) +(DS)0), etc.
- Example

XLAT
(DS) = 0300H
(BX) = 0100H
(AL) = 3FH → 6FH = ? (Question mark)

معطيات :



Address	Memory (EBCDIC)	Character
03100	00	NUL
03130	F0	0
03131	F1	1
	F2	2
	F3	3
	F4	4
	F5	5
	F6	6
	F7	7
	F8	8
	F9	9
	7A	:
	5E	;
	4C	<
	7E	=
0313E	6E	>
0313F	6F	?
03140	7C	@
03141	C1	A
	C2	B
	C3	C
	C4	D
	C5	E
	C6	F
	C7	G
	C8	H
	C9	I
0314A	D1	J

أول اشي نحسب address :-

DS : (AL+BX)

$$\begin{array}{r} 03000 \\ 3F \\ 0100 \\ \hline 0313F \end{array} +$$

ثم نخزن ← 6F في الريجستر AL .

AL befor execution = 3F
AL after execution = 6F

5.1 Data-Transfer Instructions – Load Effective Address and Load Full Pointer Instructions

- The LEA, LDS, and LES Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load effective address	LEA Reg16,EA لازم	EA → (Reg16)	None
LDS	Load register and DS	LDS Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (DS)	None
LES	Load register and ES	LES Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (ES)	None

(a)

- Load effective address instruction
  - Used to load an address pointer offset from memory into a register.
  - General format:

LEA Reg16,EA

~~[illegible]~~

~~[illegible]~~

  - Operation:

EA → (Reg16)

  - Source unaffected:
  - Flags unaffected

مثال :-

LEA Ax,[BX+1234]

يعني حاصل جمع

BX + 1234

- نخزنه في Ax

مثال:-

LEA Ax, [1234]

EA

1234 → Ax

يكافئ

⇒ Mov Ax, 1234

5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load effective address	LEA Reg16,EA	EA → (Reg16)	None
LDS	Load register and DS	LDS Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (DS)	None
LES	Load register and ES	LES Reg16,Mem32	(Mem32) → (Reg16) (Mem32+2) → (ES)	None

(a)

الفرق بينهم أن LEA تخزن القيمة كما هي داخل الريجستر. لكن LDS يأخذ العنوان وينقل الداتا منه إلى الريجستر.

- Load full pointer
  - Used to load a full address pointer from memory into a segment register and a register
  - Segment base address
  - Offset
  - General format and operation for LDS

    LDS Reg16,EA

    (EA) → (Reg16)

    (EA+2) → (DS)
  - LES operates the same, except initializes ES

مثال:-

LES CX,[SI]

باخذ العنوان

DS : SI

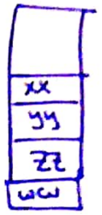
وبجيب منه الداتا

أول ٢ بايت xx yy بخزنهم في CX

و تاني ٢بايت zz ww بخزنهم في ES.

مثال:-

LDS Ax , [1234]



أول اشي بنجيب العنوان

DS : 1234

بعد ما أحسبه بجيب الداتا منه و لازم أخذ (4 byte) وبخزنهم.

بهاد المثال باخد xx yy وبخزنهم في Ax

وباخد ZZ ww وبخزنهم في DS.

5.1 Data-Transfer Instructions – Load Effective Address and Load Full Pointer Instructions

- Example

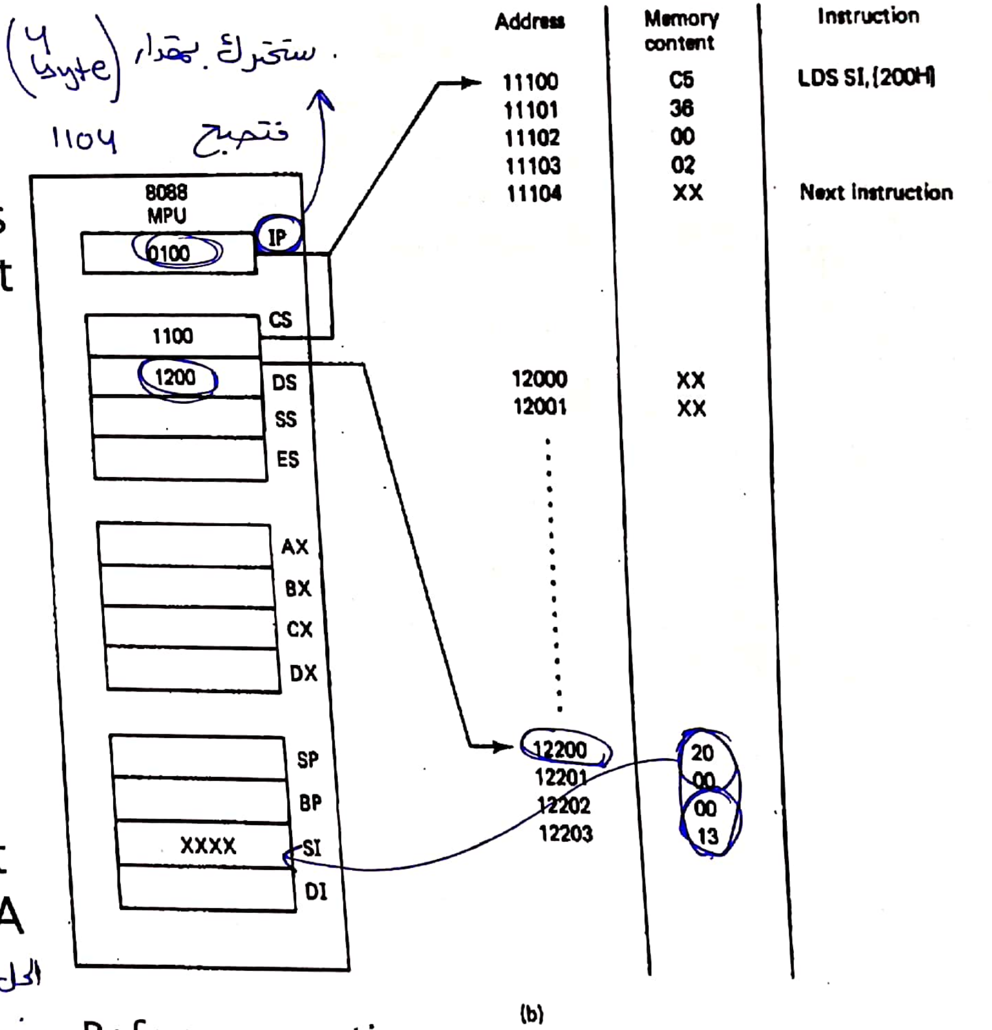
LDS SI,[200H]

Source = pointer to DS:200H→ 32 bits
Destination = SI → word pointer offset
DS → word pointer SBA
Operation: (DS:200H) → (SI)
(DS:202H) → (DS)

- State before fetch and execution

CS:IP = 1100:0100 = 11100H
LDS instruction code = C5360002H
(11102H,11103H) = (EA) = 0200H
(DS) = 1200H
(SI) = XXXX → don't care state
(DS:EA) = 12200H = 0020H = Offset
(DS:EA+2) = 12202H = 1300H = SBA

(4 byte) ستتحرك بمقدار .
1104 فتصبح



Before execution

الحل :-
نحسب العنوان DS : 200 ونأخذ منه الداتا ونخزنها في SI و DS .

$$\begin{array}{r} 12000 \\ +\ 200 \\ \hline 12200 \end{array}$$

العنوان هذا يحتوي 20 00 00 13 فتخزن SI [0020] ونخزن DS [1300]

5.1 Data-Transfer Instructions – Load Effective Address and Load Full Pointer Instructions

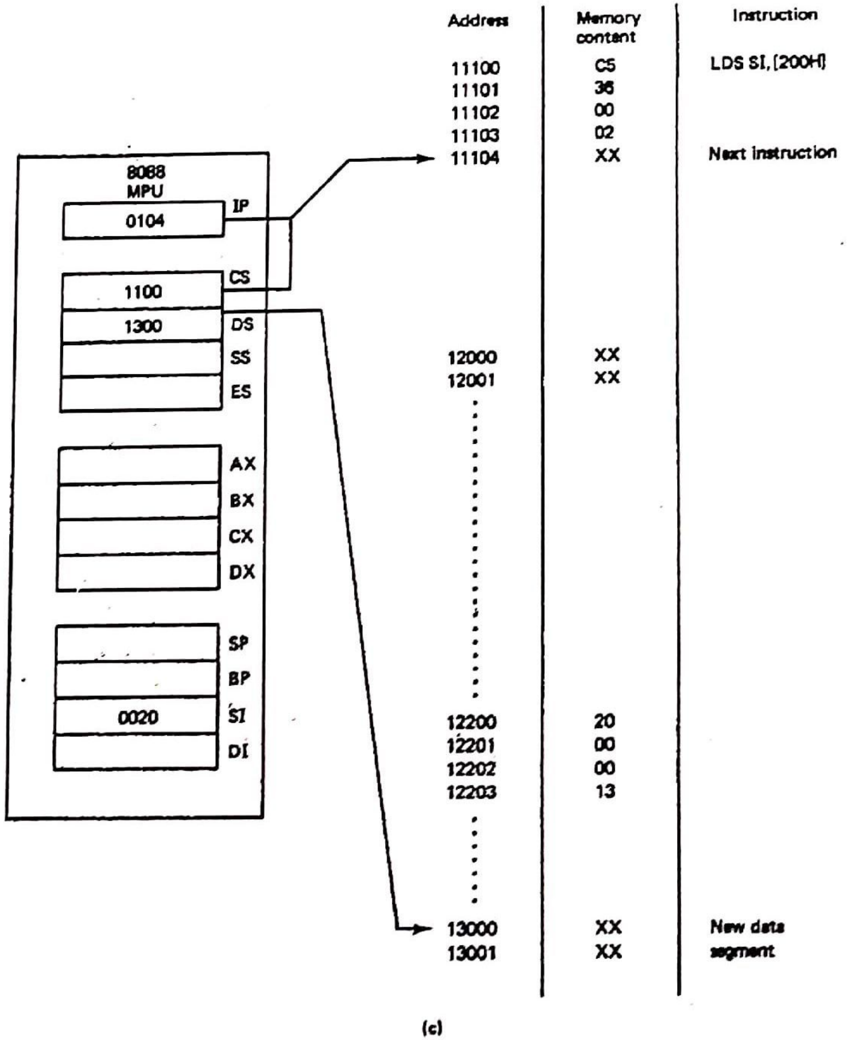
- Example
- State after execution

CS:IP = 1100:0104 = 11104H

01004H → points to next sequential instruction

(DS) = 1300H → defines a new data segment

(SI) = 0020H → defines new offset into DS



Address	Memory content	Instruction
11100	C5	LDS SI, [200H]
11101	36	
11102	00	
11103	02	
11104	XX	Next instruction
12000	XX	
12001	XX	
12200	20	
12201	00	
12202	00	
12203	13	
13000	XX	New data segment
13001	XX	

(c)

After execution

5.1 Data-Transfer Instructions –

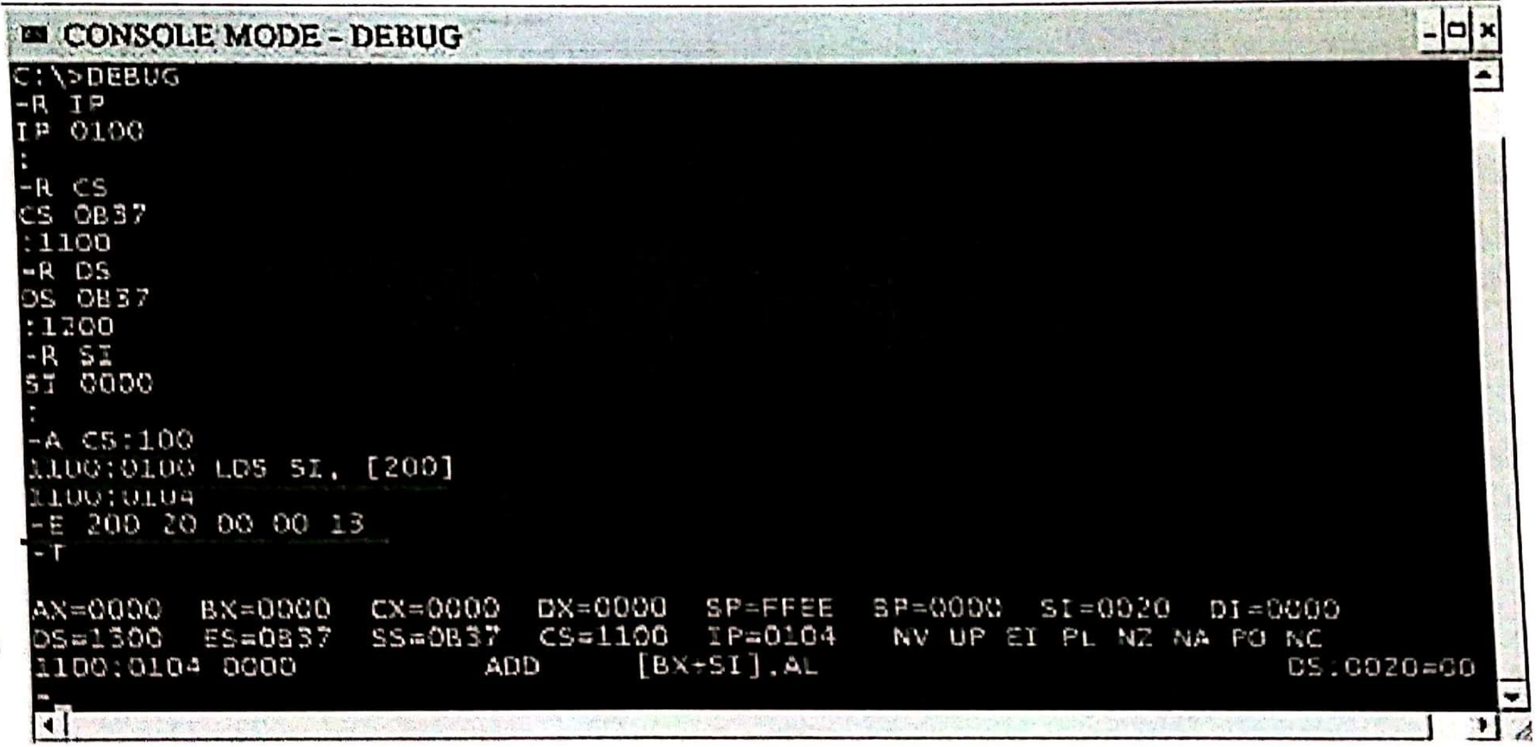
Load Effective Address and Load Full Pointer Instructions

- EXAMPLE

Verify the following instruction using DEBUG program.

LDS SI, [200H]

- Solution:



```
C:\>DEBUG
-R IP
IP 0100
:
-R CS
CS 0B37
:1100
-R DS
DS 0B37
:1200
-R SI
SI 0000
:
-A CS:100
1100:0100 LDS SI, [200]
1100:0104
-E 200 20 00 00 13
-T

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0020  DI=0000
DS=1300  ES=0B37  SS=0B37  CS=1100  IP=0104   NV UP EI PL NZ NA PO NC
1100:0104 0000          ADD     [BX+SI],AL                   DS:0020=00
-
```

5.1 Data-Transfer Instructions - Load Effective Address and Load Full Pointer Instructions

- EXAMPLE

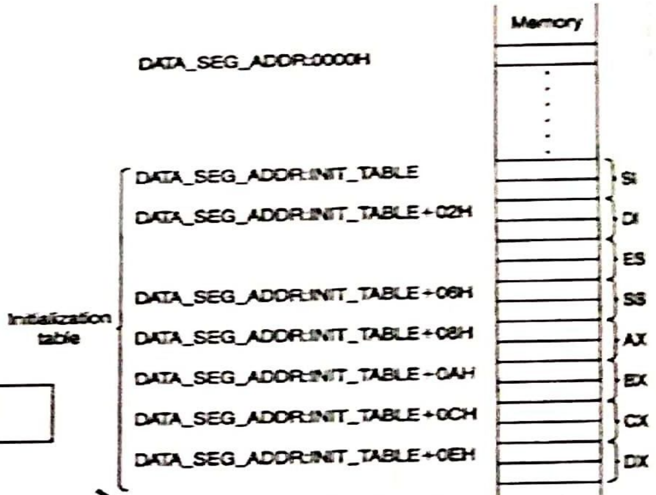
Initializing the internal registers of the 8088 from a table <u>in memory</u>.

- Solution:

- DS loaded via AX with immediate value using move instructions
DATA_SEG_ADDR → (AX) → (DS)
- Index register SI loaded with move from table
(INIT_TABLE,INIT_TABLE+1) → SI
- DI and ES are loaded with load full pointer instruction
(INIT_TABLE+2,iNIT_TABLE+3) → DI
(INIT_TABLE+4,INIT_TABLE+5) → ES

LES DI,[INIT_TABLE+2]

- SS loaded from table via AX using move instructions
(INIT_TABLE+6,INIT_TABLE+7) → AX → (SS)
- Data registers loaded from table with move instructions
(INIT_TABLE+8,INIT_TABLE+9) → AX
(INIT_TABLE+A,INIT_TABLE+B) → BX
(INIT_TABLE+C,INIT_TABLE+D) → CX
(INIT_TABLE+E,INIT_TABLE+F) → DX



~~MOV AX, [INIT_TABLE]~~
~~MOV SS, AX~~
~~LDS SI, [INIT_TABLE+02H]~~
~~LES DI, [INIT_TABLE+06H]~~
~~MOV AX, [INIT_TABLE+0AH]~~
~~MOV BX, [INIT_TABLE+0CH]~~
~~MOV CX, [INIT_TABLE+0EH]~~
~~MOV DX, [INIT_TABLE+10H]~~

5.2 Arithmetic Instructions

- The arithmetic instructions include
  - Addition
  - Subtraction
  - Multiplication
  - Division
- Data formats
  - Unsigned binary bytes
  - Signed binary bytes
  - Unsigned binary words
  - Signed binary words
  - Unpacked decimal bytes
  - Packed decimal bytes
  - ASCII numbers

BCD and ASCII Arithmetic: **The microprocessor allows arithmetic manipulation of both BCD (Binary Coded Decimal) and ASCII data.**

ASCII and BCD

- ASCII representation
  - * Numbers are stored as a string of ASCII characters
    - » Example: 1234 is stored as 31 32 33 34H

      ASCII Code for 0,1, ..,9: 30H, 31H, …, 39H
- BCD representation
  - * Unpacked BCD
    - » Example: 1234 is stored as 01 02 03 04H
      - – Additional byte is used for sign
        - ✈Sign byte: 00H for + and 80H for –
  - * Packed BCD
    - » Saves space by packing two digits into a byte
      - – Example: 1234 is stored as 12 34H

Note: ASCII = Unpacked + 30H

Processing ASCII Numbers

- 8086 provides four instructions

  `aaa` – ASCII adjust after addition

  `aas` – ASCII adjust after subtraction

  `aam` – ASCII adjust after multiplication

  `aad` – ASCII adjust before division
  - * These instructions do not take any operands
    - » Operand is assumed to be in AL

Processing Packed BCD Numbers

- Two instructions to process packed BCD numbers

  `daa` – Decimal adjust after addition

  ✈Used after **`add`** or **`adc`** instruction

  `das` – Decimal adjust after subtraction

  ✈Used after **`sub`** or **`sbb`** instruction
  - * No support for multiplication or division
    - » For these operations
      - – Unpack the numbers
      - – Perform the operation
      - – Repack them

ASCCI Adjustment Instructions

- Arithmetic operations are performed on numbers expressed in ASCII format , but we want the final result in decimal. (this saves conversions!)
  - Result must be in AL
- After the arithmetic operation, an adjustment must be performed on the result to convert it to the equivalent decimal result.
- This is main principle for all ASCCI adjust operations.

5.2 Arithmetic Instructions – Addition Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
ADD	Addition	ADD D, S	(S) + (D) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
ADC	Add with carry	ADC D, S	(S) + (D) + (CF) → (D) Carry → (CF)	OF, SF, ZF, AF, PF, CF
INC	Increment by 1	INC D	(D) + 1 → (D)	OF, SF, ZF, AF, PF
AAA	ASCII adjust for addition	AAA		AF, CF OF, SF, ZF, PF undefined
DAA	Decimal adjust for addition	DAA		SF, ZF, AF, PF, CF, OF, undefined

(a)

half بتشبه adder

full بتشبه adder

مثال :-
INC Ax
يعني Ax+1 → Ax

مثل :-
INC [1234]
يعني بروح عالميموري وبجمع واحد للداتا يلي في الميموري يلي عنوانها
DS: 1234

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

(a) Addition Instructions. (b) Allowed operands for ADD and ADC (c) Allowed operands for INC

- Variety of arithmetic instruction provided to support integer addition—core instructions are
  - ADD → Addition
  - ADC → Add with carry
  - INC → Increment
- Addition Instruction—ADD
  - ADD format and operation:

    ADD D,S

    (S) +(D) → (D)
  - Add values in two registers

    ADD AX,BX

    (AX) + (BX) → (AX)
  - Add a value in memory and a value in a register

    ADD [DI],AX

    (DS:DI) + (AX) → (DS:DI)
  - Add an immediate operand to a value in a register or memory

    ADD AX,100H

    (AX) + IMM16 → (AX)
- Flags updated based on result
  - CF, OF, SF, ZF, AF, PF

5.2 Arithmetic Instructions – Addition Instructions

- EXAMPLE

  Assume that the AX and BX registers contain 1100_{16} and $0ABC_{16}$, respectively. What is the result of executing the instruction ADD AX, BX?

- Solution:

$$(BX)+(AX) = 0ABC_{16} + 1100_{16} = 1BBC_{16}$$

The sum ends up in destination register AX. That is

$$(AX) = 1BBC_{16}$$

$$CF = 0$$

يعني لا يوجد
Carry out

Ax = 1100
Bx = 0ABC

$$\text{ADD Ax, Bx} = \begin{array}{r} 1100 + \\ 0ABC \\ \hline 1BBC \end{array}$$

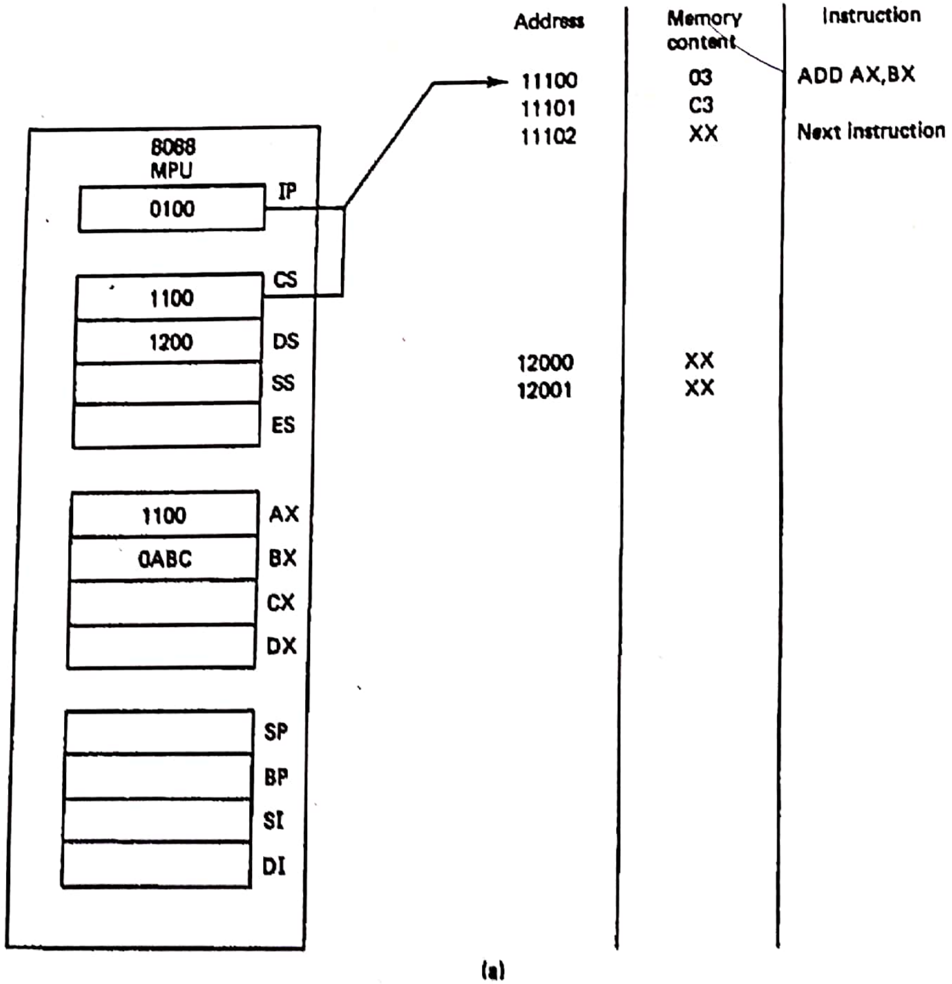
ونخزنه في Ax

$$AX \ \boxed{1BBC}$$

5.2 Arithmetic Instructions – Addition Instructions

- State before fetch and execution

CS:IP = 1100:0100 = 11100H
ADD machine code = 03C3H
(AX) = 1100H
(BX) = 0ABCH
(DS) = 1200H
(1200:0000) = 12000H = XXXX



Before execution

5.2 Arithmetic Instructions – Addition

Instructions

- State after execution

CS:IP = 1100:0102 = 11102H

11102H → points to next sequential instruction

- Operation performed

(AX) + (BX) → (AX)

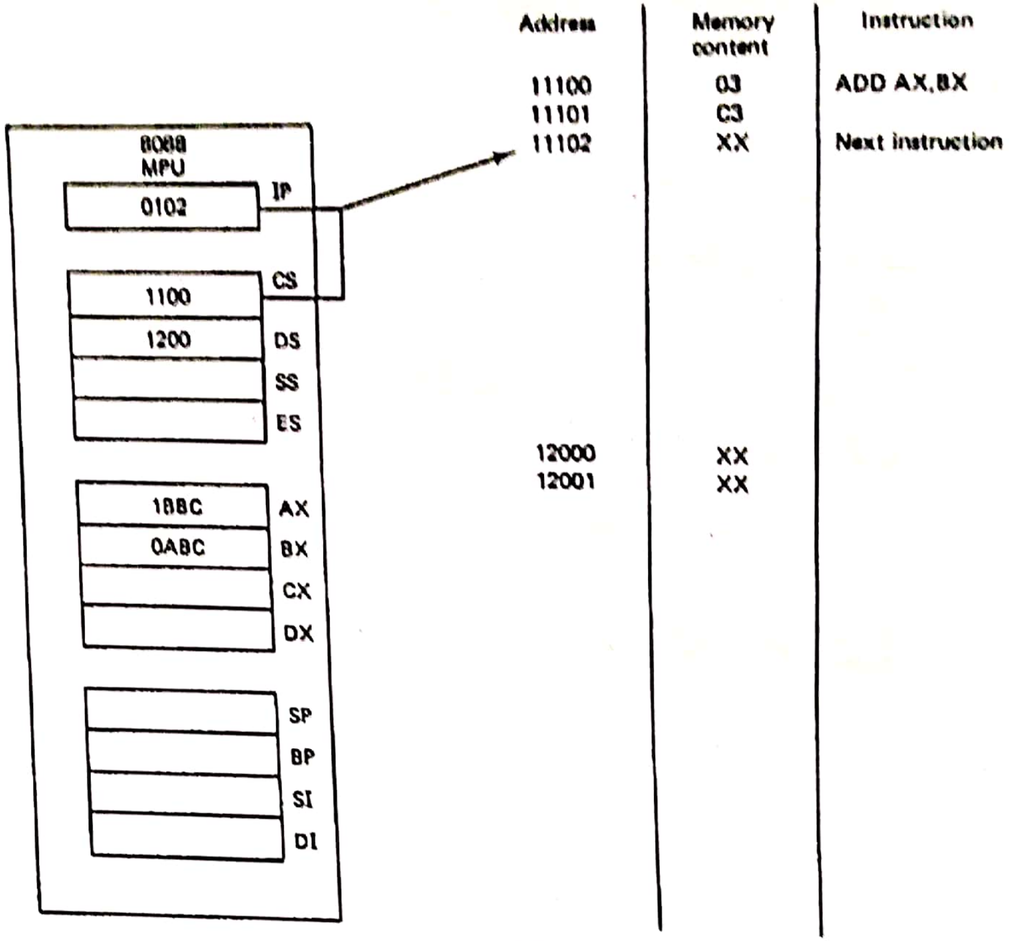
(1100H) + (0ABCH) → 1BBCH

(AX) = 1BBCH

$= 00011011\underline{10111100}_2$

(BX) = unchanged

- Impact on flags
- CF = 0 (no carry resulted)
- ZF = 0 (not zero)
- SF = 0 (positive)
- PF = 0 (odd parity)—parity flag is only based on the bits of the least significant byte



Address	Memory content	Instruction
11100	03	ADD AX,BX
11101	C3	
11102	XX	Next instruction
12000	XX	
12001	XX	

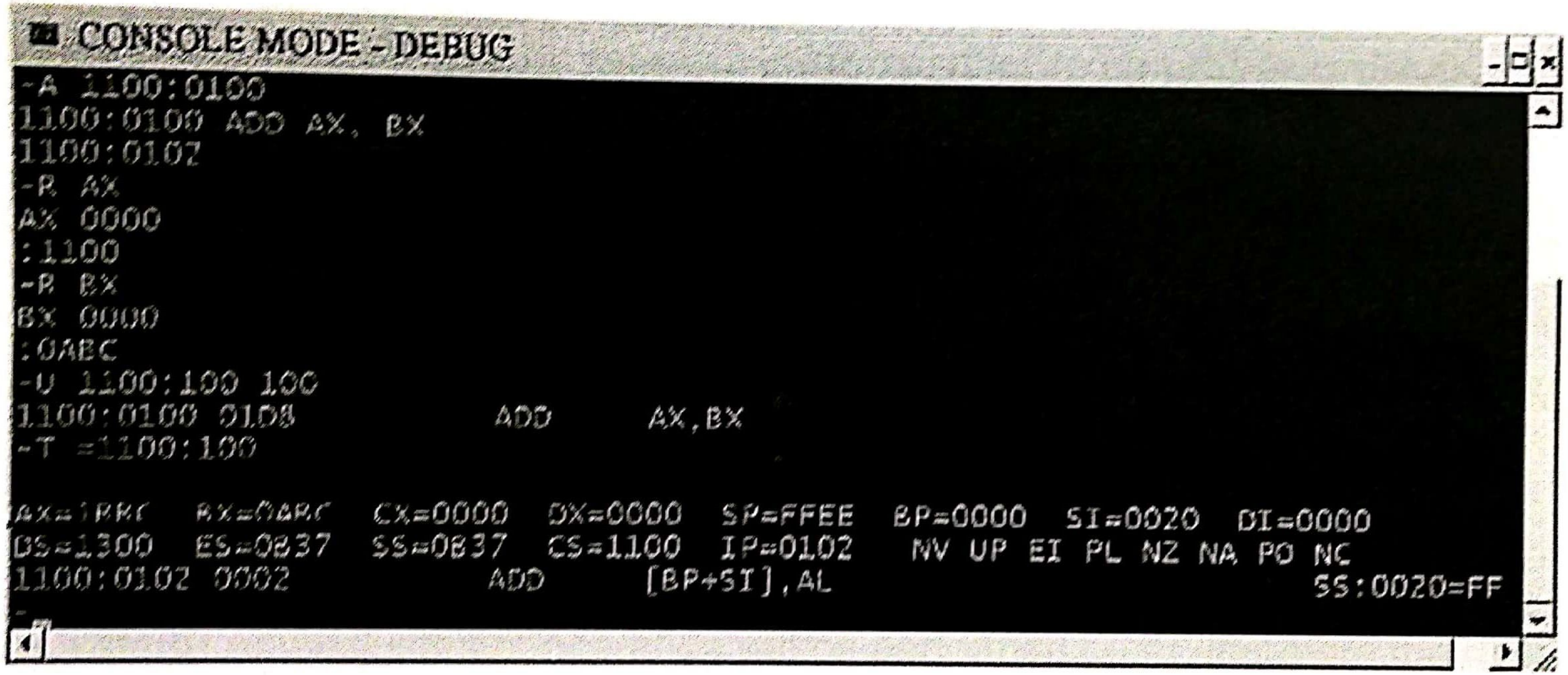
After execution

5.2 *Arithmetic Instructions* – Addition Instructions

- EXAMPLE

Verify the previous example using DEBUG program.

Solution:



```
-A 1100:0100
1100:0100 ADD AX, BX
1100:0102
-R AX
AX 0000
:1100
-R BX
BX 0000
:0ABC
-U 1100:100 100
1100:0100 01D8          ADD     AX,BX
-T =1100:100

AX=1BBC  BX=0ABC  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0020  DI=0000
DS=1300  ES=0837  SS=0837  CS=1100  IP=0102   NV UP EI PL NZ NA PO NC
1100:0102 0002          ADD     [BP+SI],AL                    SS:0020=FF
-
```

5.2 Arithmetic Instructions – Addition Instructions

- EXAMPLE

The original contents of AX, BL, word-size memory location SUM, and carry flag (CF) are 1234_{16}, AB_{16}, $00CD_{16}$, and 0_{16}, respectively. Describe the results of executing the following sequence of instruction?

```
ADD AX, [SUM]
ADC BL, 05H
INC WORD PTR [SUM]
```

- Solution:

$(AX) \leftarrow (AX)+(SUM) = 1234_{16} + 00CD_{16} = 1301_{16}$

$(BL) \leftarrow (BL)+imm8+(CF) = AB_{16} + 5_{16}+0_{16} = B0_{16}$

$(SUM) \leftarrow (SUM)+ 1_{16} = 00CD_{16} + 1_{16} = 00CE_{16}$

5.2 Arithmetic Instructions – Addition Instructions

- EXAMPLE

What is the result of executing the following instruction sequence?

ADD AL, BL

AAA

Assuming that AL contains 32_{16} (ASCII code for 2) and BL contains 34_{16} (ASCII code 4), and that AH has been cleared.

- Solution:

$(AL) \leftarrow (AL)+(BL) = 32_{16} + 34_{16} = 66_{16}$

The result after the AAA instruction is

$(AL) = 06_{16}$

$(AH) = 00_{16}$

with both AF and CF remain cleared

Important: Any adjustment operation will be performed on AL therefore the result must be always placed in AL before executing the adjustment operation

ASCII Adjustment after ADD
AAA Examples

Code	Registers
MOV AL, '2'	AL = 32H
MOV BL, '3'	BL = 33H
ADD AL, BL	AL = 65H
AAA	AL = 05

If low nibble of **AL <= 9**
- **Clear the high nibble of AL**
- AF = 0
- CF = 0

Code	Registers
MOV AL, ‘5'	AL = 35H
MOV BL, ‘6'	BL = 36H
ADD AL, BL	AL = 6BH
AAA	AX = 01H AL=01H

If low nibble of **AL > 9** or AF = 1

Clear the high nibble of AL
- AL = AL + 6
- AH = AH + 1
- AF = 1
- CF = 1

Code	Registers
MOV AL, ‘9'	AL = 39H
MOV BL, ‘9'	BL = 39H
ADD AL, BL	AL = 72H
AAA	AX = 01H AL=08H

If low nibble of AL > 9 or **AF = 1**

Clear the high nibble of AL
- AL = AL + 6
- AH = AH + 1
- AF = 1
- CF = 1

5.2 Arithmetic Instructions – Addition Instructions

- EXAMPLE

  Perform a 32-bit binary add operation on the contents of the processor's register.
- Solution:

  $(DX,CX) \leftarrow (DX,CX)+(BX,AX)$

  $(DX,CX) = FEDCBA98_{16}$

  $(BX,AX) = 01234567_{16}$

```
MOV DX, FEDCH
MOV CX, BA98H
MOV BX, 0123H
MOV AX, 4567H
ADD CX, AX
ADC DX, BX              ; Add with carry
```

5.2 Arithmetic Instructions – Subtraction Instructions

مثال:- NEG Ax

يعني 0-Ax

وخزن الناتج في Ax

ودائماً في ~~[illegible]~~ CF=1

- Variety of arithmetic instruction provided to support integer subtraction—core instructions are
  - SUB → Subtract
  - SBB → Subtract with borrow
  - DEC → Decrement
  - NEG → Negative

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	(D) – (S) → (D) Borrow → (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	(D) – (S) – (CF) → (D)	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	(D) – 1 → (D)	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	0 – (D) → (D) 1 → (CF)	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

يعني ضرب الاشارة السالبة

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Destination
Register
Memory

(d)

(a) Subtraction Instructions. (b) Allowed operands for SUB and SBB (c) Allowed operands for DEC (d) Allowed operands for NEG

مثال:- SUB Ax, Bx

يعني Ax-Bx

وخزن الناتج في Ax

والـ borrow بخزنه في CF

carry flag

مثال:- DEC Ax

يعني Ax-1

وخزنه الناتج في Ax

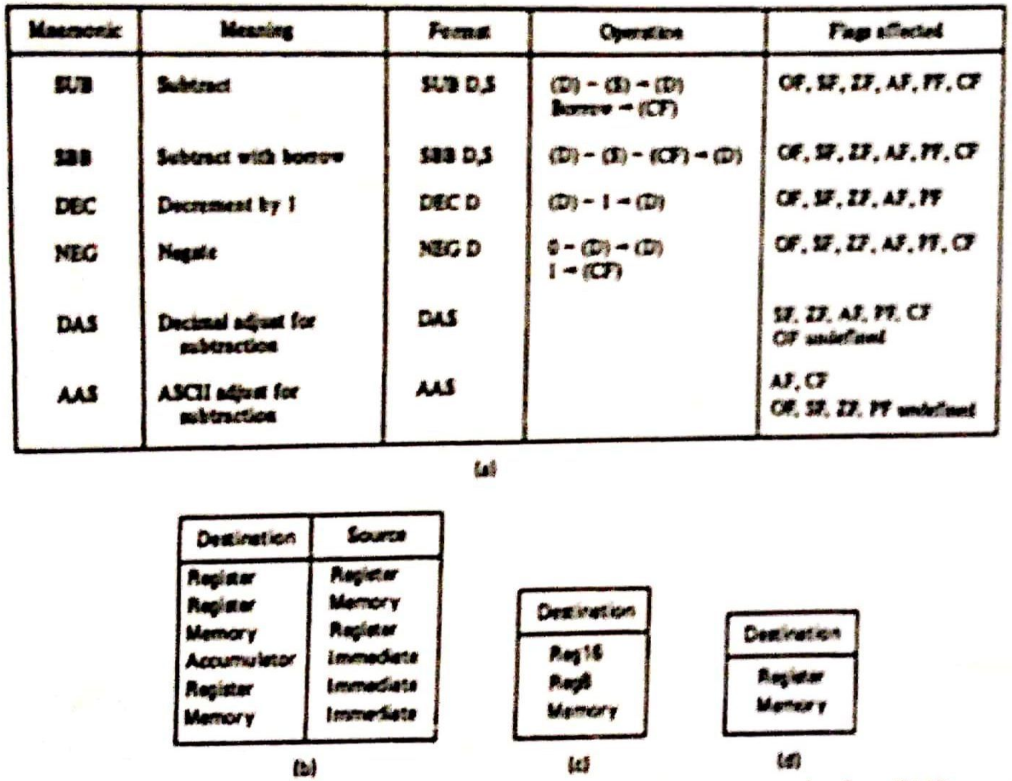
مثال:- SBB Ax, Bx

يعني Ax-Bx-CF

وخزنه الناتج في Ax

5.2 Arithmetic Instructions –

Subtraction Instructions



Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	(D) − (S) → (D) Borrow → (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	(D) − (S) − (CF) → (D)	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	(D) − 1 → (D)	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	0 − (D) → (D) 1 → (CF)	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Destination
Register
Memory

(d)

(a) Subtraction Instructions. (b) Allowed operands for SUB and SBB (c) Allowed operands for DEC (d) Allowed operands for NEG

- Subtract Instruction—SUB
  - SUB format and operation:

    SUB D,S

    (D) - (S) → (D)
  - Subtract values in two registers

    SUB AX,BX

    (AX) - (BX) → (AX)
  - Subtract a value in memory and a value in a register

    SUB [DI],AX

    (DS:DI) - (AX) → (DS:DI)
  - Subtract an immediate operand from a value in a register or memory

    SUB AX,100H

    (AX) - IMM16 → (AX)
- Flags updated based on result
  - CF, OF, SF, ZF, AF, PF

5.2 Arithmetic Instructions –

Subtraction Instructions

- Subtract with borrow instruction—SBB
  - SBB format and operation:

    SBB D,S

    (D) – (S) – (CF) → (D)
  - Used for extended subtractions
  - Subtracts two registers and carry (borrow)

    SBB AX,BX
  - Example:

    SBB BX,CX

    (BX) = 1234H

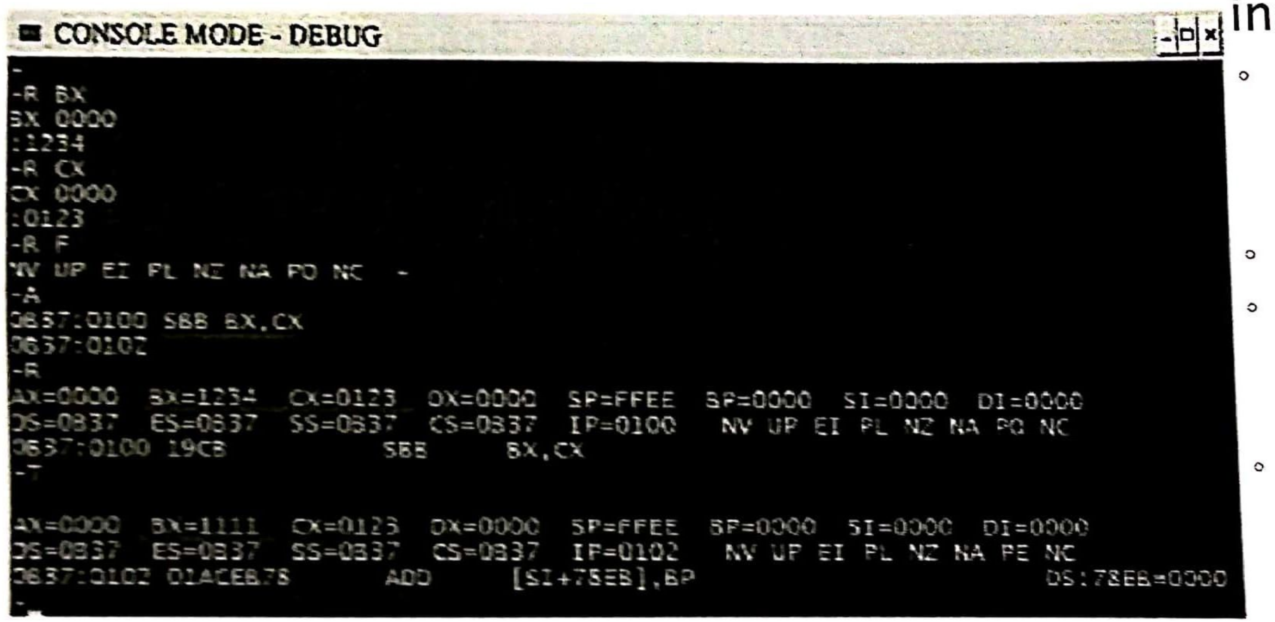
    (CX) = 0123H

    (CF) = 0

    (BX) – (CX) – (CF) → (BX)

    1234H – 0123H – 0H = 1111H

    (BX) = 1111H
  - What about CF? CF=0



```
-
-R BX
BX 0000
:1234
-R CX
CX 0000
:0123
-R F
NV UP EI PL NZ NA PO NC  -
-A
0837:0100 SBB BX,CX
0837:0102
-R
AX=0000  BX=1234  CX=0123  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0837  ES=0837  SS=0837  CS=0837  IP=0100   NV UP EI PL NZ NA PO NC
0837:0100 19CB          SBB     BX,CX
-T

AX=0000  BX=1111  CX=0123  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0837  ES=0837  SS=0837  CS=0837  IP=0102   NV UP EI PL NZ NA PE NC
0837:0102 01ACEB78      ADD     [SI+78EB],BP                       DS:78EB=0000
-
```

If we execute instead:

SBB CX, BX

The result will be:

CX= EEEF

CF= 1

SF=1

PF=0

5.2 Arithmetic Instructions – Subtraction Instructions

- Negate instruction—NEG (2's complement)
  - NEG format and operat

    NEG D

    (0) – (D) → (D)

    (1) → (CF)
  - Example:

    NEG BX

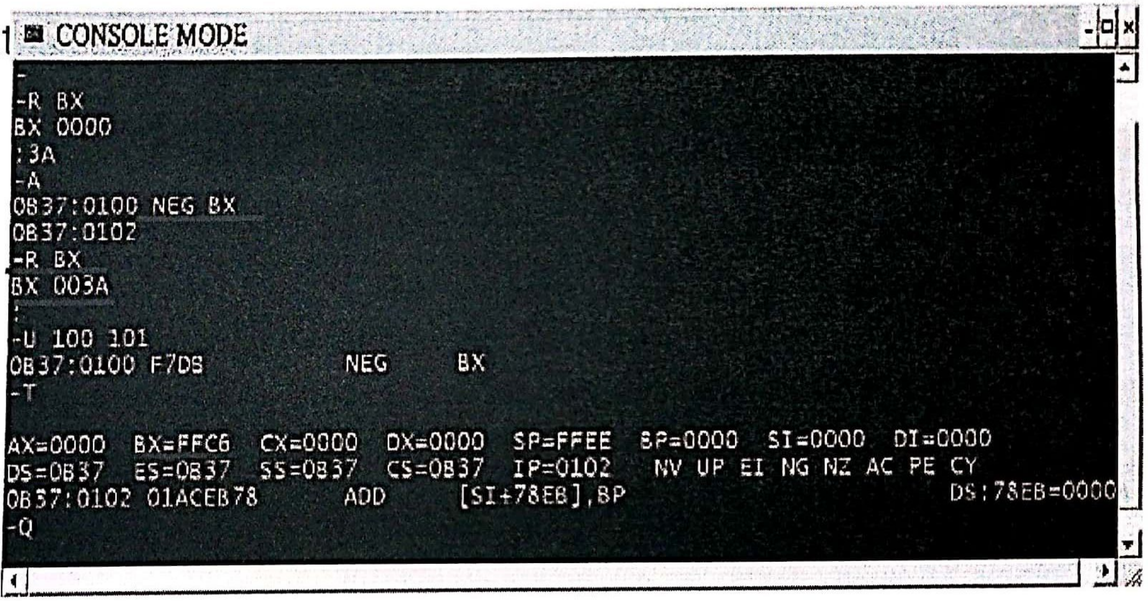
    (BX) =003AH

    (0) – (BX) → (BX)

    0000H - 003AH=

    0000H + FFC6H (2's complement) = FFC6H

    (BX) =FFC6H ; CF =1

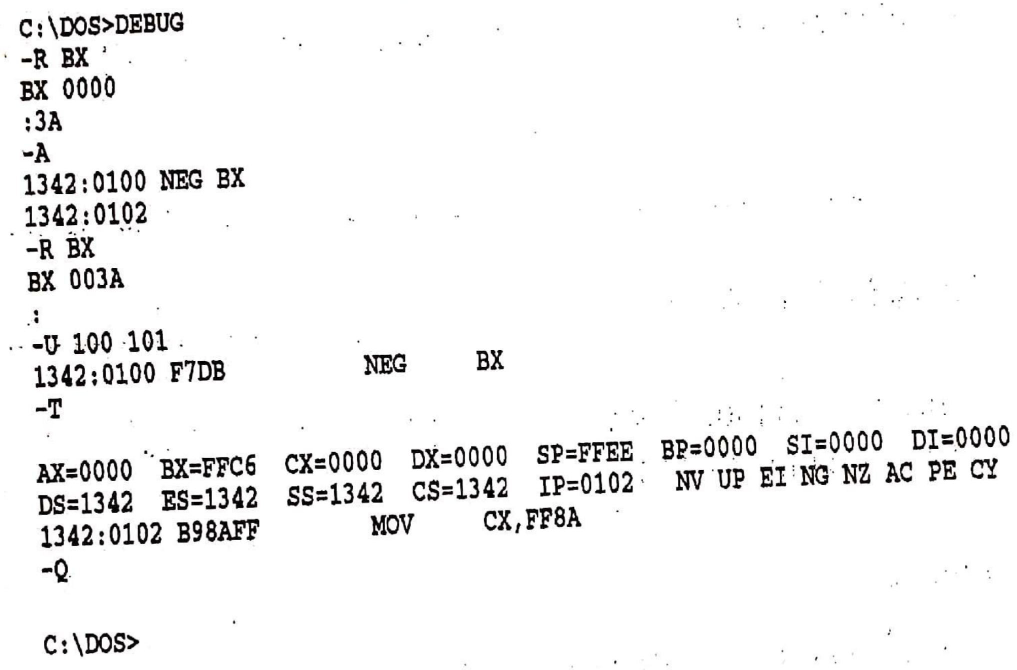


```
-
-R BX
BX 0000
:3A
-A
0B37:0100 NEG BX
0B37:0102
-R BX
BX 003A
:

-U 100 101
0B37:0100 F7DB          NEG     BX
-T

AX=0000  BX=FFC6  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0102   NV UP EI NG NZ AC PE CY
0B37:0102 01ACEB78      ADD     [SI+78EB],BP                       DS:78EB=0000
-Q
```

- Since no carry is generated in this add operation, the carry flag is complemented to give CF =1.

5.2 Arithmetic Instructions -

Subtraction Instructions



```
C:\DOS>DEBUG
-R BX
BX 0000
:3A
-A
1342:0100 NEG BX
1342:0102
-R BX
BX 003A
:
-U 100 101
1342:0100 F7DB          NEG     BX
-T

AX=0000  BX=FFC6  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0102   NV UP EI NG NZ AC PE CY
1342:0102 B98AFF        MOV     CX,FF8A
-Q

C:\DOS>
```

- Decrement instruction—DEC
  - DEC format and operation

    DEC D

    (D) – 1 → (D)
  - Used to decrement pointer—addresses
    - Example

      DEC SI

      (SI) = 0FFFH

      (SI) – 1 → SI

      0FFFH – 1 = 0FFEH

      (SI) = 0FFEH

ASCII Adjustment after SUB: AAS Examples

Code	Registers
MOV AL, '3'	AL = 33H
MOV BL, '2'	BL = 32H
SUB AL, BL	AL = 01H
AAS	AL = 01

If low nibble of AL <= 9
- **Clear the high nibble of AL (no need)**
- AF = 0
- CF = 0

Code	Registers
MOV AL, '1'	AL = 31H
MOV BL, '9'	BL = 39H
SUB AL, BL	AH= 0 AL=F8H
AAS	AH = FFH AL=02H Answer= -(10)+2=-8

if low nibble of **AL > 9** or AF = 1
Clear the high nibble of AL
- AL = AL - 6
- AH = AH - 1
- AF = 1
- CF = 1

Code	Registers
MOV AL, '2'	AL = 32H
MOV BL, '3'	BL = 33H
SUB AL, BL	AH= 0 AL=FFH
AAS	AH = FFH AL=09H Answer= -(10)+9=-1

if low nibble of **AL > 9 or AF = 1**
Clear the high nibble of AL
- AL = AL - 6
- AH = AH - 1
- AF = 1
- CF = 1

5.2 Arithmetic Instructions -

Subtraction Instructions

- EXAMPLE

  Perform a 32-bit binary subtraction for variable X and Y.
- Solution:

```
MOV SI, 200H ; Initialize pointer for X
MOV DI, 100H ; Initialize pointer for Y
MOV AX, [SI] ; Subtract LS words
SUB AX, [DI]
MOV [SI],AX ; Save the LS word of result
MOV AX, [SI]+2 ; Subtract MS words
SBB AX, [DI]+2
MOV [SI]+2, AX ; Save the MS word of result
```

DX
32 —— 16 15 —— Ax 0

فبحتاج 2 ريجستريز للتخزين

16 bits
16 bits *
32 bits

مثال:- MUL CX
معناه
CX · Ax → 32 bits
أول 16 بتخزنهم في Ax
والباقي بتخزنهم في Dx

مثال:- MUL BL
معناه
AL · BL → Ax
8 bits 8 bits 16 bits

8 bits
8 bits *
16 bits
أقصى حد

5.2 *Arithmetic Instructions* – Multiplication Instructions

كتبت ملاحظة في الصفحة التالية.

Mnemonic	Meaning	Format	Operation	Flags Affected
MUL	Multiply (unsigned)	MUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is FF_{16} in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
IMUL	Integer multiply (signed)	IMUL S	(AL) · (S8) → (AX) (AX) · (S16) → (DX),(AX)	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) Q((AX)/(S8)) → (AL) R((AX)/(S8)) → (AH) (2) Q((DX,AX)/(S16)) → (AX) R((DX,AX)/(S16)) → (DX) If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than 8001_{16}, then type 0 interupt occurs	OF, SF, ZF, AF, PF, CF undefined
AAM	Adjust AL for multiplication	AAM	Q((AL)/10) → (AH) R((AL)/10) → (AL)	SF, ZF, PF OF, AF,CF undefined
AAD	Adjust AX for division	AAD	(AH) · 10 + (AL) → (AL) 00 → (AH)	SF, ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) → (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSB of AX) → (All bits of DX)	None

(a)

destination لا يوجد

حالتين
ميموري
ريجستر

Source
Reg8
Reg16
Mem8
Mem16

(b)

(a) Multiplication and Division Instructions. (b) Allowed operands

- Integer multiply instructions—MUL and IMUL
  - Multiply two unsigned or signed byte or word operands
- General format and operation
  - MUL S = Unsigned integer multiply
  - IMUL S = Signed integer multiply

(AL) X (S8)→ (AX)

product gives 16 bit result

(AX) X (S16) → (DX), (AX)

16- bit product gives 32 bit result

  - Source operand (S) can be an 8-bit or 16-bit value in a register or memory
  - AX assumed to be destination for 16 bit result
  - DX,AX assumed destination for 32 bit result
  - Only CF and OF flags updated; other undefined

AL · (S8) → Ax

Ax · (S16) → (Dx , Ax)

higher 16 bits

lower 16 bits

5.2 Arithmetic Instructions –
Multiplication Instructions

- EXAMPLE

  The 2's-complement signed data contents of AL are -1 and that of CL are -2. What result is produced in AX by executing the following instruction?

  MUL CL and IMUL CL

- Solution:

  $(AL) = -1$ (as 2's complement) $= 11111111_2 = FF_{16}$

  $(CL) = -2$ (as 2's complement) $= 11111110_2 = FE_{16}$

Executing the MUL instruction gives

$(AX) = 11111111_2 x 11111110_2 = 1111110100000010_2 = FD02_{16}$

Executing the IMUL instruction gives

$(AX) = -1_{16} \times -2_{16} = 2_{16} = 0002_{16}$

If the operation is MUL CX ➔ multiply CX by AX and store the higher order word of the result in DX and the low order word of the result in AX

5.2 *Arithmetic Instructions* – Multiplication Instructions

- ***In general:***

1 – The multiplication may take one of two forms

- Multiply AL by 8-bit operand ➔ result will be 16-bit saved in AX.
- Multiply AX by 16-bit operand ➔ result will be 32 but saved in DX,AX.

2- To perform unsigned multiplication convert the two numbers into binary and perform the multiplication.

3- To perform signed multiplication

- If both operands are positive or both are negative ➔ ignore the sign and multiply the numbers normally
- If one operand is positive and the other is negative ➔ multiply the numbers and perform 2's complement for the result

* $\frac{16 \text{ bits}}{8 \text{ bits}}$ فإن الناتج 16 bits

* مثال :- Div CL

معناه $Q\left(\frac{AX}{CL}\right) \longrightarrow AL$

وباقي القسمة يتم تخزينه في AH

$R\left(\frac{AX}{CL}\right) \rightarrow AH$

* مثال :- Div Bx

معناه $\frac{Q(DX, AX)}{BX} \rightarrow AX$

$R\left(\frac{DX, AX}{BX}\right) \rightarrow DX$

$\frac{\text{٣٢ بت}}{\text{١٦ بت}}$

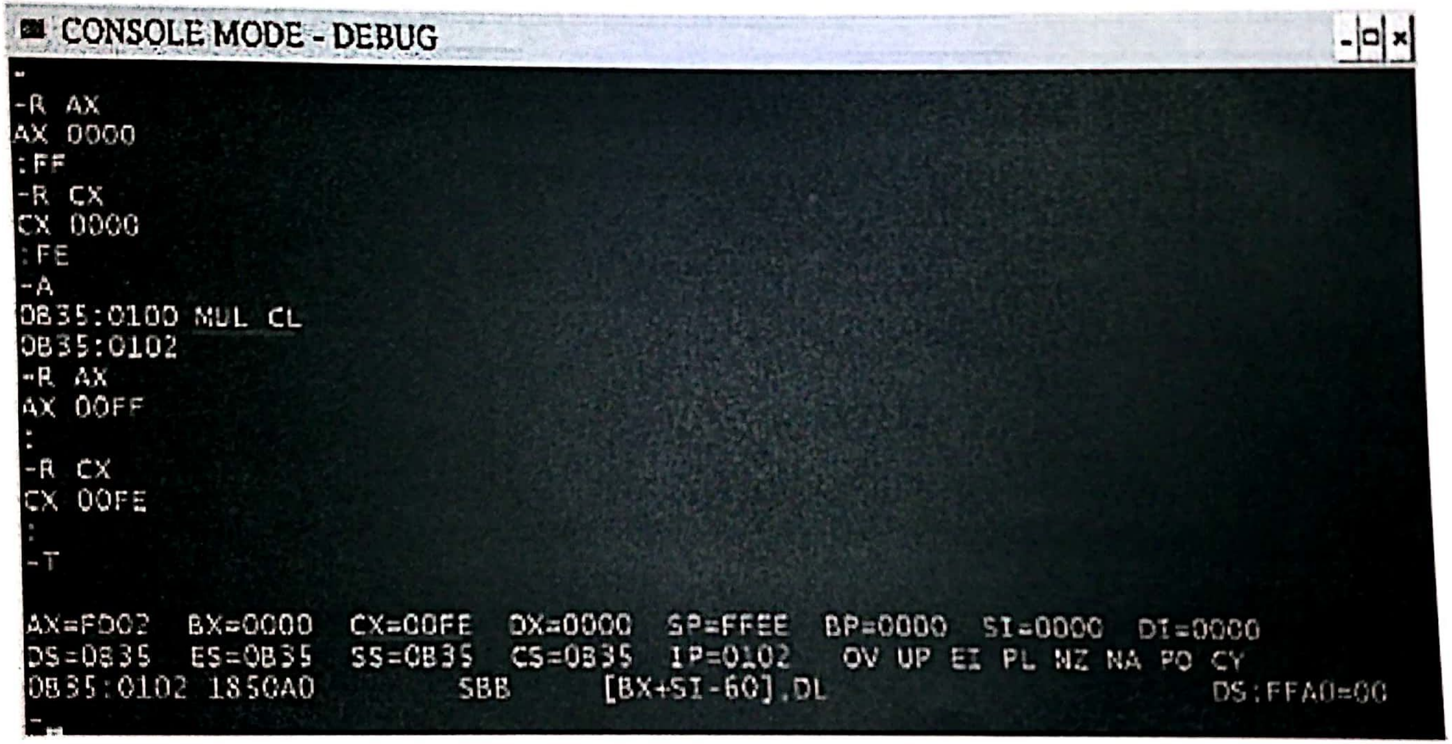
5.2 Arithmetic Instructions -

Multiplication Instructions

- EXAMPLE

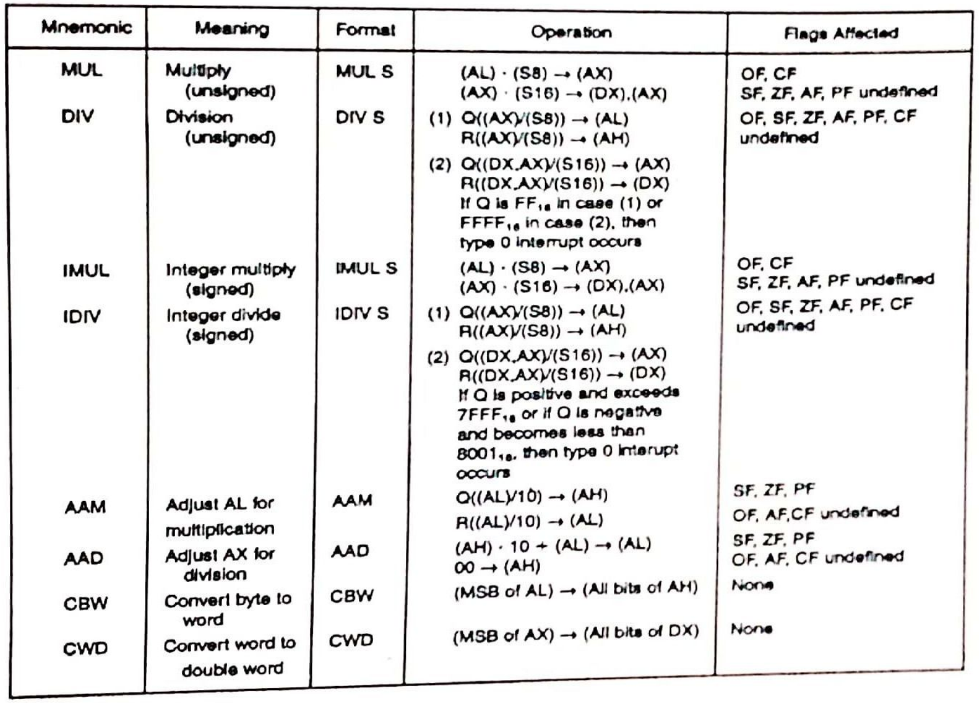
Verify the previous example using DEBUG program.

Solution:



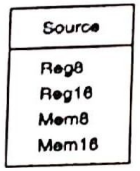
```
-
-R AX
AX 0000
:FF
-R CX
CX 0000
:FE
-A
0B35:0100 MUL CL
0B35:0102
-R AX
AX 00FF
:
-R CX
CX 00FE
:
-T

AX=FD02  BX=0000  CX=00FE  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0102   OV UP EI PL NZ NA PO CY
0B35:0102 1850A0        SBB     [BX+SI-60],DL                      DS:FFA0=00
-
```

5.2 *Arithmetic Instructions* - Division Instructions



Mnemonic	Meaning	Format	Operation	Flags Affected
MUL	Multiply (unsigned)	MUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX),(AX)$	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$ (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is FF_{16} in case (1) or $FFFF_{16}$ in case (2), then type 0 interrupt occurs	OF, SF, ZF, AF, PF, CF undefined
IMUL	Integer multiply (signed)	IMUL S	$(AL) \cdot (S8) \rightarrow (AX)$ $(AX) \cdot (S16) \rightarrow (DX),(AX)$	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) $Q((AX)/(S8)) \rightarrow (AL)$ $R((AX)/(S8)) \rightarrow (AH)$ (2) $Q((DX,AX)/(S16)) \rightarrow (AX)$ $R((DX,AX)/(S16)) \rightarrow (DX)$ If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than 8001_{16}, then type 0 interupt occurs	OF, SF, ZF, AF, PF, CF undefined
AAM	Adjust AL for multiplication	AAM	$Q((AL)/10) \rightarrow (AH)$ $R((AL)/10) \rightarrow (AL)$	SF, ZF, PF OF, AF,CF undefined
AAD	Adjust AX for division	AAD	$(AH) \cdot 10 + (AL) \rightarrow (AL)$ $00 \rightarrow (AH)$	SF, ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) → (All bits of AH)	None
CWD	Convert word to double word	CWD	(MSB of AX) → (All bits of DX)	None

(a)



Source
Reg8
Reg16
Mem8
Mem16

(b)

(a) Multiplication and Division Instructions. (b) Allowed operands

- Integer divide instructions—DIV and IDIV
  - Divide unsigned- DIV S
  - Operations:

(AX) / (S8) → (AL) =quotient
(AH) = remainder

- 16 bit dividend in AX divided by 8-bit divisor in a register or memory.
- Quotient of result produced in AL
- Remainder of result produced in AH

(DX,AX) / (S16) → (AX) =quotient
(DX) = remainder

- 32 bit dividend in DX,AX divided by 16-bit divisor in a register or memory
- Quotient of result produced in AX
- Remainder of result produced in DX

  - Divide error (Type 0) interrupt may occur.

Division Examples

UNSINGED DIV	Registers (in Hex)
MOV AX, 14	AH = 00 , AL =0E
MOV BL, 3	BL = 03
DIV BL	AH = 02 , AL =04
MOV AX, 14	AH = 00 , AL =0E
MOV BL, -3	BL = FD
DIV BL	Operation = 14/253 AH = 0E, AL =00
MOV AX, -14	AH = FF , AL =F2
MOV BL, 3	BL = 03
DIV BL	Operation = 65522/3 ERROR : INT0 (Divide overflow)
MOV AX, -14	AH = FF , AL =F2
MOV BL, -3	BL = FD
DIV BL	ERROR : INT0 (Divide overflow)

SIGNED DIV	Registers (in Hex)
MOV AX, 14	AH = 00 , AL =0E
MOV BL, 3	BL = 03
IDIV BL	AH = 02 , AL =04
MOV AX, 14	AH = 00 , AL =0E
MOV BL, -3	BL = FD
IDIV BL	AH = 02 , AL =FC
MOV AX, -14	AH = FF , AL =F2
MOV BL, 3	BL = 03
IDIV BL	AH =FE , AL =FC
MOV AX, -14	AH = FF , AL =F2
MOV BL, -3	BL = FD
IDIV BL	AH = FE , AL =04

SIGNED DIV:
The sign for the remainder == sign of the dividend

5.2 Arithmetic Instructions – Convert Instructions

- Used to sign extension signed numbers for division
- Operations
  - CBW = convert byte to word

  (MSB of AL) → (all bits of AH)
  - CWD = convert word to double word

  (MSB of AX) → (all bits of DX)
- Application:
  - To divide two signed 8-bit numbers, the value of the dividend must be sign extended in AX
    - Load into AL
    - Use CBW to sign extend to 16 bits

5.2 Arithmetic Instructions – Division Instructions

- ***In general:***

1– The division may take one of two forms

- Divide AX by 8–bit operand ➔ The division is performed between AX/ 8–bit operand. AL will contain the quotient of the result and AH will contain the remainder of the result. IF quotient is FF then interrupt occurs.
- Divide DX,AX by 16–bit operand ➔ The division is performed between DX,AX/ 16–bit operand. AX will contain the quotient of the result and DX will contain the remainder of the result. IF quotient is FFFF then interrupt occurs.

2– The way in which you perform either a singed or unsigned division is similar to the mechanism used in the multiplication instruction

3– The sign for the remainder is always similar to the sign of the dividend
ex. –26 / 8 ➔ Quotient=–3 and Remainder = –2

5.2 *Arithmetic Instructions* – Convert Instructions

- EXAMPLE

  What is the result of executing the following instructions?

  MOV AL, 0A1H
  CBW
  CWD

- Solution:

  $(AL) = A1_{16} = 10100001_2$

Executing the CBW instruction extends the MSB of AL

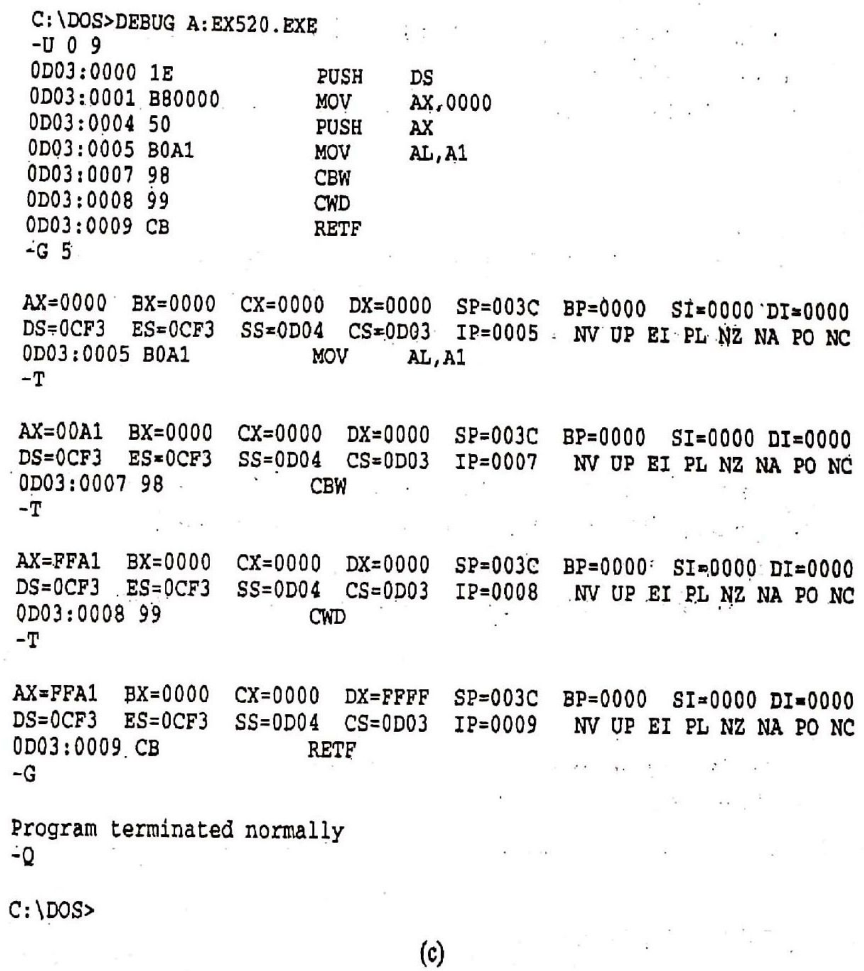
$(AH) = 11111111_2 = FF_{16}$ or $(AX) = 1111111110100001_2$

Executing CWD instruction, we get

$(DX) = 1111111111111111_2 = FFFF_{16}$

That is,

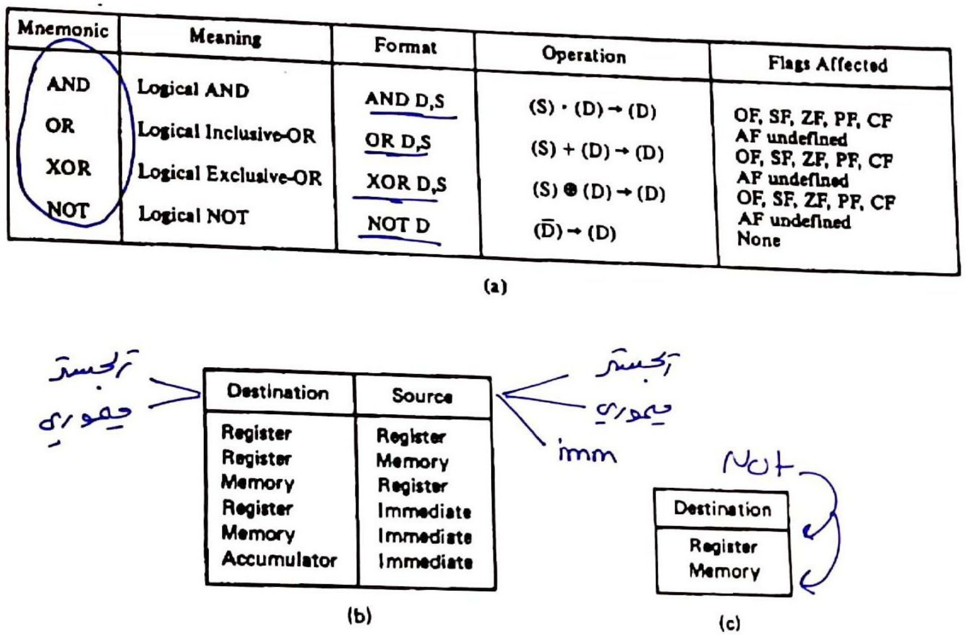
$(AX) = FFA1_{16}$ $(DX) = FFFF_{16}$



```
C:\DOS>DEBUG A:EX520.EXE
-U 0 9
0D03:0000 1E            PUSH    DS
0D03:0001 B80000        MOV     AX,0000
0D03:0004 50            PUSH    AX
0D03:0005 B0A1          MOV     AL,A1
0D03:0007 98            CBW
0D03:0008 99            CWD
0D03:0009 CB            RETF
-G 5

AX=0000  BX=0000  CX=0000  DX=0000  SP=003C  BP=0000  SI=0000 DI=0000
DS=0CF3  ES=0CF3  SS=0D04  CS=0D03  IP=0005   NV UP EI PL NZ NA PO NC
0D03:0005 B0A1          MOV     AL,A1
-T

AX=00A1  BX=0000  CX=0000  DX=0000  SP=003C  BP=0000  SI=0000 DI=0000
DS=0CF3  ES=0CF3  SS=0D04  CS=0D03  IP=0007   NV UP EI PL NZ NA PO NC
0D03:0007 98            CBW
-T

AX=FFA1  BX=0000  CX=0000  DX=0000  SP=003C  BP=0000  SI=0000 DI=0000
DS=0CF3  ES=0CF3  SS=0D04  CS=0D03  IP=0008   NV UP EI PL NZ NA PO NC
0D03:0008 99            CWD
-T

AX=FFA1  BX=0000  CX=0000  DX=FFFF  SP=003C  BP=0000  SI=0000 DI=0000
DS=0CF3  ES=0CF3  SS=0D04  CS=0D03  IP=0009   NV UP EI PL NZ NA PO NC
0D03:0009 CB            RETF
-G

Program terminated normally
-Q

C:\DOS>
```

(c)

5.3 Logic Instructions



Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND D,S	(S) · (D) → (D)	OF, SF, ZF, PF, CF AF undefined
OR	Logical Inclusive-OR	OR D,S	(S) + (D) → (D)	OF, SF, ZF, PF, CF AF undefined
XOR	Logical Exclusive-OR	XOR D,S	(S) ⊕ (D) → (D)	OF, SF, ZF, PF, CF AF undefined
NOT	Logical NOT	NOT D	($\bar{D}$) → (D)	None

(a)

رجستر
ميموري

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

رجستر
ميموري
imm

(b)

NOt

Destination
Register Memory

(c)

(a) Logic Instructions. (b) Allowed operands for AND, OR, and XOR (c) Allowed operands for NOT

* مثال:-

NOT Ax

معناه

$\overline{Ax} \xrightarrow{\text{بخزنها في}} Ax$

تقارن كل بت لوحدة

* مثال:-

Ax → 1100 0011 1111 0111

Bx → 1111 1111 0000 0110

AND Ax, Bx

الحل:- 1100 0011 0000 0110

- Variety of logic instructions provided to support logical computations
  - AND → Logical AND
  - OR → Logical inclusive-OR
  - XOR → Logical exclusive-OR
  - NOT → Logical NOT
- Logical AND Instruction—AND
  - AND format and operation:

    AND D,S

    (S) AND (D) → (D)
  - Logical AND of values in two registers

    AND AX,BX

    (AX) AND (BX) → (AX)
  - Logical AND of a value in memory and a value in a register

    AND [DI],AX

    (DS:DI) AND (AX) → (DS:DI)
  - Logical AND of an immediate operand with a value in a register or memory

    AND AX,100H

    (AX) AND IMM16 □(AX)
  - Flags updated based on result
    - CF, OF, SF, ZF, PF
    - AF undefined

:- (logic instruction) الأبليكيشن على

AND ← مرتبطة بـ Mask ①
OR ← مرتبطة بـ Set ②

5.3 Logic Instructions

- EXAMPLE

Describe the results of executing the following instructions?

MOV AL, 01010101B → خزن القيمة في AL → AL 01010101
AND AL, 00011111B → اعمل Add للرقم مع القيمة الموجودة في AL → 00010101
OR AL, 11000000B → " " " " " OR " → 11010101
XOR AL, 00001111B
NOT AL
⋮

- Solution:

$(AL)=01010101_2 \cdot 00011111_2= 00010101_2=15_{16}$

Executing the OR instruction, we get

$(AL)= 00010101_2 +11000000_2= 11010101_2=D5_{16}$

Executing the XOR instruction, we get

$(AL)= 11010101_2 \oplus 00001111_2= 11011010_2=DA_{16}$

Executing the NOT instruction, we get

$(AL)= (NOT)11011010_2 = 00100101_2=25_{16}$

5.3 Logic Instructions

- EXAMPLE

  Verify the previous example using DEBUG program.

- Solution:

```
C:\DOS>DEBUG
-A
1342:0100 MOV AL,55
1342:0102 AND AL,1F
1342:0104 OR AL,C0
1342:0106 XOR AL,0F
1342:0108 NOT AL
1342:010A
-T

AX=0055  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0102   NV UP EI PL NZ NA PO NC
1342:0102 241F          AND     AL,1F
-T

AX=0015  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0104   NV UP EI PL NZ NA PO NC
1342:0104 0CC0          OR      AL,C0
-T

AX=00D5  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0106   NV UP EI NG NZ NA PO NC
1342:0106 340F          XOR     AL,0F
-T

AX=00DA  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=0108   NV UP EI NG NZ NA PO NC
1342:0108 F6D0          NOT     AL
-T

AX=0025  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1342  ES=1342  SS=1342  CS=1342  IP=010A   NV UP EI NG NZ NA PO NC
1342:010A 2B04          SUB     AX,[SI]                      DS:0000=20CD
-Q

C:\DOS>
```

X•0 → 0

X•1 → X

(ANding with zero) يعني

5.3 *Logic Instructions* – Mask Application

- Application- Masking bits with the logic instructions
  - Mask—to clear a bit or bits of a byte or word to 0
    - AND operation can be used to perform the mask operation:
    - 1 AND 0 → 0; 0 and 0 → 0
      - bit or bits are masked by ANDing with 0
    - 1 AND 1 → 1; 0 AND 1 → 0
      - ANDing a bit or bits with 1 results in no change
  - Example: Masking the upper 12 bits of a value in a register

AND AX,000FH

(AX) =FFFF

IMM16 AND (AX) → (AX)

000FH AND FFFFH = 0000000000001111_2 AND 1111111111111111_2

= 0000000000001111_2 بحط مكان الخانة 0 , 12 , 4 أصفار , والباقي وحدات

= 000FH

لو طلب مني

Mask Ax

بعمل AND Ax,0000

لأن لو حطيت

صفر واحد ع ع

بعمل and لخانة

واحدة فقط [4 bits].

لو طلب ماسك فقط

لأول 2 bits

بعمل

AND Ax , 1100

مثال :- Mask A_{12}, A_4, A_0 → Ax

الحل :-

AND Ax, $\underbrace{0111}_{7}\ \underbrace{1111}_{F}\ \underbrace{0111}_{7}\ \underbrace{0111}_{7}$

AND Ax, 7F77

دقيقة 38

Setting : ORing with one

5.3 Logic Instructions – ~~Mask~~ Application

لو طلب set A0,A14

- OR operation can be used to set a bit or bits of a byte or word to 1 OR Ax,0010 0000 0000 0001
  = OR Ax,2001
  - X OR 0 → X; result is unchanged
  - X or 1 → 1; result is always 1
- Example: Setting a control flag in a byte memory location to 1

```
MOV AL,[CONTROL_FLAGS]
OR AL, 10H ; 00010000 sets fifth bit -b4
MOV [CONTROL_FLAGS],AL
```

- Executing the above instructions, we get

$(AL) = XXXXXXXX_2$ OR $00010000_2 = XXX1XXXX_2$

تغير الموقع تبع الداتا لليمين أو لليسار .

5.4 Shift Instructions

* بالنسبة لل (count) ، اذا بدي أحرك الداتا بمقدار ابيت ، بحط رقم ١ ، أما ازا أكتر بحط CL .

Mnemonic	Meaning	Format	Operation	Flags Affected
SAL/SHL	Shift arithmetic left/shift logical left	SAL/SHL D,Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bits positions on the right with zeros	CF, PF, SF, ZF AF undefined OF undefined if count ≠ 1
SHR	Shift logical right	SHR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with zeros	CF, PF, SF, ZF AF undefined OF undefined if count ≠ 1
SAR	Shift arithmetic right	SAR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with the original most significant bit	SF, ZF, PF, CF AF undefined OF undefined if count ≠1

نفس المعاملة → SAL/SHL

عادي بعوض صفر مكان الخانة الفارغة → SHR

لازم أحافظ على ال sign → SAR

(a)

logical shift → بكون الداتا عبارة عن AND , OR

arithmetic shift → بكون الداتا عبارة عن أرقام

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

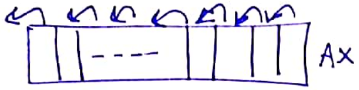
ميموري رجستر

(b)

(a) Shift Instructions. (b) Allowed operands

* مثال:- SHL Ax,1

يعني محتوى الريجستر Ax -يجب أن تتحرك بمقدار ابت باتجاه اليسار



Every shift operation is equivalent to :

*** multiplication by 2 for Shift left**

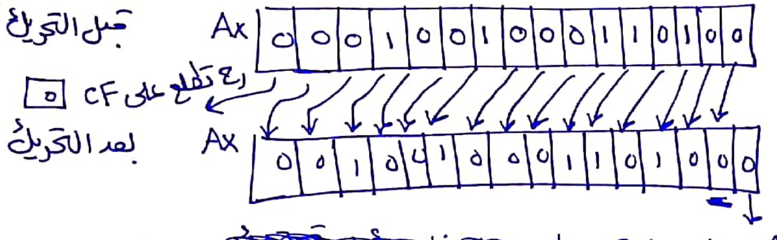
*** dividing by 2 for Logical Shift right**

- Variety of shift instructions provided
  - SAL/SHL → Shift arithmetic left/shift logical left
  - SHR → Shift logical right
  - SAR → Shift arithmetic right

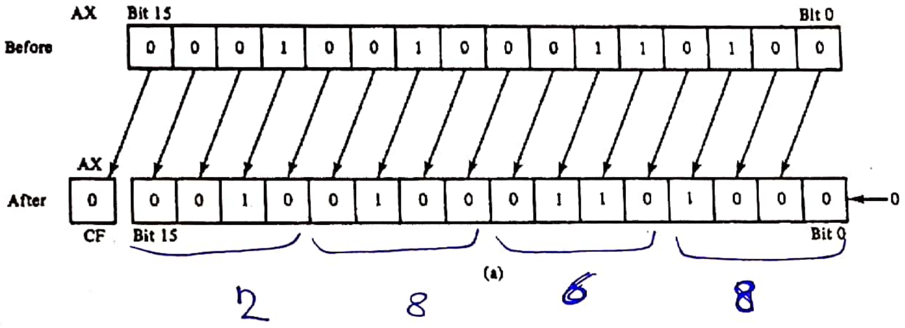
* Mov CL,3
SHL Ax,CL
يعني حرك الداتا بمقدار ٣ .

- Perform a variety of shift left and shift right operations on the bits of a destination data operand
- Basic shift instructions—SAL/SHL, SHR, SAR
  - Destination may be in either a register or a storage location in memory
  - Shift count may be:
    - 1 = one bit shift
    - CL = 1 to 255 bit shift
- Flags updated based on result
  - CF, SF, ZF, PF
  - AF undefined
  - OF undefined if Count ≠ 1

5.4 Shift Instructions – Operation of the SAL/SHL Instruction



شو بحط هون؟ بحط 0 [illegible]



(a)

يعني (Ax القديمة *2) وتذهب bit 15 إلى CF .

فإذا عملت 2 shift بضرب Ax القديمة *4 وتذهب 14 إلى CF

واذا عملت 3 shift بضرب Ax القديمة *8 وتذهب 13 إلى CF

- Typical instruction—count of 1

  SHL AX,1 يعني بدي أحولك لليسار. بمقدار (1)

- Before execution

  Dest = (AX) = 1234H

  = 00010010001101002

  Count = 1

  CF = X

- Operation:
  - The value in all bits of AX are shifted left one bit position
  - Emptied LSB is filled with 0
  - Value shifted out of MSB goes to carry flag

- After execution

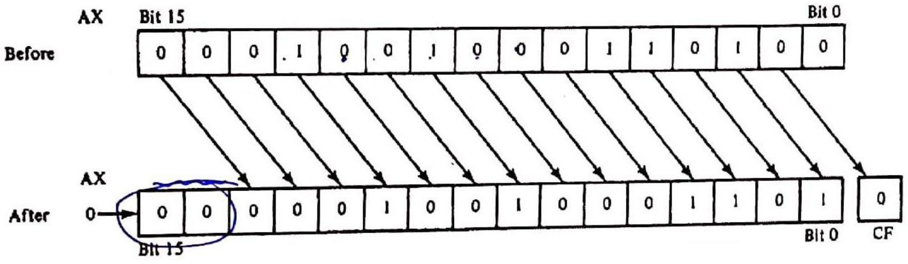
  Dest = (AX) = 2468H

  = 0010010001101000_2

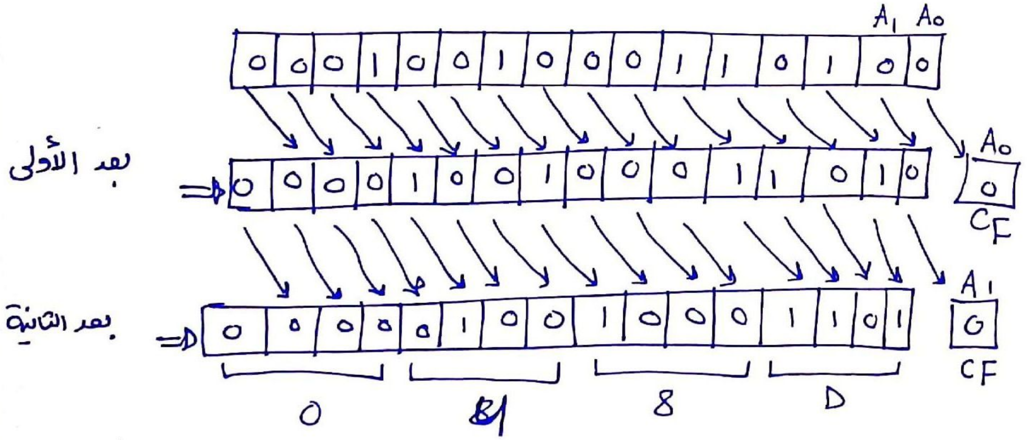
  CF = 0

- Conclusion:
  - MSB has been isolated in CF and can be acted upon by control flow instruction- conditional jump
  - Result has been multiplied by 2

5.4 *Shift Instructions* – Operation of the SHR Instruction



(b)



- Typical instruction—count in CL

  للیمین $\boxed{\text{SHR AX,CL}}$

- Before execution

  Dest = (AX) = 1234H = 4660_{10}

  = 0001001000110100_2

  Count = (CL) = 02H يعني تحريك مرتين

  CF = X

- Operation: (2bit بمقدار.]
  - The value in all bits of AX are shifted right two bit positions
  - Emptied MSBs are filled with 0s
  - Value shifted out of LSB goes to carry flag
- After execution

  Dest = (AX) = 048DH = 1165_{10}

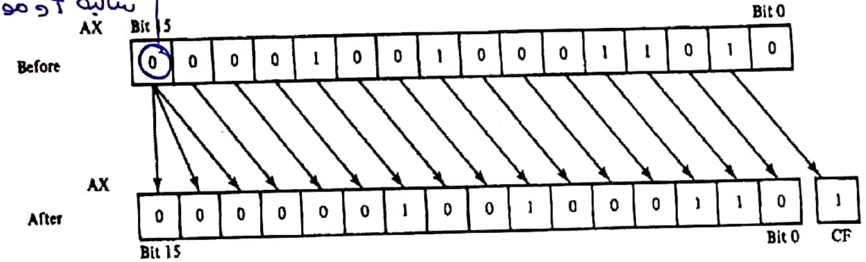
  = 0000010010001101_2

  CF = 0

- Conclusion
  - Bit 1 has been isolated in CF and can be acted upon by control flow instruction- conditional jump
  - Result has been divided by 4

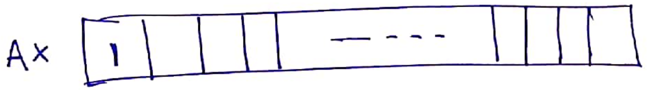
5.4 Shift Instructions – Operation of the SAR Instruction

تعبر عن sign ، يجب
الحفاظ على الاشارة
سالبة أو موجبة.

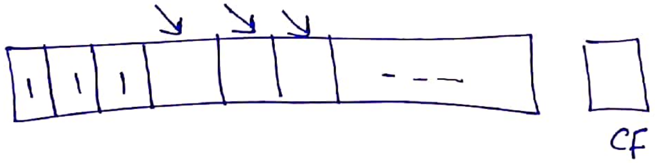


- Typical instruction—count in CL
  SAR AX,CL
- Before execution—arithmetic implies signed numbers
  Dest = (AX) = 091AH
  = 0000100100011010$_2$ = +2330
  Count = CL = 02H
  CF = X
- Operation:
  - The value in all bits of AX are shifted right two bit positions
  - Emptied MSB is filled with the value of the sign bit
  - Values shifted out of LSB go to carry flag
- After execution
  Dest = (AX) = 0246H
  = 0000001001000110$_2$ = +582
  CF = 1
- Conclusion
  - Bit 1 has been isolated in CF and can be acted upon by control flow instruction– conditional jump
  - Result has been signed extended
  - Result value has been divided by 4 and rounded to integer: 4 X +582 = +2328

مثال :-



لو بدي أعمل shift right ثلاث مرات ، لازم أكرر آخر خانة ثلاث مرات حتى أحافظ على ال sign فيصير هيك :-



5.4 Shift Instructions – Operation of the SAR Instruction

- EXAMPLE

Assume that CL contains 02_{16} and AX contains $091A_{16}$. Determine the new contents of AX and the carry flag after the instruction SAR AX, CL is executed.

- Solution:

Initial $(AX)=0000100100011010_2$
shift AX right twice: $(AX)=\underline{00}00001001000110_2=0246_{16}$

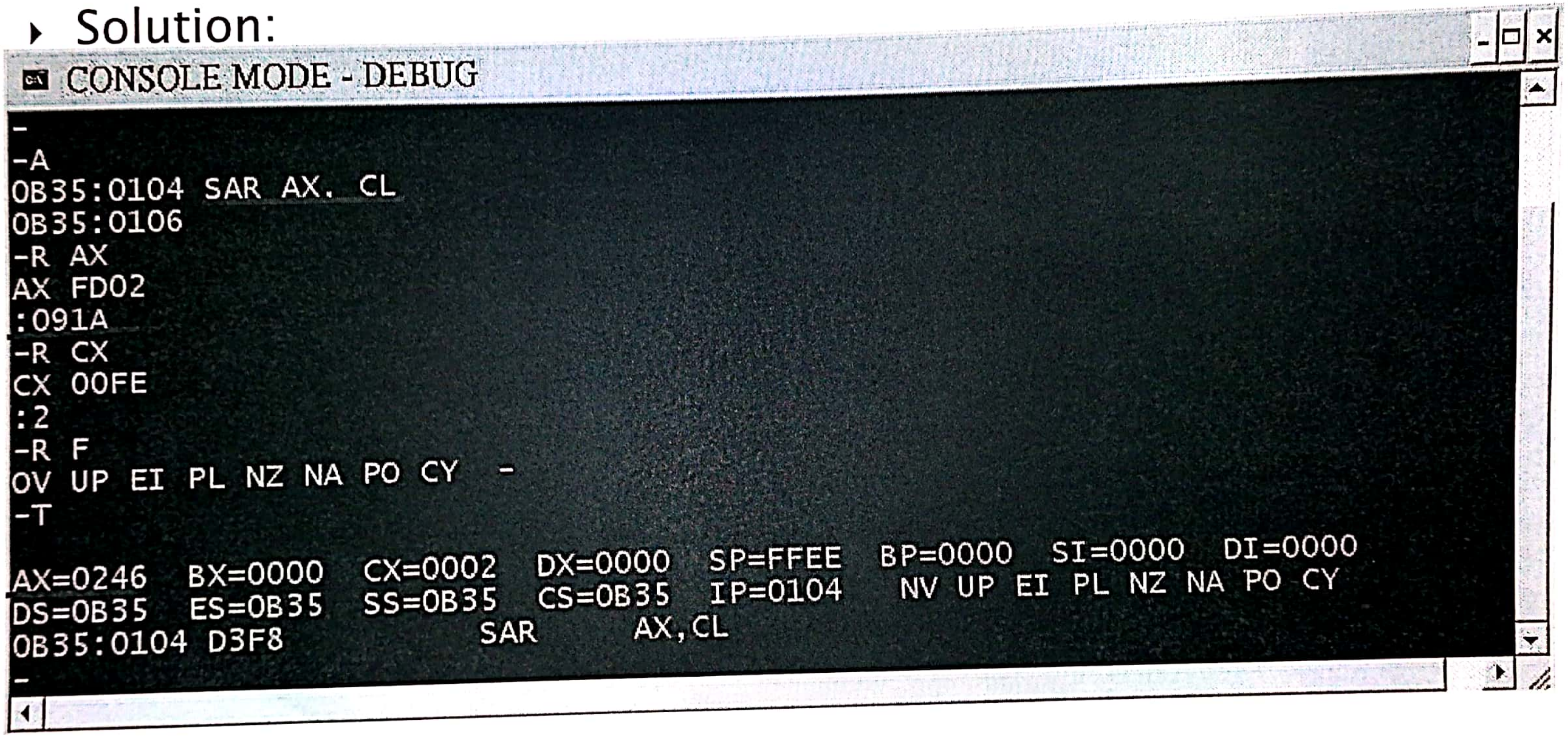
and the carry flag is $(CF)=1_2$

5.4 Shift Instructions – Operation of the SAR Instruction

- EXAMPLE

  Verify the previous example using DEBUG program.

- Solution:



```
-
-A
0B35:0104 SAR AX. CL
0B35:0106
-R AX
AX FD02
:091A
-R CX
CX 00FE
:2
-R F
OV UP EI PL NZ NA PO CY  -
-T

AX=0246  BX=0000  CX=0002  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0104   NV UP EI PL NZ NA PO CY
0B35:0104 D3F8           SAR      AX,CL
-
```

5.4 Shift Instructions – Operation of the SAR Instruction

- EXAMPLE

Isolate the bit B_3 of the byte at the offset address CONTROL_FLAGS.

- Solution:

```
MOV AL, [CONTROL_FLAGS]
MOV CL, 04H
SHR AL, CL
```

Executing the instructions, we get

$(AL)=0000B_7B_6B_5B_4$

and

$(CF)=B_3$

5.5 Rotate Instructions :-

هي حالة خاصة من الـ shift

بس الفرق إنه بالـ shift كان البت الزايدة تروح على CF .

بس بالـ (rotate)، الخانة يلي بتطلع بتحل محل الخانة الفاضية وبتعطي copy للـ CF

RoL Ax , count

ازا بدي نعمل واحد بكتب (1)

وازا بدي أكثر من واحد بكتب (CL)

CF ليس جزء من rotation path

CF يُعتبر جزء من الـ Path فبصير عنا 17 bit

Mnemonic	Meaning	Format	Operation	Flags Affected
ROL	Rotate left	ROL D,Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the leftmost bit goes back into the rightmost bit position.	CF OF undefined if count ≠ 1
ROR	Rotate right	ROR D,Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost bit goes into the leftmost bit position.	CF OF undefined if count ≠ 1
RCL	Rotate left through carry	RCL D,Count	Same as ROL except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1
RCR	Rotate right through carry	RCR D,Count	Same as ROR except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1

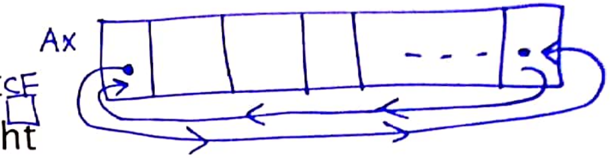
(a)

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

(b)

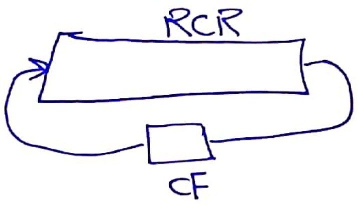
(a) Rotate Instructions. (b) Allowed operands

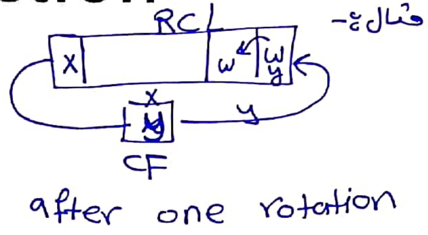
- Variety of rotate instructions provided:
  - ROL → Rotate left
  - ROR → Rotate right
  - RCL → Rotate left through carry
  - RCR → Rotate right through carry

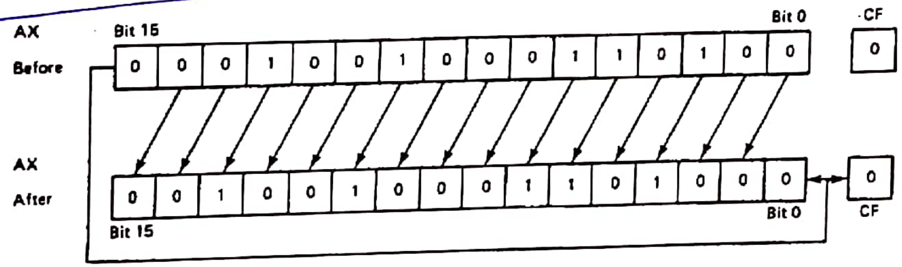


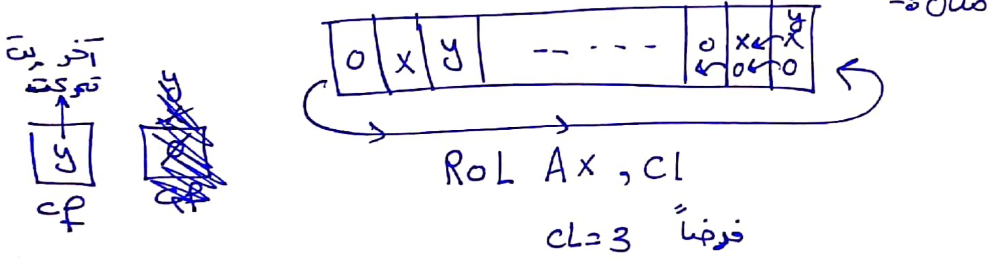
- Perform a variety of rotate left and rotate right operations on the bits of a destination data operand
  - Overview of function
  - Destination may be in either a register or a storage location in memory
  - Rotate count may be:
    - 1 = one bit rotate
    - CL = 1 to 255 bit rotate
  - Flags updated based on result
    - CF
    - OF undefined if Count ≠ 1
- Used to rearrange information

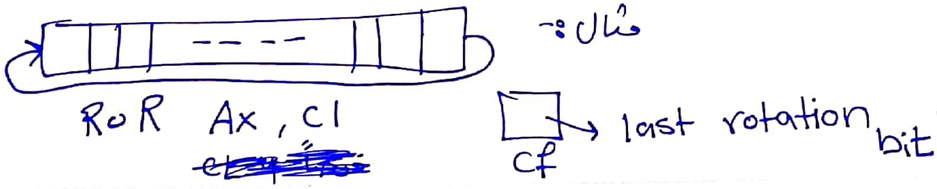
5.5 Rotate Instructions – Operation of the ROL Instruction











- Typical instruction—count of 1

  ROL AX,1

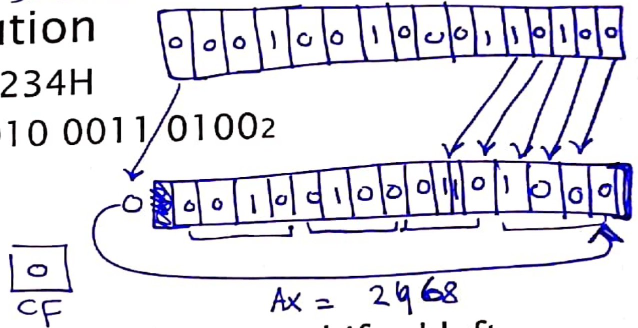
- Before execution

  Dest = (AX) = 1234H

  = 0001 0010 0011 0100_2

  Count = 1

  CF = 0



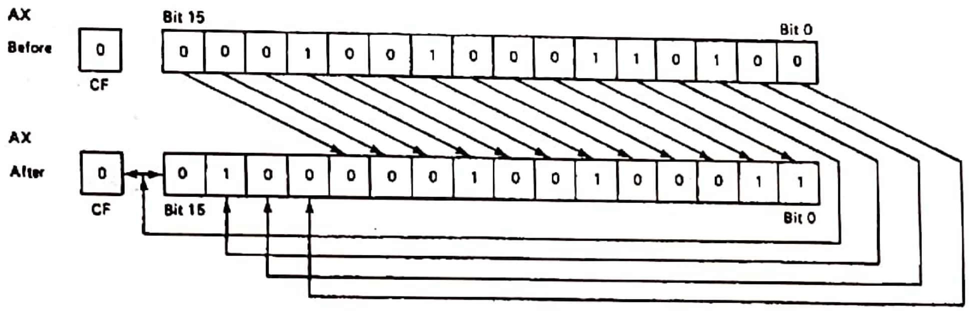
- Operation
  - The value in all bits of AX are shifted left one bit position
  - Value rotated out of the MSB is reloaded at LSB
  - Value rotated out of MSB is copied to carry flag
- After execution

  Dest = (AX) = 2468H

  = 0010 0100 0110 1000_2

  CF = 0

5.5 Rotate Instructions – Operation of the ROR Instruction



- Typical instruction—count in CL

  ROR AX,CL
- Before execution

  Dest = (AX) = 1234H

  = 0001001000110100_2

  Count = 04H

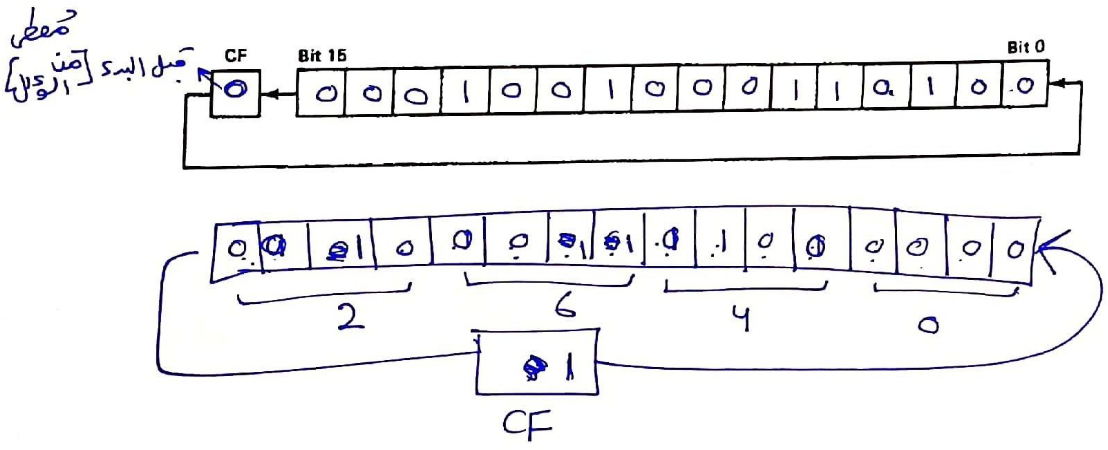
  CF = 0
- Operation
  - The value in all bits of AX are rotated right four bit positions
  - Values rotated out of the LSB are reloaded at MSB
  - Values rotated out of MSB copied to carry flag
- After execution

  Dest = (AX) = 4123H

  = 0100000100100011_2

  CF = 0
- Conclusion:
  - Note that the position of hex characters in AX have be rearranged

5.5 *Rotate Instructions* – Operation of the RCL Instruction



- Typical instruction—count in CL

  RCL BX,CL
- Before execution

  Dest = (BX) = 1234H

  = 0001001000110100₂

  Count = (CL) = 04H

  CF = 0
- Operation
  - The value in all bits of AX are rotated left four bit positions
  - Emptied MSBs are rotated through the carry bit back into the LSB
  - First rotate loads prior value of CF at the LSB
  - Last value rotated out of MSB retained in carry flag
- After execution

  Dest = (BX) = 2340H

  = 0010001101000000₂

  CF = 1

5.5 Rotate Instructions – Operation of the RCR Instruction

- EXAMPLE

What is the result in BX and CF after execution of the following instructions?

RCR BX, CL

Assume that, prior to execution of the instruction, $(CL)=04_{16}$, $(BX)=1234_{16}$, and $(CF)=0$

- Solution:

The original contents of BX are

$(BX) = 0001001000110100_2 = 1234_{16}$

Execution of the RCR command causes a 4-bit rotate right through carry to take place on the data in BX, the results are

$(BX) = 1000000100100011_2 = 8123_{16}$

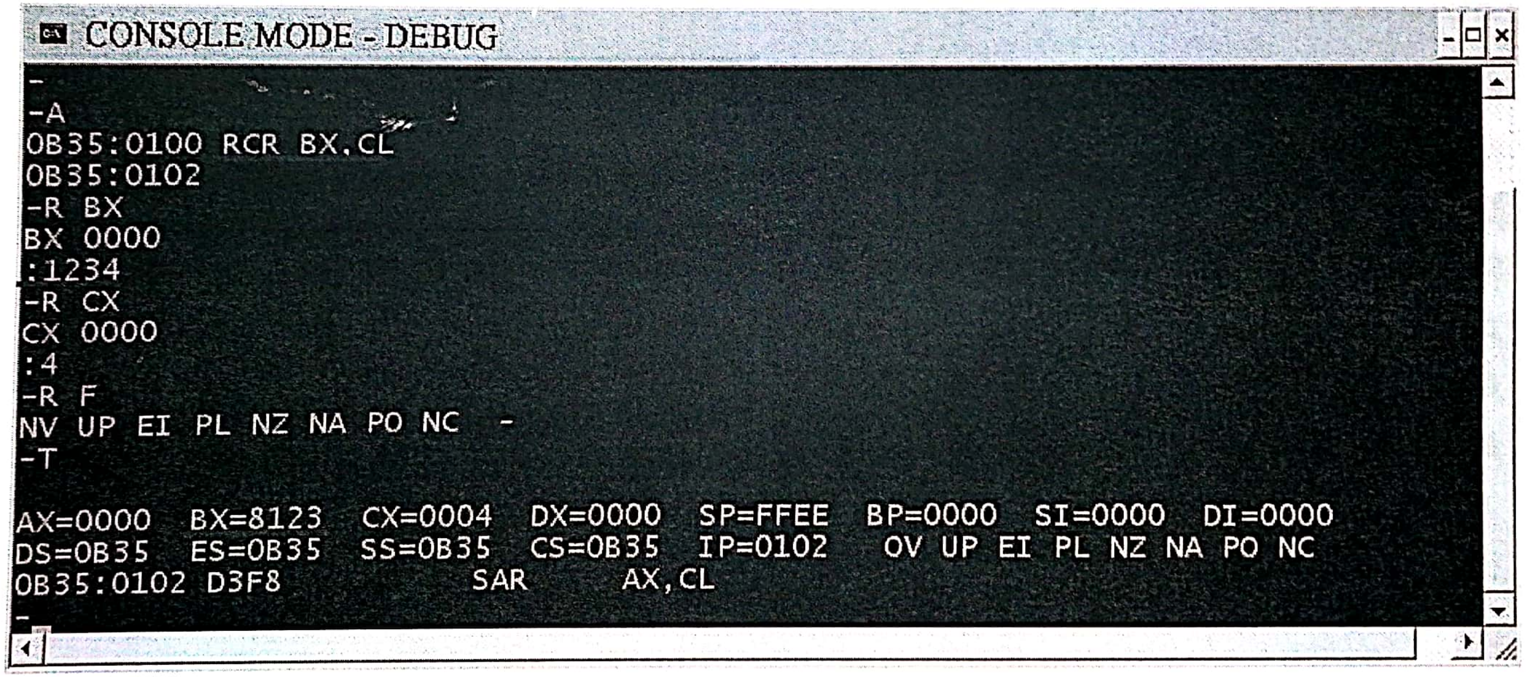
$(CF) = 0_2$

5.5 Rotate Instructions – Operation of the RCR Instruction

- EXAMPLE

  Verify the previous example using DEBUG program.

- Solution:



```
-
-A
0B35:0100 RCR BX.CL
0B35:0102
-R BX
BX 0000
:1234
-R CX
CX 0000
:4
-R F
NV UP EI PL NZ NA PO NC  -
-T

AX=0000  BX=8123  CX=0004  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B35  ES=0B35  SS=0B35  CS=0B35  IP=0102   OV UP EI PL NZ NA PO NC
0B35:0102 D3F8          SAR     AX,CL
-
```

5.5 Rotate Instructions

- EXAMPLE

Disassembly and addition of 2 hexadecimal digits stored as a byte in memory.

- Solution:

```
MOV  AL,[HEX_DIGITS]
MOV  BL,AL
MOV  CL,04H
ROR  BL,CL
AND  AL,0FH
AND  BL,0FH
ADD  AL,BL
```

1st Instruction → Loads AL with byte containing two hex digits
2nd Instruction → Copies byte to BL
3rd Instruction → Loads rotate count
4th instruction → Aligns upper hex digit of BL with lower digit in AL
5th Instruction → Masks off upper hex digit in AL
6th Instruction → Masks off upper four bits of BL
7th Instruction → Adds two hex digits

H.W. #5

- ☐ Solve the following problems from Chapter 5 from the course textbook:

  1, 10, 26, 38, 47

6.1 Flag-Control Instructions

:- عبارة عن (9 bits) نميه مجموعة من الinstruction للتعامل مع الـ(flag) .

- The flag-control instructions, when executed, directly affect the state of the flags. These instructions include:
  - ☐ LAHF (Load AH from flags)
  - ☐ SAHF (Store AH into flags)
  - ☐ CLC (Clear carry)
  - ☐ STC (Set carry)
  - ☐ CMC (Complement carry)
  - ☐ CLI (Clear interrupt)
  - ☐ STI (Set interrupt)

مهمتهم يحطوا داتا في (SR) أو يطلعوا داتا من (SR) .

CLC → مهمته يعمل (clear) للـ (carry flag)

لا يأخذ operands أم يدز .

Ax	AH	AL

لو بدي أخزن على الـ(flag) ، بخزن أولاً في (A H) بترتيب معين وبعدين بعمل (Store) للقيمة المُخزنة في AH حتى يودّيها على (SR) .

وحتى أقرأ الداتا المخزنة في الـ(flag) بستعمل (LAHF) وأخزن قيمة القراءة في (AH) .

6.1 Flag-Control Instructions –

Loading, Storing, and Modifying Flags

- Variety of flag control instructions provide support for loading, saving, and modifying content of the flags register
  - LAHF/SAHF → Load/store control flags
  - CLC/STC/CMC → Modify carry flag
  - CLI/STI → Modify interrupt flag
  - Modifying the carry flag—CLC/STC/CMC
  - Used to initialize the carry flag
  - Clear carry flag

    CLC

    0 → (CF)
  - Set carry flag

    STC

    1 → (CF)
  - Complement carry flag

    CMC

(CF*) → (CF) * stands for over bar (NOT)

  - Modifying the interrupt flag—CLI/STI
  - Used to turn on/off external hardware interrupts
  - Clear interrupt flag

    CLI

    0 → (IF) Disable interrupts
  - Set interrupt flag

    STI

    1 → (IF) Enable interrupts

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) ← $(\overline{CF})$	CF
CLI	Clear interrupt flag	(IF) ← 0	IF
STI	Set interrupt flag	(IF) ← 1	IF

ال oper ation تبعهم hidden

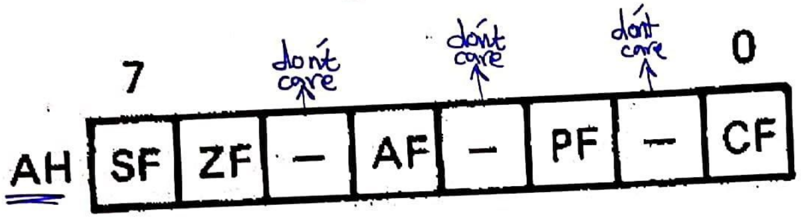
بيرجع على CF ويعملها inverting

بعمل set ال interrupt F بيشبه STC

بعمل clear ال interr F بيشبه clc

6.1 Flag-Control Instructions -

Loading and Storing the Flags Register



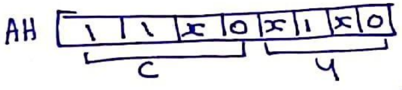
- SF = Sign flag
- ZF = Zero flag
- AF = Auxiliary
- PF = Parity flag
- CF = Carry flag
- — = Undefined (do not use)

* عن طريق (SAH) و (LAH) بنقدر نتحكم بخمسة من ال(flag) مو كلهم . SF | ZF/AF/PF | CF

مثال :- لنفرض انه بدي قيم ←

CF = 0
PF = 1
AF = 0
ZF = 1
SF = 1

بخزنهم في AH



فأنا هيك عملت Mov AH,C4 →

- Format of the flags in the AH register
- All loads and stores of flags take place through the AH register
  - B0 = CF
  - B2 = PF
  - B4 = AF
  - B6 = ZF
  - B7 = SF
- Load the AH register with the content of the flags registers

  LAHF
  (Flags) → (AH)
  Flags unchanged
- Store the content of AH into the flags register

  SAHF
  (AH) → (Flags)
  SF,ZF,AF,PF,CF → updated
- Application: saving a copy of the flags in memory and initializing with new values from memory

ولو بعدين عملت SAHF
يعني بدي أخزن الداتا يلي
موجودة ب AH (SR).

6.1 Flag-Control Instructions –
Loading and Storing the Flags Register

- EXAMPLE

  Of the three carry flag instructions CLC, STC, and CMC, only one is really independent instruction. That is, the operation that it provides cannot be performed by a series of the other two instructions. Determine which one of the carry instruction is the independent instruction.

- Solution:

* CLC ⇔ STC followed by a CMC
* STC ⇔ CLC followed by a CMC

Therefore, only CMC is the independent instruction.

يعني ضع في CF رقم واحد ثم اعكسها فتصبح صفر
هاي الخطوتين تكافئ خطوة واحدة وهي (CLC) أي ضع صفر
في CF .

نفس الشيء لو عملت
clear في البداية أي
وضعت صفر في CF ثم
عملت CMC أي عكستها .
هدول الخطوتين تكافئ خطوة
واحدة هي STC .

ممكن يجي السؤال كالتالي :-

اعمل clear لل CF بدون استعمال CLC .

or اعمل set لل CF بدون استعمال STC .

6.2 Compare Instruction :-

هو نوع من أنواع (arthemtic instruction)

- Compare instruction
  - Used to compare two values of data and update the state of the flags to reflect their relationship
- General format:

CMP D,S

- Operation: Compares the content of the source to the destination; updates flags based on result

(D) – (S) → Flags updated to reflect relationship

- Source and destination contents unchanged
- Allowed operand variations:
  - Values in two registers
  - Values in a memory location and a register
  - Immediate source operand and a value in a register or memory
- Allows SW to perform conditional control flow—typically testing of a flag by jump instruction
  - ZF = 1 → D = S = Equal
  - ZF = 0, CF = 1 → D < S = Unequal, less than
  - ZF = 0, CF = 0 → D > S = Unequal, greater than

Mnemonic	Meaning	Format	Operation	Flags affected
CMP	Compare	CMP D,S	(D) – (S) is used in setting or resetting the flags	CF,AF,OF,PF,SF,ZF

(a)

* مثال :-

CMP Ax, BX

يعني Ax-Bx

بس ما بنخزن الناتج في AX ، لأنه الهدف نشوف الناتج إما صفر وإما سالب وإما موجب .

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

* مثال :- لو عملنا A-b وكان ZF قيمته واحد ، معناته A=b

* لو بدي أقارن بين B, A ، بعمل طرح بينهم فإذا كان الناتج صفر (A-B =0) يعني بكون A ~~[illegible]~~ تساوي B .

ولو كان الجواب سالب بكون A<B وإذا الجواب موجب فإن A>B

6.2 Compare Instruction

- EXAMPLE

Describe what happens to the status flags as the sequence of instructions that follows is executed.

MOV AX, 1234H → Ax = 1234
MOV BX, ABCDH → Bx = ABCD
CMP AX, BX → Ax - Bx

- Solution: ①234 - Ⓐ BCD

* بلزمني 2 flag
حتى أعرف نتيجة CMP

$(AX) = 1234_{16} = 0001001000110100_2$ واضح من ال MSb انه Ax < Bx

$(BX) = ABCD_{16} = 1010101111001101_2$ ZF = 0 , CF = 1

$(AX) - (BX) = 0001001000110100_2 - 1010101111001101_2$

$= 0110\ 0110\ 0110\ 0111_2$

Therefore, ZF = 0, SF = 0, OF = 0, PF = 0, CF = 1, AF = 1

6.3 Control Flow and Jump Instructions

conditional (Jcc)

unconditional (JMP)

- **Control flow:** alternate the execution path of instructions in the program
- What are the pointers that keep track of the instructions being executed ?
- How it is possible to alternate the sequence of instructions being executed ?

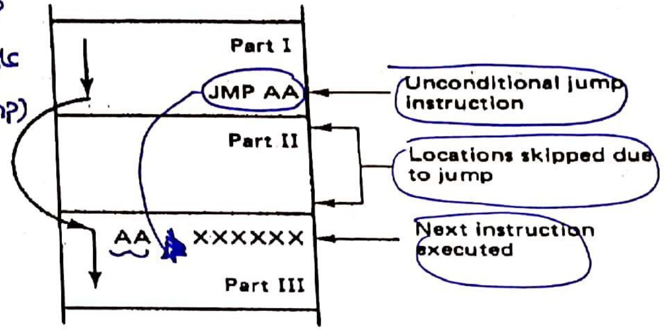
☑ Unconditional jump instruction
☑ Conditional jump instruction
☑ Branching structure - IF-THEN
☑ Loop program structure - REPEAT- UNTIL and WHILE-DO
☑ Applications using the loop and branch software structures

6.3 Control Flow and Jump Instructions

Unconditional and Conditional Jump Control Flow

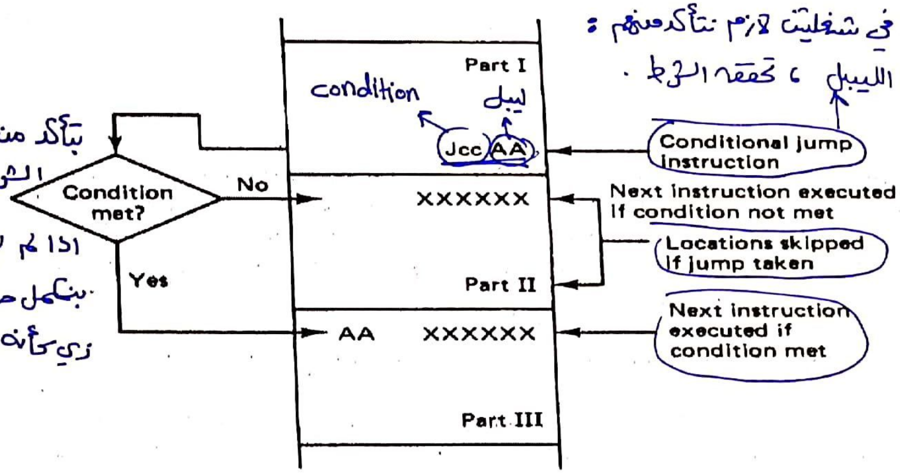
IP يتم تحديدها بناءً على الـ (Leable) يلي مع (Jmp).

تؤثر على ip و cs



(a)

في شغلتين لازم نتأكد منهم: الليبل ، تحقق الشرط.



بتأكد من تحققه الشرط.

اذا لم يتحقه بنكمل حل عادي زي كأنه ماخي Jmp

(b)

* لنفرض إنه عنا مجموعة من الـ (instruction) ، أكيد بتم تطبيقهم ورا بعض بالترتيب ، وظيفة (Jmp) إنه بنقلني من instruction لواحد تاني بمكان آخر. [مو بعده فوراً].

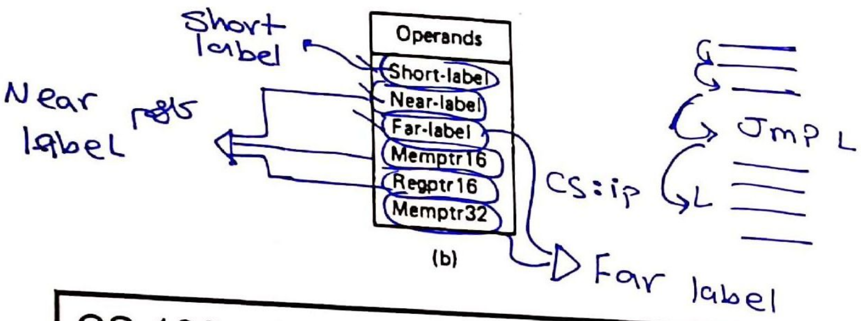
- Jump operation alters the execution path of the instructions in the program—flow control
  - Unconditional Jump
    - Always takes place
    - No status requirements are imposed
  - Example (part a)
    - JMP AA instructions in Part I executed
    - Control passed to next instruction identified by AA in Part III
    - Instructions in Part II skipped

6.3 Control Flow and Jump Instructions

Unconditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Affected flags
JMP	Unconditional jump	JMP Operand	Jump is initiated to the address specified by the operand	None

(a)



(b)

```
CS:100   lab    ADD BX,1234
CS:104          INC AX
CS:106          JMP lab
CS:108          NEG BX
```

short and near ~~Le~~ Label ⟹ بأثروا على ip فقط

Far LableL ⟹ بأثر على ip و cs

Unconditional jump instruction:

- Implements the unconditional jump operation needed by:
  - Branch program control flow structures
  - Loop program control flow structures
- General format:

JMP Operand

- Types of unconditional jumps
  - Intrasegment—branch to address is located in the current code segment
  - Only IP changes value
  - short-label
    - 8-bit signed displacement coded into the instruction
    - Immediate addressing
    - Range equal -126 to +129
    - New address computed as:

(Current IP) + short-label → (IP)

Jump to address = (Current CS) + (New IP)

  - near-label
    - 16-bit signed displacement coded in the instruction
- Example

JMP 1234H

6.3 Control Flow and Jump Instructions–Intersegment Unconditional Jump Operation

- Intersegment—branch to address is located in another code segment
  - Both CS and IP change values
  - far-label
    - 32-bit immediate operand coded into the instruction
    - New address computed as:
      - 1st 16 bits → (IP)
      - 2nd 16 bits → (CS)

    Jump to address = (New CS):(New IP)
  - memptr32
    - 32-bit value specified in memory
    - Memory indirect addressing
- Example

  JMP DWORD PTR [DI] →

  بروح على الميموري على العنوان DS:DI
  وأخذ 4 byte ، بحيث يكون
  Lower 2 byte = ip_{new}
  higher 2 byte = CS_{new}

  - Operation:

    (DS:DI) → new IP
    (DS:DI +2) → new CS
    Jump to address = (New CS):(New IP)

6.3 Control Flow and Jump Instructions

Conditional Jump Instruction

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)

same

same

يعني Jmp اذا
تحقق الشرط A
اللي هو CF=0
ZF=0

الكونديشن
بكونوا عبارة عن
· Flags

- Condition jump instruction
- Implements the conditional jump operation
- General format:

Jcc Operand

- cc = one of the supported conditional relationships
- Supports the same operand types as unconditional jump
- Operation: Flags tested for conditions defined by cc and:

If cc test True:

IP, or IP and CS are updated with new value

- Jump is taken
- Execution resumes at jump to target address

If cc test False:

IP, or IP and CS are unchanged

- Jump is not taken
  - Execution continues with the next sequential instruction
- Examples of conditional tests:

JC = jump on carry → CF = 1

JPE/JP = jump on parity even → PF =1

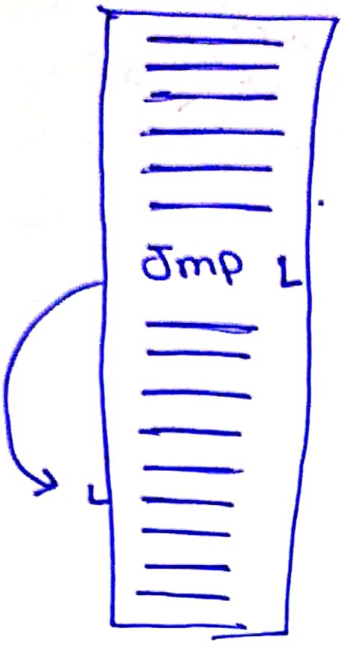
JE/JZ = jump on equal → ZF = 1

These instructions are associated with the compare instruction usually

Short label
حجمه 1 byte

Near label
حجمه 2 byte

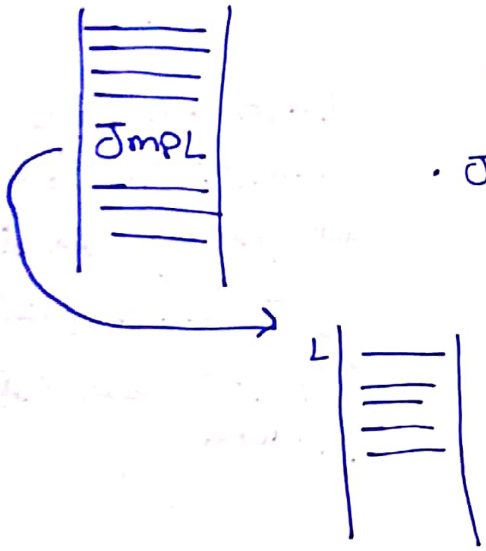
Far label
حجمه 4 byte



اذا كانوا
الinstruction يلي بدي أوزع عليهم
موجودين بنفس
الكود سيغمنت يلي منهم الJmp .
بتكون يا Short label يا Near label

بهاي الحالة بسمّي السيغمنت
intra - segment Jmp

فهون بتغير بس ip ، لكن CS
تبقى نفسها .



اذا كانوا الinstruction
موجودين بسيغمنت تاني
غير السيغمنت يلي منه Jmp .
بتكون Far .
بهاي الحالة ع أغير CS
و ip ،
وبصير اسمها
inter segment Jmp

مثال :-

Jmp 12_H

Short label = 1 byte

لنفرض ip الحالية = 20

[relative Jmp بنسميها]

$$ip_{new} = ip_{old} + Label$$

$$= 20 + 12$$

$$= 32_H$$

وتبقى cs ثابتة.

مثال :-

Jmp 1234_H

لنفرض أن ip الحالية = 20

$$ip_{new} = Label = 1234_H$$

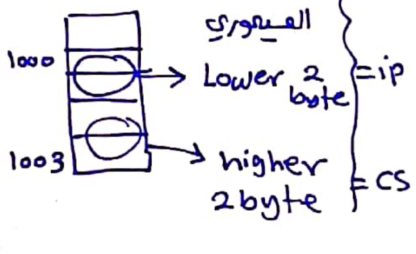
[هاد اسمه absolute Jmp]

وتبقى cs ثابتة .

لما يكون الليبل (طوله 2byte) = (1word) زي 1234 أو Ax فإن حجم الليبل يساوي 2 byte ويكون اسمه (Near label)

مثال :-

ممكن يكون مخزن داخل الميموري



Jmp 12345676_H

$$ip_{new} = Low\ 2byte = 5676$$

$$CS_{new} = high\ 2byte = 1234$$

لما يكون الليبل حجمه 4 byte يكون اسمه Far

6.3 Control Flow and Jump Instructions – Loop Program Structures

*

- EXAMPLE

  Implement an instruction sequence that calculates the absolute difference between the contents of AX and BX and places it in DX.

- Solution:

$DX = |Ax - Bx|$

```
        CMP AX, BX
        JC DIFF2
DIFF1:  MOV DX, AX
        SUB DX, BX ; (DX)=(AX)-(BX)
        JMP DONE
DIFF2:  MOV DX, BX
        SUB DX, AX ; (DX)=(BX)-(AX)
DONE:   NOP
```

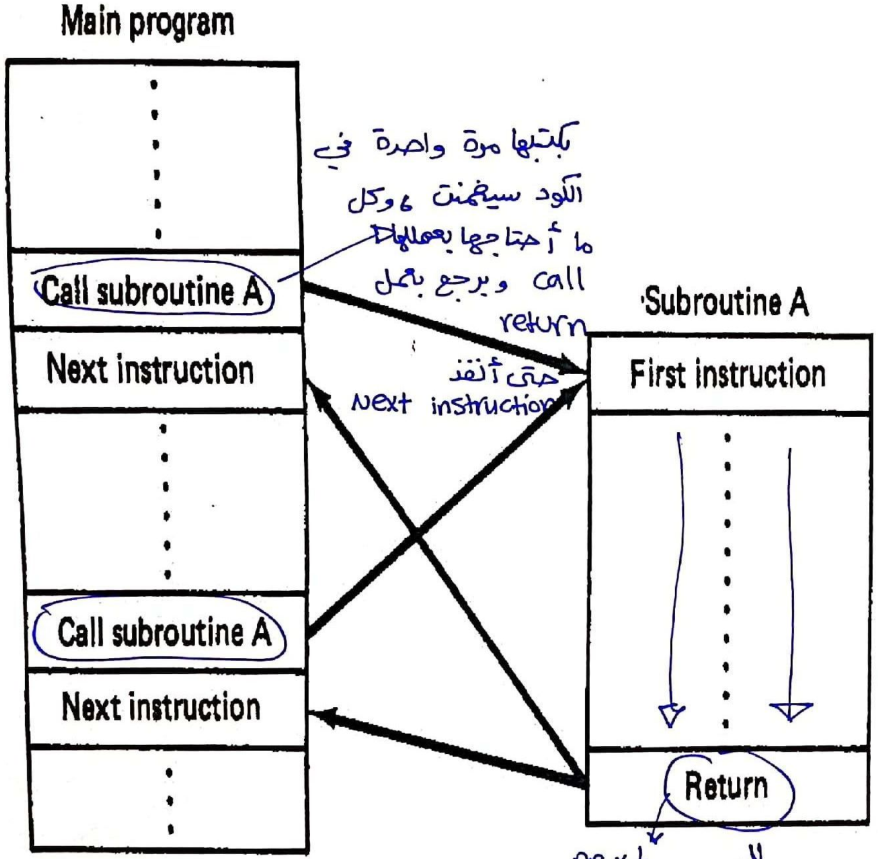
6.4 Subroutines and Subroutine-Handling Instructions

نفرض انه عنا مجموعة من الانستركشن وفي مجموعة من الانستركشن بتكرر:
في عنا اشي اسمه (density of the instruction) كل ما زادت، بزيد الحجم المطلوب لتخزين هاي الانستركشن في الكود سيغمنت.
وحتى نقلل من الانستركشن المتكررة يمكن أن نستبدلها ونكتبها على شكل فنكشن (function) بقدر أعمله (call).

- Subroutine—special segment of program that can be called for execution from any point in a program (like function)
- A subroutine is also known as a procedure.
- Program structure that implements HLL “functions” and “procedures”
- Written to perform an operation (function/procedure) that must be performed at various points in a program
- Written as a subroutine and only included once in the program
- A return instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.
- CALL and RET instructions
- PUSH and POP instructions

6.4 Subroutines and Subroutine-Handling Instructions

* call بتختلف عن Jump انه في return .



بكتبها مرة واحدة في الكود سيغمنت ، وكل ما أحتاجها بعملها call وبرجع بعمل return

حتى أنفذ Next instruction

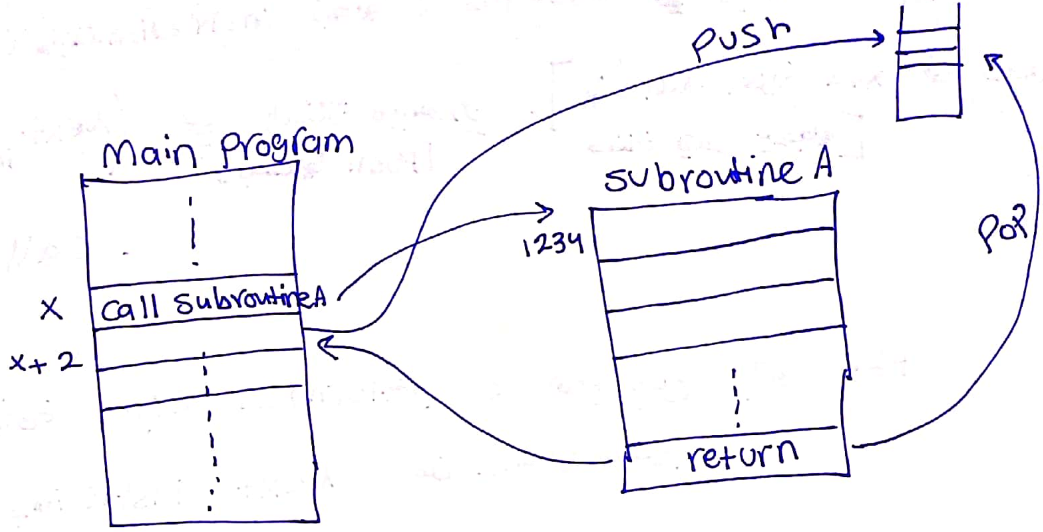
لل next instruction يلي بعد subrotine

- Example:
  - Instruction in Main part of program calls "Subroutine A"
  - Program flow of control transferred to first instruction of Subroutine A
  - Instructions of Subroutine A execute sequentially
  - Return initiated by last instruction of Subroutine A
  - Same sequence repeated when the subroutine is called again later in the program
- Instructions:
  - Call instruction—initiates the subroutine from the main part of program
  - Return instruction—initiates return of control to the main program at completion of the subroutine
  - Push and pop instructions used to save register content and pass parameters

The subroutine may be called and executed more than one time, but it is written one time

اد ١٢. ٤ـ

مثال :-
توضيحي



* اول اشي شو هو subroutine؟

بكون عنا برنامج عادي بحتوي مجموعة من instruction ، في بعض الأحيان تتكرر بعض ال (instruction) ، فبدل ما تتكرر وتوخذ مساحة أكبر ، بعوّض عنها ب function ومتى ما أحتاج تنفيذ هاي المجموعة من ال instruction بعملها call وبس أنتهي من تنفيذها برجع لباقي الـ instruction وبكمل تنفيذ .

* على سبيل المثال (ip = X) وصلنا لعند call subroutine A ، لازم أروح على ال instruction الخاصة فيه ، بكون كاتب في المثال call 1234 فبتتغير قيمة ip وبتصير (ip = 1234) ، بنفذ كل instruction وبس أخلصهم بعمل return حتى يرجعني للبرنامج الأساسي .

لما أرجع للبرنامج الاساسي بترجع ip لوضعها الطبيعي ، هي كانت قيمتها X هلأ بتصير قيمتها (ip = X+2) .

طبعاً return كيف بدها تعرف انها لازم ترجع لل (Next instruction) ؟

البرنامج وهو قاعد بنفّذ بالinstruction ، لوصل لعند (Call subroutine) فقبل ما يروح

على (subrotine) وينفذ instruction تبعتها ، لازم -تخزن قيمة IP

الخاصة بـ (Next instruction) في الستاك ميموري . [بهاد المثال . تخزن x+2 في tos

[Push بعملها] عشان يقدر يرجعها]

وبعدها بيعمل Call .

هلا صارت واضحة وظيفة الـ (return) ، هي بس بتعمل Pop

لـ ip الخاص بـ Next instruction من الستاك ميموري

هاد كله خاص بـ Near Label

أما بالنسبة لـ (Far Label) ، بتحدث نفس العملية بالزبط بس الفرق

الوحيد إنه بجيب Push لـ ip و CS على الستاك الميموري

وبعد إنهاء الـ (call) تقوم return باسترجاع (Pop) كل من ip و CS

من الستاك ميموري .

∴ return بتشتغل عكس Call .

6.4 Subroutines and Subroutine- Handling Instructions – Call Instruction

- regptr16
  - 16-bit value of IP specified as the content of a register
  - Register addressing
  - Example:

    CALL BX

    Cs : No change

    ip = label = BX
  - Operation:
  - Same as near-proc except

    (BX) → New IP
- memptr16
  - 16-bit value of IP specified as the content of a storage location in memory
  - Memory addressing modes—register addressing
  - Example

    CALL [BX] →

    يعني بروح على
    الميموري على العنوان
    DS : Bx
    وبجيب الداتا منه وبحسب
    قيمتها بحسب CS ، ip .
  - Same as near-proc except

    (DS:BX) → New IP

6.4 *Subroutines and Subroutine- Handling Instructions* – Call Instruction

- Intersegment—start address of the subroutine points to another code segment
  - Both CS and IP change values
  - far-proc
    - 32-bit immediate operand coded into the instruction
    - New address computed as:
      - 1st 16 bits → New IP
      - 2nd 16 bits → New CS

    Subroutine starts at = New CS:New IP
  - memptr32
    - 32-bit value specified in memory
    - Memory addressing modes—register indirect addressing
    - Example

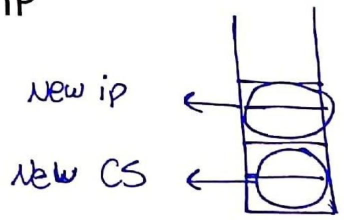
      CALL DWORD PTR [DI] → بروح على الميموري على العنوان DS:Di

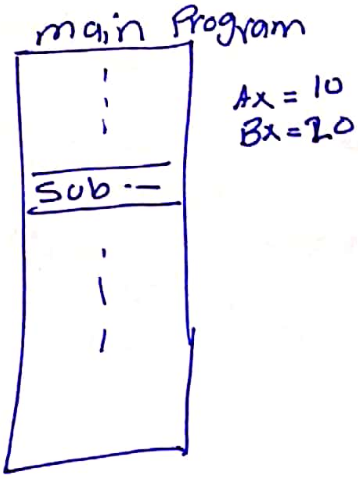
      وبجيب منه الداتا ، بكونوا أربعة بايت
    - Operation:

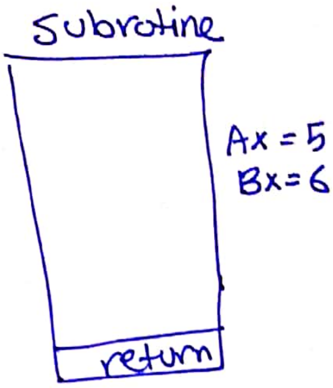
      (DS:DI) → New IP

      (DS:DI +2) → New CS

      Starting address of subroutine = New CS:New IP







لنفرض of Main Program تتعامل مع قيم Ax , Bx

وعند تنفيذ instruction وصلنا إلى call subroutine

وأيضاً تعاملنا مع قيم Ax , Bx وعند تنفيذ الـ instruction

عُدنا للوضع الطبيعي ، فهل نكمل تنفيذ البرنامج على قيم Ax ، Bx

الموجودة في main Program أم التي موجودة في subrotine؟

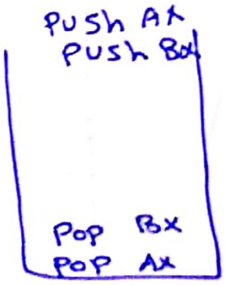
المفروض انه نكمل على قيم Ax، Bx الموجودة في Subroutine.

أما في حال أنا بدي التعامل مع Ax , Bx الموجودات في البرنامج الأساسي

لازم أعدل على structure تبع Subroutine .

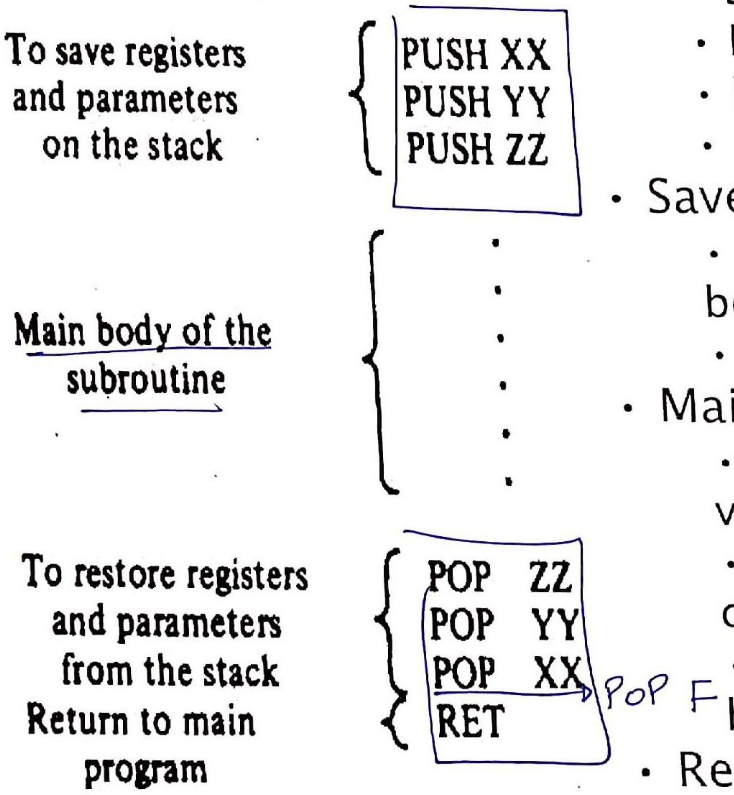
يعني بحتفظ بالداتا تبعت Ax , Bx فالي بعمله :-

بعمل Push Ax
Push Bx على Subroutine وبعمل Pop Bx
Pop ~~Bx~~ Ax بالاتجاه المعاكس



6.4 *Subroutines and Subroutine- Handling Instructions* - Structure of a Subroutine

Push F حتى احتفظ بال flag قبل sub



To save registers and parameters on the stack

```
PUSH XX
PUSH YY
PUSH ZZ
.
.
.
.
.
.
POP  ZZ
POP  YY
POP  XX
RET
```

Main body of the subroutine

To restore registers and parameters from the stack

Return to main program

POP F

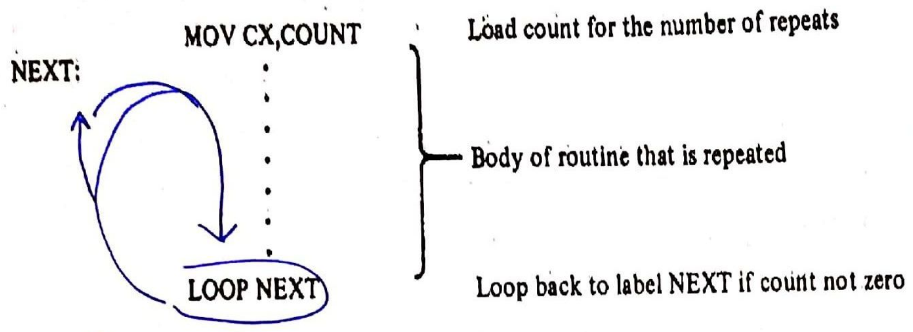
- Elements of a subroutine
  - Save of information to stack—PUSH
  - Main body of subroutine—Multiple instructions
  - Restore of information from stack—POP
  - Return to main program—RET
- Save of information
  - Must save content of registers/memory locations to be used or other program parameters (FLAGS)
  - PUSH, PUSHF
- Main body
  - Retrieve input parameters passed from main program via stack—stack pointer indirect address
  - Performs the algorithm/function/operation required of the subroutine
  - Prepare output parameters/results for return to main body via stack—stack pointer indirect addressing
- Restore information
  - Register/memory location contents saved on stack at entry of subroutine must be restored before return to main program—POP, POPF

6.5 The Loop and the Loop-Handling Instructions – Loop Instructions

Lable: عبارة عن al instruction
ودائماً يأخذ Short

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label تتأكد انه الكاونتر ما وصل الصفر.	(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 equall يعني ZF=1 تتأكد من قيمة الكاونتر انه ما وصل الصفر وانه (ZF=1)	(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 ~~ZF~~ تتأكد من الكاونتر ما وصل صفر ، وانه ZF = 0	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 0; otherwise, execute next sequential instruction

6.5 *The Loop and the Loop-Handling Instructions* – Loop Program Structure and Operation



(a)

لما تصير قيمة Cx صفر

بوقف اللوب .

مثال :- Loop cx

Cx=10

ع يضله يعمل لوب

وكل لوب بنقص

لحد ما يوصل لصفر

وقتها بوقف اللوب .

```
        MOV    AX,DATASEGADDR
        MOV    DS,AX
        MOV    SI,BLK1ADDR
        MOV    DI,BLK2ADDR
        MOV    CX,N
NXTPT:  MOV    AH,[SI]
        MOV    [DI],AH
        INC    SI
        INC    DI
        LOOP   NXTPT
        HLT
```

(b)

- Structure of a loop
- Initialization of the count in CX
- Body—instruction sequence that is to be repeated; short label identifying beginning
- Loop instruction– determines if loop is complete or if the body is to repeat
- Example

1. Initialize data segment, source and destination block pointers, and loop count
2. Body of program is executed—source element read, written to destination, and then both pointers incremented by 1
3. Loop test
   a. Contents of CX decremented by 1
   b. Contents of CX check for zero
   c. If CX = 0, loop is complete and next sequential instruction (HLT) is executed
   d. If CX ≠ 0, loop of code is repeated by returning control to the instruction corresponding to the Short-Label (NXTPT:) operand

6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

- EXAMPLE

Given the following sequence of instructions, explain what happens as they are executed.

```
          MOV DL, 05
          MOV AX, 0A00H
          MOV DS, AX
          MOV SI, 0
          MOV CX, 0FH
AGAIN:    INC SI
          CMP [SI], DL
          LOOPNE AGAIN
```

يعني بدي ألف 15 مرة .

بقارن بين DL وبين الداتا يلي داخل العنوان DS:SI

بدور على رقم (5) ، بشوف هل SI تساوي خمسة

ازا ما كانت 5 ، بكمل اللوب وبتصير SI تأشر

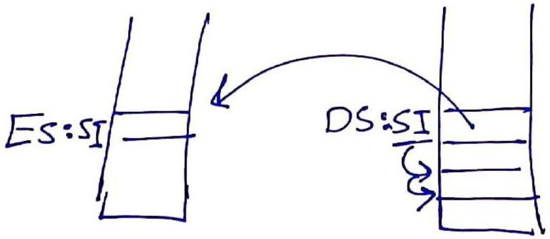
على next byte وهكذا...

6.6 String and String-Handling Instructions – String Instructions

وظيفتها تنقل
داتا من
داتا سيقمنت
إلى اكسترا
سيقمنت.

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

بنحطهم
بـ loop
ما بنستخدم
repeat معهم



ازا عملنا Mov
لـ byte
بنزيد SI بمقدار
(1) وازا عملنا
move لـ word بنزيد (SI). بمقدار 2.

6.6 String and String-Handling Instructions –

Move String Instruction

- Move string instruction
  - Used to move an element of data between a source and destination location in memory:
  - General format:

    MOVSB—move string byte
    MOVSW—move string word
  - Operation: Copies the content of the source to the destination; autoincrements/decrements both the source and destination addresses

    ((DS)0+(SI)) → ((ES)0+(DI))
    (SI) ±1 or 2 → (SI)
    (DI) ±1 or 2 → (DI)
  - Direction flag determines increment/decrement

    DF = 0 → autoincrement → $SI = SI + \substack{1 \\ 2}$
    $DI = DI + \substack{1 \\ 2}$
    DF = 1 → autodecrement ↳ $SI = SI - \substack{1 \\ 2}$
    $DI = DI - \substack{1 \\ 2}$